

Algorithms and Software for Intrusive UQ

Eric Phipps (etphipp@sandia.gov)
Sandia National Laboratories

**Fourth QUEST Uncertainty Quantification
Summer School**

August 19-21, 2015



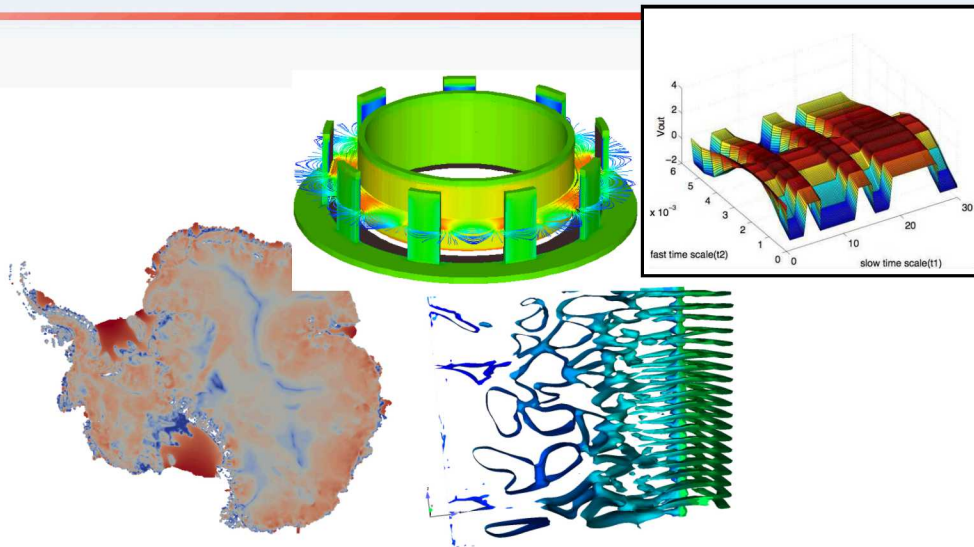
About Me

- **Sandia National Laboratories**
 - Center for Computing Research (CCR)
 - Computer Science Research Institute (CSRI)
 - Optimization and Uncertainty Quantification Department 1441
- **Been at Sandia ~13 years**
 - ~3 years as LTE (like a post-doc)
 - ~10 years as staff member
- **Work focuses on embedded (a.k.a. intrusive) analysis algorithms for large-scale science/engineering simulations:**
 - Continuation and bifurcation analysis of dynamical systems
 - Automatic differentiation for sensitivity analysis, optimization, UQ
 - Stochastic Galerkin UQ methods
 - UQ methods for emerging computer architectures
- **Helped develop several software packages:**
 - LOCA: Library of Continuation Algorithms
 - Sacado: Automatic Differentiation of C++ Codes
 - Stokhos: Embedded Stochastic Galerkin and Sampling Methods
 - Albany: Agile Components/Embedded Analysis Demonstration Application

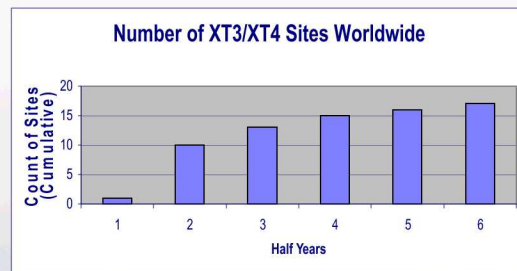
Multidisciplinary Research within CCR



Leading Edge Algorithms and
Enabling Technologies



State-of-Art Computational
Science Applications



Scalable HPC Architectures
and Systems

Model within Sandia for External Collaborations

Computer Science Research Institute (CSRI)

- Annual Statistics
 - 14 projects
 - 4 workshops
 - 102 additional visitors from 67 institutions
 - 37 summer students
 - 3 sabbaticals
- 73% of CCR's math and computer science hires in 2005-2007 had prior contact through CSRI collaborations



What Are Embedded/Intrusive UQ Methods?

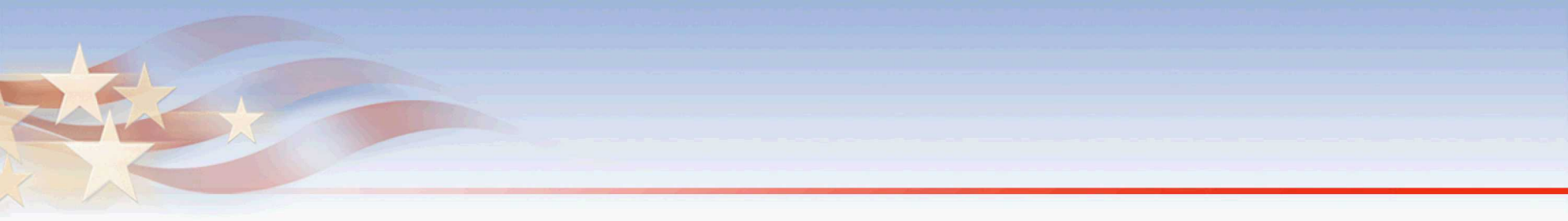
- Any method that is not “non-intrusive” or “black-box”
 - I.e., require more from the application than “parameter to response mapping”.
- Examples:
 - Methods that link a sampling engine into their code
 - E.g., Dakota library-mode
 - Methods that require local Taylor series
 - Stochastic Galerkin methods
 - Methods that propagate more than one sample at a time
 - Methods that rely on embedded optimization
- Why would we ever want to use an intrusive method?
 - Potentially improved performance, robustness, scalability or accuracy
 - With the right software infrastructure, they aren’t necessarily that much more difficult to do
 - The goal of these lectures





Agenda

- **Lecture 1: Derivative Methods and Software**
 - Use of derivatives in UQ
 - Software for computing derivatives easily and efficiently
 - Sacado package demo
- **Lecture 2: Stochastic Galerkin Methods and Software**
 - Overview of stochastic Galerkin methods
 - Approaches for implementing stochastic Galerkin methods
 - Stokhos package demo
- **Lecture 3: Embedded UQ Methods in Albany**
 - How to design C++ codes effectively using templates
 - Overview of Albany code structure
 - Demo on various UQ-related Albany topics
- **Lecture 4: Uncertainty Propagation for Next-generation Architectures**
 - Ensemble propagation
 - Stochastic Galerkin
 - Code demo



Derivative Methods and Software



Derivatives

- Derivatives are a very useful tool for simulation analysis in general
 - UQ
 - Error estimation
 - Optimization
 - Stability analysis
- Derivatives can be computed efficiently with good scalability to large numbers of independent variables
 - First derivatives, especially adjoint methods

Mathematical Model

$$\begin{aligned}0 &= f(\dot{u}(t), u(t), y, t), \quad t \in [t_0, t_f], \quad y \in \Gamma \subset \mathbb{R}^M \\ u(t_0) &= u_0(y) \\ \dot{u}(t_0) &= \dot{u}_0(y) \\ v(y) &= \int_{t_0}^{t_f} g(\dot{u}(t), u(t), y, t) dt + h(\dot{u}(t_f), u(t_f), y) \\ \xi : \Omega &\rightarrow \Gamma \subset \mathbb{R}^M, \text{ density } \rho, \xi_i \text{ independent}\end{aligned}$$





Derivatives in UQ

- **Approximate variance from first derivatives**

$$v(\xi) \approx v(0) + \sum_{i=1}^M \frac{\partial v}{\partial y_i}(0) \xi_i \implies \text{Var}(v) \approx \sum_{i=1}^M \left(\frac{\partial v}{\partial y_i}(0) \right)^2, \quad \xi_i \text{ i.i.d, } \mathbb{E}\xi_i = 0, \mathbb{E}\xi_i^2 = 1$$

- Only accurate if response is (nearly) linear over range of uncertain variables
- Only relevant if uncertain inputs are (nearly) Gaussian
- Very common approximation in neutronics calculations

- **Piecewise-linear response surface**

- Apply the above Taylor series locally at some chosen set of points
- Estep, et al

- **Quick and dirty dimension reduction**

- Only include random variables with significant sensitivity

- **Guide sampling of posterior PDF in Bayesian inference problems**

- Ghattas, et al

- **Derivative-enhanced sampling**

- PCE (example now)
- VPS (example later)

- **Derivative-based stochastic optimization**



Polynomial Chaos Expansions (PCE)

- **Steady-state finite dimensional model problem:**

Find $u(\xi)$ such that $f(u, \xi) = 0$, $\xi : \Omega \rightarrow \Gamma \subset R^M$, density ρ

- **(Global) Polynomial Chaos approximation:**

$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi), \quad \langle \psi_i \psi_j \rangle \equiv \int_{\Gamma} \psi_i(y) \psi_j(y) \rho(y) dy = \delta_{ij} \langle \psi_i^2 \rangle$$

- **Multivariate orthogonal polynomials**
 - **Typically constructed as tensor products with total order at most N**
 - **Can be adapted (anisotropic, local support)**
- **Non-intrusive polynomial chaos (NIPC, NISP):**

$$u_i = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} \hat{u}(y) \psi_i(y) \rho(y) dy \approx \frac{1}{\langle \psi_i^2 \rangle} \sum_{k=0}^Q w_k u^k \psi_i(y^k), \quad f(u^k, y^k) = 0$$

- **Sparse-grid quadrature methods for scalability to moderate stochastic dimensions**
- **Need to be careful to ensure quadrature rule preserves discrete orthogonality**



Extending Scalability Through Derivative Enhancement

- **PCE:**

- Linear regression approach for approximating PCE coefficients

$$\hat{u}(y_k) = \bar{u}_k \implies \sum_{i=0}^P u_i \psi_i(y_k) = \bar{u}_k, \quad f(\bar{u}_k, y_k) = 0, \quad k = 0, \dots, Q$$

- Mitigate ill-conditioning by over-sampling $Q \sim 2 \cdot P$
 - Equality-constrained least-squares
- Reduce number of samples by adding derivative equations

$$\sum_{i=0}^P u_i \psi_i(x_k) = \bar{u}_k, \quad k = 0, \dots, \frac{Q}{M+1}$$
$$\sum_{i=0}^P u_i \frac{\partial \psi_i}{\partial x}(x_k) = \frac{\partial \bar{u}_k}{\partial x_k}, \quad k = 0, \dots, \frac{Q}{M+1}$$

- **Stochastic Collocation**

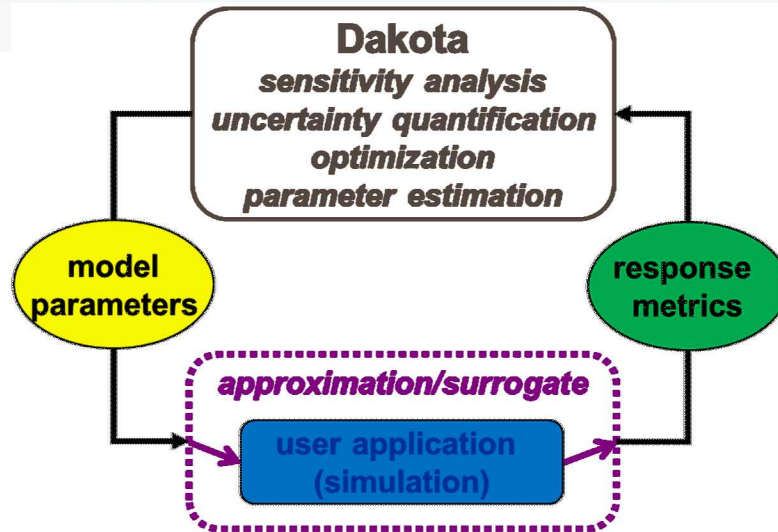
- Gradient enhanced interpolants (splines, Hermite interpolating polynomials)

DAKOTA

Research, development, & deployment of advanced algorithms for simulation-based assessment and design*



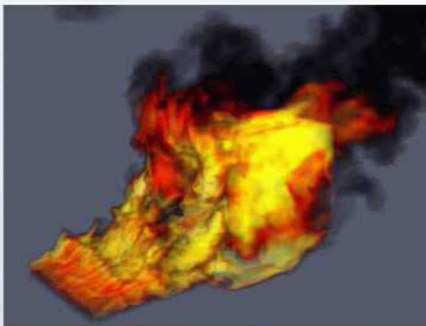
<http://dakota.sandia.gov>



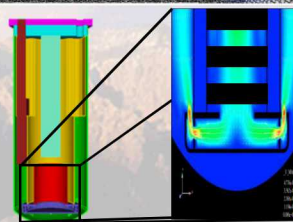
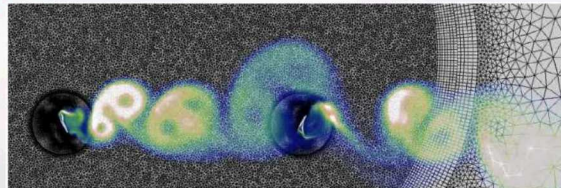
Iterative systems analysis
Multilevel parallel computing
Simulation management

Impact across a variety of DOE mission areas

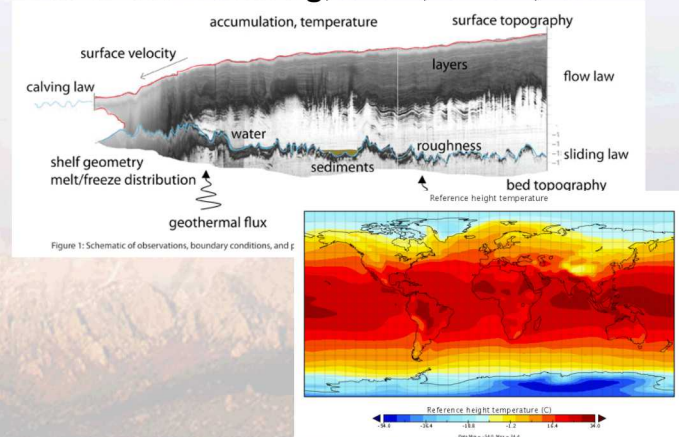
Stockpile (NNSA ASC)
 Abnormal environments



Energy (ASCR, EERE, NE)
 Wind turbines, nuclear reactors



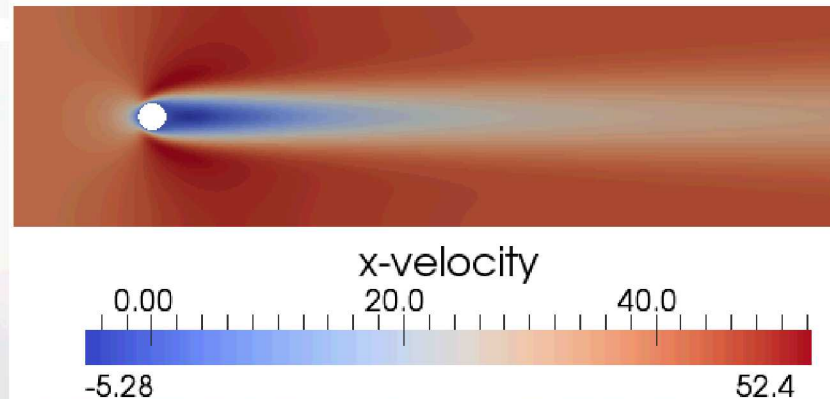
Climate (SciDAC, CSSEF)
 Ice sheet modeling, CISM, CESM, ISSM



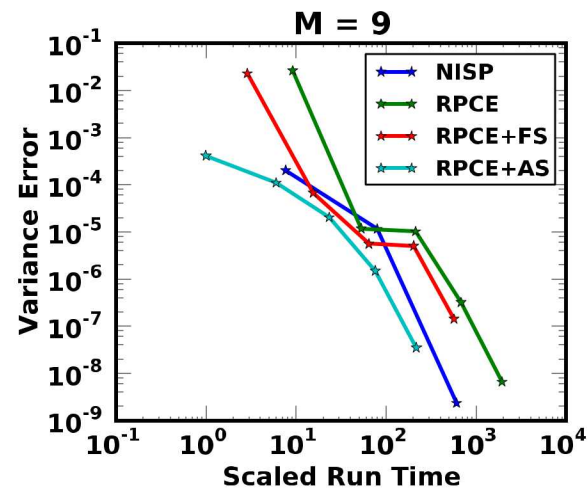
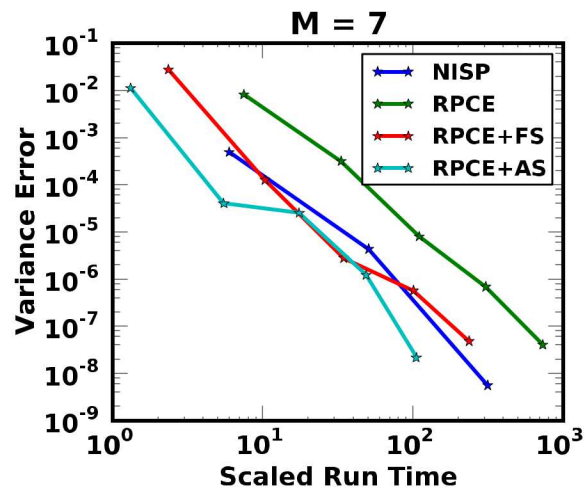
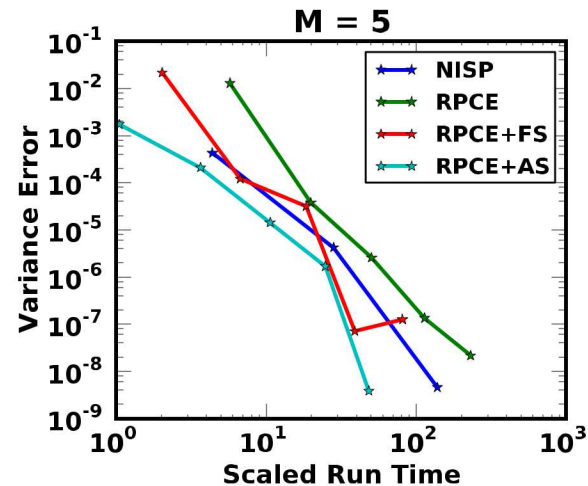
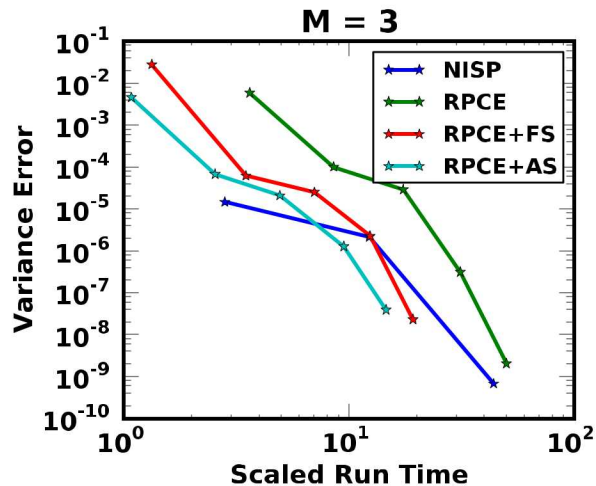
*M. Eldred, B. Adams, et al (SNL)

Small Model Problem

- 2-D incompressible fluid flow past a cylinder
 - **Uncertain viscosity field**
- Albany code (A. Salinger *et al*, SNL)
- Gradient-enhanced PCE implemented by Dakota



Comparisons on Model Problem



Computing derivatives efficiently in large-scale codes

- These techniques require accurate evaluations of partial derivatives
- These can always be derived and coded by-hand
 - Time consuming
 - Error prone
 - Distracts code developers from focusing on physics
- One alternative is numerical differentiation
 - Difficult to make accurate, robust
 - Can be very expensive
- A better alternative is automatic differentiation
 - Evaluate *analytic* derivatives automatically, efficiently



What Is Automatic Differentiation (AD) ?

- Analytic derivatives without hand-coding
- All differentiable computations are composition of simple operations
 - $\sin()$, $\log()$, $+$, $*$, $/$, etc...
- We know the derivatives of these simple operations
- We have the chain rule from calculus
- Systematic application of the chain rule through your computation differentiating each statement line-by-line.

A Simple Example

$$y = \sin(e^x + x \log x), \quad x = 2$$

$$x \leftarrow 2$$

$$t_1 \leftarrow e^x$$

$$t_2 \leftarrow \log x$$

$$t_3 \leftarrow x t_2$$

$$t_4 \leftarrow t_1 + t_3$$

$$y \leftarrow \sin t_4$$

x	$\frac{d}{dx}$
2.000	1.000
7.389	7.389
0.693	0.500
1.386	1.693
8.775	9.082
0.605	-7.233

**Analytic derivative evaluated to machine
precision**



Related Methods

$$y = \sin(e^x + x \log x), \quad x = 2$$

Automatic Differentiation

$$\begin{array}{ll}
 x \leftarrow 2 & \frac{dx}{dx} \leftarrow 1 \\
 t_1 \leftarrow e^x & \frac{dt_1}{dx} \leftarrow t_1 \frac{dx}{dx} \\
 t_2 \leftarrow \log x & \frac{dt_2}{dx} \leftarrow \frac{1}{x} \frac{dx}{dx} \\
 t_3 \leftarrow x t_2 & \frac{dt_3}{dx} \leftarrow t_2 \frac{dx}{dx} + x \frac{dt_2}{dx} \\
 t_4 \leftarrow t_1 + t_3 & \frac{dt_4}{dx} \leftarrow \frac{dt_1}{dx} + \frac{dt_3}{dx} \\
 y \leftarrow \sin t_4 & \frac{dy}{dx} \leftarrow \cos(t_4) \frac{dt_4}{dx}
 \end{array}$$

$$\frac{dy}{dx} = -7.233\ 340\ 400\ 802\ 3158$$

Symbolic Differentiation

$$\begin{aligned}
 \frac{dy}{dx} &= \cos(e^x + x \log x) \cdot \\
 &\quad (e^x + \log x + 1)
 \end{aligned}$$

$$\begin{array}{l}
 x \leftarrow 2 \\
 t_1 \leftarrow e^x \\
 t_2 \leftarrow \log x \\
 t_3 \leftarrow x t_2 \\
 t_4 \leftarrow t_1 + t_3 \\
 y \leftarrow \sin t_4
 \end{array}$$

$$\begin{array}{l}
 s_1 \leftarrow \cos t_4 \\
 s_2 \leftarrow t_1 + t_2 \\
 s_3 \leftarrow s_2 + 1
 \end{array}$$

$$\frac{dy}{dx} \leftarrow s_1 s_3$$

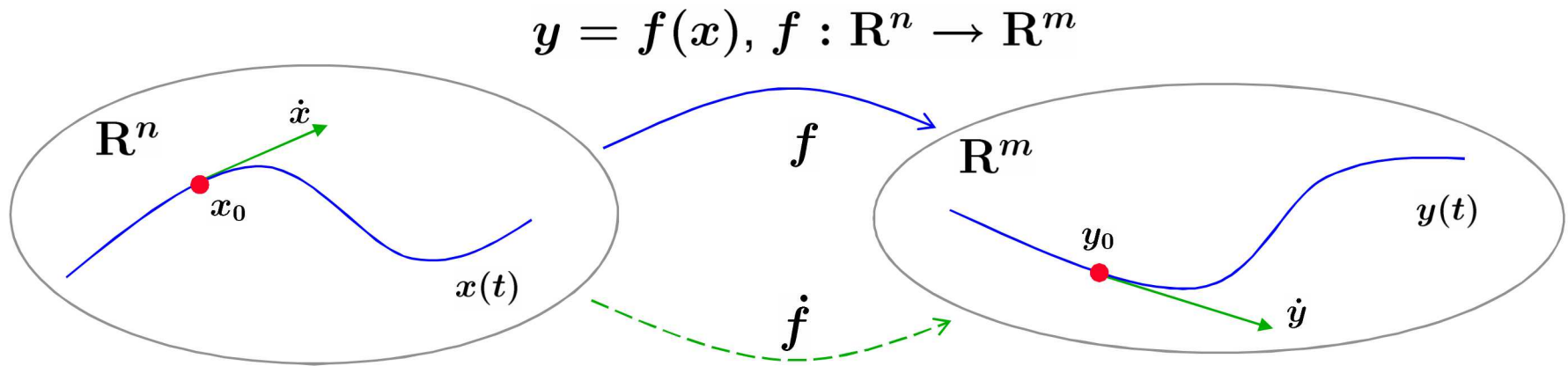
$$\frac{dy}{dx} = -7.233\ 340\ 400\ 802\ 3167$$

Finite Differencing

$$\begin{aligned}
 \frac{dy}{dx} &\approx \frac{y(2 + \varepsilon) - y(2)}{\varepsilon} \\
 &\approx -7.233\ 343\ 187
 \end{aligned}$$



Tangent Propagation



- **Tangents**

$$y(t) = f(x(t)) \implies \dot{y} \equiv \left. \frac{dy}{dt} \right|_{t=t_0} = \frac{\partial f}{\partial x} \dot{x}$$

- **For each intermediate operation**

$$c = \varphi(a, b) \implies \dot{c} = \frac{\partial \varphi}{\partial a} \dot{a} + \frac{\partial \varphi}{\partial b} \dot{b}$$

- **Tangents map forward through evaluation**

Operation	Tangent Rule
$c = a + b$	$\dot{c} = \dot{a} + \dot{b}$
$c = a - b$	$\dot{c} = \dot{a} - \dot{b}$
$c = ab$	$\dot{c} = a\dot{b} + \dot{a}b$
$c = a/b$	$\dot{c} = (\dot{a} - c\dot{b})/b$
$c = a^b$	$\dot{c} = c(\dot{b} \log(a) + \dot{a}b/a)$
$c = \sin(a)$	$\dot{c} = \cos(a)\dot{a}$
$c = \log(a)$	$\dot{c} = \dot{a}/a$

A Simple Tangent Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$

$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}$$

Given $x_1, x_2, \dot{x}_1, \dot{x}_2$:

$$s_1 \leftarrow e^{x_1}$$

$$\dot{s}_1 \leftarrow s_1 \dot{x}_1$$

$$s_2 \leftarrow x_1 x_2$$

$$\dot{s}_2 \leftarrow x_1 \dot{x}_2 + \dot{x}_1 x_2$$

$$s_3 \leftarrow s_1 + s_2$$

$$\dot{s}_3 \leftarrow \dot{s}_1 + \dot{s}_2$$

$$y_1 \leftarrow \sin(s_3)$$

$$\dot{y}_1 \leftarrow \cos(s_3) \dot{s}_3$$

$$s_4 \leftarrow x_1^2$$

$$\dot{s}_4 \leftarrow 2x_1 \dot{x}_1$$

$$s_5 \leftarrow y_1 + s_4$$

$$\dot{s}_5 \leftarrow \dot{y}_1 + \dot{s}_4$$

$$y_2 \leftarrow y_1 / s_5$$

$$\dot{y}_2 \leftarrow (\dot{y}_1 - y_2 \dot{s}_5) / s_5$$

Return $y_1, y_2, \dot{y}_1, \dot{y}_2$



Forward Mode AD via Tangent Propagation

- Choice of space curve $x(t)$ is arbitrary
- Tangent $\dot{y}$ depends only on $x_0, \dot{x}$
- Given x_0 and v :

$$y(t) = f(x_0 + vt) \implies \dot{y} = \frac{\partial f}{\partial x_0} v \quad \text{Jacobian vector product}$$

- Propagate p vectors $v_1, \dots, v_p$ simultaneously

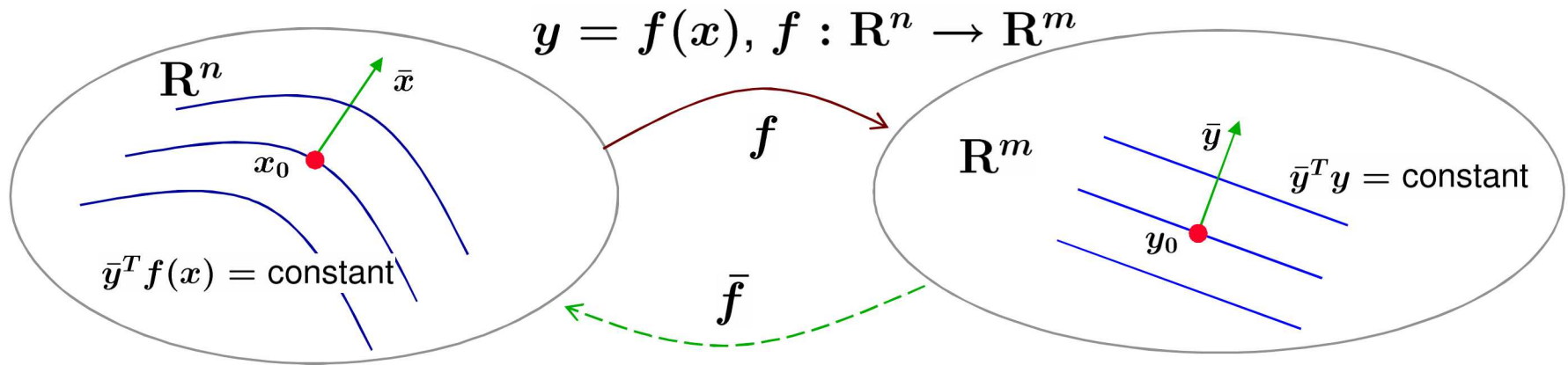
$$[\dot{y}_1 \dots \dot{y}_p] = \frac{\partial f}{\partial x_0} [v_1 \dots v_p] = \frac{\partial f}{\partial x_0} V \quad \text{Jacobian matrix product}$$

- Forward mode AD:

$$(x, V) \rightarrow \left(f(x), \frac{\partial f}{\partial x} V \right)$$

- V is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1 + 1.5p)\text{time}(f)$
- Jacobian-vector products, directional derivatives, Jacobians for $m \geq n$

Gradient Propagation



- Gradients**

$$z = \bar{y}^T y = \bar{y}^T f(x) \implies \bar{x} \equiv \left(\frac{\partial z}{\partial x} \right)^T = \left(\frac{\partial f}{\partial x} \right)^T \bar{y}$$

- For each intermediate operation**

$$c = \varphi(a, b) \implies \begin{aligned} \bar{a} &= \frac{\partial z}{\partial a} = \frac{\partial z}{\partial c} \frac{\partial c}{\partial a} = \bar{c} \frac{\partial \varphi}{\partial a}, \\ \bar{b} &= \frac{\partial z}{\partial b} = \frac{\partial z}{\partial c} \frac{\partial c}{\partial b} = \bar{c} \frac{\partial \varphi}{\partial b} \end{aligned}$$

- Gradients map backward through evaluation**

Operation	Gradient Rule
$c = a + b$	$\bar{a} = \bar{c}, \quad \bar{b} = \bar{c}$
$c = a - b$	$\bar{a} = \bar{c}, \quad \bar{b} = -\bar{c}$
$c = ab$	$\bar{a} = \bar{c}b, \quad \bar{b} = \bar{c}a$
$c = a/b$	$\bar{a} = \bar{c}/b, \quad \bar{b} = -\bar{c}c/b$
$c = a^b$	$\bar{a} = \bar{c}c \log(a), \quad \bar{b} = \bar{c}cb/a$
$c = \sin(a)$	$\bar{a} = \bar{c} \cos(a)$
$c = \log(a)$	$\bar{a} = \bar{c}/a$

A Simple Gradient Example

$$y_1 = \sin(e^{x_1} + x_1 x_2)$$

$$y_2 = \frac{y_1}{y_1 + x_1^2}$$

$$\begin{bmatrix} \bar{x}_1 \\ \bar{x}_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}^T \begin{bmatrix} \bar{y}_1 \\ \bar{y}_2 \end{bmatrix}$$

$$c = \varphi(a, b) \implies \begin{aligned} \bar{a} &= \bar{c} \frac{\partial \varphi}{\partial a}, \\ \bar{b} &= \bar{c} \frac{\partial \varphi}{\partial b} \end{aligned}$$

Given $x_1, x_2, \bar{y}_1, \bar{y}_2$:

$$s_1 \leftarrow e^{x_1}$$

$$s_2 \leftarrow x_1 x_2$$

$$s_3 \leftarrow s_1 + s_2$$

$$y_1 \leftarrow \sin(s_3)$$

$$s_4 \leftarrow x_1^2$$

$$s_5 \leftarrow y_1 + s_4$$

$$y_2 \leftarrow y_1 / s_5$$

$$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{y}_2 / s_5, \quad \bar{s}_5 \leftarrow -y_2 \bar{y}_2 / s_5$$

$$\bar{y}_1 \leftarrow \bar{y}_1 + \bar{s}_5, \quad \bar{s}_4 \leftarrow \bar{s}_5$$

$$\bar{x}_1 \leftarrow 2\bar{s}_4 x_1$$

$$\bar{s}_3 \leftarrow \bar{y}_1 \cos(s_3)$$

$$\bar{s}_1 \leftarrow \bar{s}_3, \quad \bar{s}_2 \leftarrow \bar{s}_3$$

$$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_2 x_2, \quad \bar{x}_2 \leftarrow \bar{s}_2 x_1$$

$$\bar{x}_1 \leftarrow \bar{x}_1 + \bar{s}_1 s_1$$

Return $y_1, y_2, \bar{x}_1, \bar{x}_2$

Reverse Mode AD via Gradient Propagation

- Choice of normal $\bar{y}$ is arbitrary
- Gradient $\bar{x}$ depends only on $x_0, \bar{y}$
- Given x_0 and w :

$$\bar{y} = w, y = f(x) \implies \bar{x} = \left(\frac{\partial f}{\partial x} \right)^T w \quad \text{Jacobian-transpose vector product}$$

- Propagate q vectors $w_1, \dots, w_q$ simultaneously

$$[\bar{x}_1 \dots \bar{x}_q] = \left(\frac{\partial f}{\partial x} \right)^T [w_1 \dots w_q] = \left(\frac{\partial f}{\partial x} \right)^T W \quad \text{Jacobian-transpose matrix product}$$

- Reverse mode AD:

$$(x, W) \rightarrow \left(f(x), \left(\frac{\partial f}{\partial x} \right)^T W \right)$$

- W is called the seed matrix. Setting equal to identity matrix yields full Jacobian
- Computational cost $\approx (1.5 + 2.5q)\text{time}(f)$
 $m = q = 1 \implies \text{cost} \approx 4 \text{time}(f)$
- Jacobian-transpose products, gradients, Jacobians for $n > m$

Taylor Polynomial Propagation

$$y = f(x), f : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

- Extension of tangent propagation to higher degree

- Given $d + 1$ coefficients $x_0, \dots, x_d \in \mathbb{R}^n$

$$x(t) \equiv \sum_{k=0}^d x_k t^k$$

$$y(t) \equiv f(x(t)) = \sum_{k=0}^d y_k t^k + O(t^{d+1})$$

$$y_k \equiv \left. \frac{1}{k!} \frac{d^k y}{dt^k} \right|_{t=0} = y_k(x_0, \dots, x_k)$$

- Computational cost $\approx O(d^2) \text{time}(f)$

Operation	Taylor Rule
$c = a + b$	$c_k = a_k + b_k$
$c = a - b$	$c_k = a_k - b_k$
$c = ab$	$c_k = \sum_{j=0}^k a_j b_{k-j}$
$c = a/b$	$c_k = \frac{1}{b_0} \left(a_k - \sum_{j=1}^k b_j c_{k-j} \right)$
$c = \exp(a)$	$c_k = \frac{1}{k} \sum_{j=1}^k j c_{k-j} a_j$
$c = \log(a)$	$c_k = \frac{1}{k a_0} \left(k a_k - \sum_{j=1}^{k-1} j a_{k-j} c_j \right)$
$s = \sin(a)$	$s_k = \frac{1}{k} \sum_{j=1}^k j a_j c_{k-j}$
$c = \cos(a)$	$c_k = -\frac{1}{k} \sum_{j=1}^k j a_j s_{k-j}$



Software Implementations

- Tools implementing AD have been created for many popular programming languages
 - C/C++: ADOL-C, ADIC, Sacado, ...
 - Fortran: ADIFOR, OpenAD, Tapenade, ...
 - Matlab: ADiMAT, MAD, ...
 - Python: pyADOL-C, AD, ...
- See <http://www.autodiff.org/> for a comprehensive listing
- Tools fall into two general categories
 - Source transformation
 - Operator overloading



Source Transformation

- **AD implemented by preprocessor**
 - Preprocessor reads code to be differentiated
 - Uses AD to generate derivative code
 - Writes-out differentiated code in original source language
 - Differentiated code is then compiled using a standard compiler
- **Resulting derivative computation is usually very efficient**
- **Works well for simple languages (FORTRAN, some C)**
- **ADIFOR/ADIC/OpenAD out of Argonne**
- **Extremely difficult for C++**



ADIFOR* Example

```
C      subroutine func(x, y)
C
C      double precision x(2), y(2)
C      double precision u, v, w
C
C      u = exp(x(1))
C      v = x(1)*x(2)
C      w = u+v
C      y(1) = sin(w)
C
C      u = x(1)**2
C      v = y(1) + u
C      y(2) = y(1)/v
C
C      return
C      end
```

*ADIFOR 2.0D

www.mcs.anl.gov/research/projects/adifor/

```
      subroutine g_func(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)
C
C      Initializations removed for clarity...
C
C      d2_v = exp(x(1))
C      d1_p = d2_v
C      do g_i_ = 1, g_p_
C         g_u(g_i_) = d1_p * g_x(g_i_, 1)
C      enddo
C      u = d2_v
C-----
C      do g_i_ = 1, g_p_
C         g_v(g_i_) = x(1) * g_x(g_i_, 2) + x(2) * g_x(g_i_, 1)
C      enddo
C      v = x(1) * x(2)
C-----
C      do g_i_ = 1, g_p_
C         g_w(g_i_) = g_v(g_i_) + g_u(g_i_)
C      enddo
C      w = u + v
C-----
C      d2_v = sin(w)
C      d1_p = cos(w)
C      do g_i_ = 1, g_p_
C         g_y(g_i_, 1) = d1_p * g_w(g_i_)
C      enddo
C      y(1) = d2_v
C
C      continues...
```



Operator Overloading

- **AD implemented within source language constructs**
 - New data types are created for forward, reverse, Taylor modes
 - Intrinsic operations/elementary operations are overloaded to compute derivatives as a side-effect
 - Data type (e.g., double) in original code is replaced with AD type
- **Generally easy to incorporate into C++ codes**
- **Generally slower than source transformation due to function call overhead**
 - This can generally be eliminated
- **Requires changing data types from floats/doubles to AD types**
 - C++ templates greatly help
- **ADOL-C, FAD/TFAD, Sacado**



(Naive) Operator Overloading Example

```
void func(const double x[], double y[]) {
    double u, v, w;
    u = exp(x[0]);
    v = x[0]*x[1];
    w = u+v;
    y[0] = sin(w);

    u = x[0]*x[0];
    v = y[0] + u;
    y[1] = y[0]/v;
}

void func(const Tangent x[], Tangent y[]) {
    Tangent u, v, w;
    u = exp(x[0]);
    v = x[0]*x[1];
    w = u+v;
    y[0] = sin(w);

    u = x[0]*x[0];
    v = y[0] + u;
    y[1] = y[0]/v;
}
```

```
class Tangent {
public:
    static const int N = 2;
    double val;
    double dot[N];
};

Tangent operator+(const Tangent& a, const Tangent& b) {
    Tangent c;
    c.val = a.val + b.val;
    for (int i=0; i<Tangent::N; i++)
        c.dot[i] = a.dot[i] + b.dot[i];
    return c;
}

Tangent operator*(const Tangent& a, const Tangent& b) {
    Tangent c;
    c.val = a.val * b.val;
    for (int i=0; i<Tangent::N; i++)
        c.dot[i] = a.val * b.dot[i] + a.dot[i]*b.val;
    return c;
}

Tangent exp(const Tangent& a) {
    Tangent c;
    c.val = exp(a.val);
    for (int i=0; i<Tangent::N; i++)
        c.dot[i] = c.val * a.dot[i];
    return c;
}
```

Expression Template Operator Overloading

```
void func(const Tangent x[], Tangent y[]) {
    y[0] = sin(exp(x[0]) + x[0]*x[1]);
    //...
}
```



```
SinExpr< PlusExp< ExpExpr<Tangent>,
            MultExpr<Tangent,Tangent>
        >
    >
```



```
y[0].val = sin(exp(x[0]) + x[0]*x[1]);
for (int i=0; i<N; i++) {
    y[0].dot[i] = cos(exp(x[0]) + x[0]*x[1])*
        (exp(x[0])*x[0].dot[i] +
         x[0]*x[1].dot[i] + x[1]*x[0].dot[i]);
}
```

Public domain Fad/TFad package

```
template <class E1, E2> class PlusExpr {
    double val() const { return e1.val() + e2.val(); }
    double dx(int i) const { return e1.dx(i) + e2.dx(i); }
    const E1& e1;
    const E2& e2;
};

template<class E1, class E2> PlusExpr<E1,E2>
operator+(const E1& a, const E2& b) {
    return PlusExpr<E1,E2>(a,b);
}

template <class E1> class SinExpr {
    double val() const { return sin(e1.val()); }
    double dx(int i) const { return cos(e1.val())*e1.dx(i); }
    const E1& e1;
};

template<class E1> SinExpr<E1> sin(const E1& a) {
    return SinExpr<E1>(a);
}

class Tangent {
public:
    double val() const { return val; }
    double dx(int i) const { return dot[i]; }
    template <class E> Tangent& operator=(const E& e) {
        val = e.val();
        for (int i=0; i<N; i++)
            dot[i] = e.dx(i);
    }
};
```

Sacado: AD Tools for C++ Codes

- **Several modes of Automatic Differentiation**
 - **Forward**
 - **Reverse**
 - **Univariate Taylor series**
 - **Modes can be nested for various forms of higher derivatives**
- **Sacado uses operator overloading-based approach for C++ codes**
 - **Sacado provides C++ data type for each AD mode**
 - **Replace scalar type (e.g., double) with template parameter**
 - **Instantiate template code on various Sacado AD types**
 - **Mathematical operations replaced by overloaded versions**
 - **Expression templates to reduce overhead**



<http://trilinos.sandia.gov>



Sandia National Laboratories



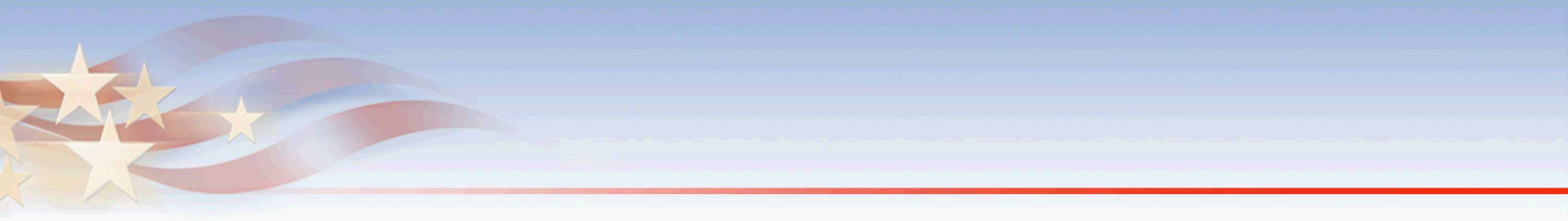
How to use Sacado

- **Template code to be differentiated:** `double -> ScalarT`
- **Replace independent/dependent variables with AD variables**
- **Initialize seed matrix**
 - **Forward:** Derivative array of i' th independent variable is i' th row of seed matrix
 - **Reverse:** Derivative array of i' th dependent variable is i' th row of seed matrix
- **Evaluate function on AD variables**
 - **Instantiates template classes/functions**
- **Extract derivatives**
 - **Forward:** Derivative components of dependent variables
 - **Reverse:** Derivative components of independent variables



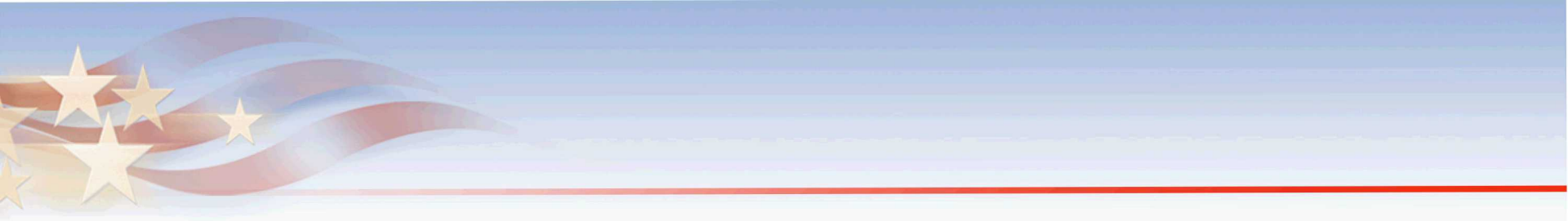
Primary Sacado AD Classes

- `#include "Sacado.hpp"`
- All classes are templated on the Scalar type
- Forward AD classes:
 - `Sacado::Fad::DFad<ScalarT>`: Derivative array is allocated dynamically
 - `Sacado::Fad::SFad<ScalarT>`: Derivative array is allocated statically and dimension must be known at compile time
 - `Sacado::Fad::SLFad<ScalarT>`: Like SFad except allocated length may be greater than "used" length
 - `Sacado::Fad::SimpleFad<ScalarT>`: Dynamically allocated array that doesn't use expression templates
- Similar forward AD classes in other namespaces that use different forward AD approaches (research ideas)
 - `Sacado::ELRFad`, `Sacado::CacheFad`, `Sacado::ELRCacheFad`
- Reverse mode AD classes:
 - `ADvar<ScalarT>`

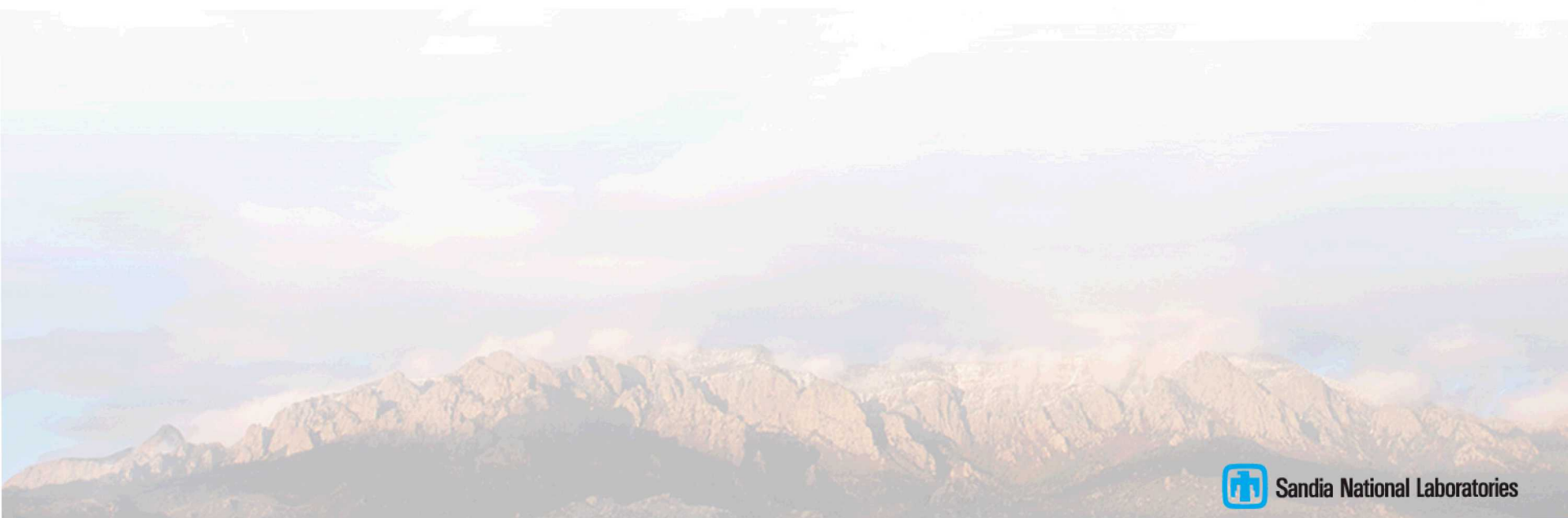


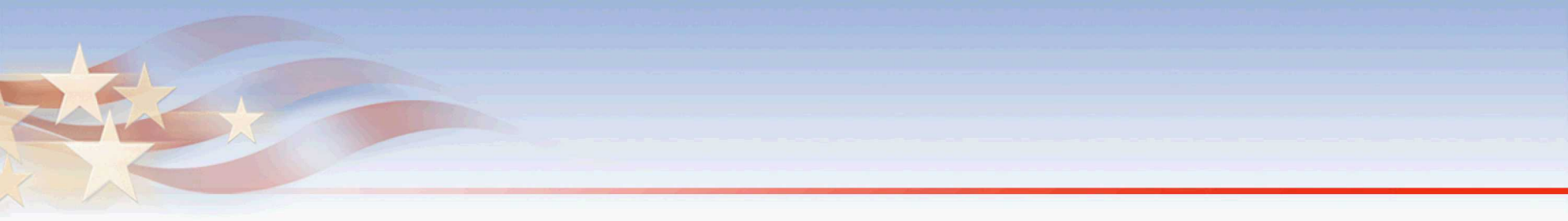
Basic Fad Example





Seed Matrix Fad Example





Basic Rad Example



Computing Higher Derivatives

- AD classes are templated, so AD classes can be nested to compute higher derivatives

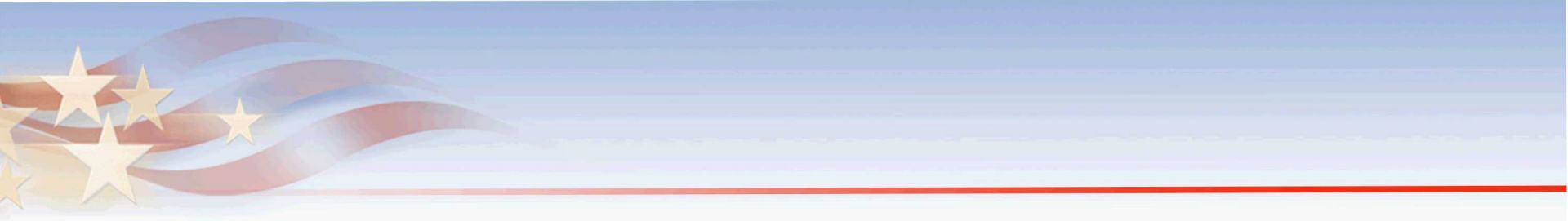
- Forward-forward: $y = f(x) \xrightarrow{\text{for}} \frac{\partial y}{\partial x} v_1 \xrightarrow{\text{for}} \frac{\partial}{\partial x} \left(\frac{\partial y}{\partial x} v_1 \right) v_2$
- Reverse-forward: $y = f(x) \xrightarrow{\text{rev}} w^T \frac{\partial y}{\partial x} \xrightarrow{\text{for}} \frac{\partial}{\partial x} \left(w^T \frac{\partial y}{\partial x} \right) v$
- Forward-Taylor: $y_0 = f(x_0) \xrightarrow{\text{for}} \frac{\partial y_0}{\partial x_0} v \xrightarrow{\text{tay}} \frac{\partial y_k}{\partial x_0} v$
- Reverse-Taylor: $y_0 = f(x_0) \xrightarrow{\text{rev}} w^T \frac{\partial y_0}{\partial x_0} \xrightarrow{\text{tay}} w^T \frac{\partial y_k}{\partial x_0}$
- Etc...



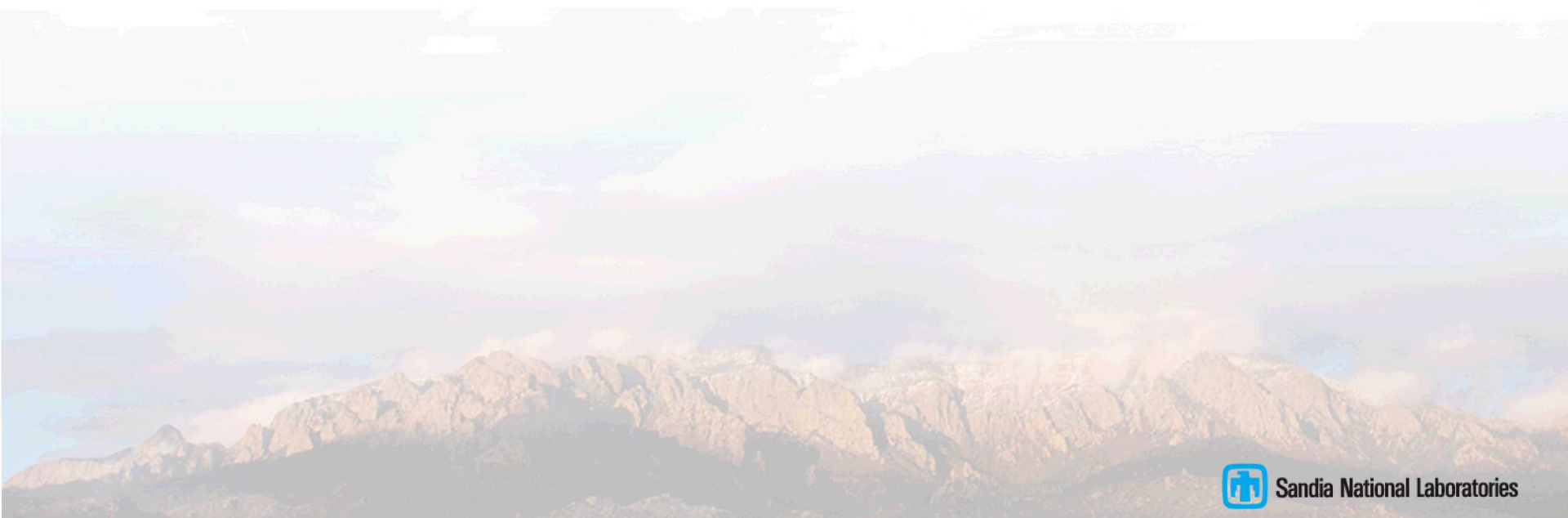
How To Compose AD Classes

Evaluation	Data Type	Derivative
Function Evaluation	double	$f(x)$ f
First Forward Derivative	xFad<double>	$\begin{bmatrix} f(x) \\ \frac{\partial f}{\partial x} V_1 \end{bmatrix}$ f.val f.dx
Second Forward Derivative	xFad< yFad<double> >	$\begin{bmatrix} f(x) \\ \frac{\partial f}{\partial x} V_2 \\ \frac{\partial^2 f}{\partial x^2} V_1 \\ \frac{\partial}{\partial x} \left(\frac{\partial f}{\partial x} V_1 \right) V_2 \end{bmatrix}$ f.val.val f.val.dx f.dx.val f.dx.dx
First Reverse Derivative	ADvar<double>	$\begin{bmatrix} f(x) \\ W^T \frac{\partial f}{\partial x} \end{bmatrix}$ f.val x.adj
Forward-Reverse Second Derivative	ADvar< xFad<double> >	$\begin{bmatrix} f(x) \\ \frac{\partial f}{\partial x} V \\ W^T \frac{\partial f}{\partial x} \\ \frac{\partial}{\partial x} \left(W^T \frac{\partial f}{\partial x} \right) V \end{bmatrix}$ f.val f.val.dx x.adj.val x.adj.dx





Nested AD Example





Forward or Reverse?

- **Forward: Computes derivatives column-wise**
 - Number of independent variables $\leq$ number of dependent variables
 - Square Jacobians for Newton's method
 - Sensitivities with small numbers of parameters
 - Algorithm naturally calls for Jacobian-vector/matrix products
 - (Block) Matrix-free Newton-Krylov
- **Reverse: Computes derivatives row-wise**
 - Number of independent variables $\gg$ number of dependent variables
 - Gradients of scalar valued functions
 - Sensitivities with respect to large numbers of parameters
 - Algorithm naturally calls for Jacobian-transpose-vector/matrix products
 - (Block) Matrix-free solves of transpose matrix
 - Optimization



Choosing AD Types

- **DFad**

- Derivative array allocated dynamically
- Most flexible
- Slowest
- Very slow in threaded environments

- **SFad**

- Derivative array size fixed at compile time
- Must know exact number of derivative components
- Fastest
- Best choice in threaded environments

- **SLFad**

- Fixed-length derivative array, can use only a portion of it at run-time
- Compromise between the two
- Usually just a little slower than SFad
- Good choice for threaded environments

- **ADvar (reverse mode)**

- Due to overhead, need substantially more independent variables than dependent variables (at least 40 more)
- Currently not appropriate for threaded environments



Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

$$f(x) = \sum_{i=1}^N Q_i^T e_{k_i}(P_i x)$$

- Jacobian computation:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^N Q_i^T J_{k_i} P_i, \quad J_{k_i} = \frac{\partial e_{k_i}}{\partial x_i}, \quad x_i = P_i x$$

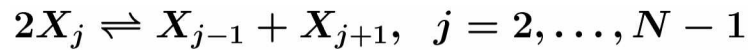
- Jacobian-transpose product computation:

$$w^T \frac{\partial f}{\partial x} = \sum_{i=1}^N (Q_i w)^T J_{k_i} P_i$$

- Hybrid symbolic/AD procedure
 - Element-level derivatives computed via AD
 - Exactly the same as how you would do this “manually”
 - Avoids parallelization issues

Performance

Set of N hypothetical chemical species:



Steady-state mass transfer equations:

$$\mathbf{u} \cdot \nabla Y_j + \nabla^2 Y_j = \dot{\omega}_j, \quad j = 1, \dots, N - 1$$

$$\sum_{j=1}^N Y_j = 1$$

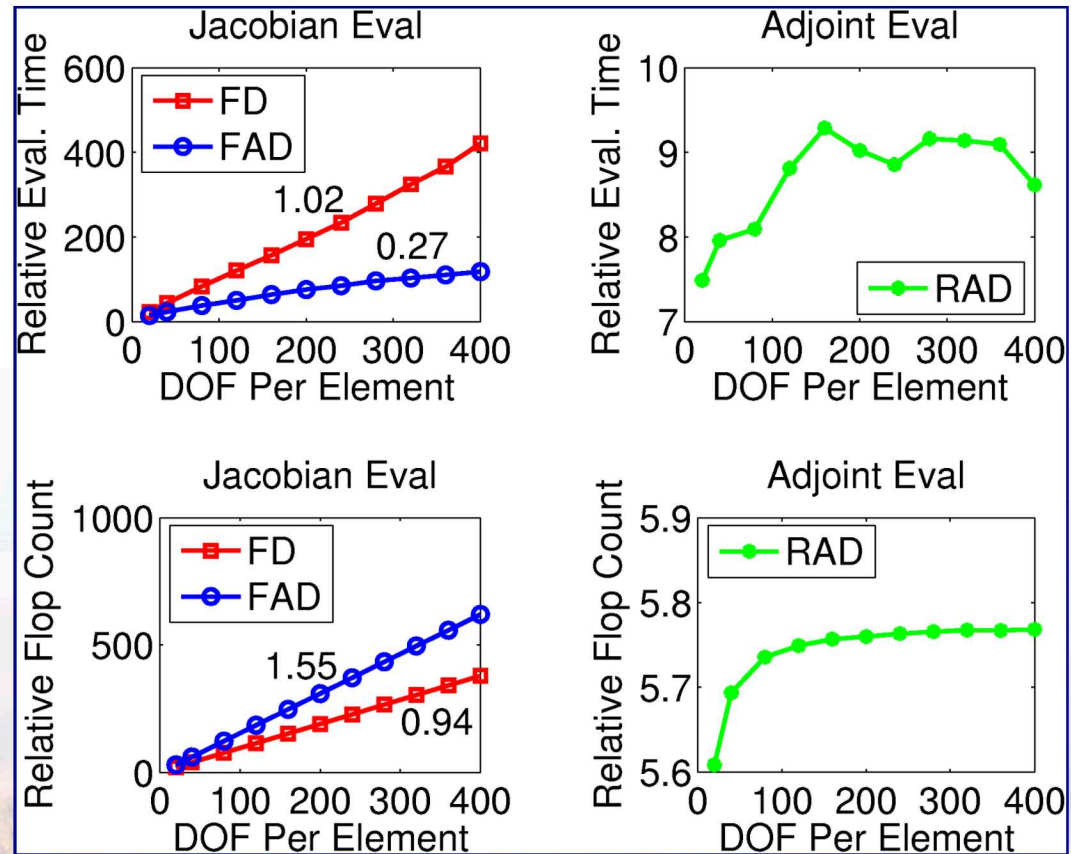
- **Forward mode AD**

- Faster than FD
- Better scalability in number of PDEs
- Analytic derivative
- Provides Jacobian for all Charon physics

- **Reverse mode AD**

- Scalable adjoint/gradient

Scalability of the element-level derivative computation



$$\text{DOF per element} = 4 * N$$



Matrix/Residual Assembly Performance Test

- Performance test for measuring Jacobian/Residual assembly using Sacado

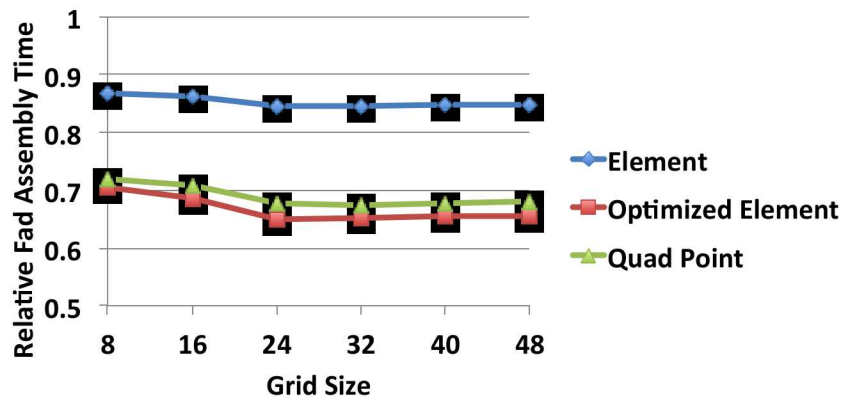
$$-\nabla \cdot (\kappa \nabla u) + \alpha v \cdot \nabla u + \beta u^2 = 0$$

- 3-D, linear FEM discretization
- 1x1x1 cube, unstructured mesh
- Derived from FENL Kokkos example (H. Carter Edwards)
- Thread-parallel matrix/residual assembly
 - Mesh cell loop parallelized with OpenMP/CUDA
 - Atomic instructions for assembling into matrix/residual
- 3 algorithms studied
 - Traditional element derivative w.r.t. nodal solution (AD size = # nodes/element x # equations)
 - Element derivative with optimized derivative of interpolation of nodal solution, gradient at quadrature points
 - Derivative at each quadrature point w.r.t. nodal solution and gradient interpolated at quadrature point (AD size = 4 x # equations)

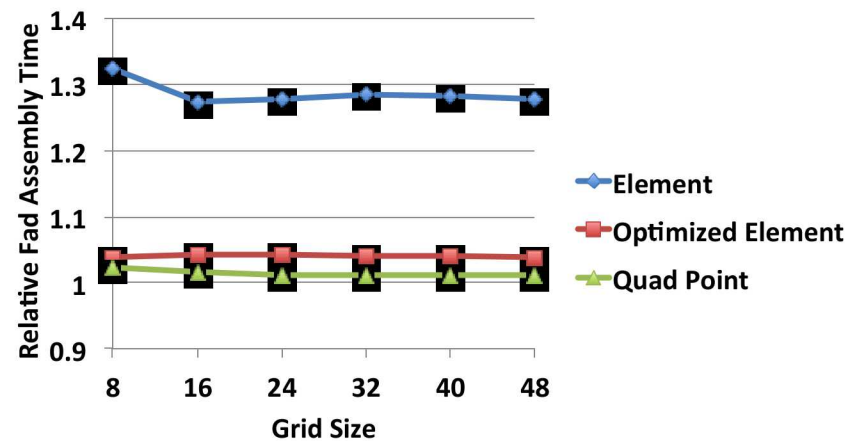


Sacado Assembly Performance

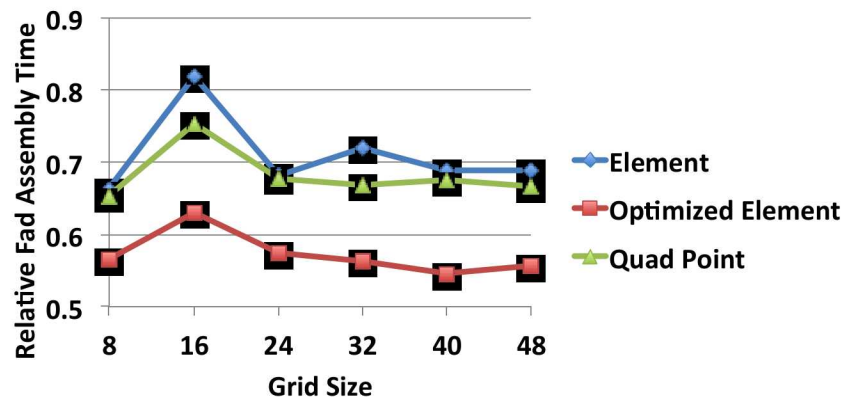
Sandy Bridge -- Linear Elements
(Single socket, 8 cores, 16 threads)



NVIDIA K20X GPU -- Linear Elements

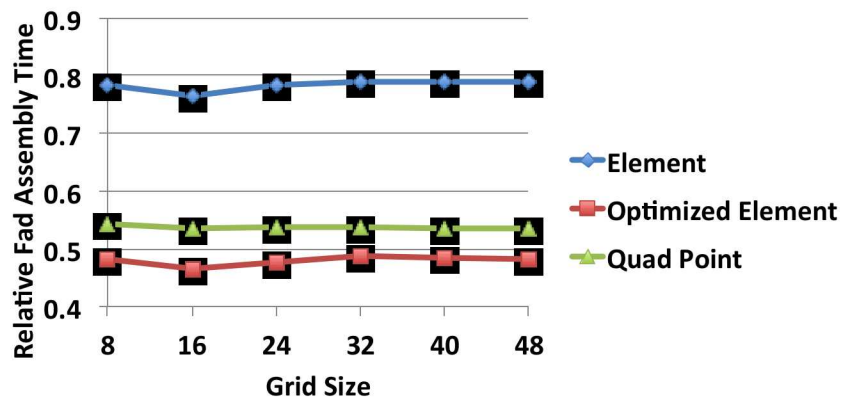


Xeon Phi 7120P -- Linear Elements
(60 cores, 240 threads)

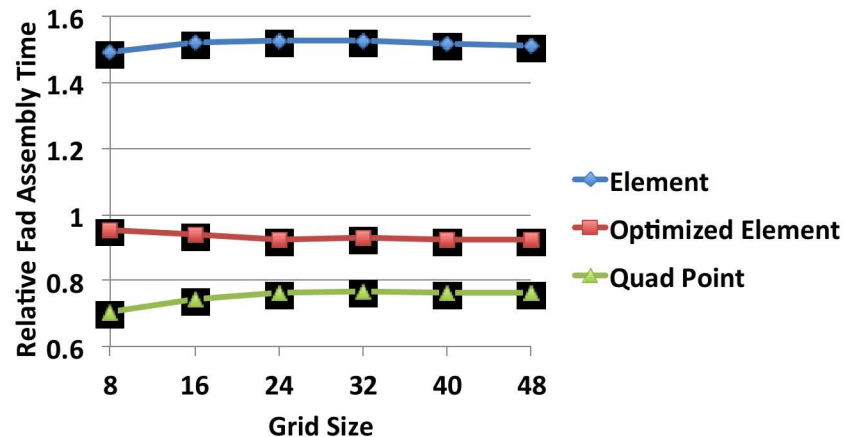


Sacado Assembly Performance

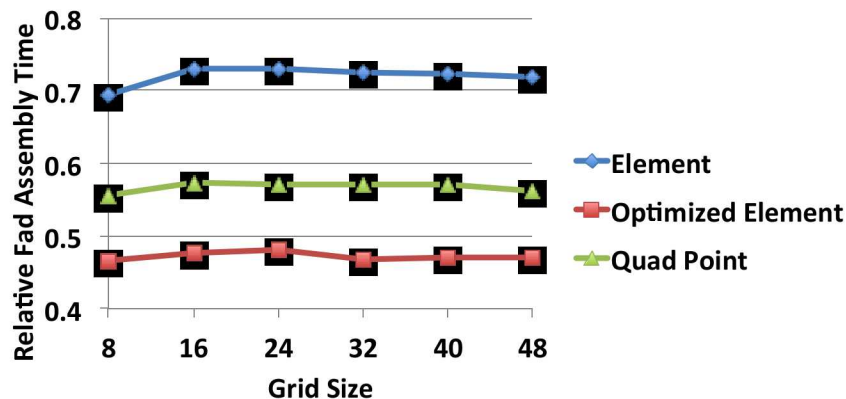
Sandy Bridge -- Quadratic Elements
(Single socket, 8 cores, 16 threads)



NVIDIA K20X GPU -- Quadratic Elements



Xeon Phi 7120P -- Quadratic Elements
(60 cores, 240 threads)



Steady-State Local Sensitivity Analysis

$$f(u, y) = 0, \quad v(y) = h(u, y)$$

Forward sensitivities

$$\frac{\partial v}{\partial y} = \frac{\partial h}{\partial u} \left(-\frac{\partial f^{-1}}{\partial u} \frac{\partial f}{\partial y} \right) + \frac{\partial h}{\partial y}$$

- **Cost scales with number of *parameters***
- **Solve system Jacobian**

Adjoint sensitivities

$$\frac{\partial v^T}{\partial y} = \frac{\partial f^T}{\partial y} \left(-\frac{\partial f^{-T}}{\partial u} \frac{\partial h^T}{\partial u} \right) + \frac{\partial h^T}{\partial y}$$

- **Cost scales with number of *observation functions***
- **Solve system Jacobian-transpose**

- **Small extension for Newton-based codes**
- **Sensitivity (linear) solves significantly cheaper than (nonlinear) state solves**
- **Accurate derivatives critical (can't use approximate Jacobian)**
- ***Simulation code* must evaluate observation functions & gradients**



Transient Local Sensitivity Analysis

Forward sensitivities

$$\frac{\partial f}{\partial \dot{u}} \left(\frac{\partial \dot{u}}{\partial y} \right) + \frac{\partial f}{\partial u} \left(\frac{\partial u}{\partial y} \right) + \frac{\partial f}{\partial y} = 0, \quad t \in [t_0, t_f],$$

$$\frac{\partial u}{\partial y}(t_0) = \frac{\partial u_0}{\partial y}, \quad \frac{\partial \dot{u}}{\partial y}(t_0) = \frac{\partial \dot{u}_0}{\partial y},$$

$$\frac{\partial v}{\partial y} = \int_{t_0}^{t_f} \left(\frac{\partial g}{\partial \dot{u}} \frac{\partial \dot{u}}{\partial y} + \frac{\partial g}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial g}{\partial y} \right) dt + \left(\frac{\partial h}{\partial \dot{u}} \frac{\partial \dot{u}}{\partial y} + \frac{\partial h}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial h}{\partial y} \right) \Big|_{t=t_f}$$

- **Linear ODE for sensitivities solved alongside original model**
- **Cost scales with number of *parameters***
- **Hindmarsh *et al***

Adjoint sensitivities

$$\frac{d}{dt} \left(\frac{\partial f^T}{\partial \dot{u}} \Lambda \right) - \frac{\partial f^T}{\partial u} \Lambda + \frac{\partial g^T}{\partial u} = 0, \quad t \in [t_0, t_f],$$

$$\left(\frac{\partial f^T}{\partial \dot{u}} \Lambda \right) \Big|_{t=t_f} = \frac{\partial h^T}{\partial u} \Big|_{t=t_f},$$

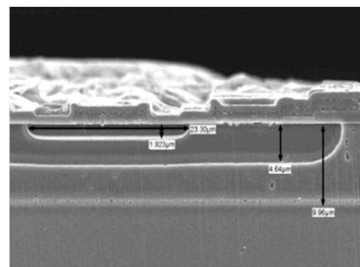
$$\frac{\partial v^T}{\partial y} = \int_{t_0}^{t_f} \left(\frac{\partial g^T}{\partial y} - \frac{\partial f^T}{\partial y} \Lambda \right) dt + \frac{\partial h^T}{\partial y} \Big|_{t=t_f} + \frac{\partial u_0^T}{\partial y} \left(\frac{\partial f^T}{\partial \dot{u}} \Lambda \right) \Big|_{t=t_0}$$

- **Linear ODE for adjoint that must be integrated backward in time**
- **Requires full forward model integration first (or check-pointing)**
- **Cost scales with number of *objective functions***
- **Petzold *et al***

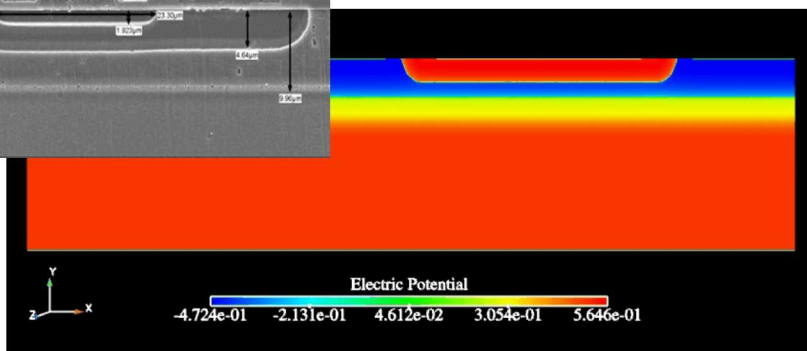
Both techniques provided by SUNDIALS:
computation.llnl.gov/casc/sundials/main.html

QASPR

Qualification of electronic devices in radiation environments

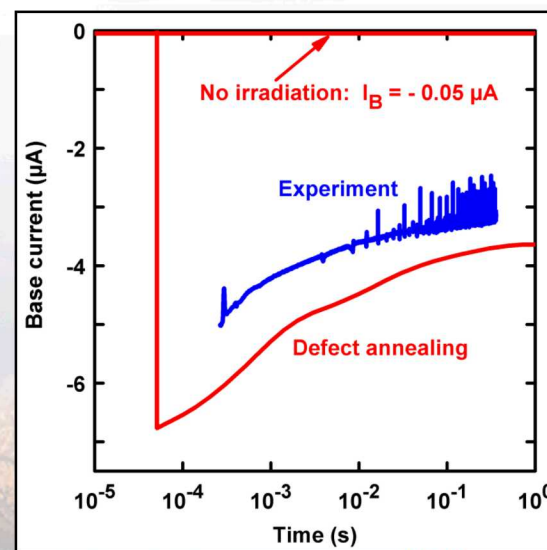
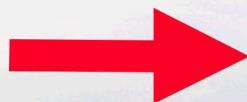
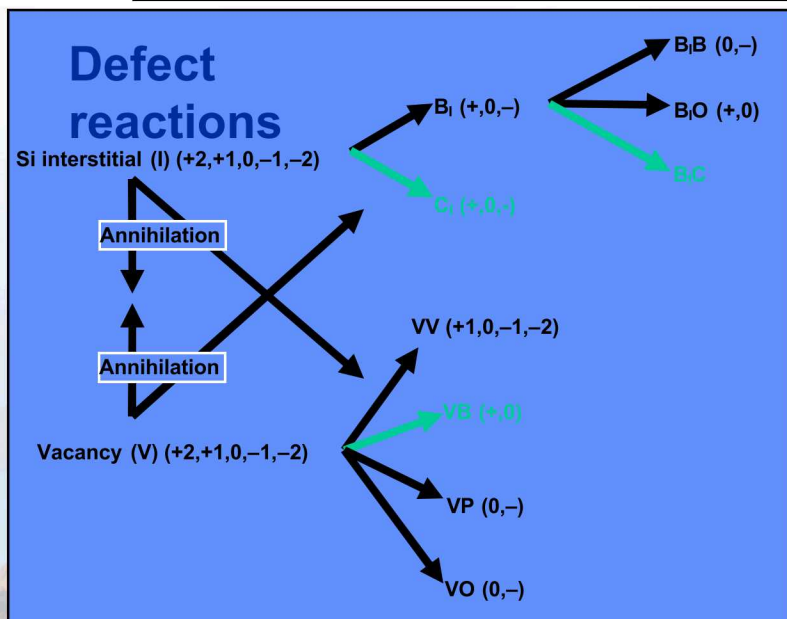


Bipolar Junction Transistor



**PDE semiconductor
device simulation**

G. Hennigan, J. Castro,
P. Lin, R. Pawlowski (SNL)



Charon Drift-Diffusion Formulation with Defects

Current
Conservation for e-
and h+

$$\begin{aligned} \frac{\partial n}{\partial t} - \nabla \cdot J_n &= -R_n(\psi, n, p, Y_1, \dots, Y_N), & J_n &= -n\mu_n \nabla \psi + D_n \nabla n \\ \frac{\partial p}{\partial t} + \nabla \cdot J_p &= -R_p(\psi, n, p, Y_1, \dots, Y_N), & J_p &= -p\mu_p \nabla \psi - D_p \nabla p \end{aligned}$$

Defect Continuity

$$\frac{\partial Y_i}{\partial t} + \nabla \cdot J_{Y_i} = -R_{Y_i}(\psi, n, p, Y_1, \dots, Y_N), \quad J_{Y_i} = -\mu_i Y_i \nabla \psi - D_i \nabla Y_i$$

Electric potential

$$-\nabla(\epsilon \nabla \psi(x)) = -q(p(x) - n(x) + N_D^+(x) - N_A^-(x)) - \sum_{i=1}^N q_i Y_i(x)$$

Recombination/
generation
source terms

R_X

Include electron capture and hole capture by defect species
and reactions between various defect species

Electron
emission/capture



$$R_{[Z^i \rightarrow Z^{i+1} + e^-]} \propto \sigma_{[Z^i \rightarrow Z^{i+1} + e^-]} Z^i \exp\left(\frac{\Delta E_{[Z^i \rightarrow Z^{i+1} + e^-]}}{kT}\right)$$

Activation Energy

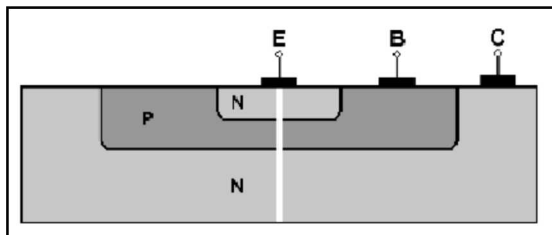
Cross section



Sensitivity Analysis of a Bipolar Junction Transistor

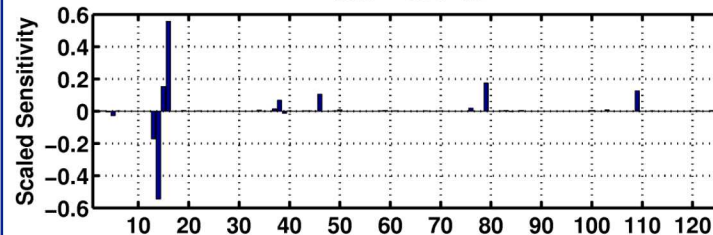


- Bipolar Junction Transistor
- Pseudo 1D strip (9x0.1 micron)
- Full defect physics
- 126 parameters

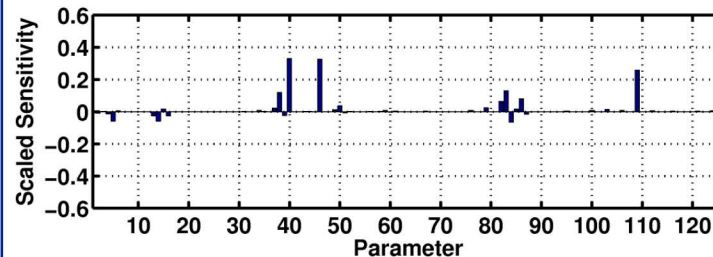


Sensitivities show dominant physics

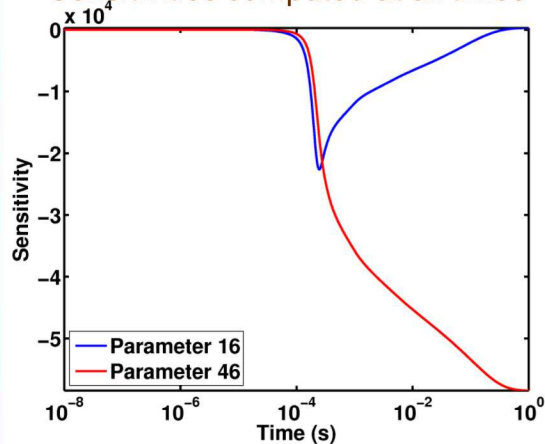
time = 1.0e-03



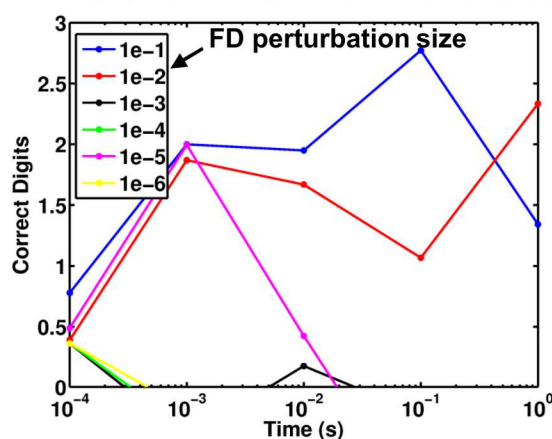
time = 1.0



Sensitivities computed at all times



1st-order Finite Difference Accuracy



Comparison to FD:

- ✓ Sensitivities at all time points
- ✓ More accurate
- ✓ More robust
- ✓ 14x faster!



Summary

- **Derivatives are useful in many forms of simulation and analysis**
 - **Discuss more uses in Part 3**
- **Automatic differentiation provides a powerful means for computing analytic derivatives in simulation codes**
- **Sacado+templating provides an effective means for implementing these ideas in large-scale C++ codes**
 - **More on this in Parts 3 and 4**



AD Research

- **Efficiently deploying AD in modern programming environments**
 - Expression templates for C++
 - AD in interpreted languages (Matlab, Python, ...)
- **Reducing overhead of reverse-mode AD**
- **AD in threaded-environments**
 - Automatically differentiating thread-parallel programs
 - Exploiting thread parallelism within AD tools
- **Finding most efficient way to differentiate a given program**
 - Column/row compression
 - Cross-country elimination
- **Efficiently evaluating higher derivatives**
- **Automatically detecting and exploiting sparsity in derivatives**



AD References

- Introduction to AD

- A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2008.
- U. Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, SIAM, 2012.

- Transient sensitivity analysis

- Y. Cao, S. Li, L. Petzold, and R. Serban. “Adjoint sensitivity analysis for differential-algebraic equations: The adjoint DAE system and its numerical solution.” *SIAM J Sci Comput*, 24(3):1076–1089, 2003.
- A. Hindmarsh,, P. Brown, K. Grant, S. Lee, R. Serban, D. Shumaker, and C. Woodward. “Sundials: Suite of nonlinear and differential/algebraic equation solvers.” *ACM Trans. Math. Softw.* 31(3): 363–396, 2005.

- Sacado AD

- E. Phipps, R. Bartlett, D. Gay, and R. Hoekstra. “Large-Scale Transient Sensitivity Analysis of a Radiation-Damaged Bipolar Junction Transistor via AD.” *Advances in Automatic Differentiation*, C. Bischof, M. Bucker, P. Hovland, U. Naumann, and J. Utke, eds., *Lecture Notes in Computational Science and Engineering*, 2008.
- E. Phipps and R. Pawlowski, “Efficient Expression Templates for Operator Overloading-based Automatic Differentiation,” in *Recent Advances in Algorithmic Differentiation*, S. Forth, P. Hovland, E. Phipps, J. Utke and A. Walther, eds., *Lecture Notes in Computational Science and Engineering*, Springer, 2012.

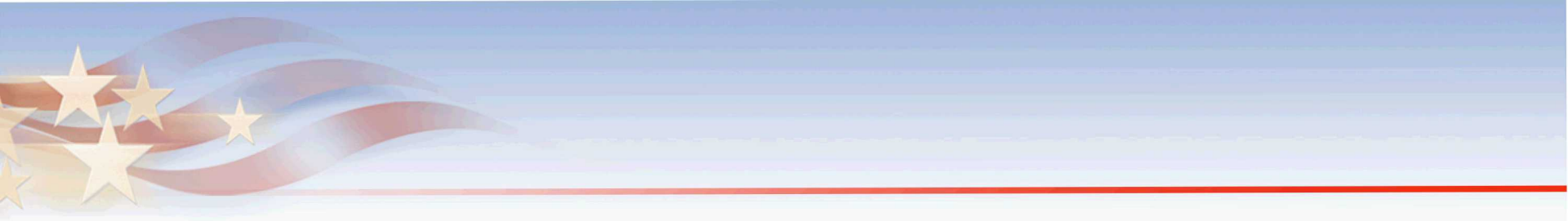
- Use of templates for automatic differentiation in large-scale codes

- R. Pawlowski, E. Phipps, and A. Salinger, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part I: template-based generic programming,” *Journal of Scientific Programming*, vol. 20 (2), 2012.
- R. Pawlowski, E. Phipps, A. Salinger, S. Owen, C. Siefert, and M. Staten, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part II: application to partial differential equations,” *Journal of Scientific Programming*, vol. 20 (3), 2012.

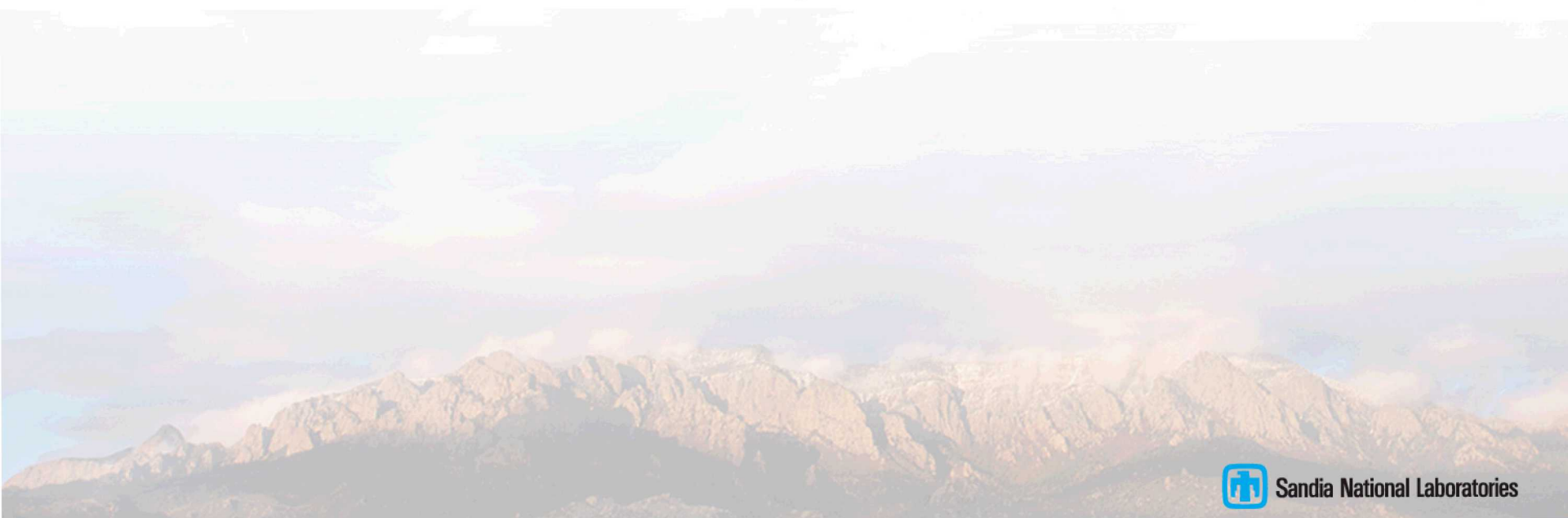
- Using adjoints/derivatives in UQ

- J. Breidt, T. Butler, D. Estep. “A measure-theoretic computational method for inverse sensitivity problems I: Basic Method and Analysis.” *SINUM*, 49: 1836–1859, 2011.

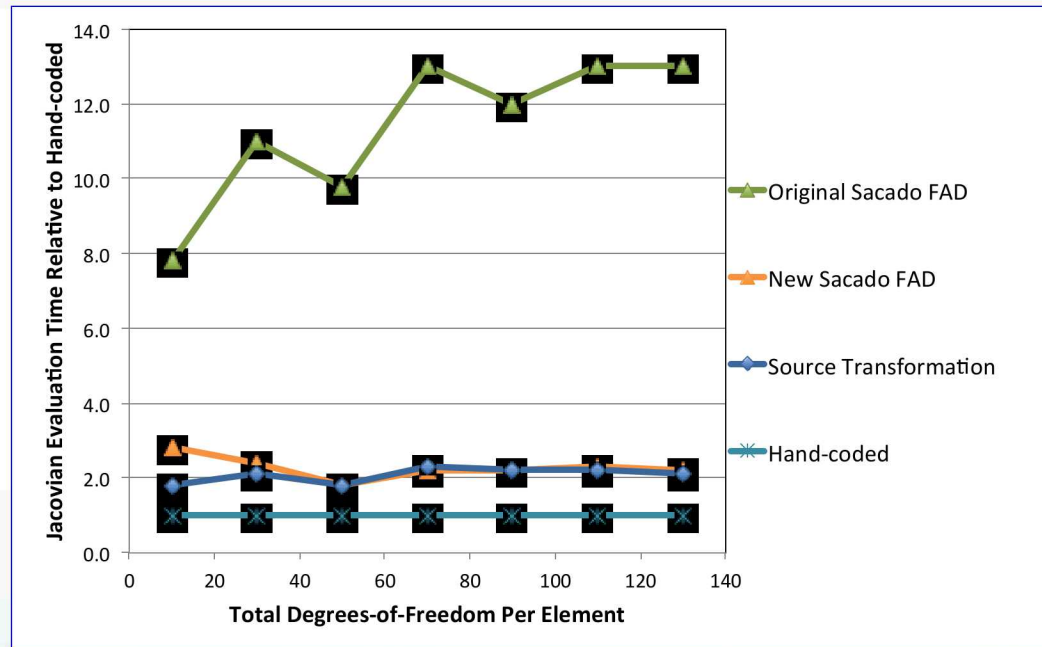




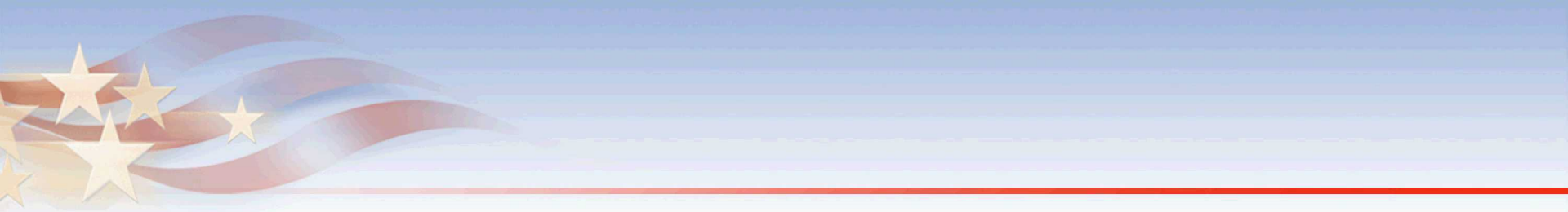
Auxiliary Slides



Sacado AD Tools Perform Extremely Well



- Simple set of representative PDEs
 - Total degrees-of-freedom = number of nodes x number of PDEs for each element
- New expression-template implementations virtually eliminate all operator overloading overhead
 - Phipps & Pawlowski, 2012
- 2x cost relative to hand-coded, optimized Jacobian (very problem dependent)



Stochastic Galerkin Methods and Software



Polynomial Chaos Expansions (PCE)

- **Steady-state finite dimensional model problem:**

Find $u(\xi)$ such that $f(u, \xi) = 0$, $\xi : \Omega \rightarrow \Gamma \subset R^M$, density ρ

- **(Generalized) Polynomial Chaos (PC, gPC) approximation (e.g., Xiu and Karniadakis, 2002):**

$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi), \quad \langle \psi_i \psi_j \rangle \equiv \int_{\Gamma} \psi_i(y) \psi_j(y) \rho(y) dy = \delta_{ij} \langle \psi_i^2 \rangle$$

- **Multivariate orthogonal polynomials**
- **Typically constructed as tensor products with total order at most N**

$$\psi_k(y) = \phi_{k_1}^1(y_1) \dots \phi_{k_M}^M(y_M), \quad k_1 + \dots + k_M \leq N, \quad k = 0, \dots, P = \frac{(M+N)!}{M!N!} - 1$$

- **where**

$$\Gamma = \Gamma_1 \times \dots \times \Gamma_M$$
$$\rho(y) = \rho_1(y_1) \dots \rho_M(y_M)$$

$$\langle \phi_{j_i}^i \phi_{k_i}^i \rangle_i \equiv \int_{\Gamma_i} \phi_{j_i}^i(y_i) \phi_{k_i}^i(y_i) \rho_i(y_i) dy_i = \delta_{j_i k_i} \langle (\phi_{j_i}^i)^2 \rangle_i$$

- **(assuming independence)**

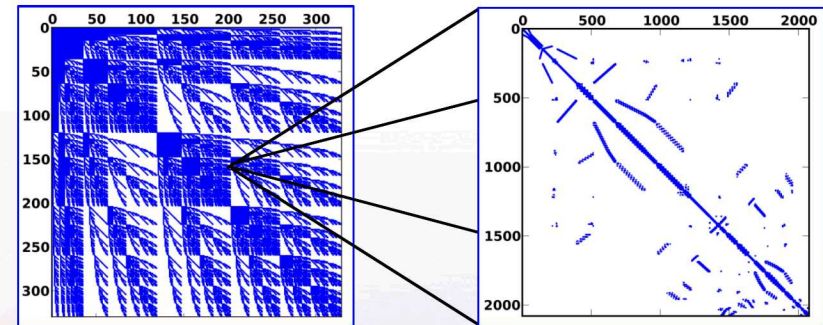
Stochastic Galerkin UQ Methods

- Stochastic Galerkin method (see Ghanem and Spanos, 1991):

$$\hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi) \rightarrow f_i(u_0, \dots, u_P) = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \dots, P$$

- Method generates new coupled spatial-stochastic nonlinear problem (intrusive)

$$0 = F(U) = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_P \end{bmatrix}, \quad U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_P \end{bmatrix} \quad \frac{\partial F}{\partial U} :$$



Stochastic sparsity

Spatial sparsity

- Advantages:
 - Often fewer stochastic degrees-of-freedom for comparable level of accuracy
- Challenges:
 - Computing SG residual and Jacobian entries in large-scale, production simulation codes
 - Solving resulting systems of equations efficiently, particularly for nonlinear problems

SG Jacobian

- Stochastic Galerkin Residual

$$\hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi) \rightarrow f_i(u_0, \dots, u_P) = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \dots, P$$

- Stochastic Galerkin Jacobian:

$$\frac{\partial f_i}{\partial u_j} = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} \frac{\partial f}{\partial u}(\hat{u}(y), y) \psi_i(y) \psi_j(y) \rho(y) dy,$$

$$\frac{\partial f}{\partial u}(\hat{u}(\xi), \xi) \approx \sum_{k=0}^P A_k \psi_k(\xi), \quad A_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\Gamma} \frac{\partial f}{\partial u}(\hat{u}(y), y) \psi_k(y) \rho(y) dy$$

$$\Rightarrow \frac{\partial f_i}{\partial u_j} \approx \sum_{k=0}^P A_k \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle} \Rightarrow \frac{\partial F}{\partial U} = \sum_{k=0}^P G_k \otimes A_k, \quad G_k(i, j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$

SG Linear Systems

- Stochastic Galerkin Newton linear systems:

$$A\Delta U = -F \implies \left(\sum_{k=0}^P G_k \otimes A_k \right) \left(\sum_{k=0}^P e_k \otimes \Delta u_k \right) = - \sum_{k=0}^P e_k \otimes f_k, \quad e_k = I(:, k) \in \mathbb{R}^{P+1}$$

- Solution methods:

- Form SG matrix directly (expensive)
- “Matrix-free” approach for iterative linear solvers (Pellissetti & Ghanem, 2000):

$$\begin{aligned} Y = AX &\implies \sum_{i=0}^P e_i \otimes y_i = \left(\sum_{k=0}^P G_k \otimes A_k \right) \left(\sum_{j=0}^P e_j \otimes x_j \right) \\ &\implies y_i = \sum_{j=0}^P \sum_{k=0}^P A_k x_j C_{ijk}, \quad C_{ijk} = G_k(i, j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle} \end{aligned}$$

- Sparsity determined by triple product tensor
- Only requires operator-apply for each operator PCE coefficient
- Organize algorithm to minimize operator-vector applies



SG Matrix-free Multiply Algorithm

$$y_i = \sum_{j=0}^P \sum_{k=0}^P A_k x_j C_{ijk}, \quad i = 0, \dots, P$$

```
for  $i = 0$  to  $P$  do
   $y_i = 0$ 
end for
for  $k = 0$  to  $P$  do
   $J = \{j \mid C_{ijk} \neq 0 \text{ for some } i\}$ 
   $U = \{x_j \mid j \in J\}$ 
   $V = A_k U$ 
  for  $j' = 0$  to  $|J|$  do
     $j = J(j')$ 
     $I = \{i \mid C_{ijk} \neq 0\}$ 
    for  $i$  in  $I$  do
       $y_i = y_i + C_{ijk} U_{j'}$ 
    end for
  end for
end for
end for
```



Stokhos: Trilinos Tools for Embedded Stochastic Galerkin UQ Methods

- Eric Phipps (lead developer) with contributions by Chris Miller, Habib Najm, Bert Deusschere, Omar Knio
- Tools for describing SG discretization
 - Stochastic bases
 - Quadrature rules
 - Sparse triple-product tensor
 - Tools for approximating nonlinear terms
- C++ operator overloading library for automatically evaluating SG residuals and Jacobians
 - Template-based generic programming: extend Sacado AD ideas to projections onto orthogonal polynomials
 - Replace low-level scalar type with orthogonal polynomial expansions
 - Incorporate into code through C++ templates
- Tools forming and solving SG linear systems
 - SG matrix operators
 - Stochastic preconditioners
 - Hooks to Trilinos parallel solvers and preconditioners
- Nonlinear SG application code interface
 - Connect SG methods to nonlinear solvers, time integrators, optimizers, ...
- Embedded ensemble propagation for sampling-based UQ methods
 - Propagate groups of samples using an ensemble scalar type
 - Discuss this in part 4.



<http://trilinos.org>



Sandia National Laboratories

1-Dimensional Basis Polynomials

- **Stokhos::OneDOrthogPolyBasis<ordinal_type, value_type>**
 - Abstract interface for orthogonal polynomials in one variable
 - Evaluate, norms, Gauss points, ...

- General implementation provided by **RecurrenceBasis**
 - Based on 3-term recurrence formula:

$$\phi_{k+1}(y) = \gamma_{k+1}((\delta_k y - \alpha_k)\phi_k(y) - \beta_k \phi_{k-1}(y))$$

- Each implementation only needs to provide recurrence coefficients

- Currently there are implementations for

- Hermite polynomials (Gaussian)
- Legendre polynomials (Uniform)
 - Gauss-Legendre quadrature points
 - Clenshaw-Curtis quadrature points
 - Gauss-Patterson quadrature points

- Jacobi polynomials

- General user-provided weight-function using the Discretized Stieltjes Procedure (W. Gautschi, 2004) e.g., Rys polynomials for truncated Gaussian.

- Various approaches for defining new bases in terms of given PC expansions, e.g., Constantine et al, 2012.

$$\alpha_k = \frac{\int y \phi_k^2(y) \rho(y) dy}{\int \phi_k^2(y) \rho(y) dy}, \quad k = 0, 1, 2, \dots,$$

$$\beta_k = \frac{\int \phi_k^2(y) \rho(y) dy}{\int \phi_{k-1}^2(y) \rho(y) dy}, \quad k = 1, 2, \dots,$$

$$\beta_0 = 1,$$

$$\gamma_k = \delta_k = 1, \quad k = 0, 1, 2, \dots$$



Multi-Dimensional Basis Polynomials

- **Stokhos::OrthogPolyBasis<ordinal_type, value_type>**
 - Abstract interface for orthogonal polynomials in multiple variables
 - Similar to 1-D interface

- Primary implementation is **CompletePolynomialBasis**
 - Total-order tensor product of 1-D orthogonal polynomials:

$$\psi_k(y) = \phi_{k_1}^1(y_1) \dots \phi_{k_M}^M(y_M), \quad k_1 + \dots + k_M \leq N$$

- Provides methods for multi-variate indexing $k \leftrightarrow (k_1, \dots, k_M)$ using a “total order” ordering scheme
- **TotalOrderBasis**
 - Similar to **CompletePolynomialBasis** but allows for different ordering schemes
 - Total order, lexicographic, Mortan-Z
 - Lexicographic ordering yields more “compact” Cijk tensor, leading to improved matrix-vector product performance in some cases





Multi-Dimensional Basis Polynomials

- **TensorProductBasis**
 - Full tensor product
- **SmolyakBasis**
 - Product basis built to be consistent with Smolyak sparse grid
 - More on this later
- **Several additional research-level bases used within various dimension reduction approaches**
 - See, e.g., Constantine et al, 2012.



Multi-Dimensional Quadrature

- **Stokhos::Quadrature<ordinal_type, value_type>**

- Abstract interface for quadrature in multiple dimensions, e.g.,

$$\int_{\Gamma} f(y) \psi_j(y) \rho(y) dy \approx \sum_{k=0}^Q w_k f(y_k) \psi_j(y_k)$$

- Implementations provide quadrature points, weights, and values of basis functions at quadrature points

- **TensorProductQuadrature**

- Tensor product of 1-D gauss points

- **SparseGridQuadrature**

- Calls John Burkardt's (<http://people.sc.fsu.edu/~jburkardt/>) sparse-grid package through Dakota
 - Requires enabling TriKota package in Trilinos
 - Unpack Dakota tarball in TriKota directory
- Can use any Stokhos basis using a kludgy interface

- **SmolyakSparseGridQuadrature**

- Limited implementation of Smolyak sparse grids directly within Stokhos



Sparse Triple Product Tensor

- **Stokhos::Sparse3Tensor<ordinal_type, value_type>**
 - Stores expectations of products of three basis functions

$$\langle \psi_i \psi_j \psi_k \rangle = \int_{\Gamma} \psi_i(y) \psi_j(y) \psi_k(y) \rho(y) dy$$

- Stored in sparse format (Najm, Debusschere, Knio)
- Entries computed via product basis leveraging tensor product structure
- New construction algorithm only iterates of non-zeros in final tensor, based on non-zeros for sparse tensor from each 1-D basis
 - Avoids P^3 algorithm



Polynomial Approximation

- **Stokhos::OrthogPolyApprox<ordinal_type, value_type>**
 - Stores coefficients of a scalar polynomial chaos expansion

$$u(y) \approx \sum_{k=0}^P u_k \psi_k(y)$$

- Provides several convenience routines for managing expansion (evaluation, global indexing, (dimension, order) indexing, statistics, ...)

Intrusive polynomial chaos through operator overloading

$$f(u, \xi) = 0, \quad \hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi)$$

$$\rightarrow F_i(u_0, \dots, u_P) = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \dots, P$$

- By orthogonality of the basis polynomials

$$(\psi_i, \psi_j) = \langle \psi_i \psi_j \rangle = \int_{\Gamma} \psi_i(y) \psi_j(y) \rho(y) dy = \langle \psi_i^2 \rangle \delta_{ij}$$

- The F_i are just the first $P + 1$ coefficients of the polynomial chaos expansion

$$f(\hat{u}(y), y) = \sum_{i=0}^{\infty} F_i \psi_i(y)$$

- Basic idea is to compute such a truncated polynomial chaos expansion for each intermediate operation in the calculation of $f(u, y)$

$$\text{Given } a(y) = \sum_{i=0}^P a_i \psi_i(y), \quad b(y) = \sum_{i=0}^P b_i \psi_i(y), \quad \text{find } c(y) = \sum_{i=0}^P c_i \psi_i(y)$$

$$\text{such that } \int_{\Gamma} (c(y) - \phi(a(y), b(y))) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \dots, P$$

- Implement rules in operator overloading library, similar to Sacado for AD

– See Le Maître and Knio, 2010.



SG Projections of Arithmetic Operations

- Addition/subtraction

$$c = a \pm b \Rightarrow c_i = a_i \pm b_i$$

- Multiplication

$$c = a \times b \Rightarrow \sum_i c_i \psi_i = \sum_i \sum_j a_i b_j \psi_i \psi_j \rightarrow c_k = \sum_i \sum_j a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

- Division

$$c = a/b \Rightarrow \sum_i \sum_j c_i b_j \psi_i \psi_j = \sum_i a_i \psi_i \rightarrow \sum_i \sum_j c_i b_j \langle \psi_i \psi_j \psi_k \rangle = a_k \langle \psi_k^2 \rangle$$



Projections of Transcendental Operations

e.g., $c = \exp(a)$

- Taylor series approximations (Debusschere *et al*, UQ Toolkit)

$$c \approx \sum_{k=0}^n \frac{a^k}{k!}$$

- Use arithmetic rules for evaluating Taylor polynomial
- Convergence can cause problems

- Time integration (Debusschere *et al*, UQ Toolkit)

$$u(x) = \exp(x) \text{ is a solution to ODE } \frac{du}{dx} = u$$

- Translate this to an ODE on coefficients of c
- Call time integration package (e.g., CVODE)
- More accurate and robust, but more expensive
- Both approaches provided by `Stokhos::ForUQTKOrthogPolyExpansion`
- Requires the Fortran version of their UQ toolkit

- Quadrature

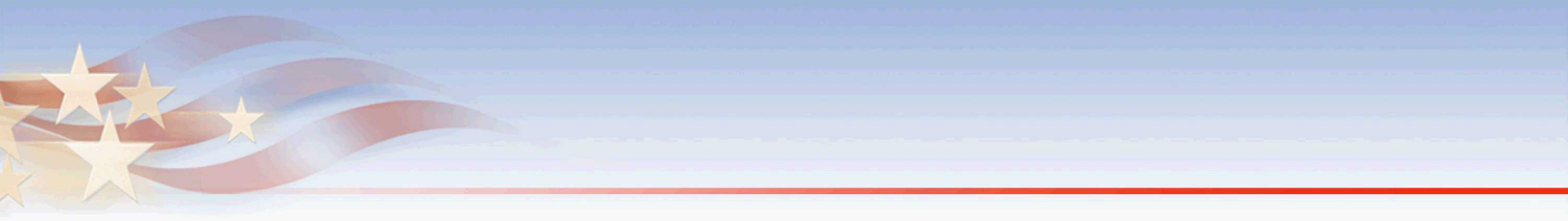
$$c_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\Gamma} \exp(a(y)) \rho(y) dy \approx \sum_{l=0}^Q w_l \exp(a(y_l))$$

- I found to be more robust than Taylor series, more efficient than time integration
- Take advantage of sparse-grid technology
- `Stokhos::QuadOrthogPolyExpansion`

Galerkin Expansion Classes and Scalar Types

- **Stokhos::OrthogPolyExpansion<ordinal_type, value_type>**
 - Abstract interface for computing Galerkin PC expansion/projection for each type of elementary operation (+, -, *, /, exp, log, sin, cos, ...), e.g.,
 - One method for each elementary operation
 - Coefficients stored in OrthogPolyApprox<> objects
- **Stokhos::PCE::OrthogPoly<>, Stokhos::ETPCE::OrthogPoly<>**
 - Stochastic Galerkin scalar types
 - Internally store coefficients in OrthogPolyApprox<> objects
 - Use OrthogPolyExpansion<> to implement overloaded operators
 - “ET” version uses limited expression templates (addition, subtraction, multiplication)
 - For quadrature expansion approach, a more complete ET implementation could be useful, but not implemented





Simple PCE Example





Solving Stochastic Problems

- Tools discussed so far provide means of generating stochastic Galerkin residual & Jacobian entries

$$F_i = \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy, \quad \langle \cdot \rangle = \int_{\Gamma} \cdot \rho(y) dy$$
$$\frac{\partial F_i}{\partial u_j} \approx \sum_{k=0}^P J_k \langle \psi_i \psi_j \psi_k \rangle, \quad J_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\Gamma} \frac{\partial f}{\partial u}(\hat{u}(y), y) \psi_k(y) \rho(y) dy$$

- To solve problems, we need
 - Parallel data structures
 - Solver algorithms
 - Interfaces between solvers and application codes



Parallel Linear Algebra

- **Epetra: Trilinos Linear Algebra Services Package**
 - Concrete data structures for distributing vectors and matrices across parallel machine
 - Hard-coded to double precision floating point numbers and 32-bit integer indexing (local & global)
 - Thyra: Abstract *interfaces* to linear algebra
 - Tpetra: Next-generation Epetra templated on scalar and ordinal types
 - Foundation for many solver/preconditioner packages
 - AztecOO, Amesos, Ifpack, ML, ...
 - Often encapsulates Trilinos interface into application code
- **Epetra_Map**
 - Describes layout of linear algebra objects across machine
- **Epetra_Vector/Epetra_MultiVector**
 - Parallel distributed vector/multi-vector
- **Epetra_CrsMatrix**
 - Parallel distributed sparse matrix format
- **Epetra_Operator**
 - Abstract operator interface
- **Stokhos parallel tools are currently built on Epetra**
 - Assume application code provides deterministic vectors/matrices through Epetra objects
 - On-going work to build on Tpetra (more on this in part 4).



Vector/Matrix/Operator Polynomials

- **Stokhos::VectorOrthogPoly<coeff_type>**
 - **Analog of Stokhos::OrthogPolyApprox where coefficients are vectors/matrices/operators instead of scalars**

- **SG residual polynomial:**
 - **VectorOrthogPoly<Epetra_Vector>**

$$F(y) = \sum_{k=0}^P F_k \psi_k(y), \quad F_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\Gamma} f(u(y), y) \psi_k(y) \rho(y) dy$$

- **F_k = Epetra_Vector**

- **SG Jacobian matrix/operator polynomial:**
 - **VectorOrthogPoly<Epetra_CrsMatrix>**
 - **VectorOrthogPoly<Epetra_Operator>**

$$J(y) = \sum_{K=0}^P J_K \psi_K(y), \quad J_K = \frac{1}{\langle \psi_K^2 \rangle} \int_{\Gamma} \frac{\partial f}{\partial u}(u(y), y) \psi_K(y) \rho(y) dy$$

- **J_k = Epetra_CrsMatrix or Epetra_Operator**



Product Vectors/Multi-Vectors

- **EpetraExt::BlockVector, BlockMultiVector**
 - Map collection Epetra_Vectors/EpetraMultiVectors (with same parallel distribution) to a single product vector

$$\{F_0, \dots, F_p\} \rightarrow \begin{bmatrix} F_0 \\ \vdots \\ F_P \end{bmatrix}$$

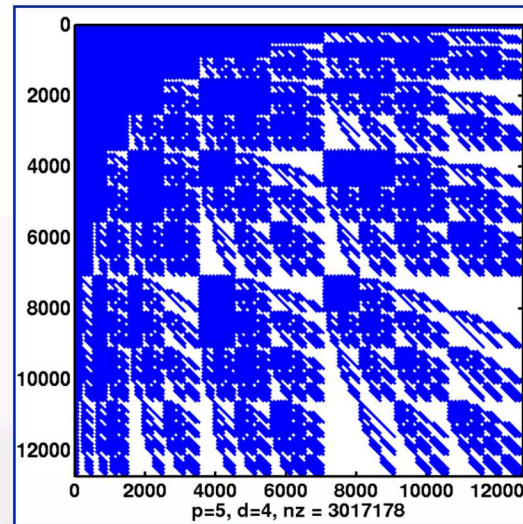
- Supports arbitrary parallel redistribution of resulting block vector/multivector
 - E.g., supports additional parallelism over blocks
- Is a Epetra_Vector/Epetra_MultiVector (inheritance)
- Allows easy construction of SG solution & residual vector from VectorOrthogPoly coefficients



Product Matrices

- **EpetraExt::BlockCrsMatrix**
 - Map collection of Epetra CRS matrices to a single sparse CRS matrix

$$\{J_0, \dots, J_p\} \rightarrow$$



- Stokhos can do this, but it is expensive and unnecessary (for iterative solvers)

SG Matrix Free Operator

- **Stokhos::MatrixFreeEpetraOp**
 - Implements **SG operator-vector apply** using formula

$$\frac{\partial F_i}{\partial u_j} \approx \sum_{k=0}^P J_k \langle \psi_i \psi_j \psi_k \rangle \implies \left(\frac{\partial F}{\partial U} V \right)_i \approx \sum_{j=0}^P \sum_{k=0}^P J_k v_j \langle \psi_i \psi_j \psi_k \rangle$$

- **Requires**
 - **VectorOrthogPoly<Epetra_Operator>**
(Epetra_Operator interface for each J_k)
 - **Sparse3Tensor**

SG Matrix-free Multiply Algorithm

$$y_i = \sum_{j=0}^P \sum_{k=0}^P A_k x_j C_{ijk}, \quad i = 0, \dots, P$$

```
for  $i = 0$  to  $P$  do
   $y_i = 0$ 
end for
for  $k = 0$  to  $P$  do
   $J = \{j \mid C_{ijk} \neq 0 \text{ for some } i\}$ 
   $U = \{x_j \mid j \in J\}$ 
   $V = A_k U$ 
  for  $j' = 0$  to  $|J|$  do
     $j = J(j')$ 
     $I = \{i \mid C_{ijk} \neq 0\}$ 
    for  $i$  in  $I$  do
       $y_i = y_i + C_{ijk} U_{j'}$ 
    end for
  end for
end for
end for
```



Preconditioning

- **Stokhos::MeanEpetraOp**

- Preconditioner for SG operator using a preconditioner for the mean block, applied on the diagonal

$$P_0^{-1} \approx J_0^{-1}, \quad P = \begin{bmatrix} P_0 & 0 & \dots & 0 \\ 0 & P_0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & P_0 \end{bmatrix}$$

- Simply an approximate block-diagonal preconditioner

- **Extensions of this to more complex block preconditioners:**

- Jacobi (Stokhos::ApproxJacobiPreconditioner)
- Gauss-Seidel (Stokhos::ApproxGaussSeidelPreconditioner)
- Kronecker product (Stokhos::KroneckerProductPreconditioner) (E. Ullman, SISC, 2010)
- Schur-complement (Stokhos::ApproxSchurComplementPreconditioner)(Sousedik et al, NLA, 2014)

- **Each of these can use several Trilinos preconditioners to approximate inverse of the blocks**

- Ifpack (incomplete LU/Cholesky, relaxation, polynomial, overlapping Schwarz)
- ML (algebraic multigrid)

- **With these tools, we can support preconditioned iterative linear solvers**

- E.g., GMRES and CG by AztecOO and Belos



(Nonlinear) Application Code Interface

- **Originally, each nonlinear analysis package had its own application code interface**
 - **Nonlinear solver, time integration, optimization, stability analysis, ...**
 - **Difficult for applications to support**
 - **Reinventing the wheel for each new package**
- **ModelEvaluator**
 - **Single interface for all analysis packages**
 - **Application only needs to support one interface**
 - **Concrete (EpetraExt – for applications) and abstract (Thyra – for abstract analysis packages) versions**



EpetraExt::ModelEvaluator

- Interface to get residuals, Jacobians, etc...from an application code
- Single evalModel routine
 - Often more efficient to evaluate quantities together instead of separate calls
 - E.g., residual and Jacobian
 - InArgs and OutArgs store inputs and outputs
- InArgs: Struct storing all the things the solution might depend on
 - Solution vector x ,
 - Solution time derivative vector dx/dt
 - Time t ,
 - Parameters p
 - ...
- OutArgs: Struct storing all the things model can compute
 - Residual vector f
 - Transient Jacobian $W = a*df/dx + b*df/(dx/dt)$,
 - Parameter derivatives df/dp ,
 - Response functions g ,
 - ...
- How is this implemented?
 - Application code specifies what InArgs and what OutArgs it supports
 - Application checks for those InArgs and OutArgs, and computes whatever was requested



Stochastic Galerkin ModelEvaluator Extensions

- For most InArgs and OutArgs, SG versions added for stochastic Galerkin analog
 - E.g., `x_sg`, `f_sg`, `W_sg`, `p_sg`, ...
 - Each is a `Stokhos::VectorOrthogPoly` of corresponding coefficient type
 - `VectorOrthogPoly<Epetra_Vector>` for `x_sg`, `f_sg`,
 - `VectorOrthogPoly<Epetra_Operator>` for `W`
- SG-enabled application just checks for these InArgs and OutArgs and computes them based on its implementation
 - E.g., operator overloading, quadrature, ...



Stokhos SG Model Evaluator

- Translates Stochastic Galerkin problem to a standard deterministic problem
 - Converts SG in/out-args to standard (block) in/out args:
 - $\text{IN_ARG_X_SG (VectorOrthogPoly<Epetra_Vector>)} \leftrightarrow \text{IN_ARG_X (Epetra_Vector)}$

$$\hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi) \leftrightarrow U = \begin{bmatrix} u_0 \\ u_1 \\ \vdots \\ u_P \end{bmatrix}$$

- $\text{OUT_ARG_F_SG (VectorOrthogPoly<Epetra_Vector>)} \leftrightarrow \text{OUT_ARG_F (Epetra_Vector)}$

$$\hat{f}(\xi) = \sum_{i=0}^P f_i \psi_i(\xi) \leftrightarrow F = \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_P \end{bmatrix}$$

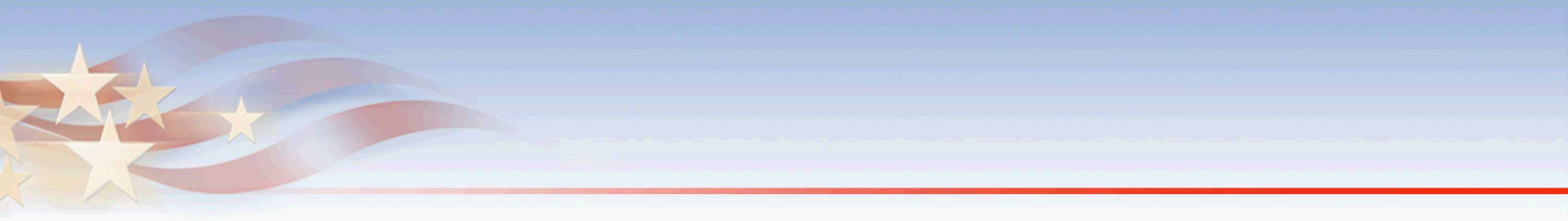
- $\text{OUT_ARG_W_SG (VectorOrthogPoly<Epetra_Operator>)} \leftrightarrow \text{OUT_ARG_W (Epetra_Operator)}$

$$\hat{A}(\xi) = \sum_{i=0}^P A_i \psi_i(\xi) \leftrightarrow A = \frac{\partial F}{\partial U} \approx \sum_{k=0}^P G_k \otimes A_k$$



Stokhos SG ModelEvaluator

- **SG ModelEvaluator can then be given to standard solvers**
 - **NOX (nonlinear solvers)**
 - **Rythmos (time integration)**
 - ...
- **Also provides interface for creating customized preconditioners**
- **Encapsulated within Piro package**
 - **More on this in Part 3.**



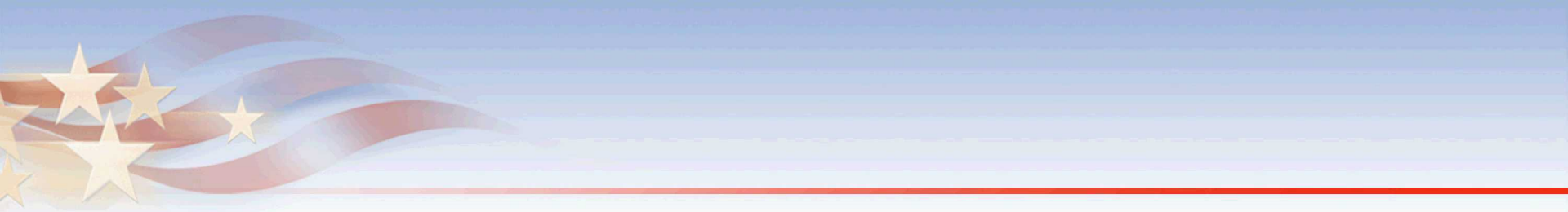
Simple Nonlinear Solver Examples



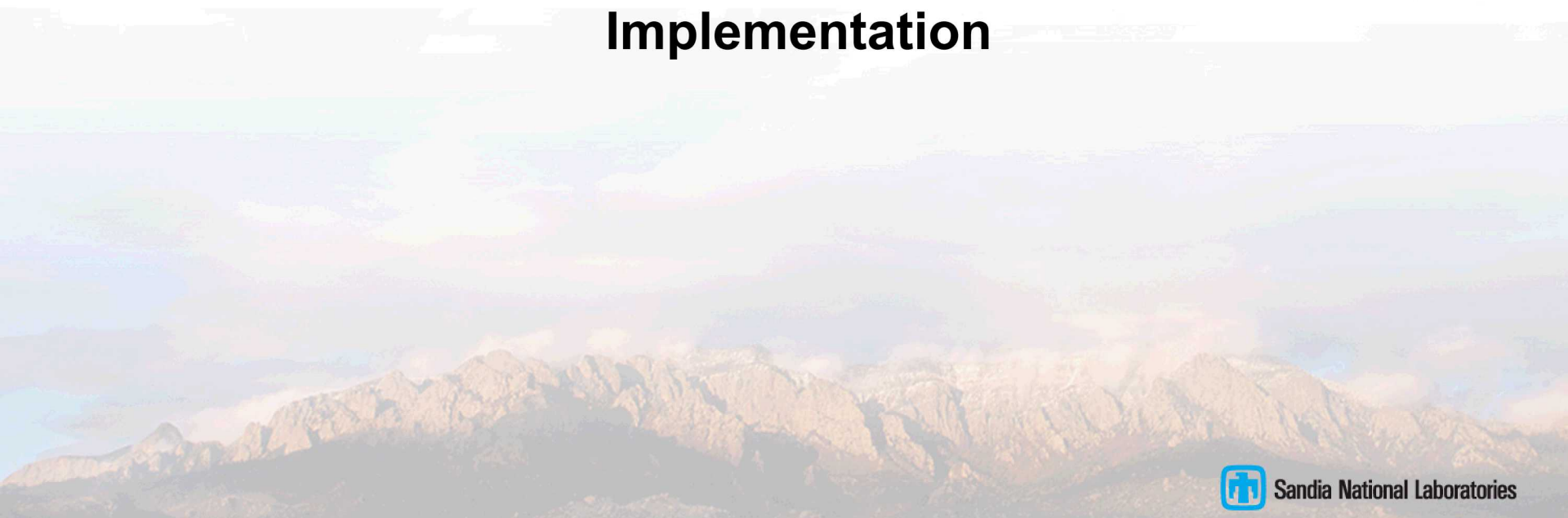


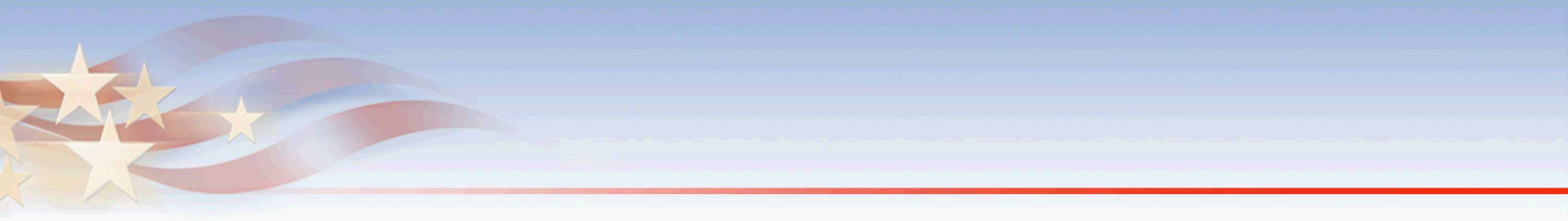
Factories

- **Stokhos employs the “factory” pattern for many of its objects**
 - **Basis**
 - **Quadrature**
 - **Pseudospectral operators**
 - **Galerkin operators**
 - **Preconditioners**
 - ...
- **These handle creating the object for you**
 - **Insulate setup code from needing to know all possible options**
 - **Easy to drive through ParameterList’s**
- **Use extensively in Piro and Albany**
 - **More on this in Part 3**



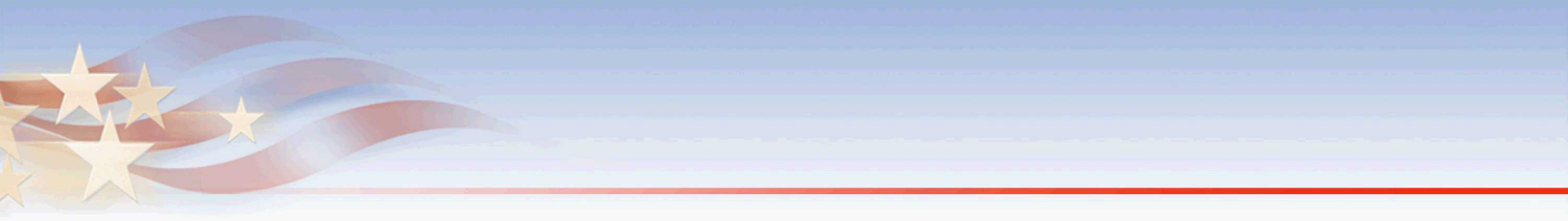
Advanced Topic: MatrixFreeEpetraOp Implementation





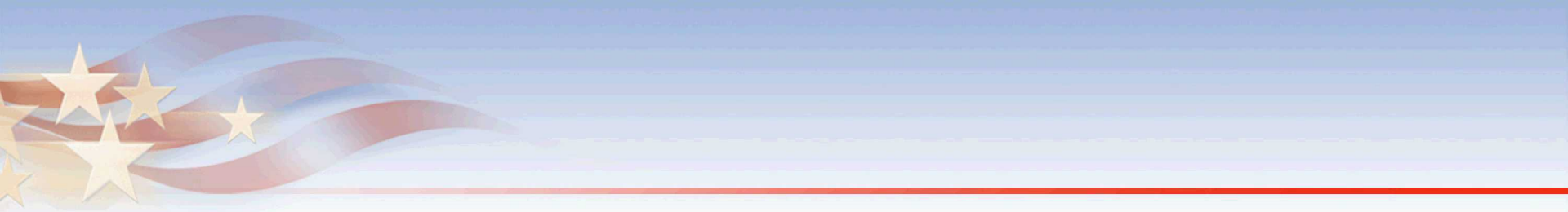
Advanced Topic: Preconditioner Implementations





Advanced Topic: Preconditioner Factory





Advanced Topic: Smolyak Basis and Pseudospectral Operator



Pseudospectral Approximations

- When approximating PCE coefficients numerically, must be careful to preserve discrete orthogonality of the PCE basis
- First discovered with regards to sparse grid quadrature for pseudospectral approximations
 - Constantine et al, CMAME, 2012
 - Later analyzed in Conrad and Marzouk, SISC, 2013.

$$f_i = \int f(y) \psi_i(y) \rho(y) dy \approx \sum_{k=0}^Q w_k f(y_k) \psi_i(y_k)$$

- Must have

$$\sum_{k=0}^Q w_k \psi_i(y_k) \psi_j(y_k) = \delta_{ij}, \quad i, j = 0, \dots, P$$

- If this is not true, this introduces $O(1)$ error into the approximated PCE coefficients
 - Called aliasing error
 - See Conrad & Marzouk
- Since a Gaussian quadrature rule with $n+1$ points exactly integrates polynomials of order $2n+1$, this is straightforward for
 - 1-D Gaussian integration
 - Full tensor product integration with Gaussian rules
- For sparse grids, this creates difficulty in matching sparse grid rule to PCE basis



Smolyak Operator and Basis

- Remedied by using Smolyak formula directly as a pseudospectral operator

$$A(\mathcal{K}, M, \mathcal{L}) = \sum_{k \in \mathcal{K}} c_k \mathcal{L}_{k_1}^1 \otimes \cdots \otimes \mathcal{L}_{k_M}^M$$

- where

$$\mathcal{L}_{k_i}^i(f) = \sum_{j=0}^{k_i} f_j^i \psi_j^i, \quad f_j^i = \sum_{l=0}^{k_i} w_l^i f(y_l^i) \psi_l^i(y_l^i)$$

- The Smolyak basis is the set of multivariate polynomials where the Smolyak operator preserves discrete orthogonality
 - **Stokhos::SmolyakBasis** implements this
- **Stokhos::SmolyakPseudospectralOperator** implements the operator
 - Can be used within operator overloading rules (**Stokhos::PseudospectralOrthogPolyExpansion**)
- Can use quadrature directly when using
 - **Total order tensor product basis**
 - **Sparse grid based on Gaussian abscissas with linear growth rule**





References

- **Polynomial Chaos and Stochastic Galerkin**

- R. G. Ghanem and P. D. Spanos. *Stochastic Finite Elements: A Spectral Approach*. Springer-Verlag, New York, 1991.
- M. Pellissetti and R. Ghanem, “Iterative solution of systems of linear equations arising in the context of stochastic finite elements,” *Advances in Engineering Software*, vol. 31, no. 8, 2000.
- D. Xiu and G. Karniadakis, “The Wiener-Askey polynomial chaos for stochastic differential equations,” *SISC*, vol. 24, no. 2, 2002.
- M. Reagan, H. Najm, R. Ghanem, O. Knio, “Uncertainty quantification in reacting-flow simulations through non-intrusive spectral projection,” *Combustion and Flame*, vol. 132, no. 3, 2003.
- B. Debusschere, H. Najm, P. Pebay, O. Knio, R. Ghanem, and O. L. Maitre. “Numerical challenges in the use of polynomial chaos representations for stochastic processes.” *SISC*, 26(2): 698–719, 2004.
- W. Gautschi, *Orthogonal Polynomials*, Oxford University Press, 2004.
- O.P. Le Maître and O.M. Knio, *Spectral Methods for Uncertainty Quantification*, Springer, 2010.
- E. Ullmann, “A Kronecker Product Preconditioner for Stochastic Galerkin Finite Element Discretizations,” *SISC*, Vol. 32, No. 2, 2010.
- P. Constantine, M. Eldred, and E. Phipps, “Sparse Pseudospectral Approximation Method,” *CMAME*, vol. 229-232, 2012.
- P. Conrad and Y. Marzouk, “Adaptive Smolyak Pseudospectral Approximations,” *SISC*, vol. 35, No. 6, 2013.
- B. Sousedik, R. Ghanem, and E. Phipps, “Hierarchical Scur Complement Preconditioner for the Stochastic Galerkin Finite Element Methods,” *NLA*, Vol 21, No. 1, 2014.

- **Template-based Generic Programming and UQ**

- R. Pawlowski, E. Phipps, and A. Salinger, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part I: template-based generic programming,” *Journal of Scientific Programming*, vol. 20 (2), 2012.
- R. Pawlowski, E. Phipps, A. Salinger, S. Owen, C. Siefert, and M. Staten, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part II: application to partial differential equations,” *Journal of Scientific Programming*, vol. 20 (3), 2012.

- **Dimension reduction (defining new polynomial bases)**

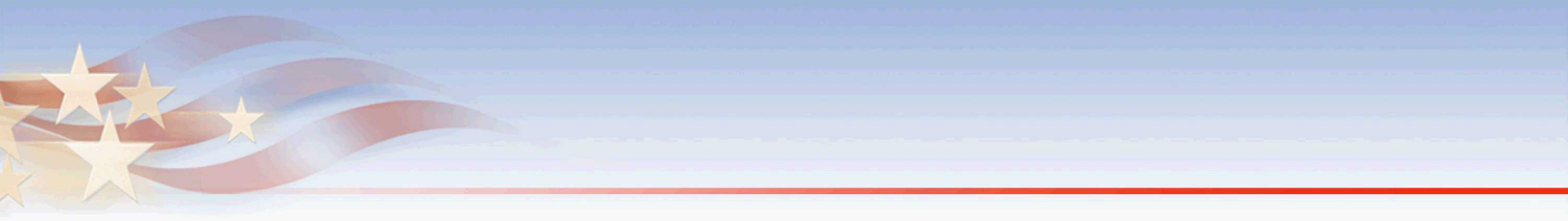
- P. Constantine and E. Phipps, “A Lanczos Method For Approximating Composite Functions,” *Applied Mathematics and Computation*, vol. 218 (24), 2012.
- P. Constantine, E. Phipps, and T. Wildey, “Efficient uncertainty propagation for network multiphysics systems,” submitted to *International Journal for Numerical Methods in Engineering*, 2013.



Stochastic Galerkin Research

- **Efficiently evaluating PC coefficients for nonlinear, transcendental operations**
- **Solvers and preconditioners**
 - Eliminate dependence on variance, polynomial order
 - Multilevel solvers/preconditioners for PC system
- **Transient problems**
 - Instabilities, stiffness
- **Multiphysics coupling**
 - Optimizing PC basis at physics/scale interfaces





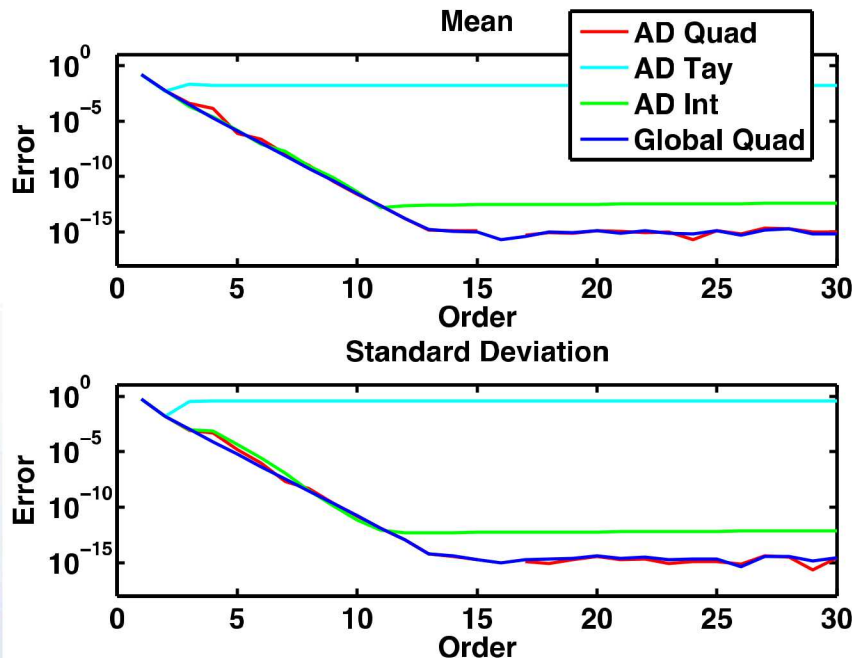
Auxiliary Slides



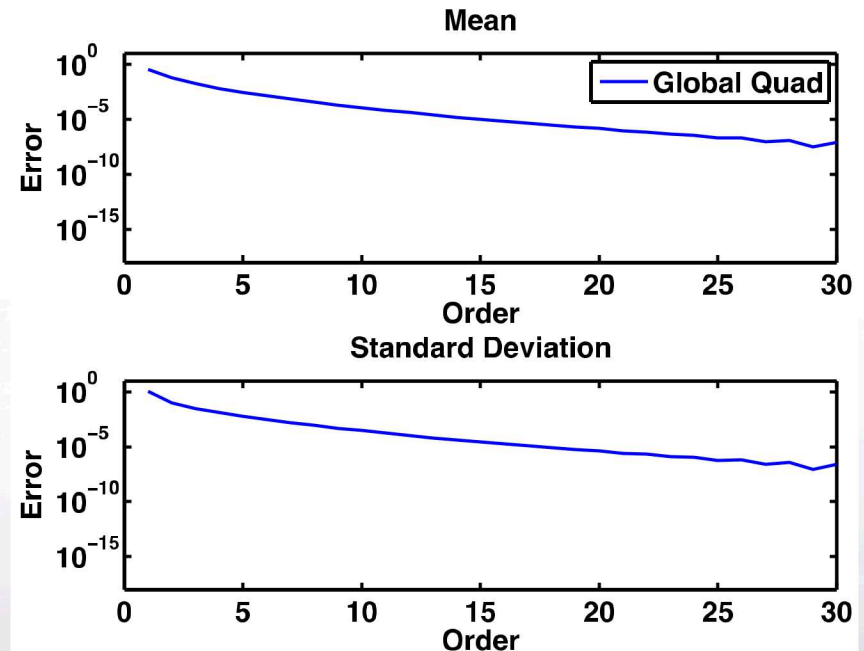
Accuracy of Operator Overloading Approaches

$$u = \log \left(\frac{1}{1 + (e^x)^2} \right)$$

Uniform U(-1,1) x

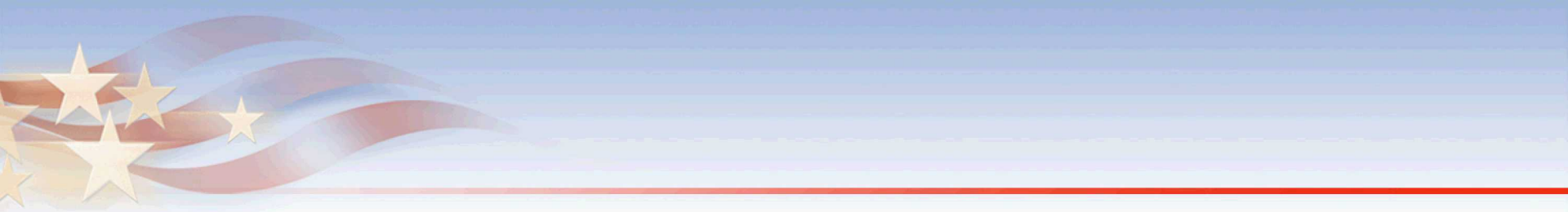


Gaussian N(0,1) x



All 3 AD approaches fail

- Operator overloading approach is usually accurate
- Truncation error *can* cause catastrophic failure



Embedded UQ Methods in Albany





Outline

- **Supporting embedded algorithms in large-scale codes**
 - Templates
 - Element-based codes and gather/scatter
- **Albany**
 - Component design
 - Piro
 - Heat equation example with sensitivities
- **Albany-Dakota**
 - Dakota
 - TriKota
 - NISP example
 - Regression-PCE w/gradients example
 - VPS w/gradients example
- **Albany-Stokhos**
 - Kokkos and changes it required
 - SGQuadModelEvaluator
 - Stochastic Galerkin example with viz
- **Albany internals**
 - Phalanx
 - Gather/scatter
 - PDE terms
 - Parameters
 - Stochastic Galerkin preconditioners





Challenges of embedded algorithms

- **Many kinds of quantities required**
 - State and parameter derivatives
 - Various forms of second derivatives
 - Polynomial chaos expansions
 - ...
- **Incorporating these directly requires significant effort**
 - Time consuming, error prone
 - Gets in the way of physics/model development
- **Requires code developers to understand requirements of algorithmic approaches**
 - Limits embedded algorithm R&D on complex problems
- **Need a framework that**
 - Allows simulation code developers to focus on complex physics development
 - Doesn't make them worry about advanced analysis
 - Allows derivatives and other quantities to be easily extracted
 - Is extensible to future embedded algorithm requirements



A Solution Through Templates

- Recall C++ templates provide API for incorporating automatic derivative calculations
- Benefits of templating
 - Developers only develop, maintain, test one templated code base
 - Developers don't have to worry about what the scalar type really is
 - Easy to incorporate new scalar types
- Templates provide a deep interface into code
 - Can use this interface for more than derivatives
 - Any calculation that can be implemented in an operation-by-operation fashion will work
 - Extension to general scalar types we call template-based generic programming



A solution Through Templates

- **Template-based generic programming**
 - Code developers write physics code templated on scalar type
 - Operator overloading libraries provide tools to propagate needed embedded quantities
 - Libraries connect these quantities to embedded solver/analysis tools
- **Plethora of scalar types enable many forms of embedded analysis**
 - Jacobians – forward mode AD
 - Adjoint – forward or reverse mode AD
 - Hessians – nested forward/reverse AD
 - Spectral UQ methods – polynomial chaos expansions
 - Sampling UQ methods – multi-point vector
 - Epistemic UQ methods – intervals, fuzzy numbers
- **Strategy:**
 - Template PDE residual evaluation on scalar type
 - Instantiate template code on appropriate scalar type for each type of analysis
 - Connect to high-level analysis algorithm through an interface

Differentiating Element-Based Codes

- Global residual computation (ignoring boundary computations):

$$f(x) = \sum_{i=1}^N Q_i^T e_{k_i}(P_i x)$$

- Jacobian computation:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^N Q_i^T J_{k_i} P_i, \quad J_{k_i} = \frac{\partial e_{k_i}}{\partial x_i}, \quad x_i = P_i x$$

- Jacobian-transpose product computation:

$$w^T \frac{\partial f}{\partial x} = \sum_{i=1}^N (Q_i w)^T J_{k_i} P_i$$

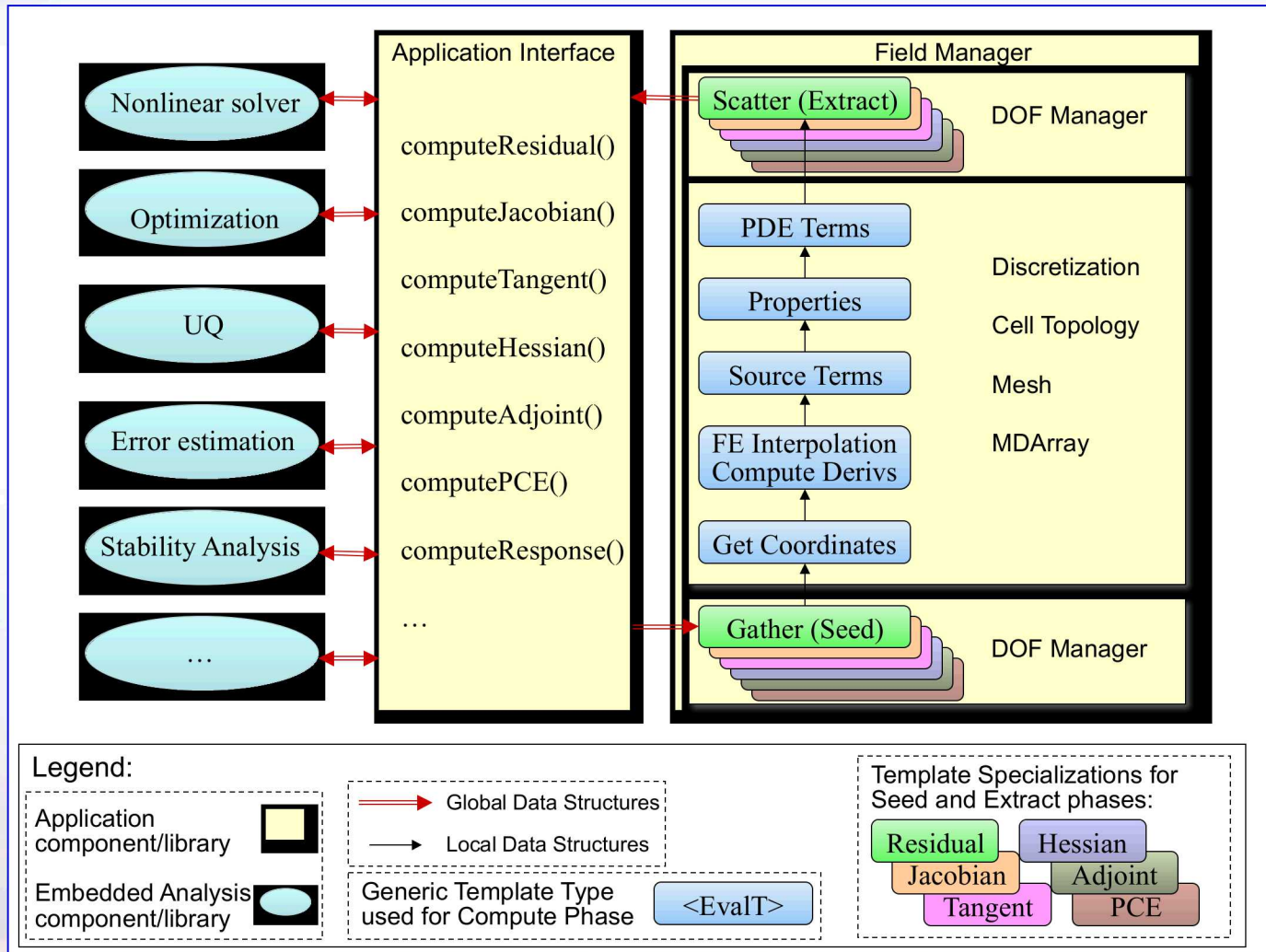
- Hybrid symbolic/AD procedure
 - Element-level derivatives computed via AD
 - Exactly the same as how you would do this “manually”
 - Avoids parallelization issues



Template-Specialized Gather/Scatter

- **Structure generalizes to all scalar types:**
 - **For each element:**
 - Extract local DOFs from global solution vector
 - Initialize local DOF scalars based on scalar type
 - Evaluate templated local element residual
 - Extract data from local element residual scalars
 - Scatter data into global data structures
- **Encapsulate into templated gather-scatter operations**
 - **Partial template specialization of each gather-scatter on relevant scalar type**

Templates Orthogonalize Physics and Embedded Algorithm R&D



Tools and techniques have been developed, implemented in SNL Albany code

Sandia National Laboratories' Albany multiphysics code

2,654 commits 3 branches 0 releases 10 contributors

branch: master Albany

adding check for quadratic input mesh

ibaned authored 3 hours ago latest commit c96d76ef13

doc	Merge remote-tracking branch 'github/master' into fix_time	a day ago
examples	Merge remote-tracking branch 'github/master' into fix_time	a day ago
project	Continue adding nodal state field support.	4 months ago
src	adding check for quadratic input mesh	3 hours ago
.gitignore	Fix ETI issues uncovered by Intel compiler.	2 days ago
ALBANY_MASTER_B...	Albany development has moved to a git repository!	4 years ago
CMakeLists.txt	Merge remote-tracking branch 'github/master' into fix_time	a day ago
README.md	Fix another typo.	11 days ago
license.txt	Replace copyright banner with a short 2012 banner for Albany ...	a year ago

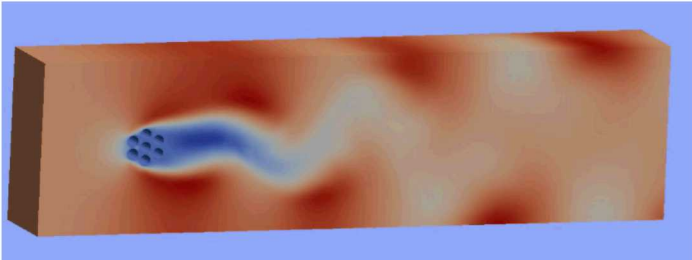
README.md

Albany

Albany is an implicit, unstructured grid, finite element code for the solution and analysis of partial differential equations.

Features

Analysis of complex multiphysics problems



<https://github.com/gahansen/Albany>

- Albany lead/PI: Andy Salinger (SNL)
- Hosts several application and algorithms R&D efforts
 - Applications:
 - Mechanics (LCM)
 - Quantum Devices (QCAD)
 - Ice Sheets (FELIX)
 - Atmosphere Dynamics (Aeras)
 - Particle-Continuum coupling (Peridigm)
 - Additive Manufacturing
 - Algorithms:
 - Adaptivity (PUMI)
 - Embedded UQ (Equinox)
 - Topological Optimization (ATO)
 - Performance Portable FEM (Kokkos, Intrepid2)
 - Scalable Solvers (MueLu)
 - Adjoint-Based Inversion
 - UQ Workflow (QUEST)
 - Goal-Oriented Adaptive Refinement
 - Model Order Reduction (RAZOR)
- Incorporates many advanced analysis techniques
- Effective test-bed for developing stochastic Galerkin algorithms and solvers

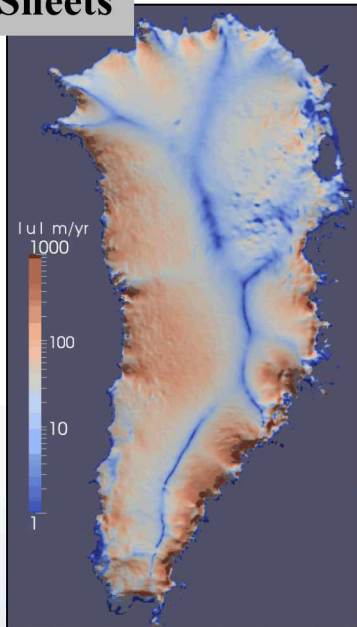


Sandia National Laboratories

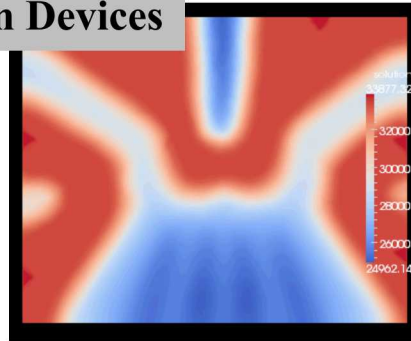


is a finite element code that supports many kinds of physics

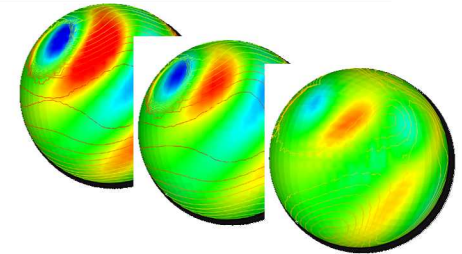
Ice Sheets



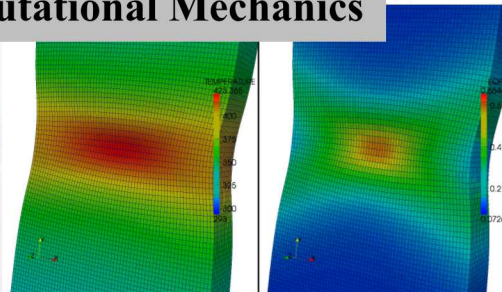
Quantum Devices



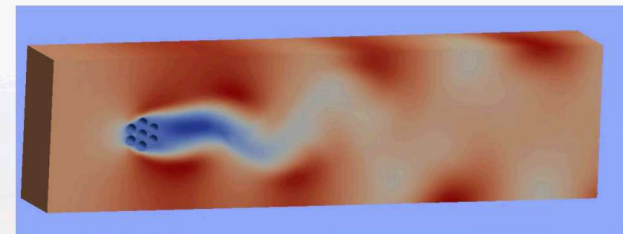
Atmosphere Dynamics



Computational Mechanics



Incompressible Flow





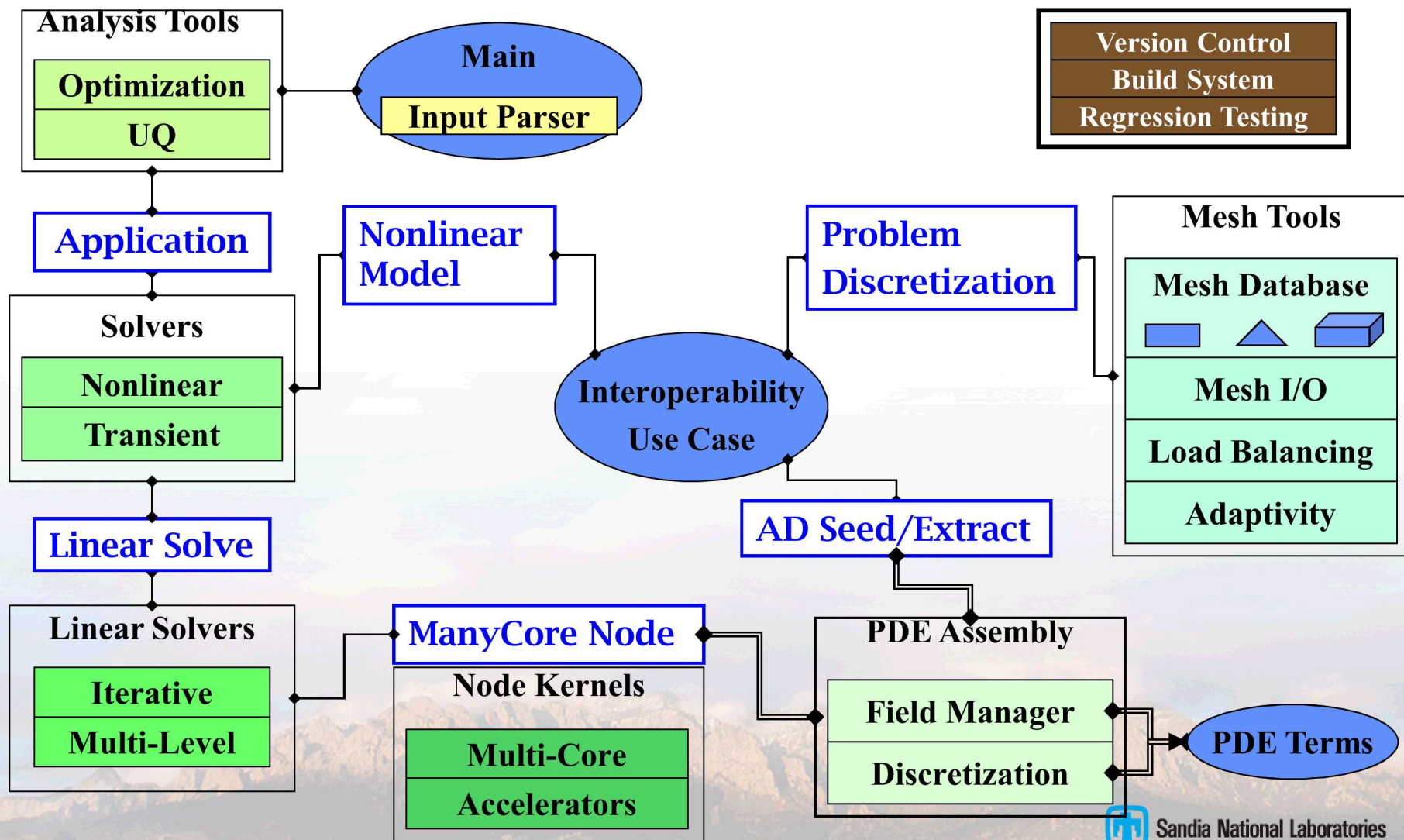
Component-Based Application:

Software Quality Tools

Libraries

Interfaces

GlueCode

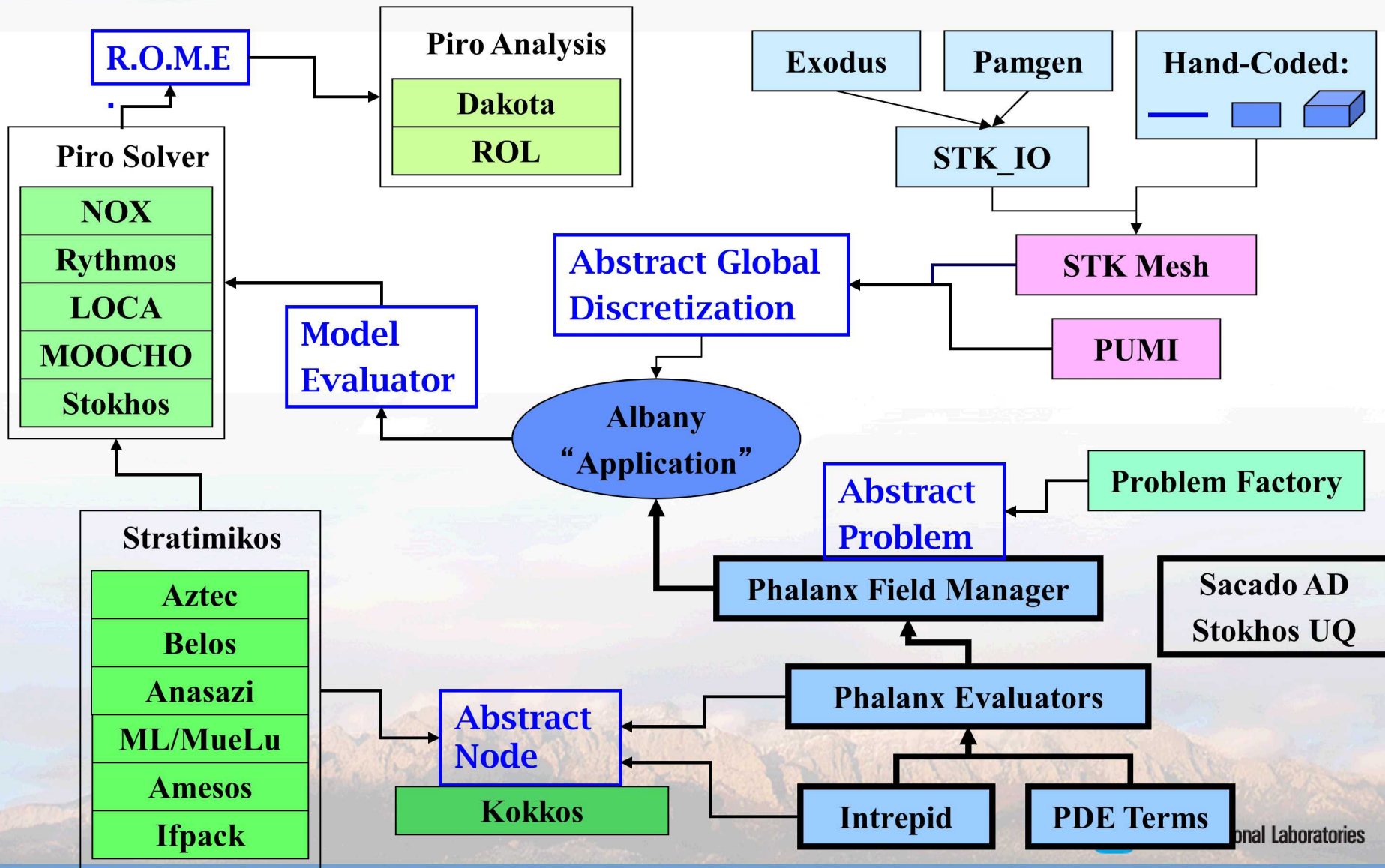




Trilinos



Albany is built from Trilinos and Dakota component libraries



Piro: Parameters-In Responses-Out

- Recall ModelEvaluator interface from part 2 (steady-state for simplicity):

$$f(u, y) = 0, \quad v(y) = h(u, y)$$

- Model inputs: solution vector u , parameters y
- Model outputs: residual vector f , response v , derivatives df/du , df/dy , dv/du , dv/dy , ...

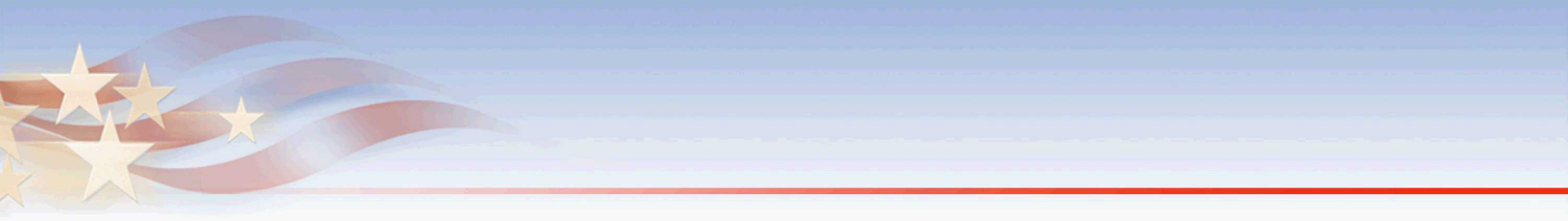
- Given application ModelEvaluator, Piro creates a ModelEvaluator that “looks” like just a mapping from parameters y to responses v (response-only ModelEvaluator):

$$v(y) = h(u(y), y) \quad \text{s.t.} \quad f(u(y), y) = 0$$

- Given parameters y , solve for u by through nonlinear solver
- Evaluate response v
- Compute response gradient via implicit function theorem:

$$\frac{\partial v}{\partial y} = \frac{\partial h}{\partial u} \left(-\frac{\partial f^{-1}}{\partial u} \frac{\partial f}{\partial y} \right) + \frac{\partial h}{\partial y} \quad \frac{\partial v^T}{\partial y} = \frac{\partial f^T}{\partial y} \left(-\frac{\partial f^{-T}}{\partial u} \frac{\partial h^T}{\partial u} \right) + \frac{\partial h^T}{\partial y}$$

- Internally Piro creates nonlinear solver (NOX) or time-integrator (Rythmos) to accomplish this
 - Entirely ParameterList driven
- Additionally, Piro provides simple interface for various analysis methods that operate on a response-only ModelEvaluator
 - Optimization (Dakota, ROL, MOOCHO, ...)
 - Sampling-based UQ (Dakota) and Stochastic Galerkin UQ (Stokhos)



Albany heat-equation demo

- **Input file**
- **Steady-state solve**
- **Visualization**
- **Sensitivities**

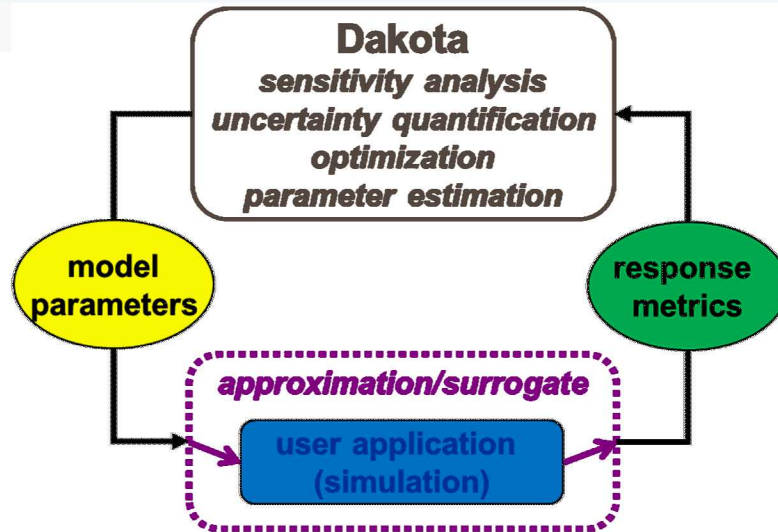


DAKOTA

Research, development, & deployment of advanced algorithms for simulation-based assessment and design*



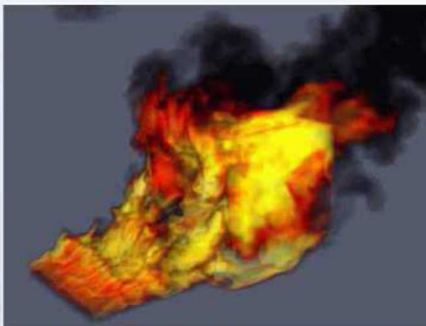
<http://dakota.sandia.gov>



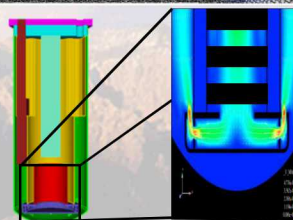
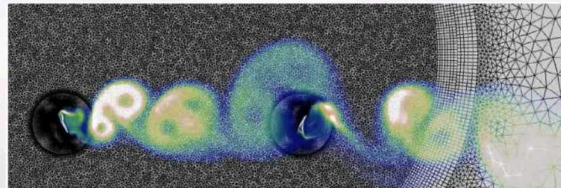
Iterative systems analysis
Multilevel parallel computing
Simulation management

Impact across a variety of DOE mission areas

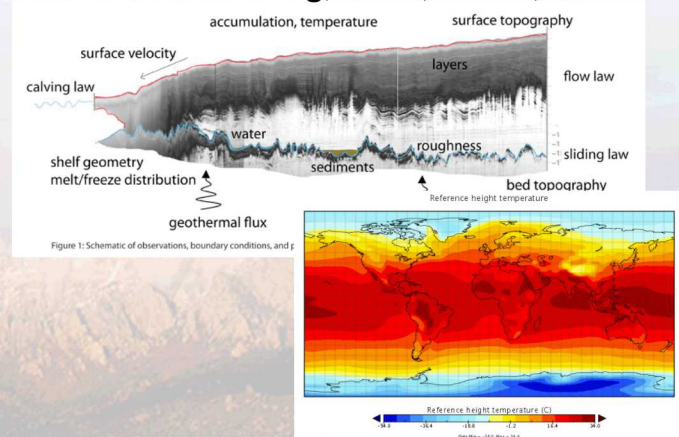
Stockpile (NNSA ASC)
 Abnormal environments



Energy (ASCR, EERE, NE)
 Wind turbines, nuclear reactors



Climate (SciDAC, CSSEF)
 Ice sheet modeling, CISM, CESM, ISSM

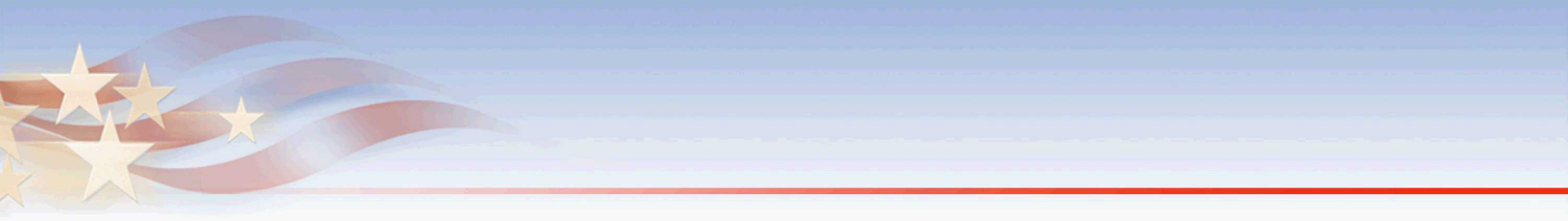


*M. Eldred, B. Adams, et al (SNL)



TriKota and Albany

- **TriKota package in Trilinos makes Dakota look like just another Trilinos package**
 - **Unpack Dakota tarball inside TriKota directory**
 - **Add `-D Trilinos_ENABLE_TriKota` to Trilinos configure**
 - **Builds Dakota as a library within Trilinos**
- **TriKota provides implementation of Dakota's embedded application interface using a Piro response-only ModelEvaluator**
 - **"Dakota library mode"**
 - **Piro can then call Dakota for any kind of Dakota UQ/Optimization analysis**
- **Makes it simple to run Dakota analysis on Albany**
 - **Just supply a Dakota input-file (no scripting necessary)**
 - **Requires Albany to implement any pre/post-processing for parameters/responses**



Albany-Dakota Examples

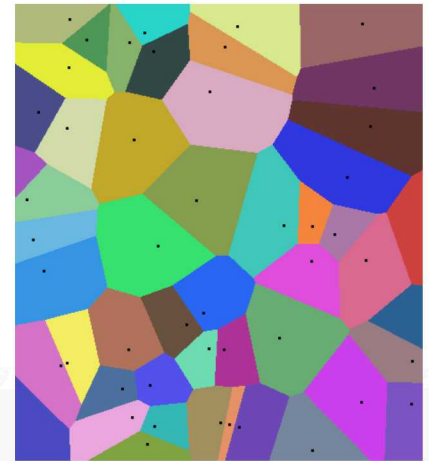
- **NISP example**
- **Regression-PCE w/gradients example**



VPS: Voronoi Piecewise Surrogates

Problem Statement: Given a d -dimensional pointset and function evaluations pairs $\{x_i, f(x_i)\}$, build a surrogate to estimate the function elsewhere.

VPS Solution: Consider the given points as *implicit Voronoi seeds*. For each cell, find the polynomial coefficients that fits the function value at the cell seed and minimizes the error at the neighbors in the *Least Squares* sense.



VPS Features:

- Only needs neighbors, not Voronoi edges.
- No explicit Voronoi Tessellation construction (no curse of dimensionality).
- Varies from local (only direct neighbors used) to global (extended neighbors).
- Better error performance compared to the global Gaussian Process Surrogate.
- A surrogate evaluation = polynomial evaluation (cheap).
- All polynomial coefficients are calculated once (less computational cost).

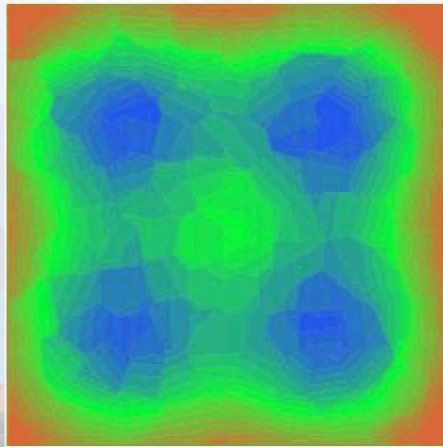


VPS with derivatives information

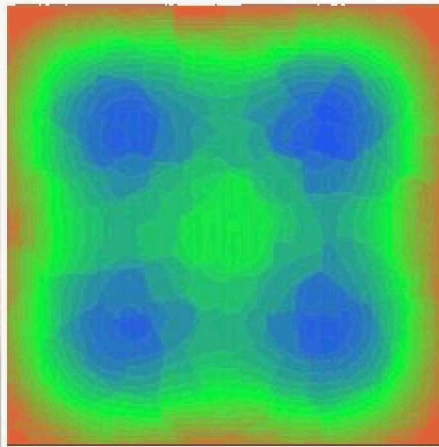
Problem Statement: Given a d -dimensional pointset $\{x_i\}$, function evaluations $\{f(x_i)\}$, as well as **gradients**, and **Hessians**, build a surrogate to estimate the function elsewhere.

VPS Solution: Consider the given points as *implicit Voronoi seeds*. For each cell, use the function **evals**, **gradients**, and **hessians** to approximate the constant, linear, and quadratic coefficients of the surrogate polynomial, in a Taylor series context. If a higher order polynomial is needed, use *Least Squares* to solve a regression problems for the remaining polynomial coefficients.

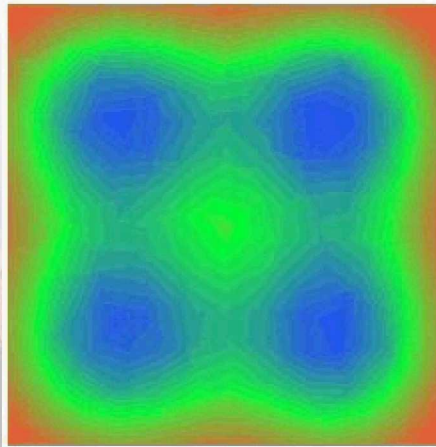
1st order VPS – 121 points
Least Squares Regression



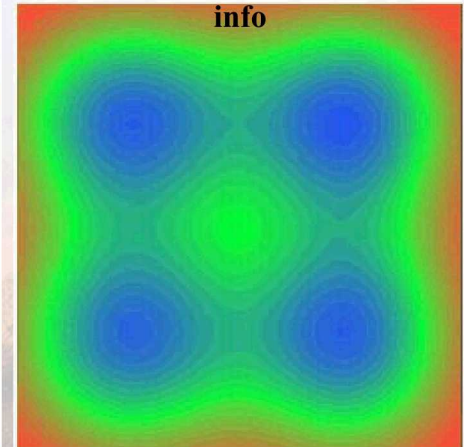
2nd order VPS – 121 points
Least Squares Regression

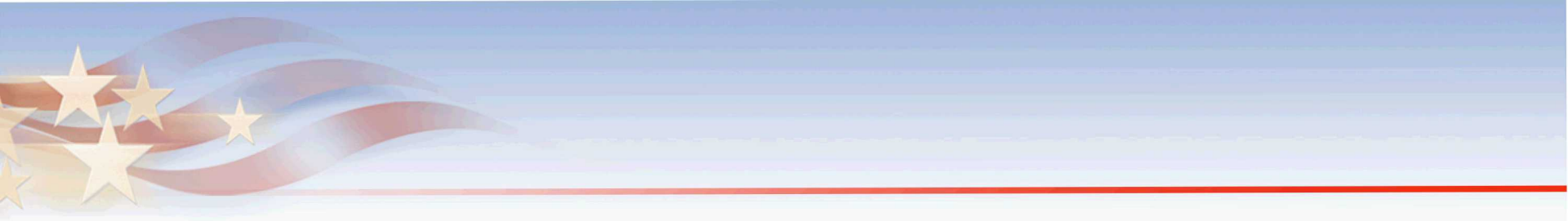


1st order VPS – 121 points
w/ **Gradient** info

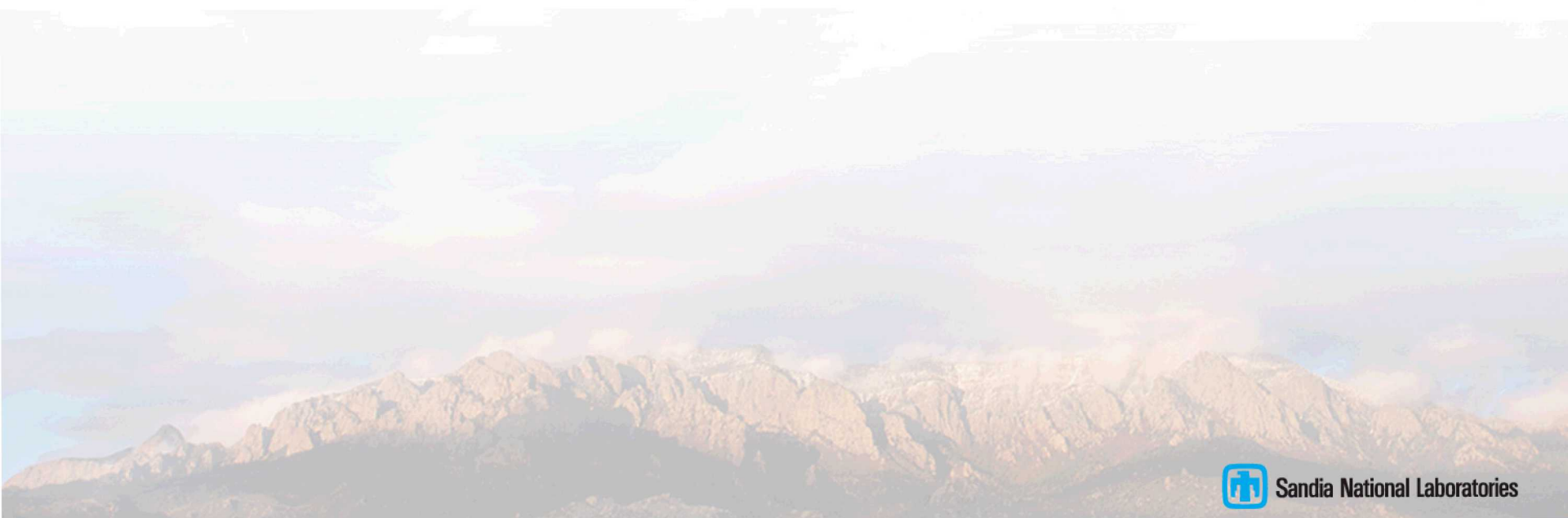


2nd order VPS – 121 points
w/ **Gradient** and **Hessian**
info



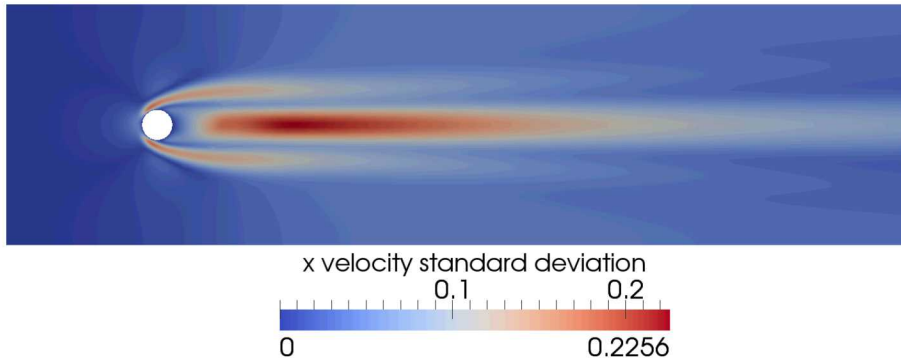


Albany VPS example



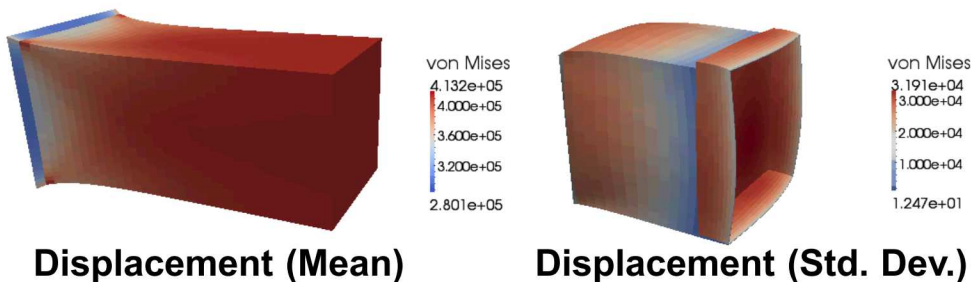
Stochastic Galerkin UQ in Albany

Navier-Stokes



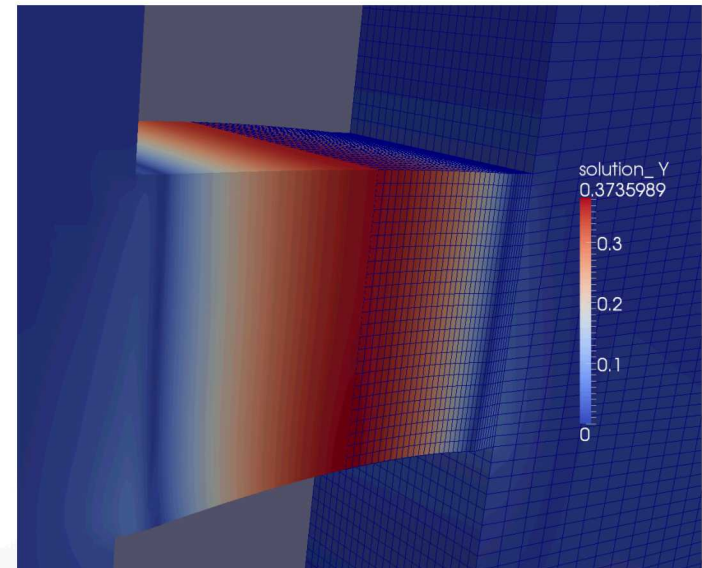
- Incompressible flow past a cylinder
 - Uncertain viscosity field
 - Standard deviation of x-velocity field

Mechanics



- Neo-Hookean nonlinear elasticity
 - Uncertain Young's Modulus field

Thermal-Electrostatics



- Sliding electromagnetic contact
 - Uncertain electrical conductivity
 - Standard deviation of maximum temperature

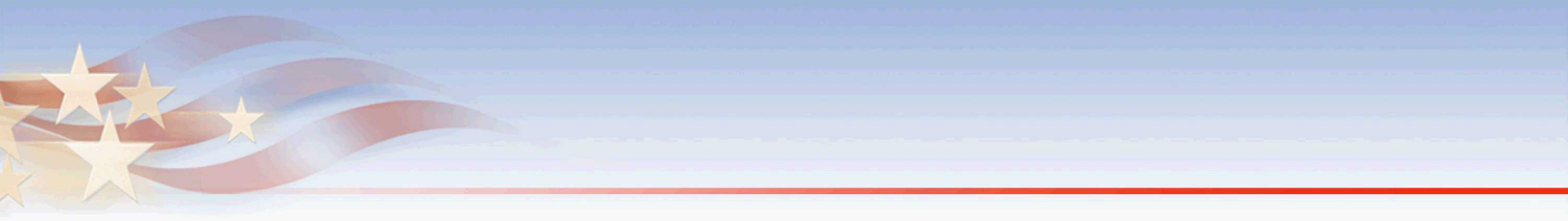
Pseudospectral SG Residual/Jacobian Evaluation

- PCE scalar type (from part 1) incorporated into Albany for SG residual/Jacobian evaluation
- Currently this is disabled due to Kokkos transition
 - Kokkos is a performance portability library for next-generation multicore CPU, GPU, Xeon Phi architectures
 - Internal PDE evaluation code being converted to use Kokkos for thread parallelism (more on this later)
 - Kokkos data structures currently don't support PCE scalar type used in Albany
 - This will be fixed in the future
- However Albany also supports a semi-intrusive pseudospectral evaluation of SG residual and Jacobian via (sparse-grid) quadrature:

$$F_i = \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy \approx \sum_{k=0}^P w_k f(\hat{u}(y_k), y_k) \psi_i(y_k)$$

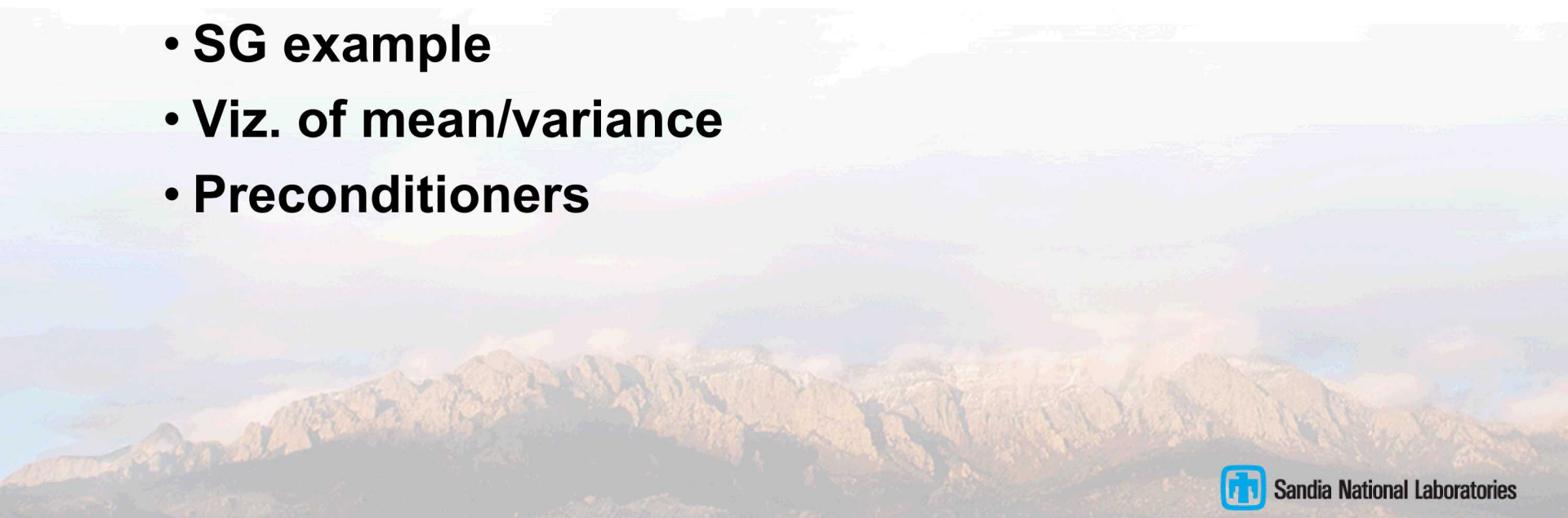
- Implemented by Stokhos::SGQuadModelEvaluator
 - Computes X_SG, W_SG, ... OutArgs by sampling given (deterministic) ModelEvaluator
 - Evaluate X_SG, P_SG at each quadrature point, evaluate model, sum results into SG OutArg
 - Provides SG capabilities to any ModelEvaluator
 - Incorporated into Piro
- In the future will also support
 - SPAM approach
 - Applying quadrature/SPAM at element-level (for better cache performance)





Albany-Stokhos Examples

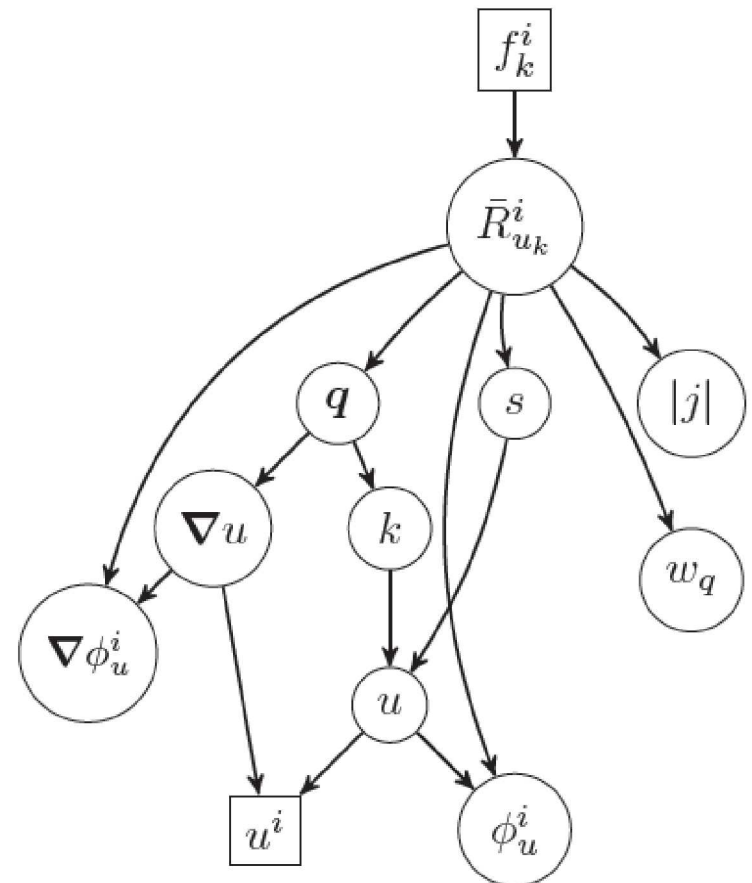
- **SG example**
- **Viz. of mean/variance**
- **Preconditioners**



Lightweight DAG-based Expression Evaluation with Phalanx (R. Pawlowski)

- Albany leverages Phalanx (R. Pawlowski) for evaluating PDE terms
- 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

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

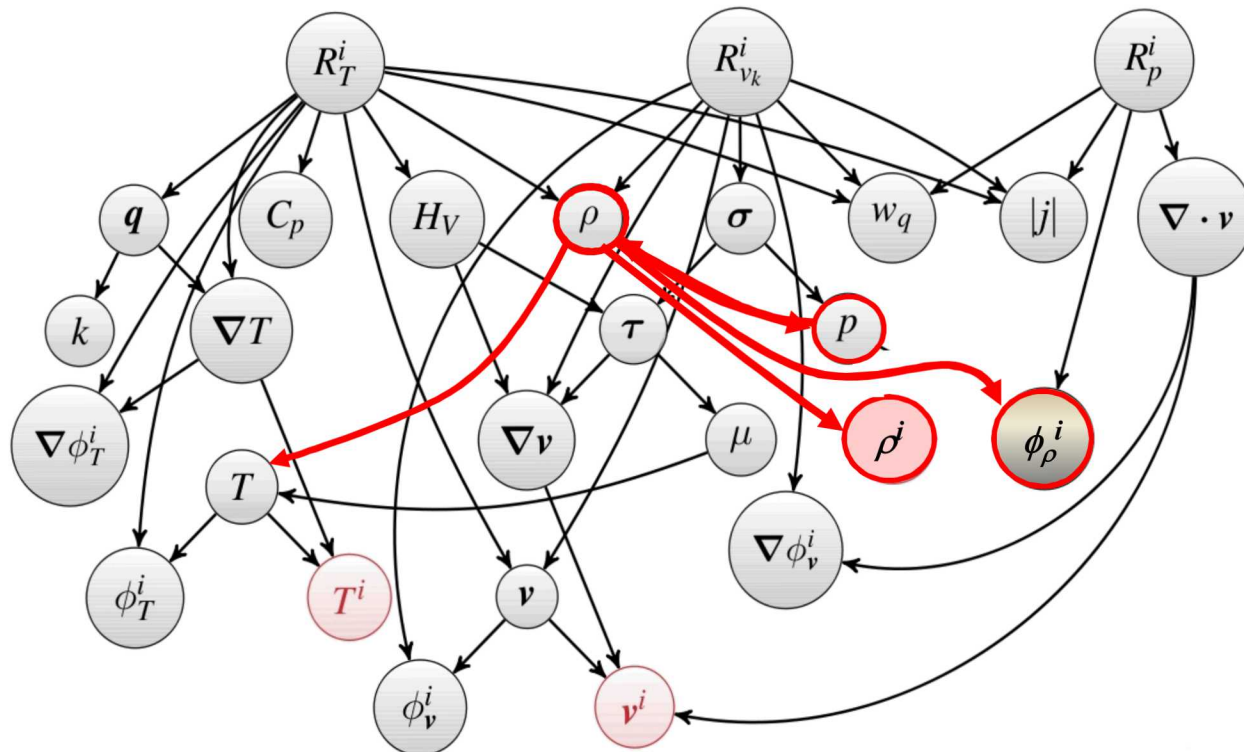


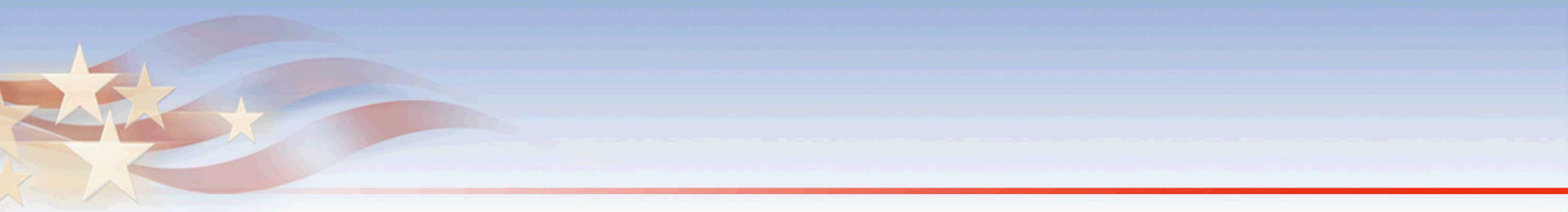
Navier-Stokes Example

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

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

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





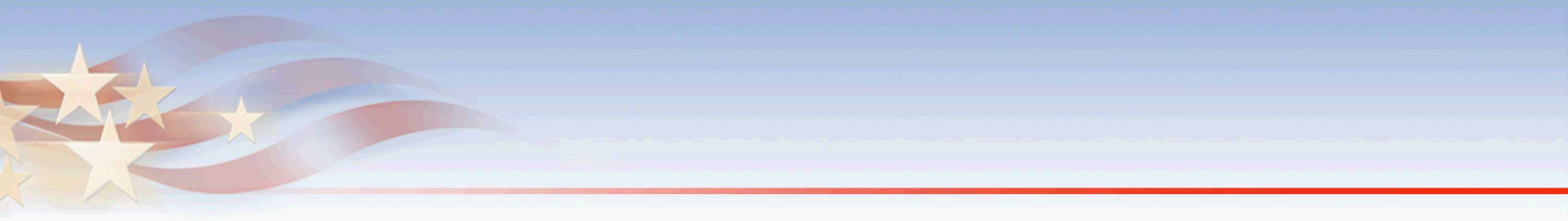
Albany source-code deep-dive

- **Heat equation evaluator setup**
- **Diffusion coefficient**
- **Source term**
- **Gather/scatter**
- **Parameters**
- **SG Preconditioners**



References

- **Use of templates for automatic differentiation in large-scale codes**
 - R. Pawlowski, E. Phipps, and A. Salinger, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part I: template-based generic programming,” *Journal of Scientific Programming*, vol. 20 (2), 2012.
 - R. Pawlowski, E. Phipps, A. Salinger, S. Owen, C. Siefert, and M. Staten, “Automating embedded analysis capabilities and managing software complexity in multiphysics simulation part II: application to partial differential equations,” *Journal of Scientific Programming*, vol. 20 (3), 2012.
- **Phalanx**
 - 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).
- **Dakota/UQ**
 - A. Rushdi, L. Swiler, S. Mitchell, and M. Ebeida, “VPS: Voronoi Piecewise Surrogates for High-Dimensional Data Fitting,” to be submitted to *SIAM/ASA Journal on Uncertainty Quantification*.
 - B.M. Adams, L.E. Bauman, W.J. Bohnhoff, K.R. Dalbey, M.S. Ebeida, J.P. Eddy, P.D. Hough, K.T. Hu, J.D. Jakeman, L.P. Swiler, , J.A. Stephens, D.M. Vigil, and T.M. Wildey, “Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis,” Version 6.2 user’s manual. Sand Report SAND2014-4633, Sandia National Laboratories, May 2014, updated 2015.



Auxiliary Slides



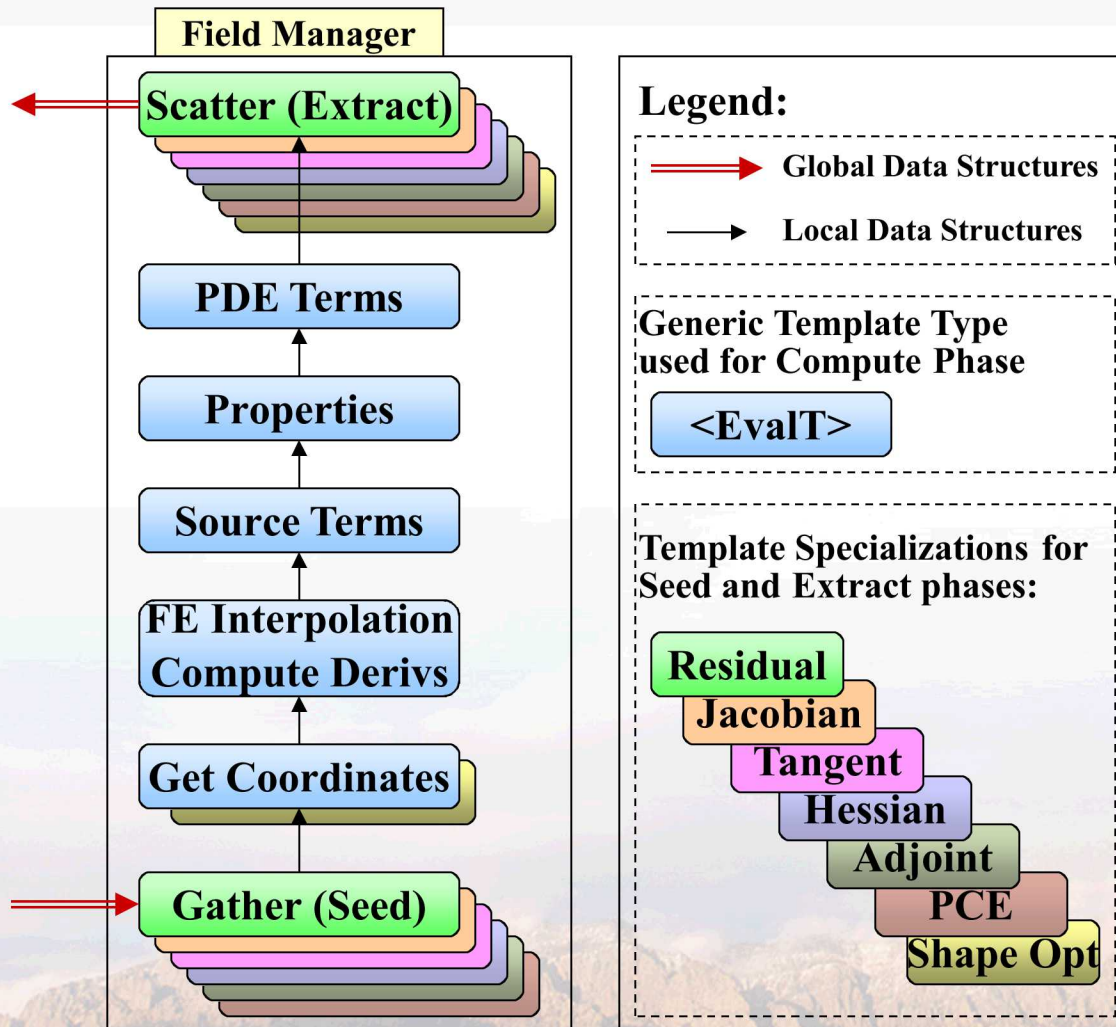
ElasticityResid Evaluator

```
template<typename EvalT, typename Traits>
void ElasticityResid<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
    for (std::size_t cell=0; cell < workset.numCells; ++cell) {
        for (std::size_t node=0; node < numNodes; ++node) {
            for (std::size_t dim=0; dim<numDims; dim++) Residual(cell,node,dim)=0.0;
            for (std::size_t qp=0; qp < numQPs; ++qp) {
                for (std::size_t i=0; i<numDims; i++) {
                    for (std::size_t dim=0; dim<numDims; dim++) {
                        Residual(cell,node,i)
                            += Stress(cell, qp, i, dim) * wGradBF(cell, node, qp, dim);
                    }
                }
            }
        }
    }

    if (workset.transientTerms && enableTransient)
        for (std::size_t cell=0; cell < workset.numCells; ++cell) {
            for (std::size_t node=0; node < numNodes; ++node) {
                for (std::size_t qp=0; qp < numQPs; ++qp) {
                    for (std::size_t i=0; i<numDims; i++) {
                        Residual(cell,node,i)
                            += uDotDot(cell, qp, i) * wBF(cell, node, qp);
                    }
                }
            }
        }
}
```



Template-Based Generic Programming: Codes are born ready for embedded algorithms



ElasticityResid Evaluator

```
template<typename EvalT, typename Traits>
void ElasticityResid<EvalT, Traits>::
evaluateFields(typename Traits::EvalData workset)
{
    for (std::size_t cell=0; cell < workset.numCells; ++cell) {
        for (std::size_t node=0; node < numNodes; ++node) {
            for (std::size_t dim=0; dim<numDims; dim++) Residual(cell,node,dim)=0.0;
            for (std::size_t qp=0; qp < numQPs; ++qp) {
                for (std::size_t i=0; i<numDims; i++) {
                    for (std::size_t dim=0; dim<numDims; dim++) {
                        Residual(cell,node,i)
                            += Stress(cell, qp, i, dim) * wGradBF(cell, node, qp, dim);
                    }
                }
            }
        }
    }

    if (workset.transientTerms && enableTransient)
        for (std::size_t cell=0; cell < workset.numCells; ++cell) {
            for (std::size_t node=0; node < numNodes; ++node) {
                for (std::size_t qp=0; qp < numQPs; ++qp) {
                    for (std::size_t i=0; i<numDims; i++) {
                        Residual(cell,node,i)
                            += uDotDot(cell, qp, i) * wBF(cell, node, qp);
                    }
                }
            }
        }
}
```

```
PHX::MDField<EvalT::ScalarT,Cell,QuadPoint,Dim,Dim> Stress;
```



Conversion of a finite element kernel to Kokkos programming model for portable node-level parallelism

```
template<typename EvalT>
Void VecGrad<EvalT>::evaluateFields()
{
    // Outer loop over a Workset of Elements
    for(int cell = 0; cell < NumCells; cell++) {
        for(int qp = 0; qp < numQPs; qp++) {
            for(int i = 0; i < numVecs; i++){
                for(int dim = 0; dim < numDims; dim++){
                    for(int nd = 0; nd < numNodes; nd++){
                        vecGrad[cell][qp][i][dim] +=
                            vec[cell][nd][i]
                            * basisGrads[cell][nd][qp][dim];
                    }
                }
            }
        }
    } // cell loop
}
```

```
template<typename EvalT>
void VecGrad<EvalT>::evaluateFields()
{
    // Outer loop over a Workset of Elements
    Kokkos::parallel_for (NumCells, *this);
}

*****
template<typename EvalT>
KOKKOS_INLINE_FUNCTION
void VecGrad<EvalT>::operator ()
    (const int cell) const
{
    for(int qp = 0; qp < numQPs; qp++) {
        for(int i = 0; i < numVecs; i++){
            for(int dim = 0; dim < numDims; dim++){
                for(int nd = 0; nd < numNodes; nd++){
                    vecGrad(cell, qp, i, dim) +=
                        vec(cell, nd, i)
                        * basisGrads(cell, nd, qp, dim);
                }
            }
        }
    }
}
```

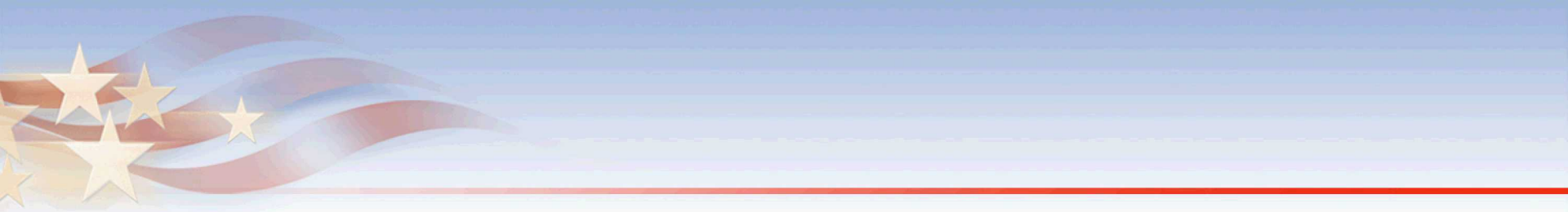
Refactoring follows simple recipe:

- **Outer loop** moved to `parallel_for(int)`
- **Inner kernel** moved to `operator(int)` functor
- **Arrays `a[i][j]`** converted to `Kokkos::Views a(i,j)`



Sacado/Stokhos- and Kokkos- ification of FE assembly

```
typedef Kokkos::OpenMP ExecutionSpace;
//typedef Kokkos::CUDA ExecutionSpace;
//typedef Kokkos::Serial ExecutionSpace;
template<typename ScalarT>
vectorGrad<ScalarT>::vectorGrad()
{
Kokkos::View<ScalarT****, ExecutionSpace> vecGrad(numCells, numQP, numVec, numDim);
}
*****
template<typename ScalarT>
void vectorGrad<ScalarT>::evaluateFields()
{
    Kokkos::parallel_for<ExecutionSpace> (numCells, *this);
}
*****
template<typename ScalarT>
KOKKOS_INLINE_FUNCTION
void vectorGrad<ScalarT>::operator() (const int cell) const
{
    for (int cell = 0; cell < numCells; cell++)
        for (int qp = 0; qp < numQP; qp++) {
            for (int dim = 0; dim < numVec; dim++) {
                for (int i = 0; i < numDim; i++) {
                    for (int nd = 0; nd < numNode; nd++) {
                        vecGrad(cell, qp, dim, i) += val(cell, nd, dim) * basisGrad(nd, qp, i);
                    }
                }
            }
        }
}
```

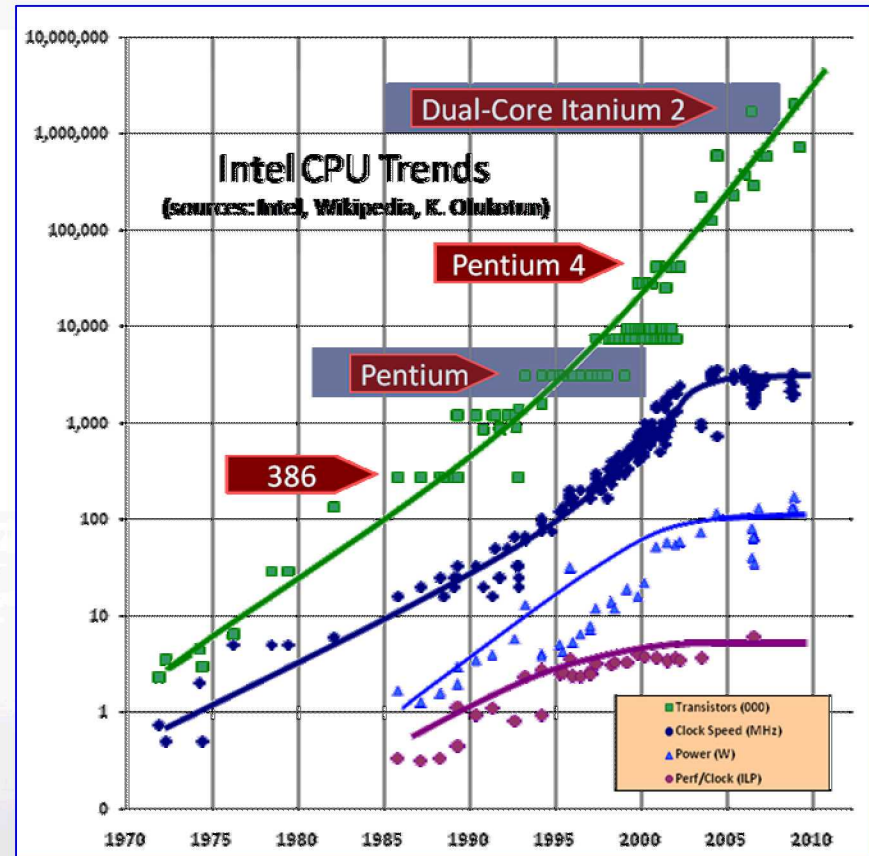


Uncertainty Quantification for Next-Generation Architectures



Computer Architectures Are Changing Dramatically

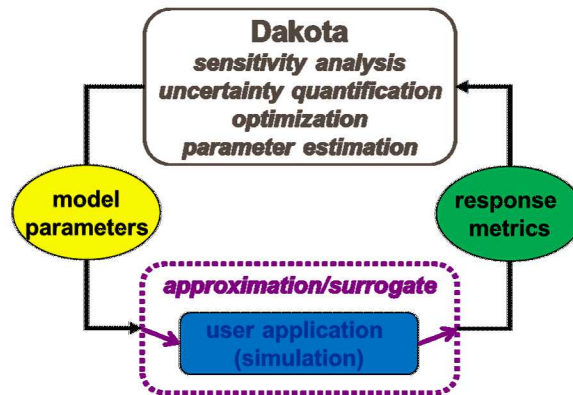
- Historically (super)computers have gotten faster by
 - Increasing clock frequency
 - Adding more compute nodes that communicate through an interconnect
- Power requirements make this approach untenable for future performance increases
- Instead performance increases are now achieved through increases in node-level fine-grained parallelism
 - Many, many threads executing simultaneously
 - Memory access, arithmetic on wide vectors
 - Complex memory hierarchies that require processing units to share data



Herb Sutter, “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”, Dr. Dobbs’s Journal

Emerging Architectures Motivate New Approaches to Predictive Simulation

- UQ approaches traditionally implemented as an outer loop:



<http://dakota.sandia.gov>

- Increasing UQ performance will require
 - Speeding-up each sample evaluation, and/or
 - Evaluating more samples in parallel
- Many important scientific simulations will struggle with upcoming architectures
 - Irregular memory access patterns (e.g., indirect accesses resulting in long latencies)
 - Inconsistent vectorization (e.g., complex loop structures with variable trip-count)
 - Poor scalability to high thread-counts (e.g., poor cache reuse results in ineffective hardware threading)
- Investigate improving performance and scalability through embedded UQ approaches that propagate some UQ information at lowest levels of simulation
 - Improve memory access patterns and cache reuse
 - Expose new dimensions of structured fine-grained parallelism
 - Reduce aggregate communication



Sparse CRS-Format Matrix-Vector Product

```
// CRS Matrix for an arbitrary floating-point type T
template <typename T>
struct CrsMatrix {
    int num_rows;    // number of rows in matrix
    int num_entries; // number of nonzeros in matrix
    int *row_map;    // starting index of each row, [0,num_rows+1)
    int *col_entry;   // column indices for each nonzero, [0,num_entries)
    T *values;       // matrix values of type T, [0,num_entries)
};

// Serial CRS matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```

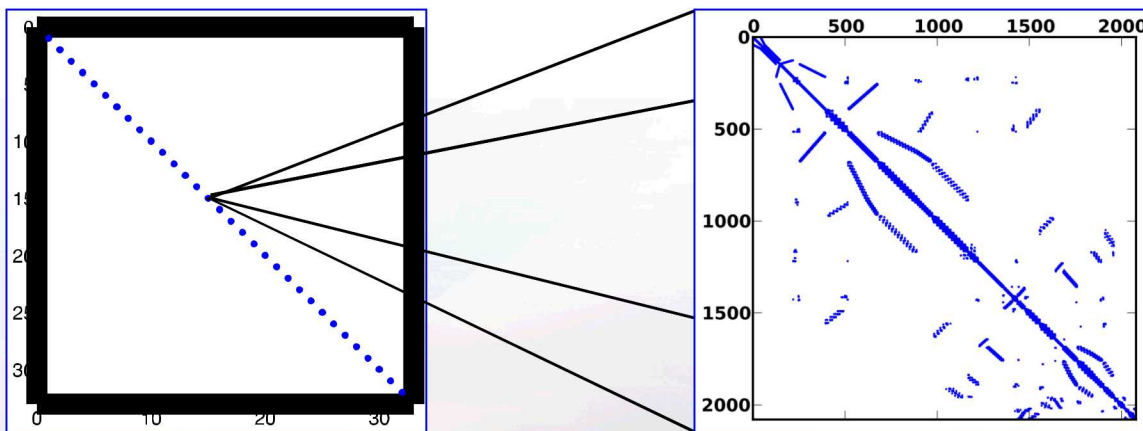

Simultaneous ensemble propagation

- PDE:

$$f(u, y) = 0$$

- Propagating m samples – block diagonal (nonlinear) system:

$$F(U, Y) = 0, \quad U = \sum_{i=1}^m e_i \otimes u_i, \quad Y = \sum_{i=1}^m e_i \otimes y_i, \quad F = \sum_{i=1}^m e_i \otimes f(u_i, y_i), \quad \frac{\partial F}{\partial U} = \sum_{i=1}^m e_i e_i^T \otimes \frac{\partial f}{\partial u_i}$$



- Spatial DOFs for each sample stored consecutively

Ensemble Matrix-Vector Product

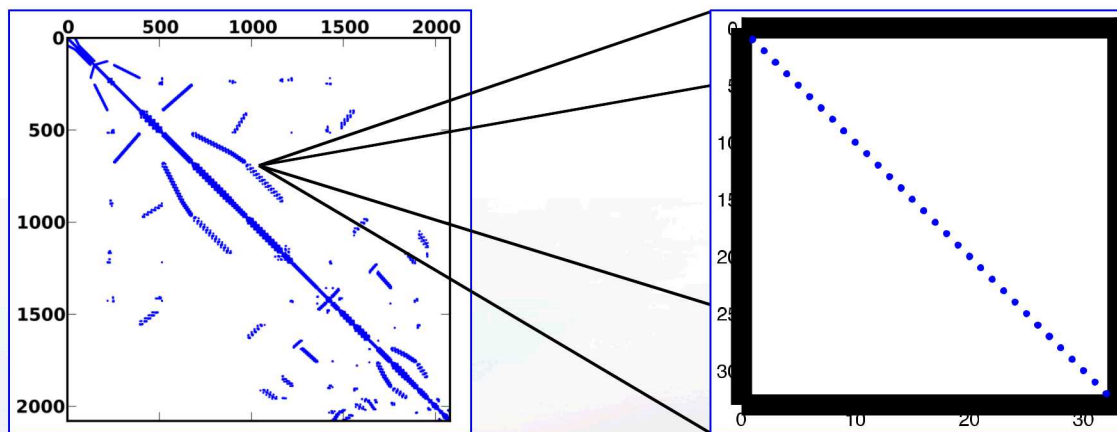
```
// Ensemble matrix-vector product
template <typename T, int m>
void ensemble_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int e=0; e < m; ++e) {
        for (int row=0; i<A.num_rows; ++row) {
            const int entry_begin = A.row_map[row];
            const int entry_end   = A.row_map[row+1];
            T sum = 0.0;
            for (int entry = entry_begin; entry < entry_end; ++entry) {
                const int col = A.col_entry[entry];
                sum += A.values[entry + e*A.num_entries] * x[col + e*A.num_rows];
            }
            y[row + e*A.num_rows] = sum;
        }
    }
}
```



Simultaneous ensemble propagation

- Commute Kronecker products:

$$F_c(U_c, Y_c) = 0, \quad U_c = \sum_{i=1}^m u_i \otimes e_i, \quad Y_c = \sum_{i=1}^m y_i \otimes e_i, \quad F_c = \sum_{i=1}^m f(u_i, y_i) \otimes e_i, \quad \frac{\partial F_c}{\partial U_c} = \sum_{i=1}^m \frac{\partial f}{\partial u_i} \otimes e_i e_i^T$$



- *m* sample values for each DOF stored consecutively

Commutated, Ensemble Matrix-Vector Product

```
// Ensemble matrix-vector product using commuted layout
template <typename T, int m>
void ensemble_commutated_crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; i<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end   = A.row_map[row+1];
        T sum[m];
        for (int e=0; e < m; ++e)
            sum[e] = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            for (int e=0; e < m; ++e) {
                sum[e] += A.values[entry*m + e] * x[col*m + e];
            }
        }
        for (int e=0; e < m; ++e)
            y[row*m + e] = sum[e];
    }
}
```

- Automatically reuse non-sample dependent data
- Sparse access latency amortized across ensemble
- Communication latency amortized across ensemble
- Math on ensemble naturally maps to vector arithmetic



C++ Ensemble Scalar Type

```
// Ensemble scalar type
template <typename U, int m>
struct Ensemble {
    U val[m];
    Ensemble(const U& v) { for (int e=0; e<m; ++e) val[e] = v; }
    Ensemble& operator=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[e] = a.val[e];
        return *this;
    }
    Ensemble& operator+=(const Ensemble& a) {
        for (int e=0; e<m; ++e) val[e] += a.val[e];
        return *this;
    }
    // ...
};

template <typename U, int m>
Ensemble<U,m> operator*(const Ensemble<U,m>& a, const Ensemble<U,m>& b) {
    Ensemble<U,m> c;
    for (int e=0; e<m; ++e) c.val[e] = a.val[e]*b.val[e];
    return c;
}

// ...
```

Ensemble Matrix-Vector Product Through Operator Overloading

- Original matrix-vector product routine, instantiated with $T = \text{Ensemble}<\text{double}, m>$ scalar type:

```
// Serial Crs matrix-vector product for arbitrary floating-point type T
template <typename T>
void crs_mat_vec(const CrsMatrix<T>& A, const T *x, T *y) {
    for (int row=0; row<A.num_rows; ++row) {
        const int entry_begin = A.row_map[row];
        const int entry_end  = A.row_map[row+1];
        T sum = 0.0;
        for (int entry = entry_begin; entry < entry_end; ++entry) {
            const int col = A.col_entry[entry];
            sum += A.values[entry] * x[col];
        }
        y[row] = sum;
    }
}
```



Ensemble Scalar Type Provided by Stokhos

- **Currently called `Sacado::MP::Vector`**

- **Uses expression templates to fuse loops**

$$d = a \times b + c = \{a_1 \times b_1 + c_1, \dots, a_m \times b_m + c_m\}$$

- **Very similar implementation as `Sacado AD` data types**



<http://trilinos.org>

- **Enabled in simulation codes through template-based generic programming**

- **Template C++ code on scalar type**
- **Instantiate template code on ensemble scalar type**

- **Integrated with Kokkos (Edwards, Sunderland, Trott) for many-core parallelism**

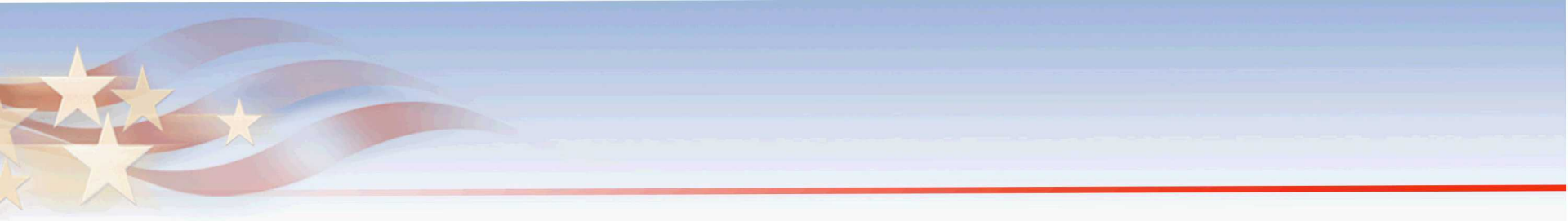
- **Specializes Kokkos data-structures, execution policies to map vectorization parallelism across ensemble**

- **Integrated with Tpetra-based solvers for hybrid (MPI+X) parallel linear algebra**

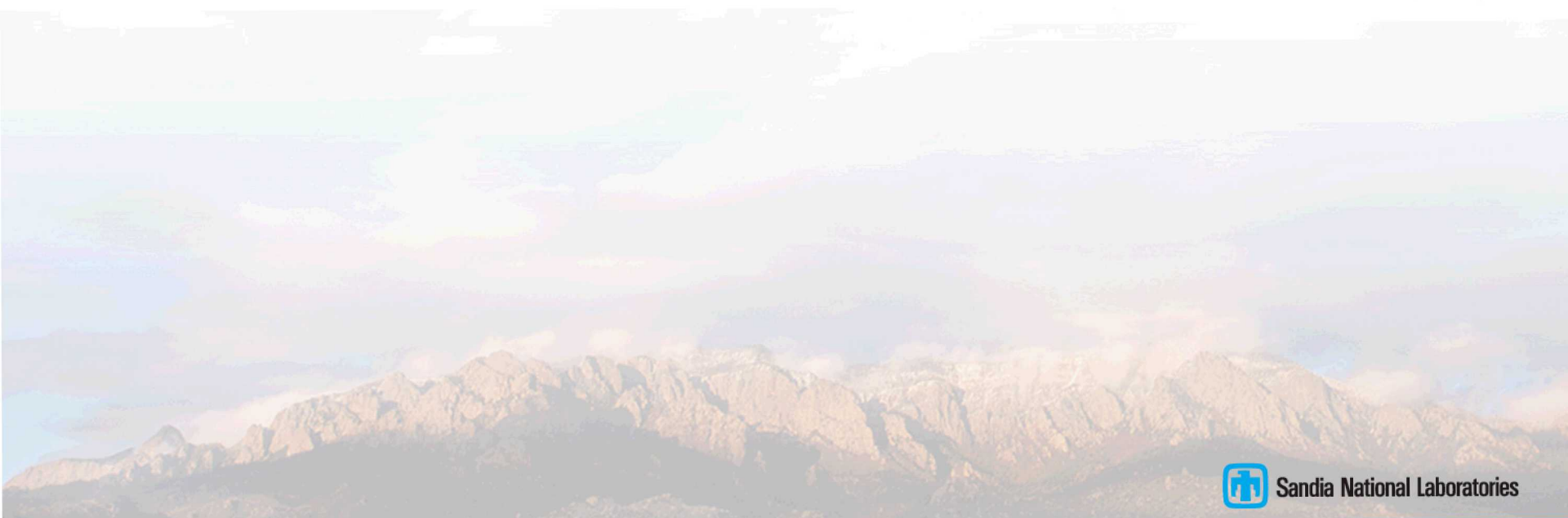
- **Exploits templating on scalar type**
- **Krylov solvers (Belos)**
- **Algebraic multigrid preconditioners (MueLu)**
- **Incomplete factorization, polynomial, and relaxation-based preconditioners/smoothers (Ifpack2)**
- **Sparse-direct solvers (Amesos2)**



Sandia National Laboratories



MP::Vector Example



Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications*

- **Standard C++ library, not a language extension**
 - **Core:** multidimensional arrays, parallel execution, atomic operations
 - **Containers:** Thread-scalable implementations of common data structures (vector, map, CRS graph, ...)
 - **LinAlg:** Sparse matrix/vector linear algebra
- **Relies heavily on C++ template meta-programming to introduce abstraction without performance penalty**
 - **Execution spaces** (CPU, GPU, ...)
 - **Memory spaces** (Host memory, GPU memory, scratch-pad, texture cache, ...)
 - **Layout of multidimensional data in memory**
 - **Scalar type**



<http://trilinos.org>

*H.C. Edwards, D. Sunderland, C. Trott (SNL)

Application & Library Domain Layer

Kokkos Sparse Linear Algebra

Kokkos Containers

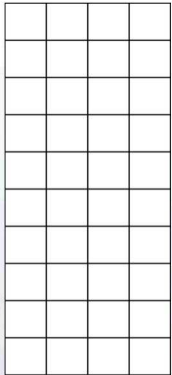
Kokkos Core

Back-ends: OpenMP, pthreads, Cuda, vendor libraries ...

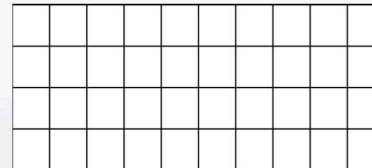
Kokkos Integration

- Kokkos views of UQ scalar type internally stored as views of 1-higher rank
 - UQ dimension is always contiguous, regardless of layout
- Facilitates
 - Fine-grained parallelism over UQ dimension
 - Efficient allocation and initialization
 - Specialization of kernels
 - Transferring data between host and device and MPI communication

```
Kokkos::View< Ensemble<double,4>*, LayoutRight, Device > view("v", 10);
```



```
Kokkos::View< Ensemble<double,4>*, LayoutLeft, Device > view("v", 10);
```



- Requires specialized kernel launch for CUDA to map warp to UQ dimension to achieve performance



Tpetra: Foundational Layer / Library for Sparse Linear Algebra Solvers on Next-Generation Architectures*

- Tpetra: Sandia's templated C++ library for distributed memory (MPI) sparse linear algebra
 - Epetra but templated
 - Scalar, local-ordinal, global ordinal, node
 - Builds distributed memory linear algebra on top of Kokkos library
 - Distributed memory vectors, multi-vectors, and sparse matrices
 - Data distribution maps and communication operations
 - Fundamental computations: axpy, dot, norm, matrix-vector multiply, ...
 - Templated on “scalar” type: float, double, automatic differentiation, polynomial chaos, ensembles, ...
- Higher level solver libraries built on Tpetra
 - Preconditioned iterative algorithms (Belos)
 - Incomplete factorization preconditioners (Ifpack2, ShyLU)
 - Multigrid solvers (MueLu)
 - All templated on the scalar type



<http://trilinos.org>

*M. Heroux, M. Hoemmen, et al (SNL)



Sandia National Laboratories

Techniques Prototyped in FENL Mini-App



<http://trilinos.org>

- Simple nonlinear diffusion equation

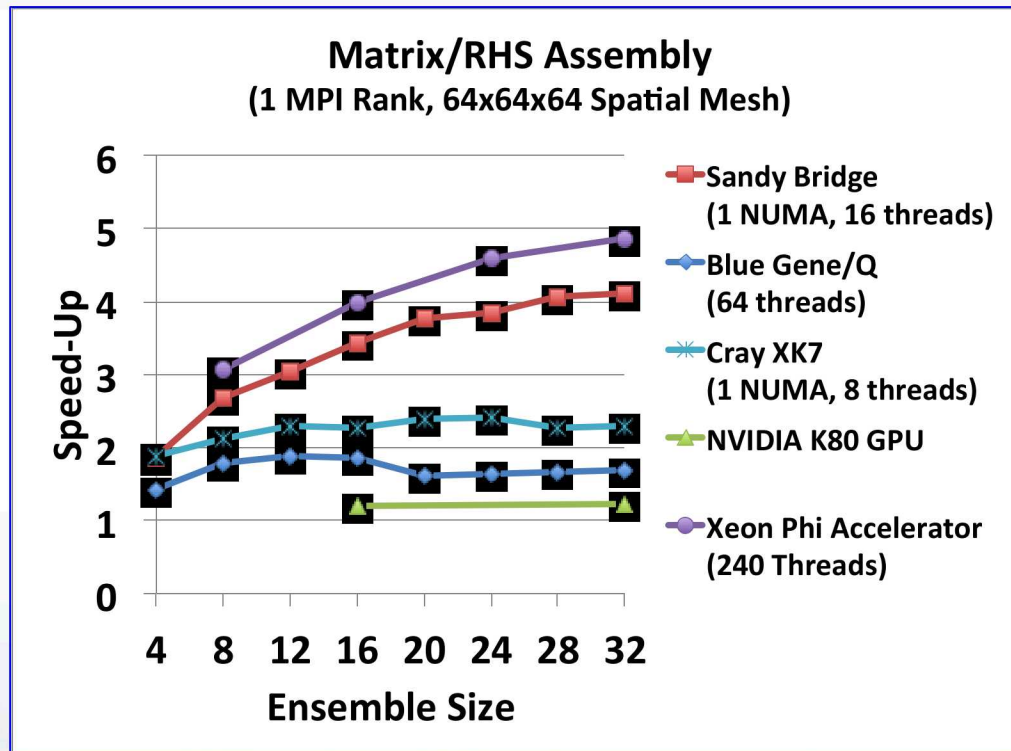
$$-\nabla \cdot (\kappa(x, y) \nabla u) + u^2 = 0$$

- 3-D, linear FEM discretization
 - 1x1x1 cube, unstructured mesh
 - KL truncation of exponential random field model for diffusion coefficient
 - Trilinos-couplings package
- Hybrid MPI+X parallelism
 - Traditional MPI domain decomposition using threads within each domain
 - Employs Kokkos for thread-scalable
 - Graph construction
 - PDE matrix/RHS assembly
 - Employs Tpetra for distributed linear algebra
 - CG iterative solver (Belos package)
 - Smoothed Aggregation AMG preconditioning (MueLu)
 - Supports embedded ensemble propagation via Stokhos through entire assembly and solve
 - Samples generated via Smolyak sparse grids



Sandia National Laboratories

Ensemble PDE Matrix/RHS Assembly Speed-Up

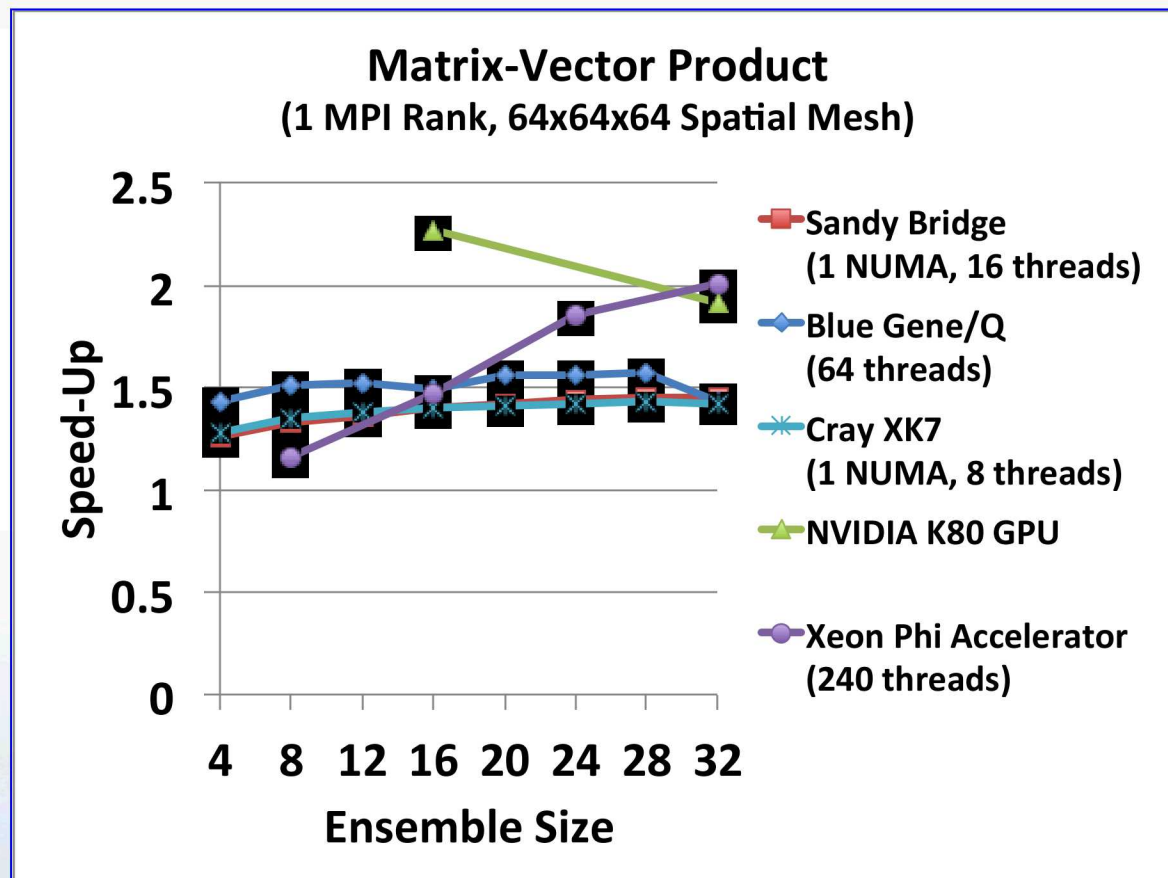


- Speed-up results from
 - Reuse of mesh, discretization data structures
 - Replacement of sparse gather with contiguous load
 - Perfect vectorization of math

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$



Ensemble Sparse Matrix-Vector Product Speed-Up

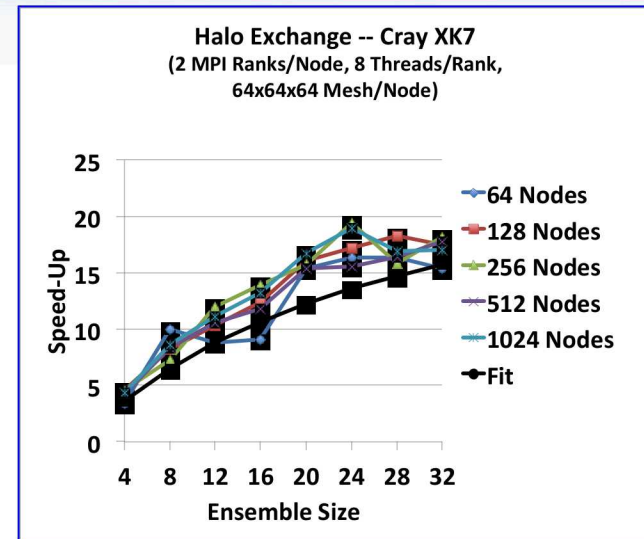
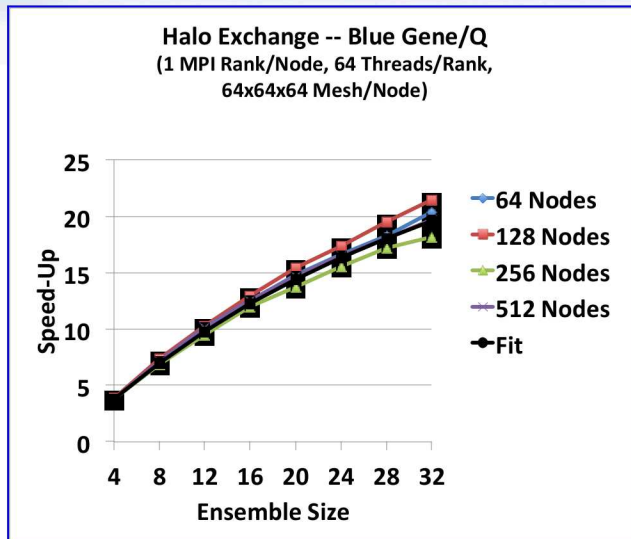


- Speed-up results from
 - Reuse of matrix graph (20%)
 - Replacement of sparse gather with contiguous load
 - Perfect vectorization of multiply-add

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$



Interprocessor Halo Exchange

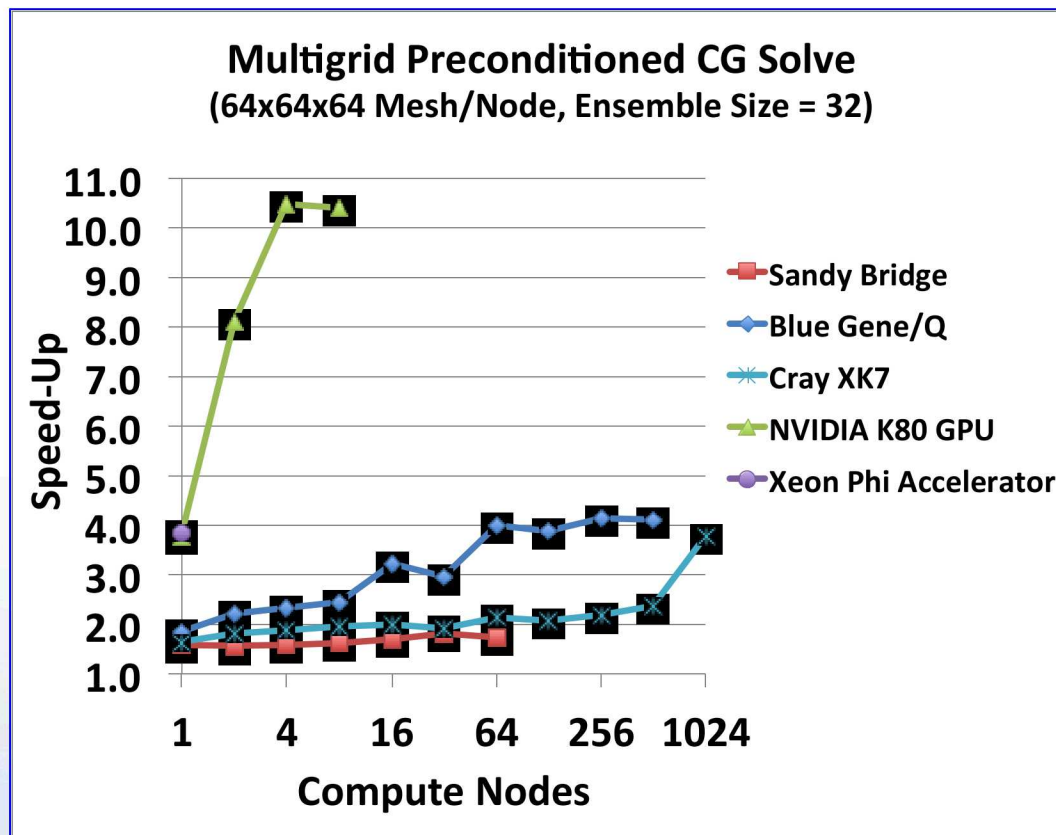


$$\text{Time} \approx a + bm$$

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$
$$\approx \frac{m(a + b)}{a + bm}$$

- **Speed-up results from reduced aggregate communication latency**
 - Fewer, larger MPI messages
 - Communication volume is the same

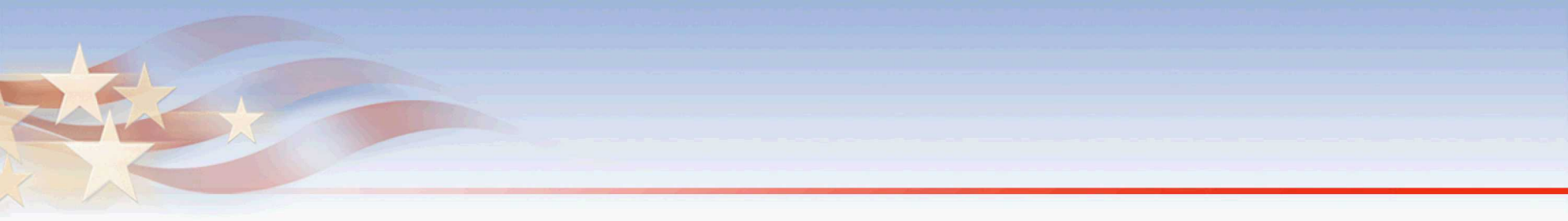
AMG Preconditioned CG Solve



- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
 - Chebyshev smoothers
 - Sparse-direct coarse-grid solver (Amesos2/Basker)
 - Multi-jagged parallel repartitioning (Zoltan2)

$$\text{Speed-Up} = \frac{\text{Ensemble size} \times \text{Time for single sample}}{\text{Time for ensemble}}$$





FENL Ensemble UQ Example



Ensemble Propagation for More Challenging Problems

- **Assuming number of CG iterations doesn't vary significantly from sample to sample**
 - True for problems with tame diffusion coefficient on regular meshes
 - Implies number of CG iterations for ensemble does not increase
- **For general problems, number of iterations will increase for ensemble system**
 - Spectrum of ensemble matrix must spread out
 - Need to group samples to group matrices with similar spectra
- **Note: Do not require smoothness (of matrix, RHS, solution) between samples!**



Summary

- **Embedded sampling approach improves aggregate UQ performance by**
 - **Eliminating sparse memory accesses**
 - **Amortizing communication/access latency**
 - **Perfect fine-grained vector/Cuda-thread parallelism**
- **Embedded sampling approach does not**
 - **Substantially change floating-point operation to memory access ratios**
 - **Increase cache reuse**
 - **Reduce communication volume**
- **To achieve this, we need some form of compression of stochastic information**
 - **Trade reduced stochastic DOFs for increased FLOPs**



Embedded Stochastic Galerkin UQ Methods

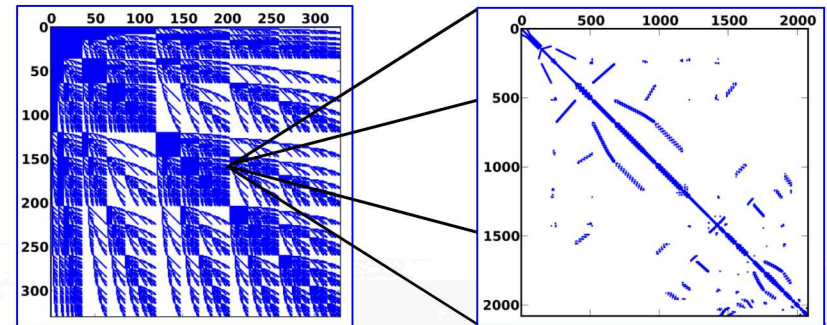
- Stochastic Galerkin method (Ghanem and many, many others...):

$$\hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi) \rightarrow f_i(u_0, \dots, u_P) \equiv \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy = 0, \quad i = 0, \dots, P$$

- Method generates new coupled spatial-stochastic nonlinear problem (intrusive)

$$F(U) = 0, \quad U = \sum_{i=1}^P e_i \otimes u_i, \quad F = \sum_{i=1}^P e_i \otimes f_i$$

$$\frac{\partial F}{\partial U} \approx \sum_{k=0}^P G_k \otimes A_k, \quad G_k(i, j) \equiv C_{ijk} \equiv \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}$$



Stochastic sparsity

Spatial sparsity

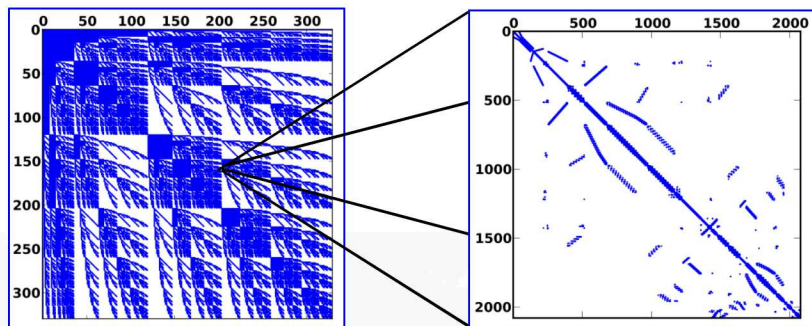
- Many fewer stochastic degrees-of-freedom for comparable level of accuracy:

N = 3			N = 5		
M	P+1	Q+1	M	P+1	Q+1
1	4	5	1	6	7
3	20	39	3	56	153
5	56	151	5	252	933
7	120	407	7	792	3697
9	220	871	9	2002	11581

Commutated SG Structure for Emerging Architectures

- DOF layout can be reorganized in similar manner to embedded sampling:
 - Store PC coefficients for each spatial DOF consecutively

$$A^{trad} = \sum_{k=0}^P G_k \otimes A_k$$

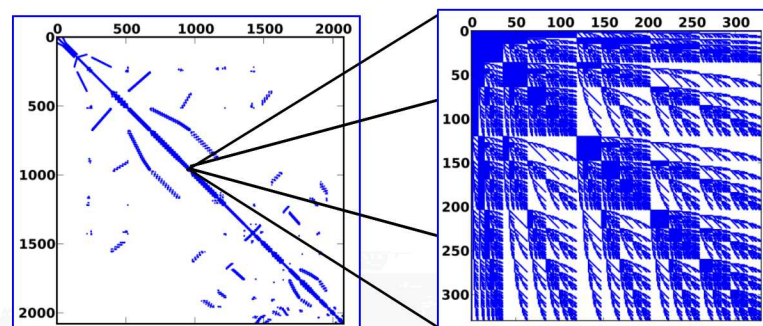


Stochastic sparsity

Spatial sparsity



$$A^{com} = \sum_{k=0}^P A_k \otimes G_k$$



Spatial sparsity

Stochastic sparsity

- Implemented in same manner as embedded sample propagation
 - Scalars replace by PC coefficient arrays
 - Apply operator overloading approach from part 2 to linear algebra
 - `Stokhos::OrthogPoly<Epetra_CrsMatrix>` replaced `Tpetra::CrsMatrix<Sacado::UQ::PCE<>,...>`
 - Approach implemented within Stokhos package

Commutated SG Matrix-Vector Multiply

$$Y^{com} = A^{com} X^{com} \implies \sum_{i=0}^P y_i \otimes e_i = \left(\sum_{k=0}^P A_k \otimes G_k \right) \left(\sum_{j=0}^P x_j \otimes e_j \right)$$

- **Two level algorithm**

- **Outer: sparse (CRS) matrix-vector multiply algorithm**
- **Inner: sparse stochastic Galerkin product**

$$\aleph_A(l) = \{m \mid A_0(l, m) \neq 0\} \quad \aleph_C(i) = \{(j, k) \mid C(i, j, k) \neq 0\}$$

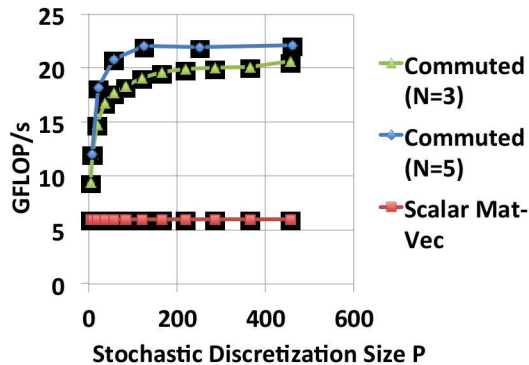
$$y(i, l) = \sum_{m \in \aleph_A(l)} \sum_{(j, k) \in \aleph_C(i)} A(k, l, m) x(j, m) C(i, j, k)$$

Diagram illustrating the components of the equation:

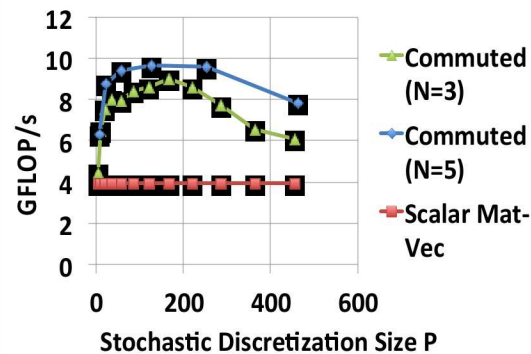
- stochastic basis** (blue box) points to $y(i, l)$
- FEM basis** (green box) points to $y(i, l)$
- stochastic bases sum** (blue box) points to the sum over $m \in \aleph_A(l)$
- FEM bases sum** (green box) points to the sum over $m \in \aleph_A(l)$
- stochastic basis** (blue box) points to $A(k, l, m)$
- FEM basis** (green box) points to $A(k, l, m)$
- stochastic basis** (blue box) points to $x(j, m)$
- FEM basis** (green box) points to $x(j, m)$
- triple product** (blue box) points to $C(i, j, k)$

Sparse Matrix-Vector Product*

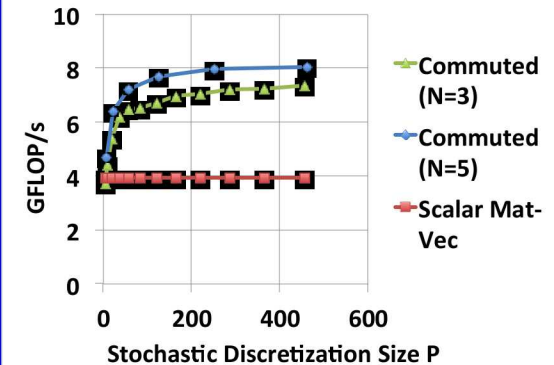
Intel Sandy Bridge CPU
(n=262k, 8 threads)



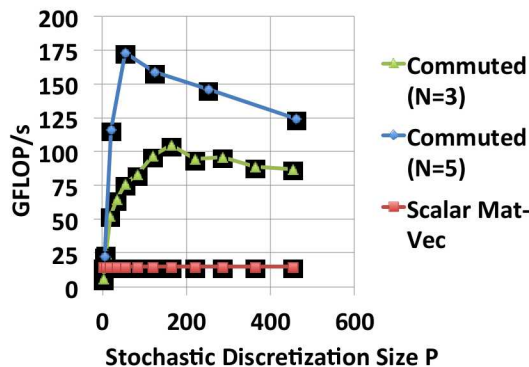
Blue Gene Q CPU
(n=32k, 64 threads)



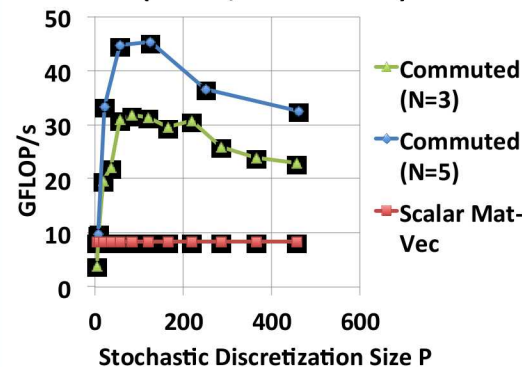
AMD Interlagos CPU
(n=32k, 8 threads)



Nvidia Kepler K80 GPU
(n=32k)



Xeon Phi 7120P Accelerator
(n=32k, 240 threads)

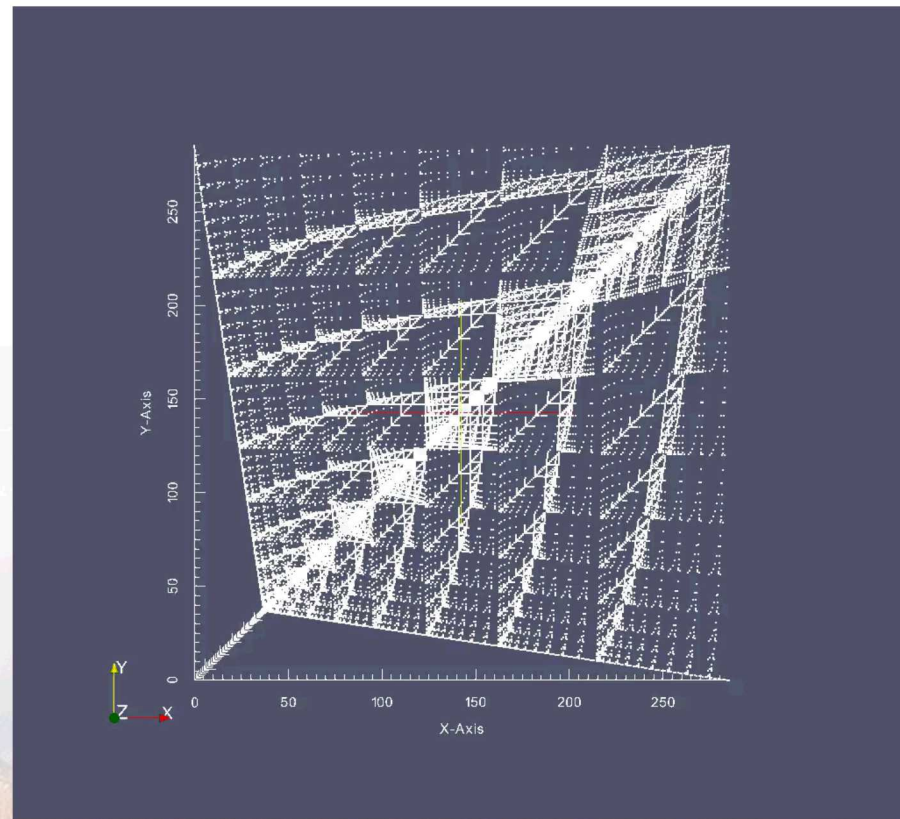


- Increased throughput arises from substantial reuse within PCE multiply

Performance driven by C(i,j,k) tensor

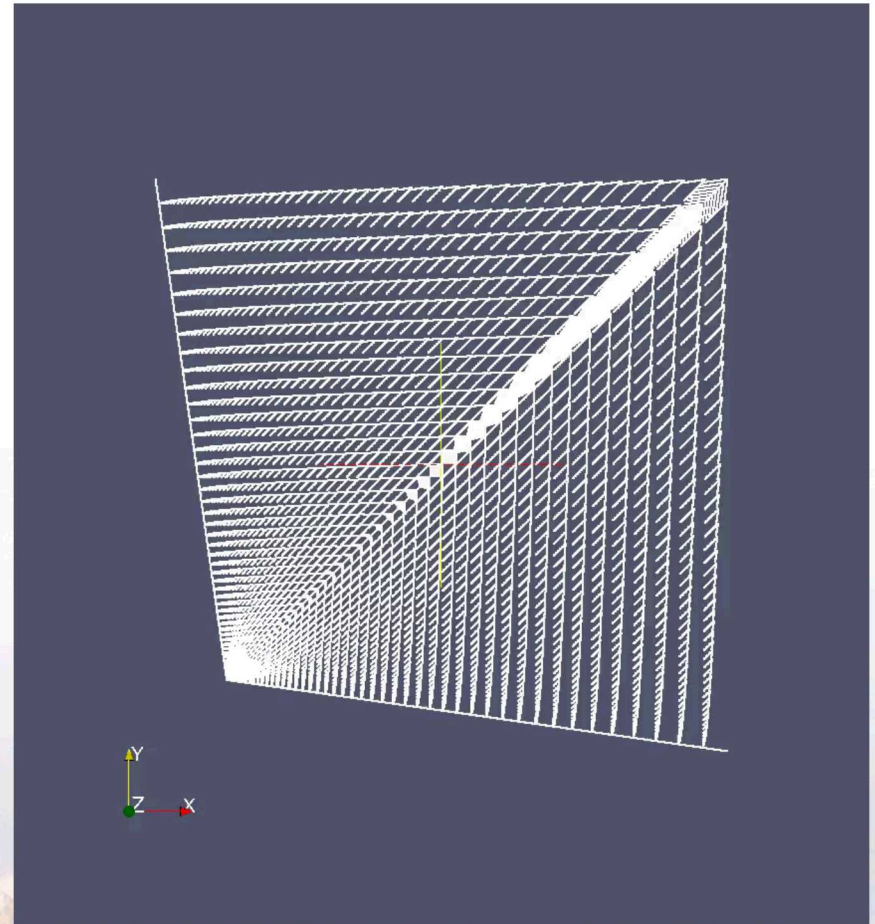
$$y(i, l) = \sum_{m \in \mathfrak{N}_A(l)} \sum_{(j,k) \in \mathfrak{N}_C(i)} A(k, l, m) x(j, m) C(i, j, k)$$

- Precompute and store C
- Given l,m, load A(:,l,m), y(:,l), x(:,m) into cache
- Iterate over non-zero C(i,j,k) entries
- Sparse accesses of A, x, but in fast cache
 - Very fast for GPU
- Lots of reuse of A, x entries
 - Effectiveness determined by ordering of nonzeros in C(i,j,k)
- Can load A, x for multiple values of l,m to reduce reads of C



Traded one bandwidth limit for another

- **NVIDIA GPU, Intel accelerator**
 - Performance no longer driven by bandwidth of reading matrix, vector entries
 - Instead limited by bandwidth of reading sparse tensor
- **Can we remedy this?**
 - Generate $C(i,j,k)$ entries on-the-fly
 - Possible for general PC discretizations, but difficult to do efficiently
 - Might be possible for quadratic basis



Stochastic Galerkin Preconditioning

- Preconditioning stochastic Galerkin system is a significant challenge
- Common approach is mean-based preconditioning:

$$(A^{com})^{-1} \approx M_{mean}^{com} = M_0 \otimes I_P, \quad M_0 \approx A_0^{-1}$$

- Applying mean preconditioner in commuted layout is very efficient:

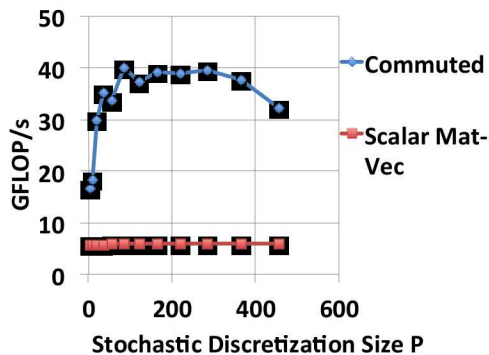
$$\begin{aligned} Y^{com} = M_{mean}^{com} X^{com} &\implies \sum_{i=0}^P y_i \otimes e_i = \left(M_0 \otimes I_P \right) \left(\sum_{j=0}^P x_j \otimes e_j \right) \\ &\implies [y_0, \dots, y_P] = M_0 [x_0, \dots, x_P] \end{aligned}$$

- **Matrix-times-multivector with row-wise layout**
 - **Vectorize over multivector columns**
 - **Reuse of matrix/graph entries**
- Applying preconditioner is often dominant cost

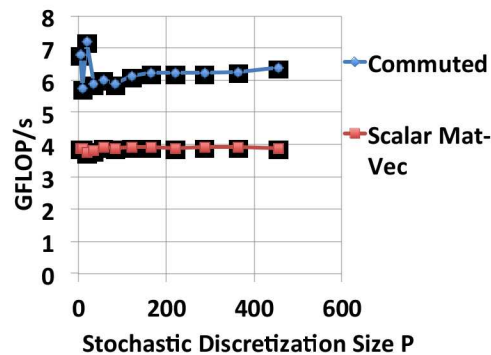


Mean Matrix-Vector Multiply

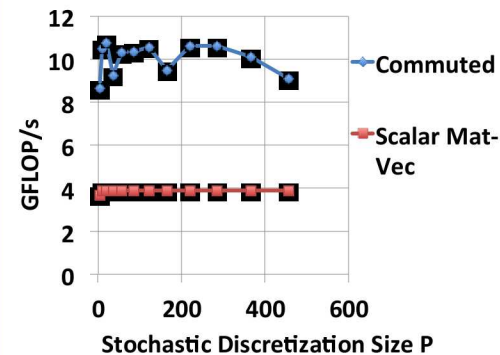
Intel Sandy Bridge CPU
(n=262k, 8 threads)



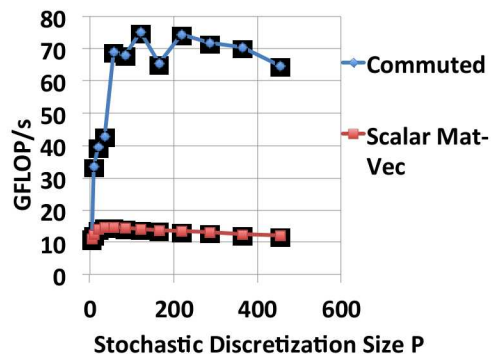
Blue Gene Q CPU
(n=32k, 64 threads)



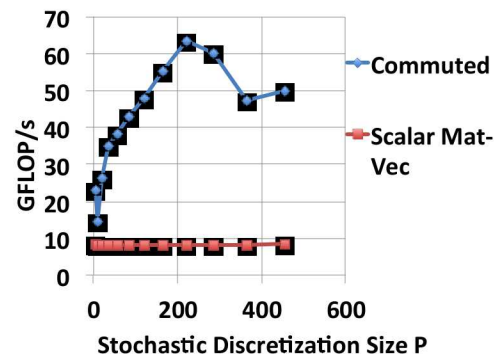
AMD Interlagos CPU
(n=32k, 8 threads)



Nvidia Kepler K80 GPU
(n=32k)



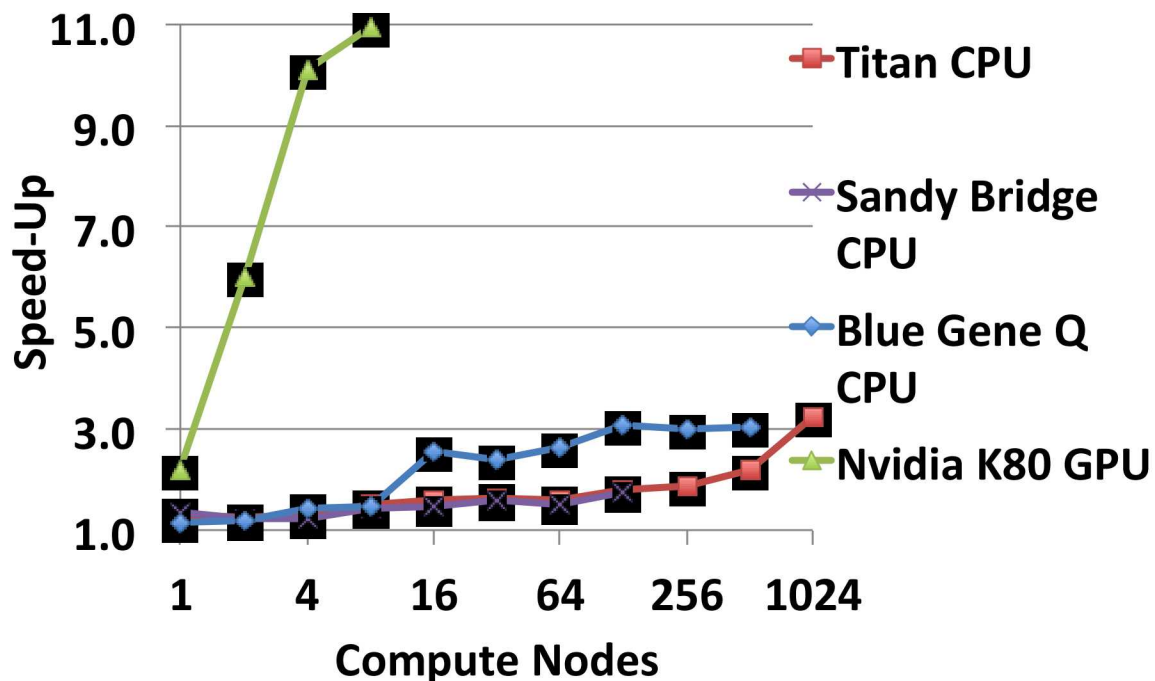
Xeon Phi 7120P Accelerator
(n=32k, 240 threads)



AMG Preconditioned CG Solve

Stochastic Galerkin CG-AMG Solve Speed-Up Over Non-intrusive Polynomial Chaos Sampling

64x64x64 Mesh/Node, $M = 5$, $N = 3$



- Speed-up arises from:

- Increased floating-point throughput
- Reduced preconditioner applies
- Reduced aggregate communication volume

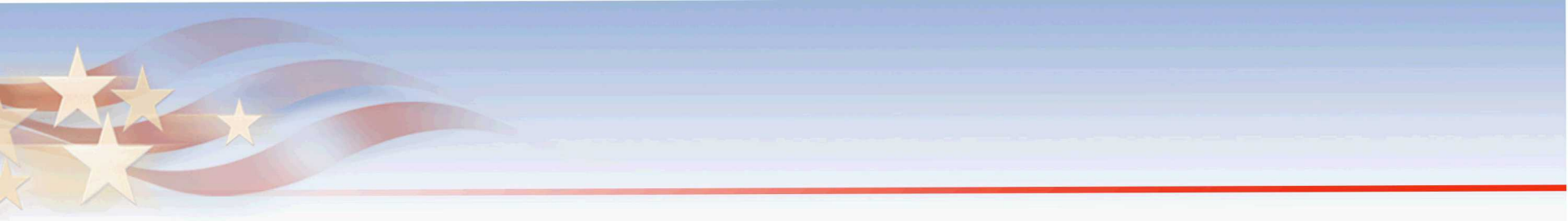


Embedded Ensemble Scalar Type for PDE “Assembly”

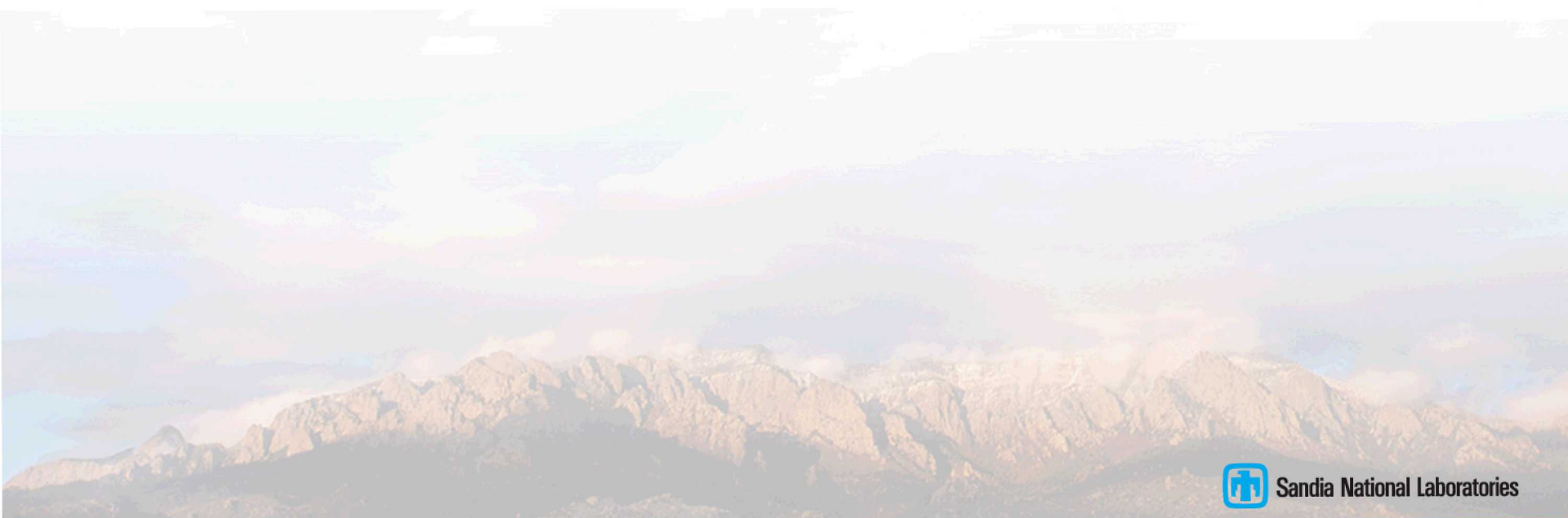
- Operator overloading approach meant to mitigate challenges with SG PDE residual/Jacobian assembly
- Difficult in multicore environment
 - Wide range in UQ problem dimension suggests “dynamic” memory allocation approach
 - But new/delete, malloc/free introduces global thread synchronization cost
 - Very large overhead
- For general nonlinear problems, found a “pseudospectral” approach most-effective:

$$F_i = \int_{\Gamma} f(\hat{u}(y), y) \psi_i(y) \rho(y) dy \approx \sum_{k=0}^P w_k f(\hat{u}(y_k), y_k) \psi_i(y_k)$$

- Sparse-grid quadrature on residual/Jacobian (either local element or global)
- Requires only two additional assembly kernels: PCE evaluation and quadrature
- Use ensemble scalar type for evaluating residual/Jacobian at multiple quadrature points simultaneously



FENL PCE UQ Example





Concluding Remarks

- Investigated reordering of UQ algorithms to propagate some UQ information at lowest levels
 - Trade coarse-grained for fine-grained UQ parallelism
 - Alleviate burden of deterministic simulation code from exploiting all fine-grained parallelism
- Propagating ensembles of samples at lowest level of simulation leads to improved aggregate UQ performance
 - Eliminate sparse memory accesses
 - Amortize communication/access latency
 - Perfect fine-grained vector/Cuda-thread parallelism
- Embedded stochastic Galerkin additionally
 - Introduces more cache reuse
 - Reduction in communication volume
 - At the expense of increased mat-vec FLOPs (which eventually dominate)
- Applying technique through C++ templates greatly facilitates implementation





Challenges/Future Work

- **Ensemble approach:**
 - Effective grouping of samples in ensembles for non-smooth, less-smooth problems
 - Dealing with code divergence (e.g., conditionals)
- **Stochastic Galerkin approach**
 - Partitioning/adapting PC basis to improve scalability, reduce memory burden
 - Low-order spectral DG over random variable space
 - Building PC basis within Voronoi cells
 - Ordering of Cijk tensor, generating on-the-fly to eliminate bandwidth limitations
 - Improved pseudospectral assembly kernels
 - Incorporating SPAM/Smolyak
 - Commuted-layout preconditioning kernels



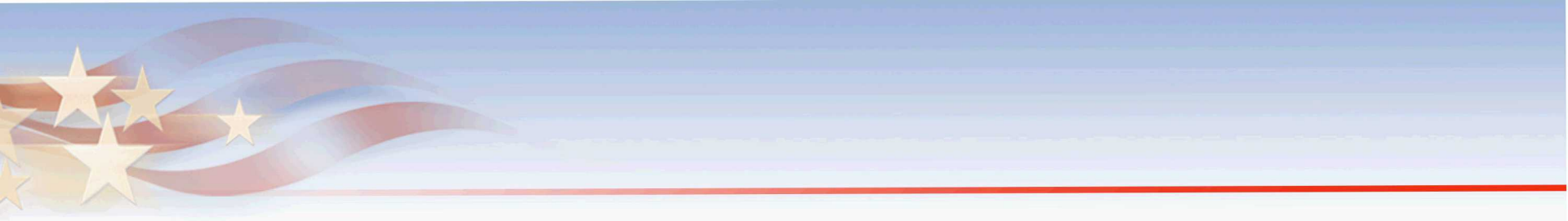
References

- **UQ for multicore architectures**

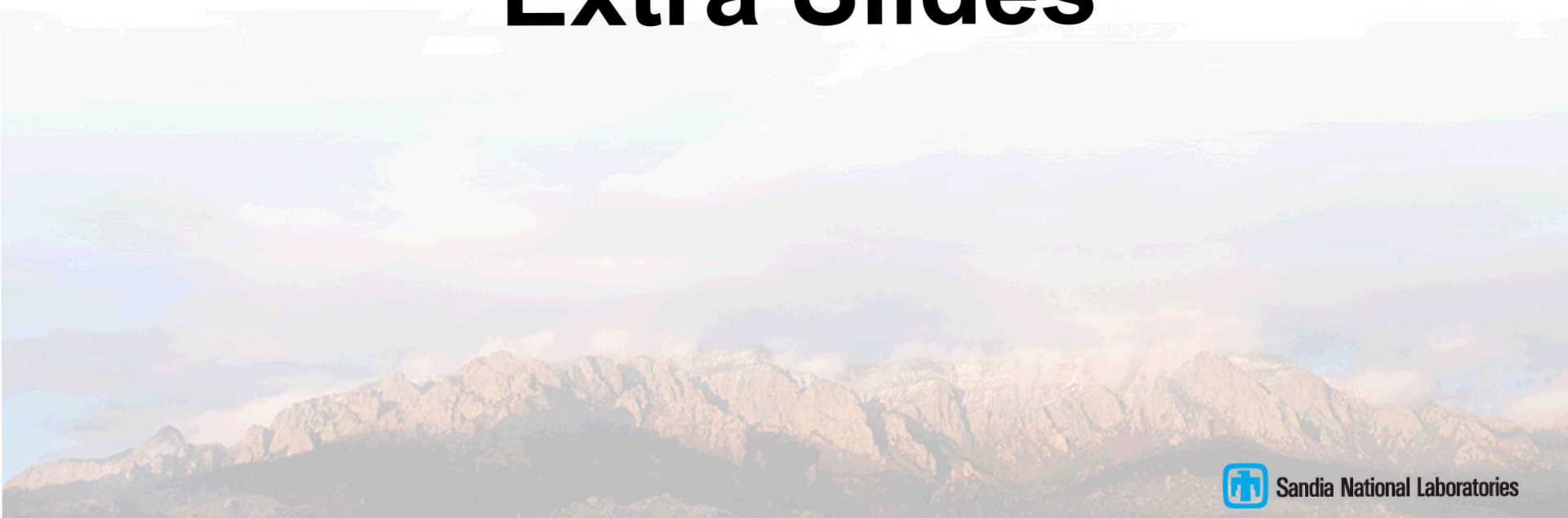
- E. Phipps, H.C. Edwards, J. Hu, and J. Ostien, “Exploring emerging manycore architectures for uncertainty quantification through embedded stochastic Galerkin methods,” *IJCM*, vol. 91, no. 4, 2013.
- E. Phipps, H.C. Edwards, J. Hu, “Exploring Heterogeneous Multicore Architectures for Advanced Embedded Uncertainty Quantification,” Sandia Technical Report, SAND2014-17875, Sandia National Laboratories, 2014.
- E. Phipps, M. D’Elia, H.C. Edwards, M. Hoemmen, J. Hu, and S. Rajamanaickam, “Embedded Ensemble Propagation for Improving Performance, Portability and Scalability of Uncertainty Quantification on Emerging Computational Architectures,” to be submitted to *ACM TOMS*, 2015 (preprint available on request).

- **Next-generation Trilinos packages**

- H. C. Edwards, D. Sunderland, V. Porter, C. Amsler, and S. Mish, “Manycore performance-portability: Kokkos multidimensional array library,” *Scientific Programming*, vol. 20, 2012.
- C. G. Baker and M. A. Heroux, “Tpetra, and the use of generic programming in scientific computing,” *Scientific Programming*, vol. 20, 2012.
- H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, 2014.
- P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, and S. Kennon, “Towards extreme-scale simulations with second-generation Trilinos,” *Parallel Processing Letters*, 2014.



Extra Slides



Polynomial Chaos Expansions (PCE)

- **Steady-state finite dimensional model problem:**

Find $u(\xi)$ such that $f(u, \xi) = 0$, $\xi : \Omega \rightarrow \Gamma \subset R^M$, density ρ

- **(Global) Polynomial Chaos approximation:**

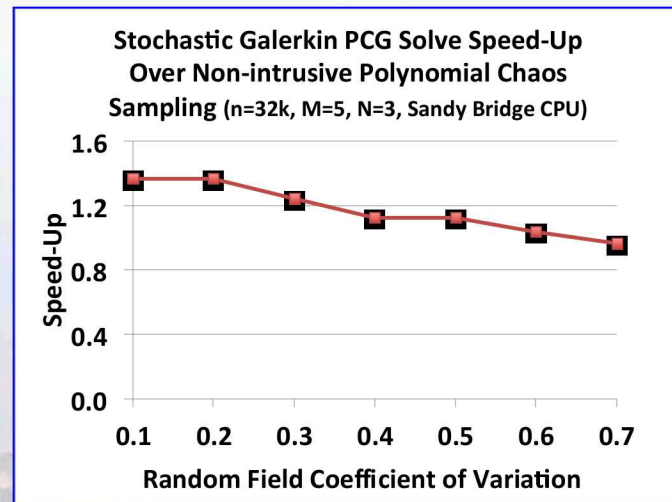
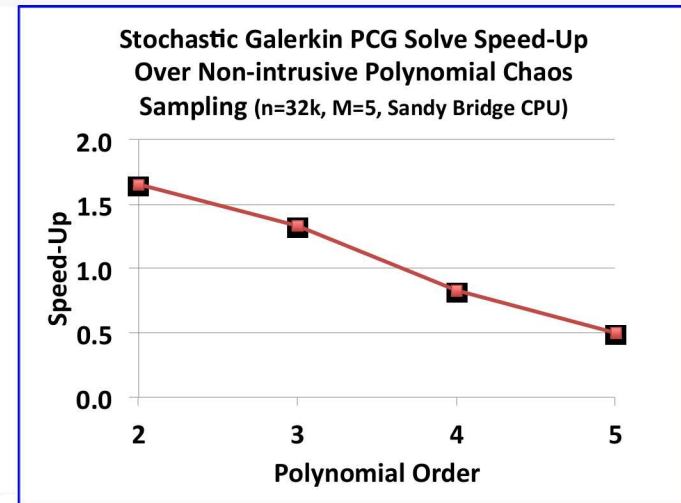
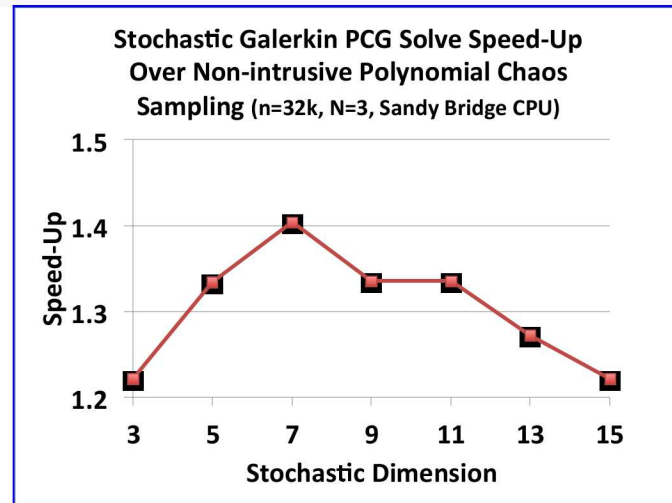
$$u(\xi) \approx \hat{u}(\xi) = \sum_{i=0}^P u_i \psi_i(\xi), \quad \langle \psi_i \psi_j \rangle \equiv \int_{\Gamma} \psi_i(y) \psi_j(y) \rho(y) dy = \delta_{ij} \langle \psi_i^2 \rangle$$

- **Multivariate orthogonal polynomials**
 - **Typically constructed as tensor products with total order at most N**
 - **Can be adapted (anisotropic, local support)**
- **Non-intrusive polynomial chaos (NIPC, NISP):**

$$u_i = \frac{1}{\langle \psi_i^2 \rangle} \int_{\Gamma} \hat{u}(y) \psi_i(y) \rho(y) dy \approx \frac{1}{\langle \psi_i^2 \rangle} \sum_{k=0}^Q w_k u^k \psi_i(y^k), \quad f(u^k, y^k) = 0$$

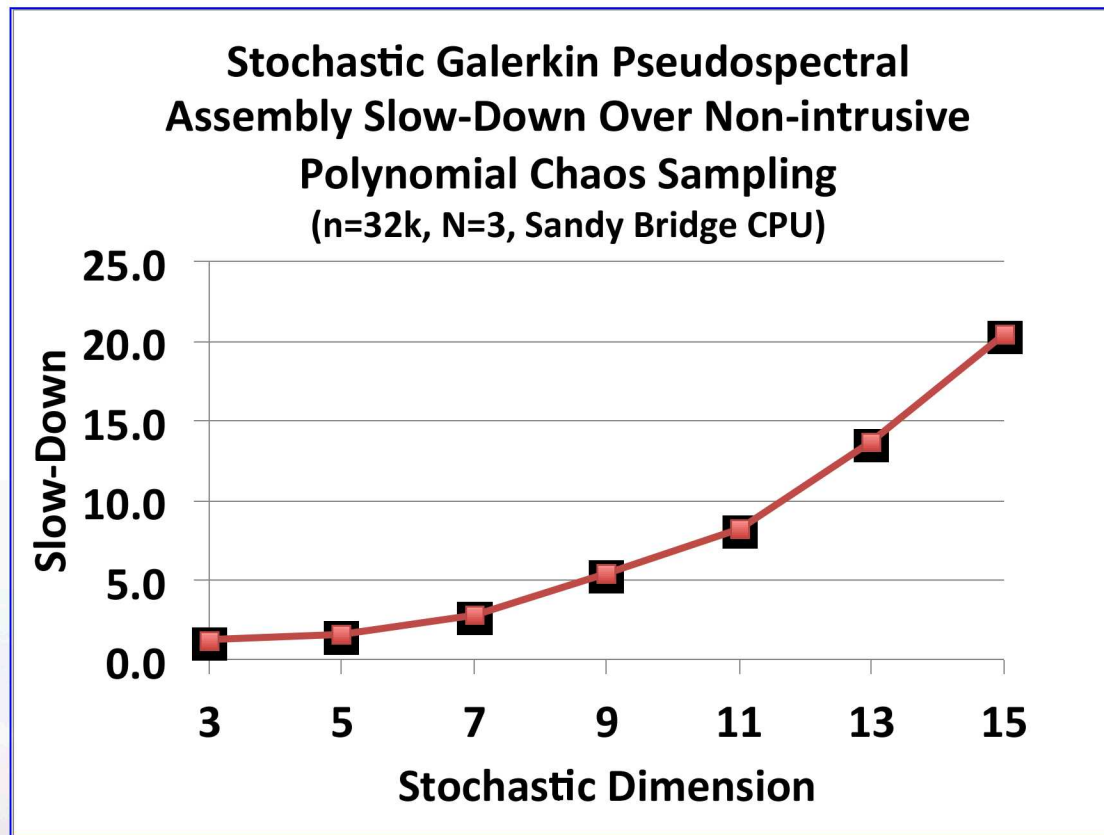
- **Sparse-grid quadrature methods for scalability to moderate stochastic dimensions**

SG Method Performs Well Over Moderate Range of Stochastic Problem Size



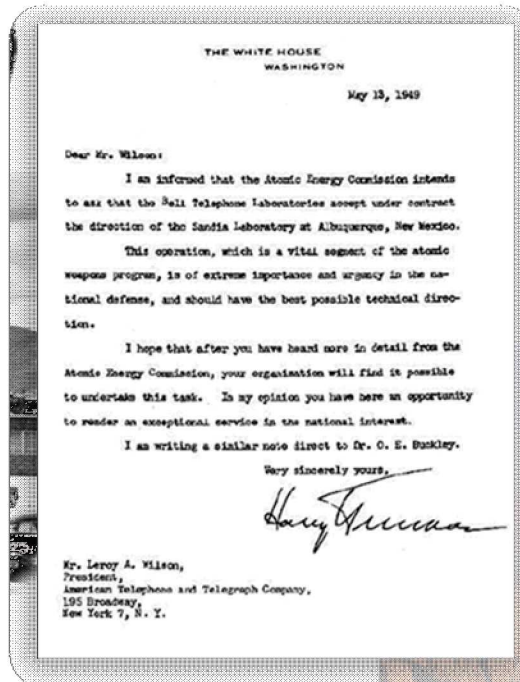
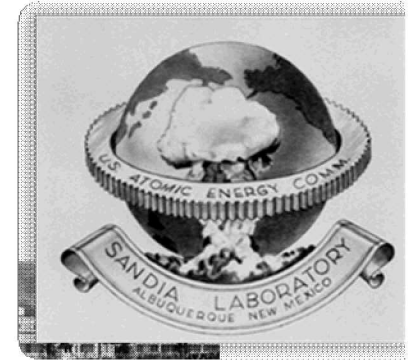
- Speed-up in time-to-solution of SG method compared to non-intrusive sampling
 - Smolyak sparse-grids for building PC basis
 - Gaussian abscissas
 - Comparable accuracy between SG solution and NISP solution
- Increased floating-point throughput (mat-vec, prec-vec) + reduced prec applies (P/Q) offset by increased FLOPs in mat-vec

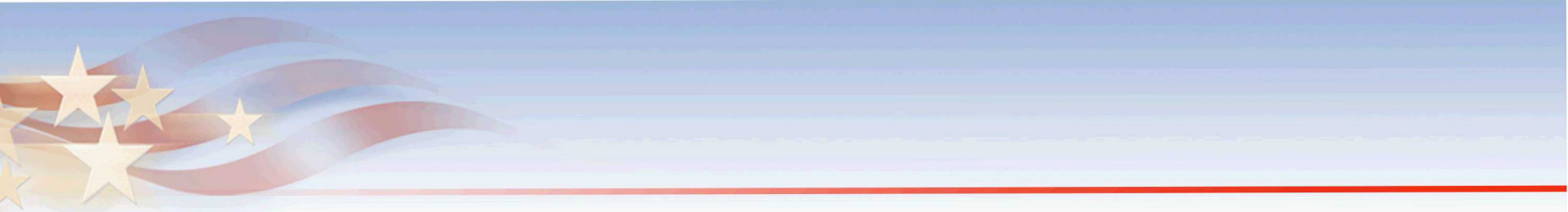
Stochastic Galerkin Assembly



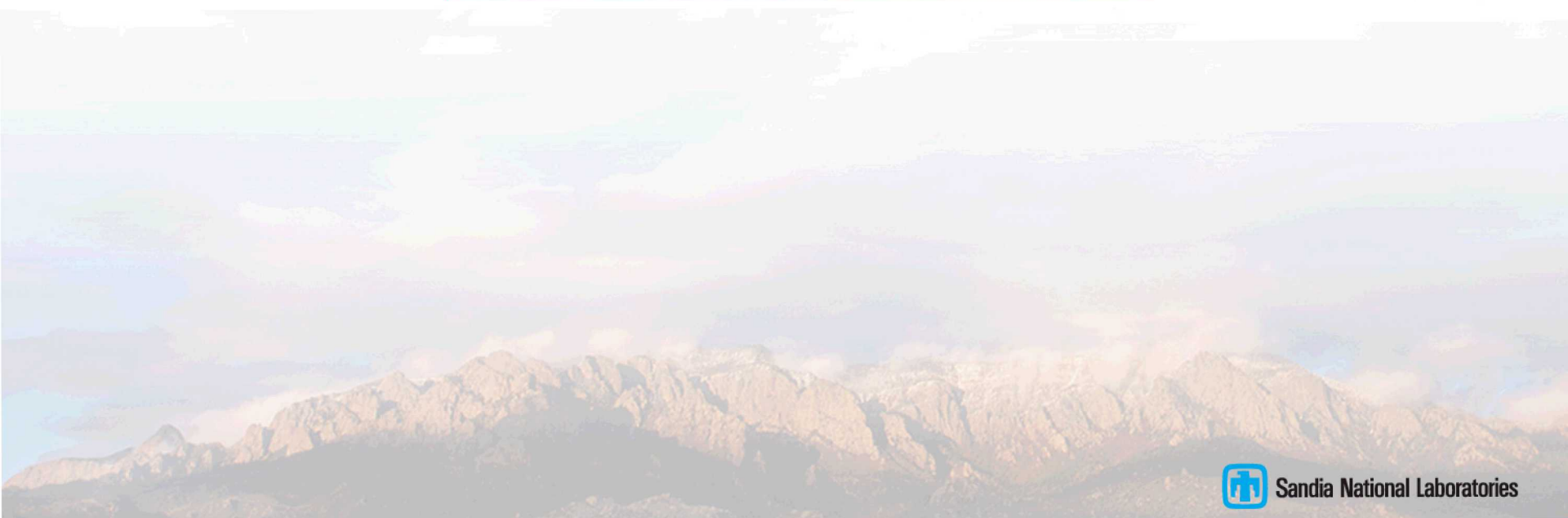
Sandia's History

"Exceptional service in the national interest"





Sandia “Who We Are” movie



Sandia's Sites

Albuquerque, New Mexico



Livermore, California

Kauai, Hawaii



Las Vegas, Nevada



WIPP, New Mexico



Pantex, Texas



Tonopah, Nevada



Sandia National Laboratories

The Evolution of Our Mission

1950s

Production
engineering and
manufacturing
engineering

1960s

Development
engineering

1970s

Multiprogram
laboratory

1980s

Research,
development and
production

1990s

Post-Cold War
transition

2000s

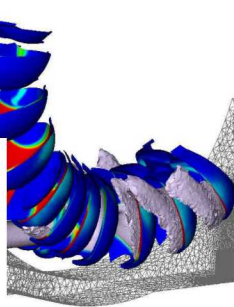
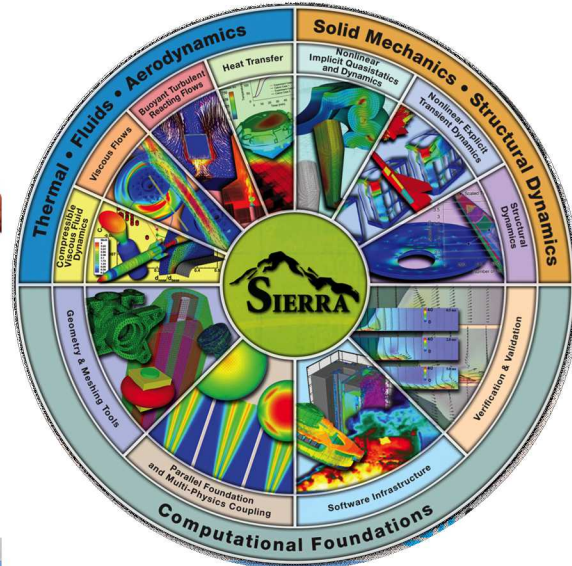
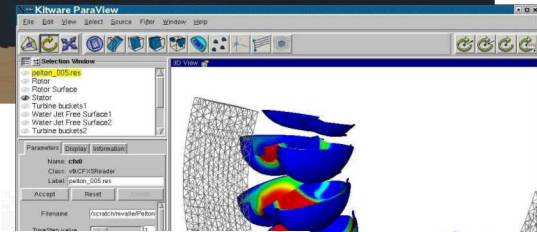
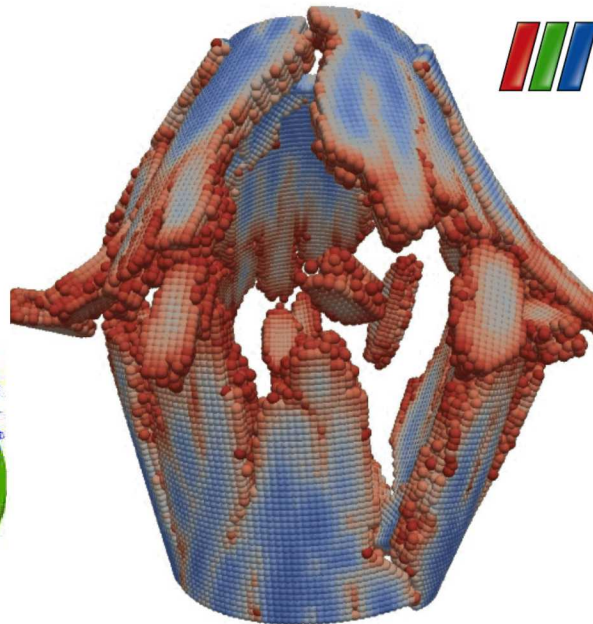
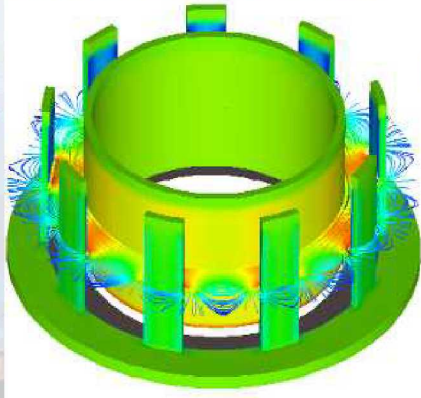
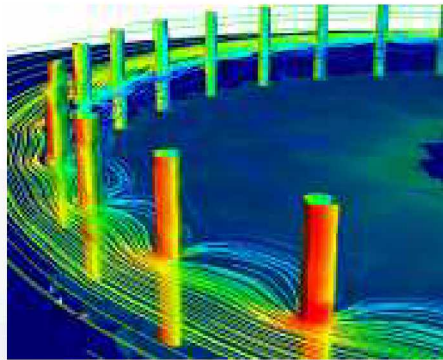
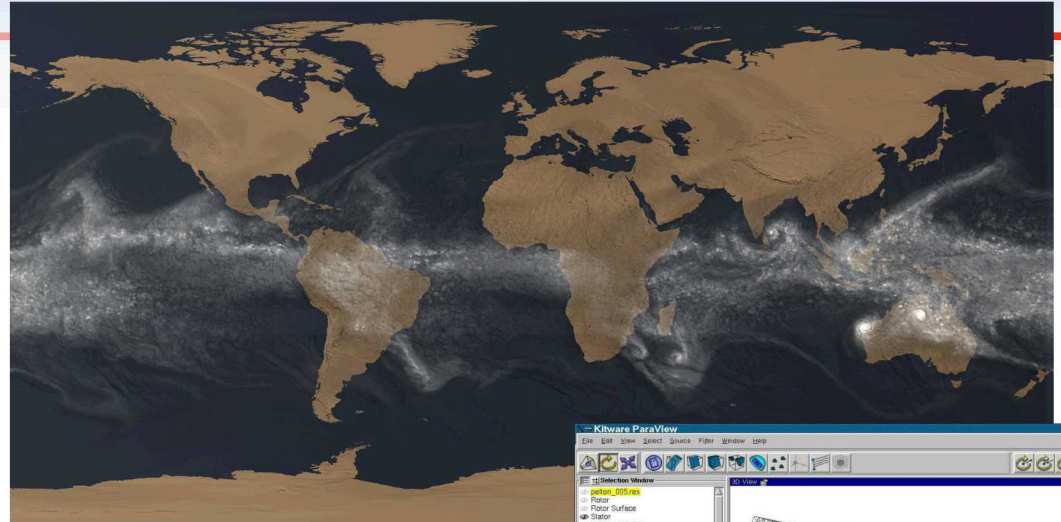
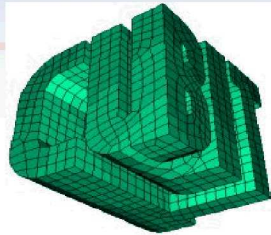
Expanded national
security role

% NW FUNDING

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%



(Some of) What we do



Department 1441 Mission

In partnership with others, we strive to transform computational simulation through the integration of new optimization, UQ, and discretization technologies into traditional engineering design and analyses.





1441 R&D Focus Areas

- **Uncertainty Quantification & Sensitivity Analysis**
 - **Methods for parameter and model UQ**
 - **Numerical uncertainty/verification**
 - **Reduced-order models/surrogates**
- **HPC Algorithms (Architecture Aware)**
 - **Inverse methods**
 - **Numerical PDE solution and multiphysics coupling**
 - **Mathematical and numerical analysis**
- **HPC Applications**
 - **V&V methodologies**
- **Optimization**
 - **Design optimization**
 - **Design of physical/computational experiments**



Computer Science Research Institute (CSRI)

- Annual Statistics
 - 14 projects
 - 4 workshops
 - 102 additional visitors from 67 institutions
 - 37 summer students
 - 3 sabbaticals
- 73% of 1400's math and computer science hires in 2005-2007 had prior contact through CSRI collaborations

