SAND2015-2029C

Exploring Embedded Uncertainty Quantification Methods on Next-Generation Computer Architectures

Eric Phipps (etphipp@sandia.gov),
H. Carter Edwards, Marta D'Elia, Jonathan Hu, and Siva Rajamanickam
Sandia National Laboratories

SIAM Conference on Computational Science and Engineering
March 13-18, 2015

SAND 2015-xxxxC

Sandia National Laboratories

Can Exascale Solve the UQ Challenge?

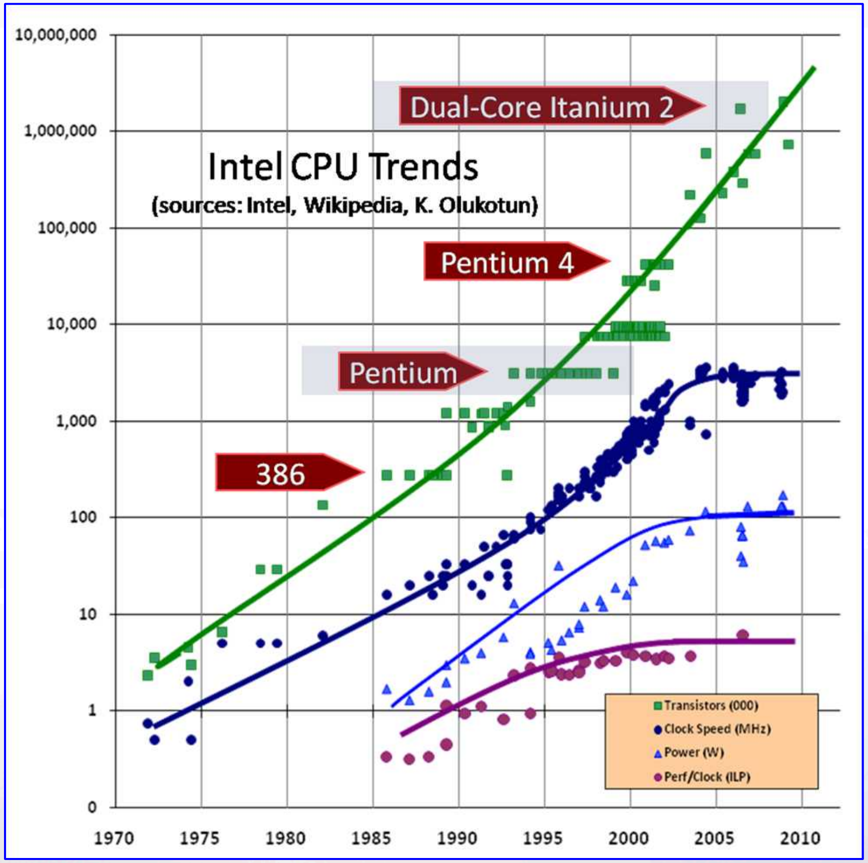
- UQ means many things
  - Best estimate + uncertainty, model validation, model calibration, …
- A key to many UQ tasks is forward uncertainty propagation
  - Given uncertainty model of input data (aleatory, epistemic, …)
  - Propagate uncertainty to output quantities of interest
- There are many forward uncertainty propagation approaches
  - Monte Carlo, stochastic collocation, polynomial chaos, stochastic Galerkin, …
- Key challenge:
  - Accurately quantifying rare events and localized behavior in high-dimensional uncertain input spaces
  - Can easily require O(10^4-10^6) expensive forward simulations
  - Often can only afford O(10^2) on today's petascale machines

Computer Architectures Are Changing Dramatically

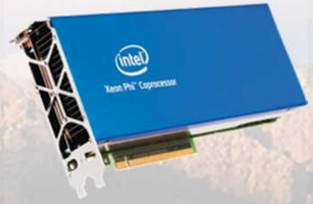
- Historically (super)computers have gotten faster by
  - Decreasing transistor size
  - Increasing clock frequency
  - Adding more compute nodes that communicate through an interconnect

- 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 threads to share data

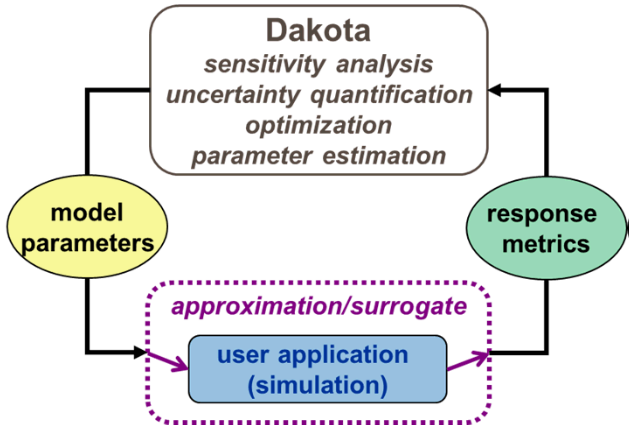


Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", Dr. Dobb's Journal



Emerging Architectures Motivate New Approaches to Predictive Simulation

- UQ approaches traditionally implemented as an outer loop:



http://dakota.sandia.gov

  - Easily exploit coarse-grained sampling parallelism by executing samples in parallel on collections of compute nodes
  - Aggregate performance limited by deterministic simulation

- Increasing UQ performance will require
  - Evaluating more samples in parallel
  - Speeding-up each sample evaluation

- 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 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

Polynomial Chaos Expansions (PCE)

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

$$\text{Find } u(\xi) \text{ such that } f(u,\xi) = 0,\ \xi : \Omega \to \Gamma \subset R^M, \text{ density } \rho$$

- **(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

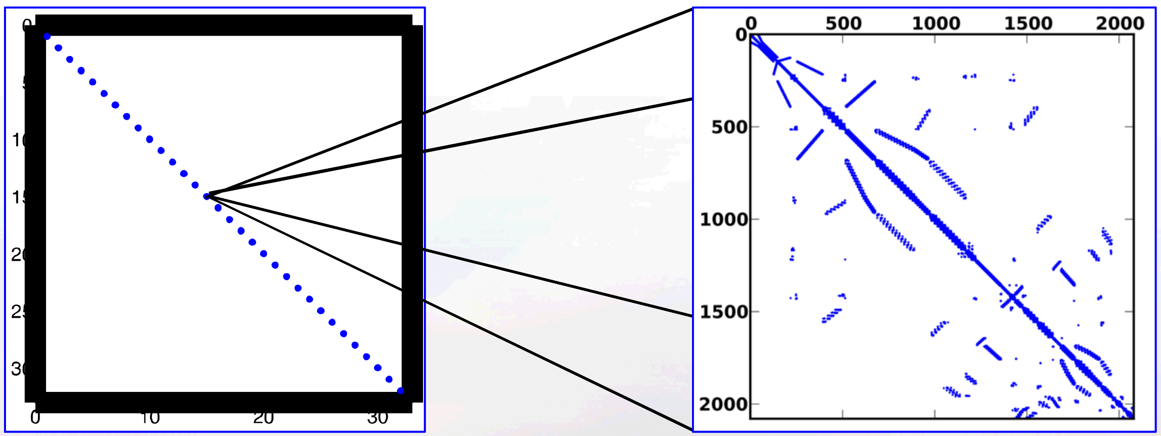
Simultaneous ensemble propagation

- **PDE:**

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

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

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

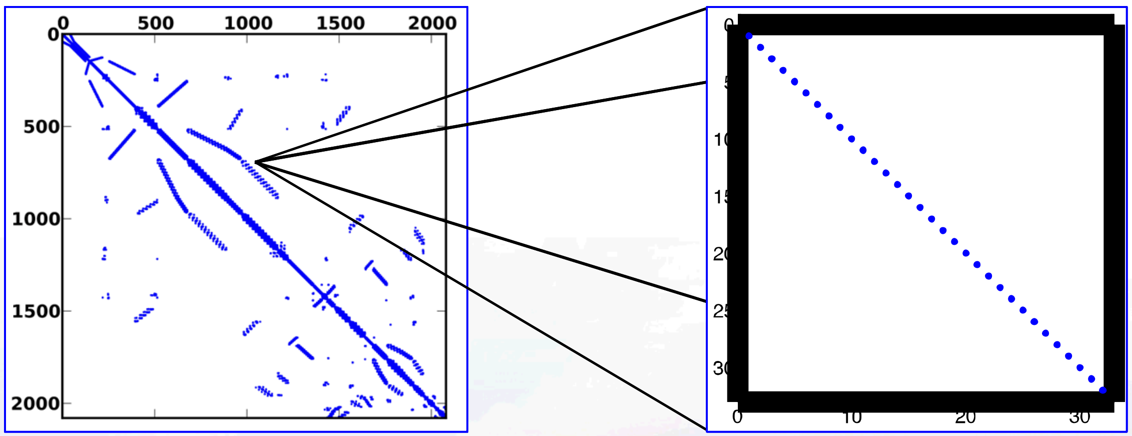


  - **Spatial DOFs for each sample stored consecutively**
  - **Implemented by ensemble loop around PDE matrix/RHS assembly, solve**

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**
  - **Implemented by placing ensemble loop at “scalar” level of PDE assembly, solve**
  - **Still have loop over ensembles around PDE assembly, solve**
    - **Suitable for coarse-grained parallelism**

Implementing simultaneous ensemble propagation

- Each sample-dependent scalar replaced by length-*m* array
  - Automatically reuse non-sample dependent data (e.g., mesh in matrix/RHS assembly, matrix-graph in solvers, …)
  - Sparse access latency amortized across ensemble (e.g., sparse mat-vecs)
  - Communication latency amortized across ensemble (sparse mat-vecs, dot-products, …)
  - Math on ensemble naturally maps to vector arithmetic (consistent vectorization)

- Could implemented this by rewriting simulation code
  - Expand size of matrix/vector data structures by *m*
  - Replace each scalar operation by a length-*m* loop

- Or automatically (in C++) by introducing an *ensemble* scalar type
  - C++ class containing an array with length fixed at compile-time
  - Overload all math operations by mapping operation across array

$$a = \{a_1, \dots, a_m\}, \quad b = \{b_1, \dots, b_m\}, \quad c = a \times b = \{a_1 \times b_1, \dots, a_m \times b_m\}$$

  - Replace floating-point type with ensemble type in
    - Matrix/vector data structures
    - Matrix/RHS assembly routines
    - Solvers

Stokhos: Trilinos Tools for Embedded UQ Methods

- **Provides ensemble scalar type**
  - **Uses expression templates to fuse loops**

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

Trilinos

http://trilinos.sandia.gov

- **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)**

Techniques Prototyped in FENL Mini-App

http://trilinos.sandia.gov

- 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 grid quadrature for NISP method

Potential Speed-up for Sparse Solvers

- Ingredients to sparse linear system solvers (CG, GMRES, …)
  - Sparse matrix-vector products

$$y(i) = \sum_{l=A.row(i)}^{A.row(i+1)} A.vals(l)x(A.col(l))$$

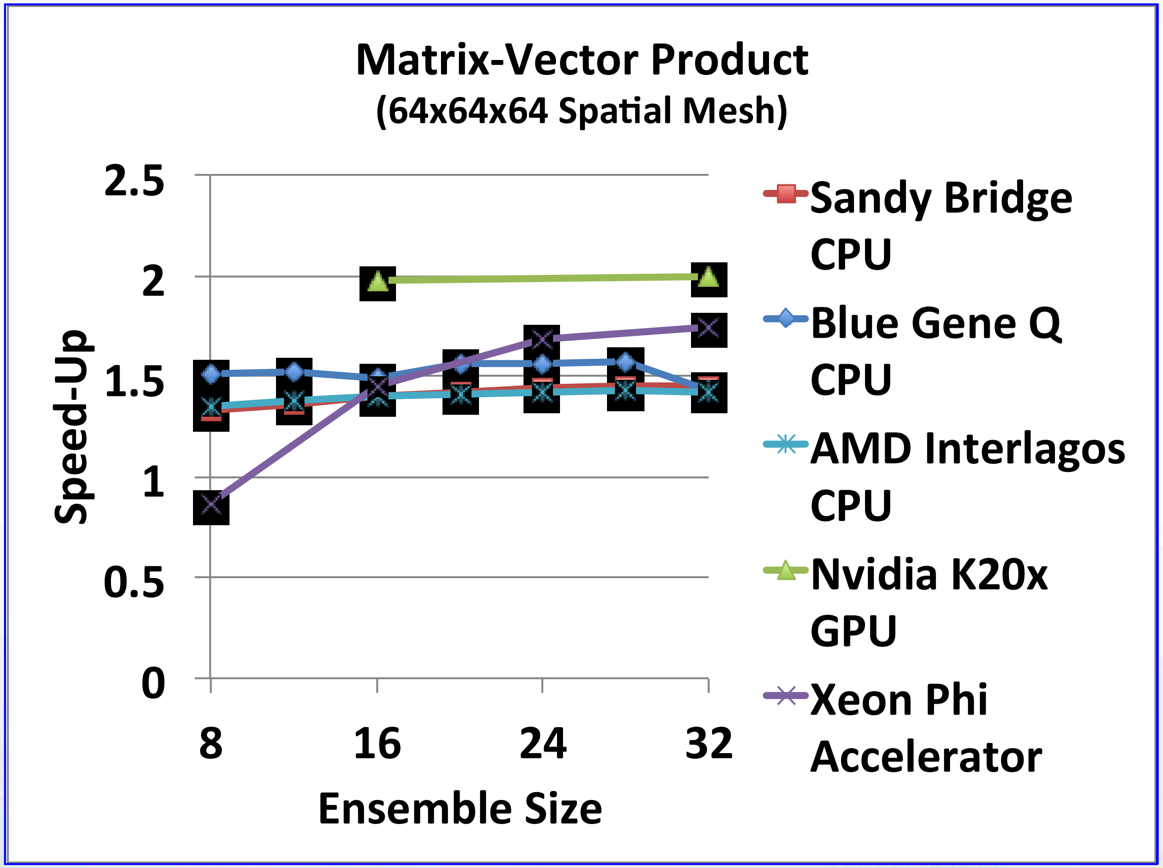
  - Dot-products
  - Preconditioners
    - Relaxation-based (Jacobi, Gauss-Seidel, …)
    - Incomplete factorizations (ILU, IC, …)
    - Polynomial (Chebyshev, …)
    - Multilevel (Algebraic/Geometric multigrid)

- Sparse matrix-vector products
  - Amortize MPI latency in halo exchange
  - Reuse matrix graph
  - Replace sparse with contiguous loads
  - Vector arithmetic

- Dot-products
  - Amortize MPI latency

- Preconditioners
  - Sparse mat-vecs
  - Sparse factorizations/triangular-solves
  - Smaller, more unstructured matrices

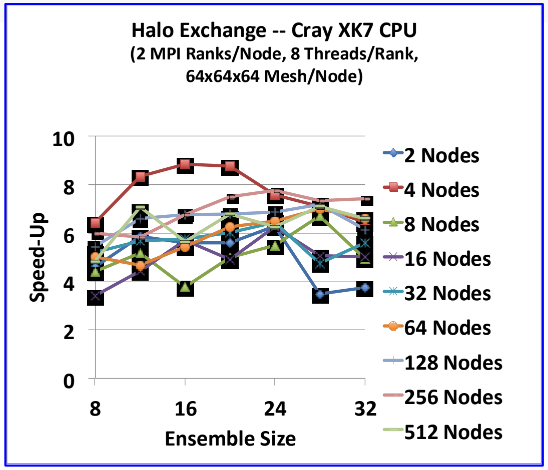
Ensemble Sparse Matrix-Vector Product Speed-Up

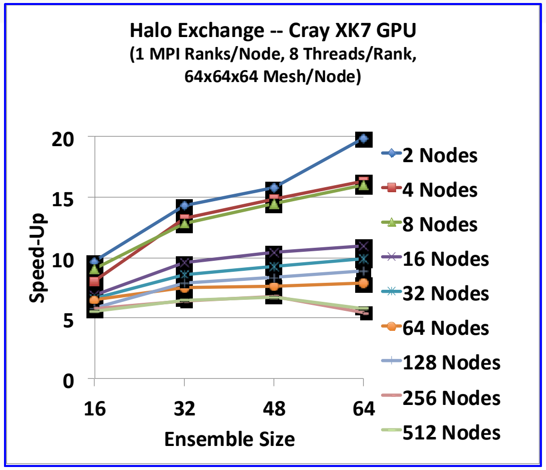


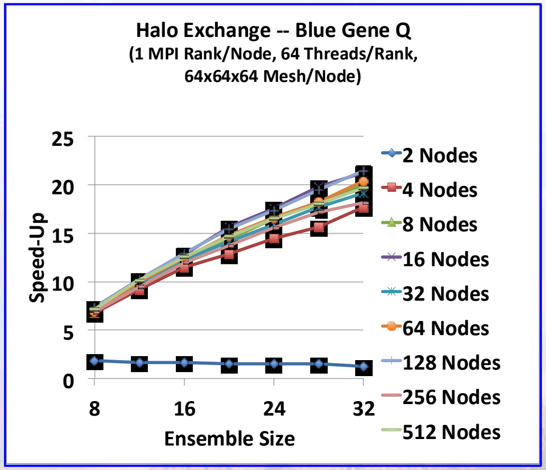
- Speed-up results from
  - Reuse of matrix graph (20%)
  - Replacement of sparse gather with contiguous load
  - Perfect vectorization of multiply-add

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

Interprocessor Halo Exchange





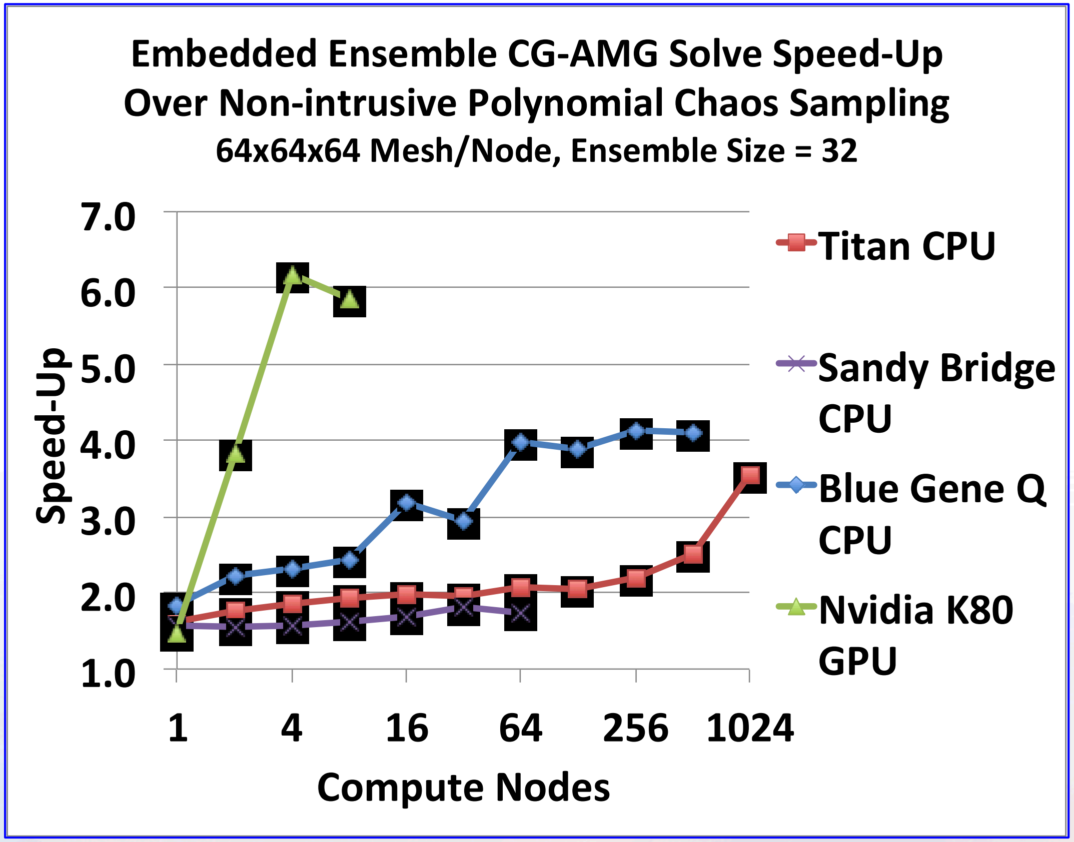


- **Speed-up results from reduced aggregate communication latency**
  - Fewer, larger MPI messages
  - Communication volume is the same

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

Sandia National Laboratories

AMG Preconditioned CG Solve



- Smoothed-aggregation algebraic multigrid preconditioning (MueLu)
  - Chebyshev smoothers
  - Sparse-direct coarse-grid solver (Amesos2/Basker)
  - Multi-jagged parallel repartioning (Zoltan2)
- Assumes number of CG iterations same for all samples
  - True for problems with tame diffusion coefficient on regular meshes
  - See poster by M. D'Elia, PP201

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

- **Embedded sampling approach does not**
  - Substantially change floating-point operation to memory access ratios
  - 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

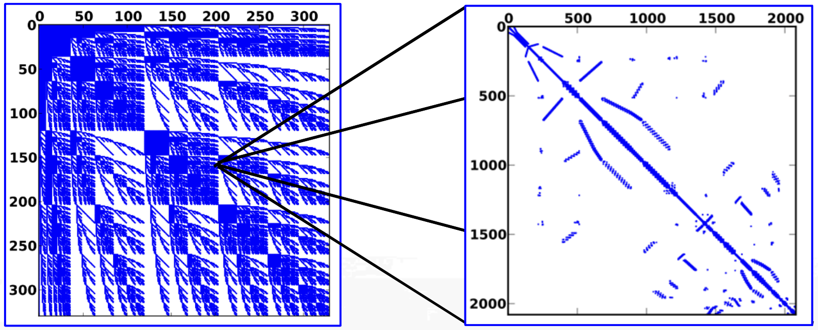
- **Stochastic Galerkin method (Ghanem and many, many others…):**

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

- **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**

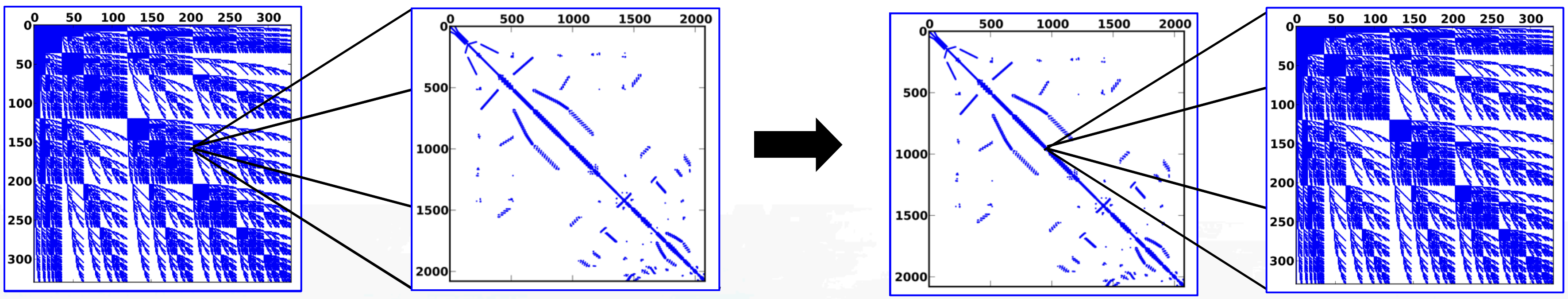
- **Many fewer stochastic degrees-of-freedom for comparable level of accuracy:**

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

Commuted SG Structure for Emerging Architectures

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

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



Stochastic sparsity — Spatial sparsity — Spatial sparsity — Stochastic sparsity

- Implemented in same manner as embedded sample propagation
  - Scalars replace by PC coefficient arrays
  - Similar C++ operator overloading approach:

$$a = \sum_{i=0}^{P} a_i \psi_i, \;\; b = \sum_{j=0}^{P} b_j \psi_j, \;\; c = ab \approx \sum_{k=0}^{P} c_k \psi_k, \;\; c_k = \sum_{i,j=0}^{P} a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}$$

  - Approach implemented within Stokhos package

Commuted 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

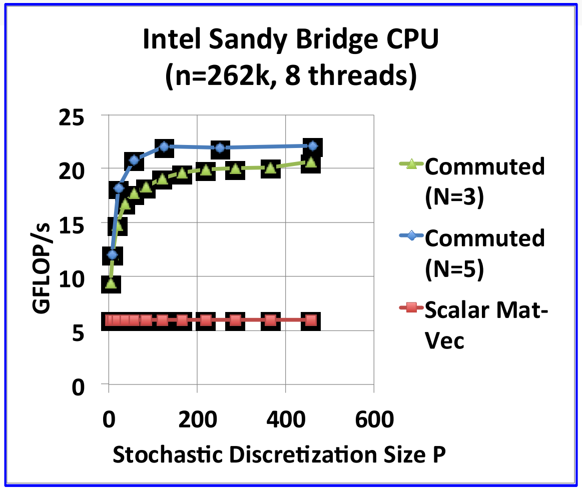
$$\aleph_A(l) = \{ m \mid A_0(l,m) \neq 0 \} \qquad \aleph_C(i) = \{ (j,k) \mid C(i,j,k) \neq 0 \}$$

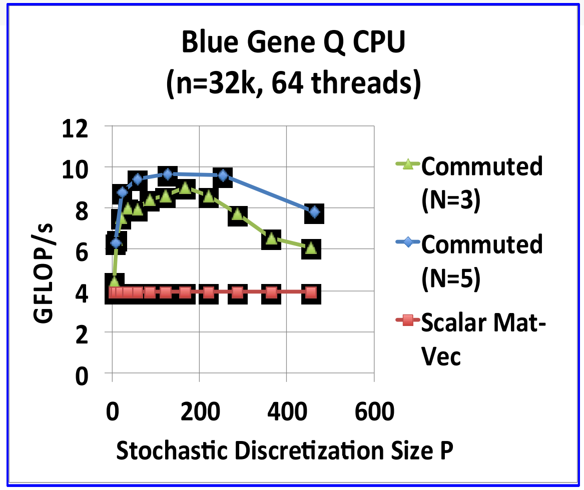
stochastic basis
stochastic bases sum
stochastic basis
stochastic basis
triple product

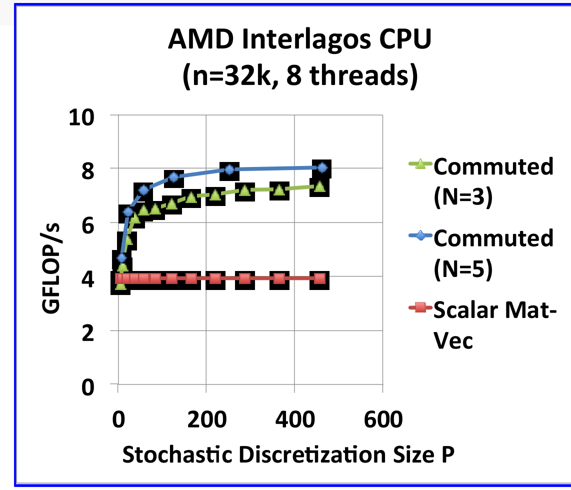
$$y(i,l) = \sum_{m \in \aleph_A(l)} \sum_{(j,k) \in \aleph_C(i)} A(k,l,m) x(j,m) C(i,j,k)$$

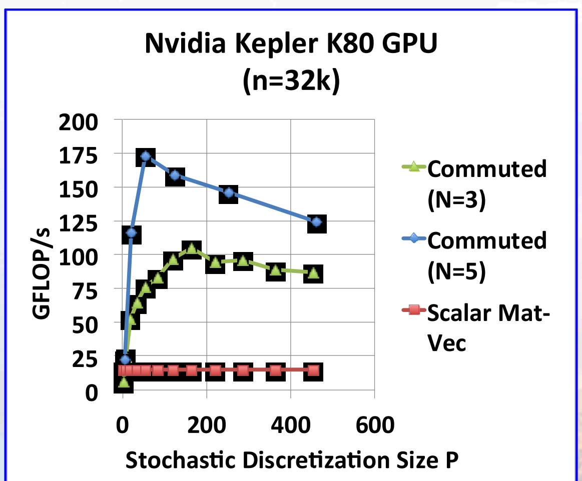
FEM basis
FEM bases sum
FEM basis
FEM basis

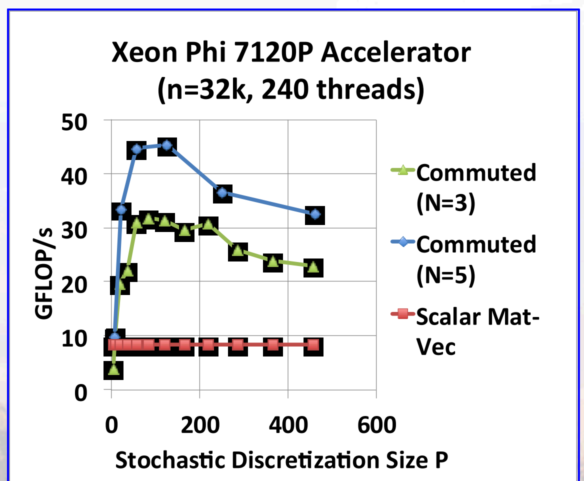
Sparse Matrix-Vector Product*











- **Increased throughput arises from substantial reuse within PCE multiply**

*Phipps, Edwards, Hu and Ostien, International Journal of Computer Mathematics, 2013.

Sandia National Laboratories

Stochastic Galerkin Preconditioning

- Preconditioning stochastic Galerkin system is a significant challenge

- Common approach is mean-based preconditioning:

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

- Applying mean preconditioner in commuted layout is very efficient:

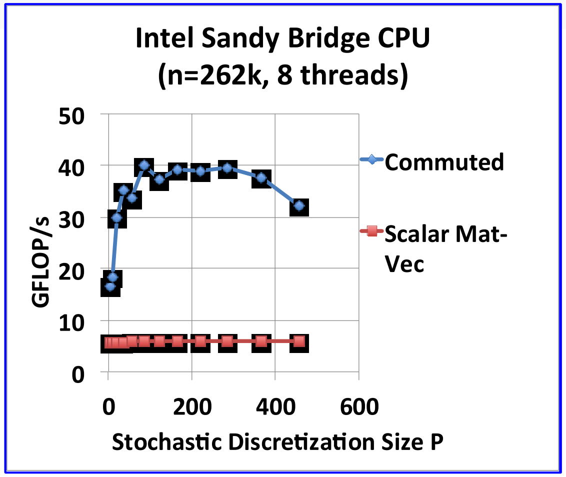
$$Y^{com} = M^{com}_{mean} X^{com} \implies \sum_{i=0}^{P} y_i \otimes e_i = \left(M_0 \otimes I_P \right) \left(\sum_{j=0}^{P} x_j \otimes e_j \right)$$

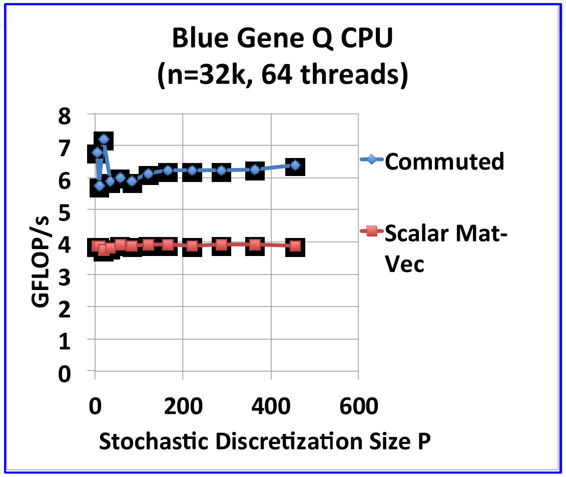
$$\implies [y_0, \ldots y_P] = M_0 [x_0, \ldots, x_P]$$

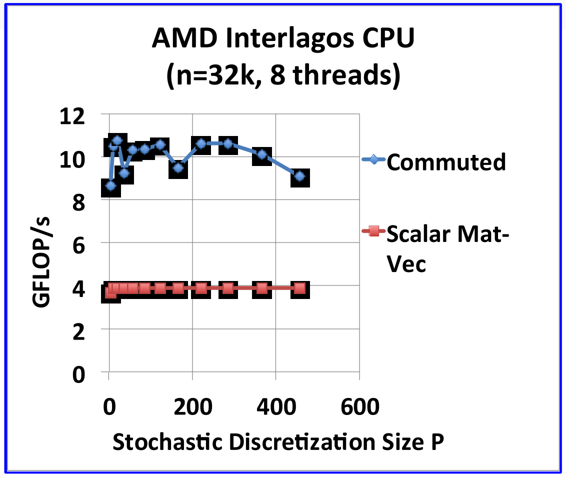
  - Matrix-times-multivector with row-wise layout
  - Vectorize over multivector columns
  - Reuse of matrix/graph entries

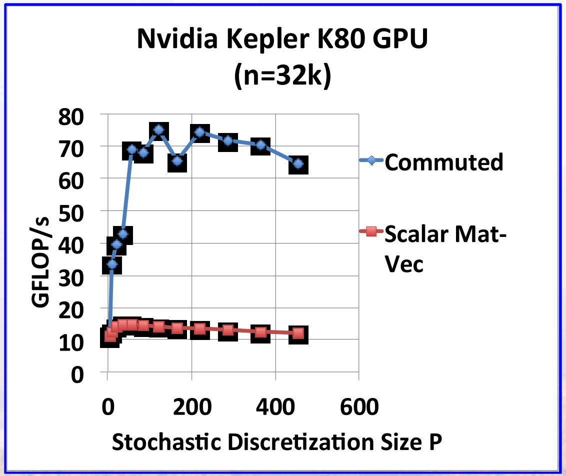
- Applying preconditioner is often dominant cost

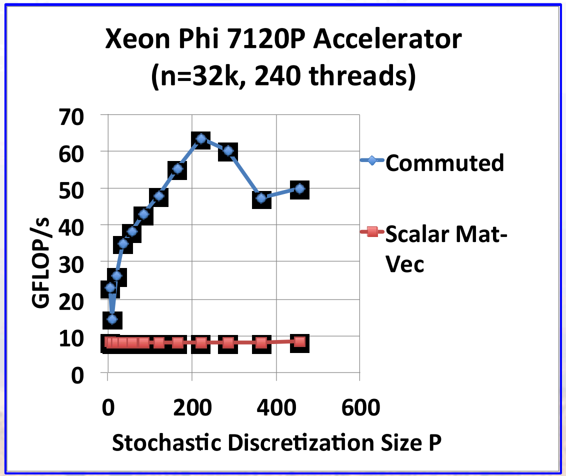
Mean Matrix-Vector Multiply





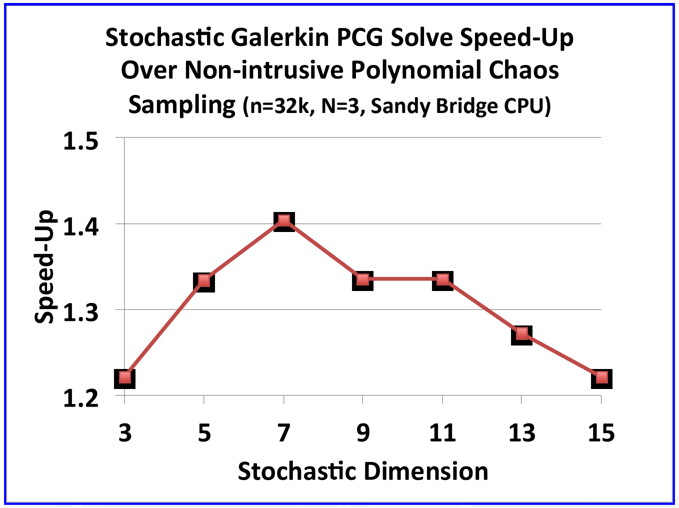


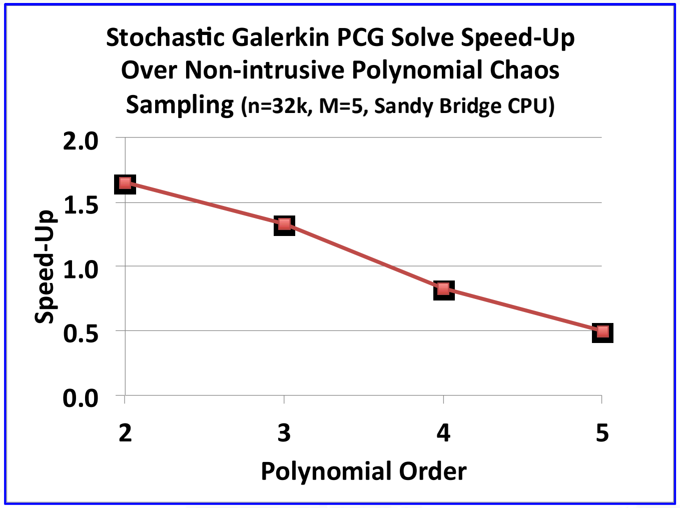


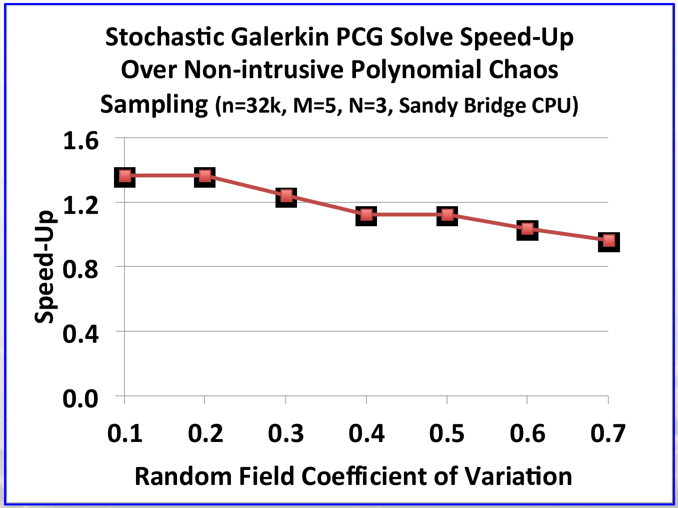


Sandia National Laboratories

SG Method Performs Well Over Moderate Range of Stochastic Problem Size

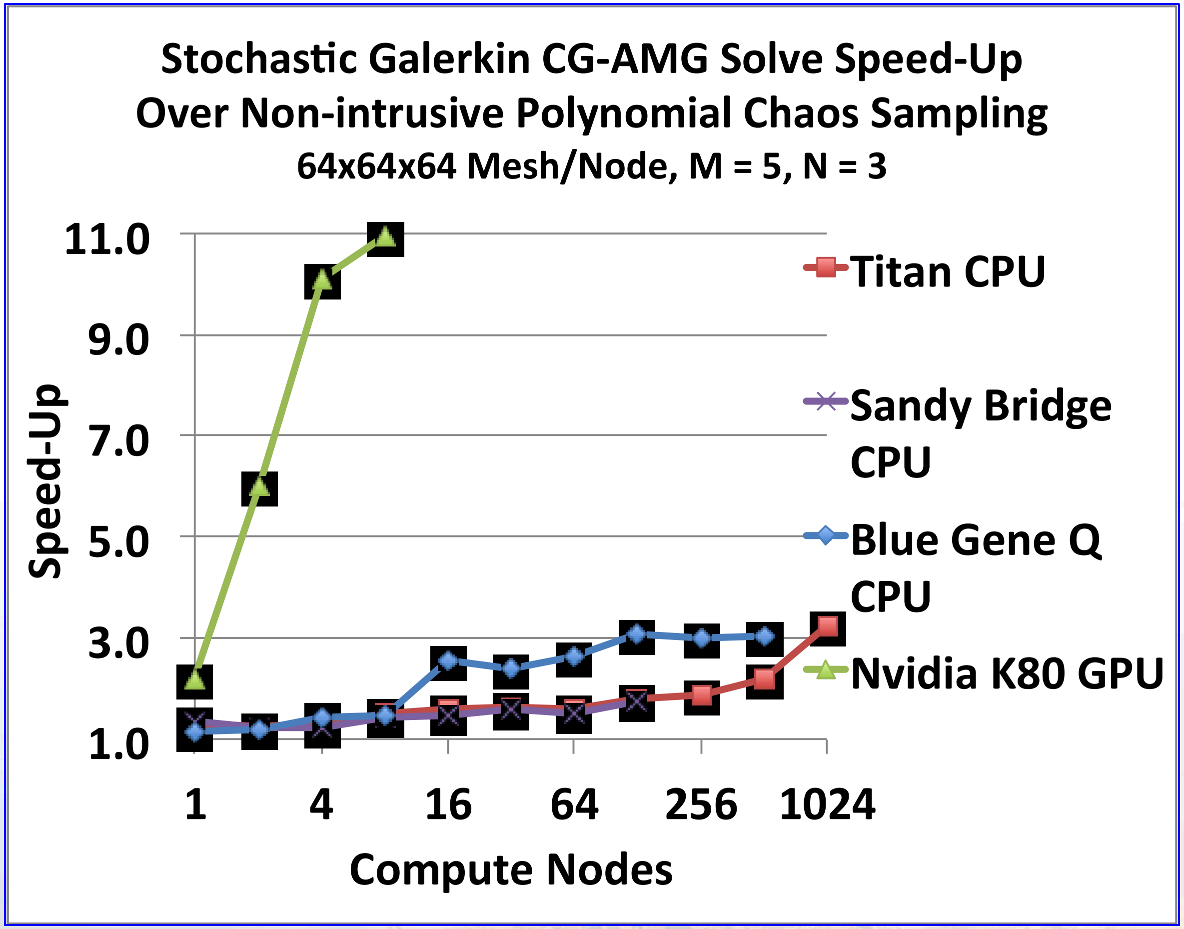






- **Speed-up in time-to-solution of SG method compared to non-intrusive sampling**
  - Smolyak sparse-grids for building PC basis
  - Gaussian abscissas
  - Comparable accuracy between SG solution and NISP solution
- **Increased floating-point throughput (mat-vec, prec-vec) + reduced prec applies (P/Q)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

AMG Preconditioned CG Solve



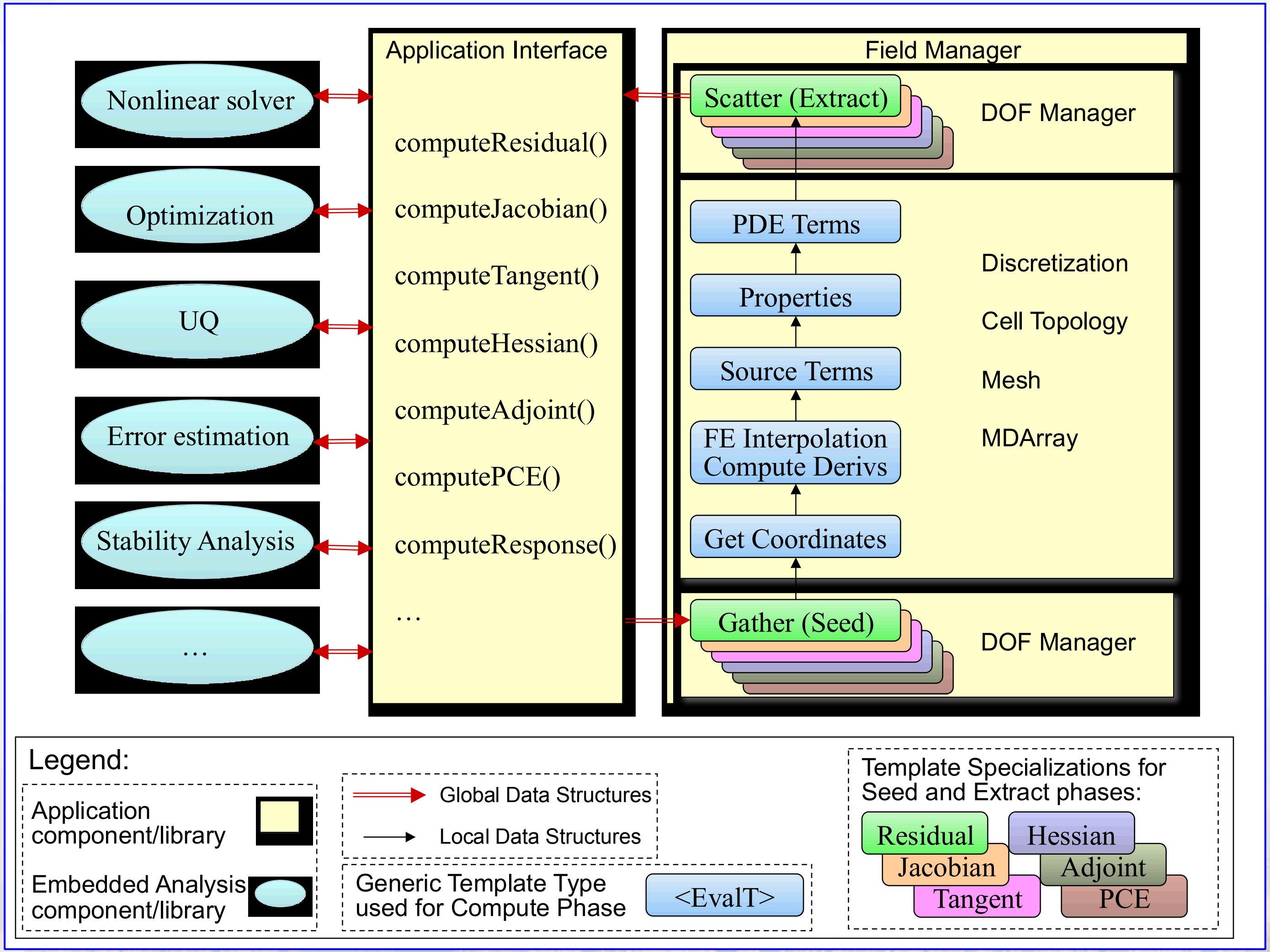
- Speed-up arises from:
  - Increased floating-point throughput
  - Reduced preconditioner applies
  - Reduced aggregate communication volume

Concluding Remarks

- Reordering UQ algorithms to propagate some UQ information at lowest levels can lead to substantial improvements in performance
  - Alleviate burden of deterministic simulation code from exploiting all fine-grained parallelism
  - Increases opportunities for fine-grained parallelism
  - Improves memory access patterns
  - Reduces aggregate memory bandwidth and communication
- Applying technique through C++ templates greatly facilitates implementation
  - Alleviate code developers from having to worry about UQ
- Significant challenges remain:
  - Effective grouping of samples in ensembles for non-smooth, less-smooth problems (See M. D'Elia poster, PP201 for first steps in this direction)
  - Dealing with code divergence (e.g., conditionals)
  - Partitioning/adapting PC basis to reduce memory burden

Extra Slides

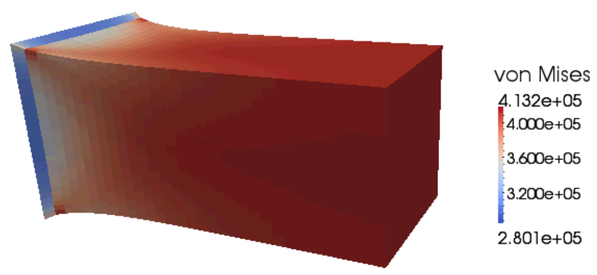
Templated Components Orthogonalize Physics and Embedded Algorithm R&D



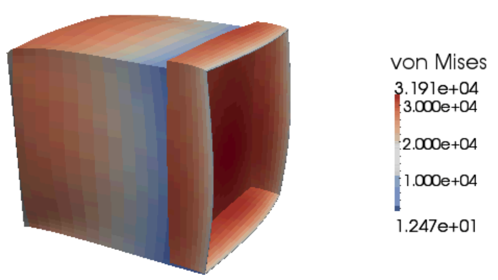
3-D Linear & Nonlinear Elasticity Model Problems[1]

- **Linear finite elements, 32x32x32 mesh**
  - **Nonlinear: neo-Hookean strain energy potential**
- **Uncertain Young's modulus random field**
  - **Truncated KL expansion (exponential covariance)**
- **Albany/LCM code (Salinger, Ostien, et al)**
  - **Trilinos discretization and solver tools**
  - **Automatic differentiation**
  - **Embedded UQ**
  - **MPI parallelism**

http://trilinos.sandia.gov



Displacement (Mean)

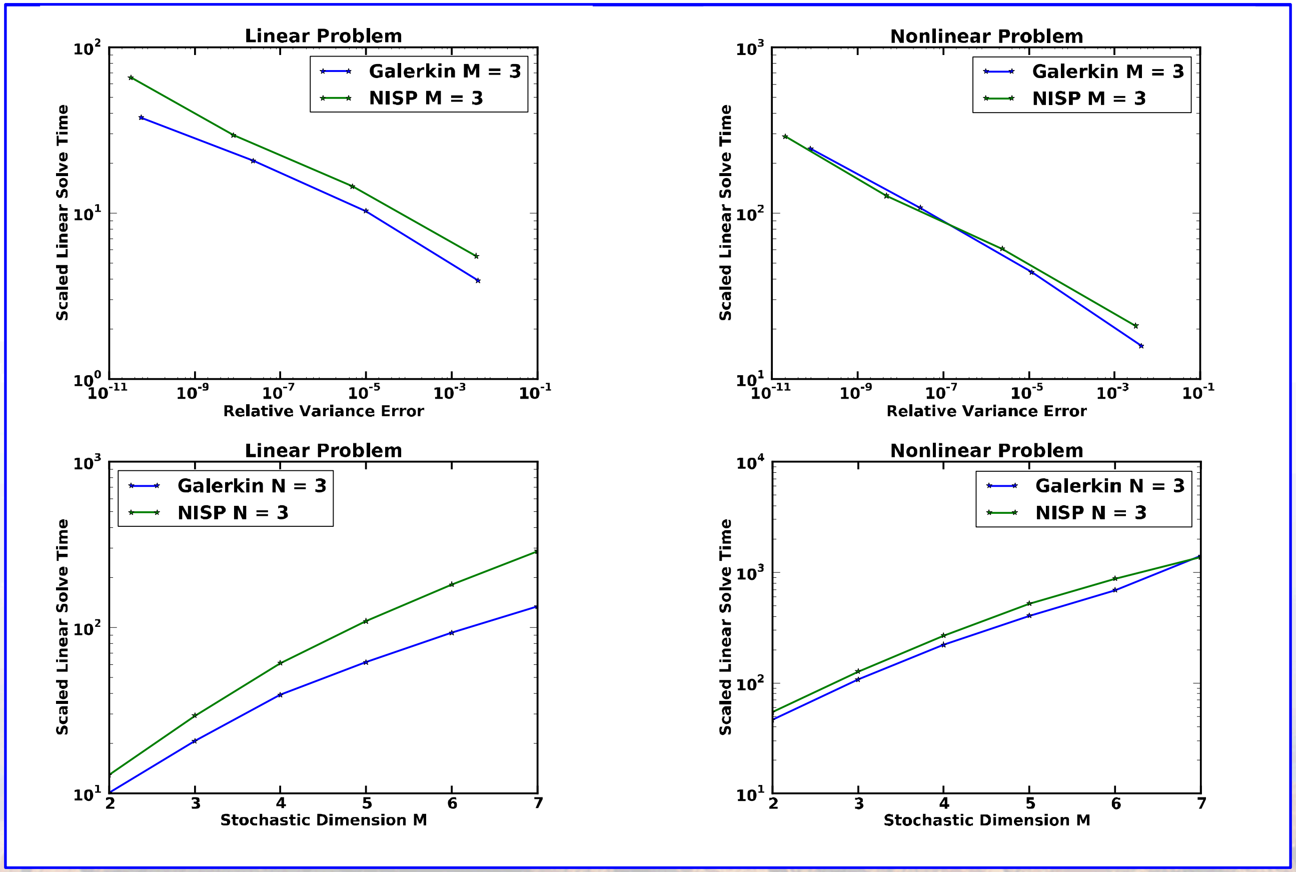


Displacement (Std. Dev.)

[1]Phipps, Edwards, Hu and Ostien, International Journal of Computer Mathematics, 2013.

Solve Performance

- Comparison to non-intrusive polynomial chaos/spectral projection (NISP)
  - Isotropic sparse-grid quadrature, Gauss-Legendre abscissas, linear growth rules
  - GMRES, algebraic multigrid preconditioning

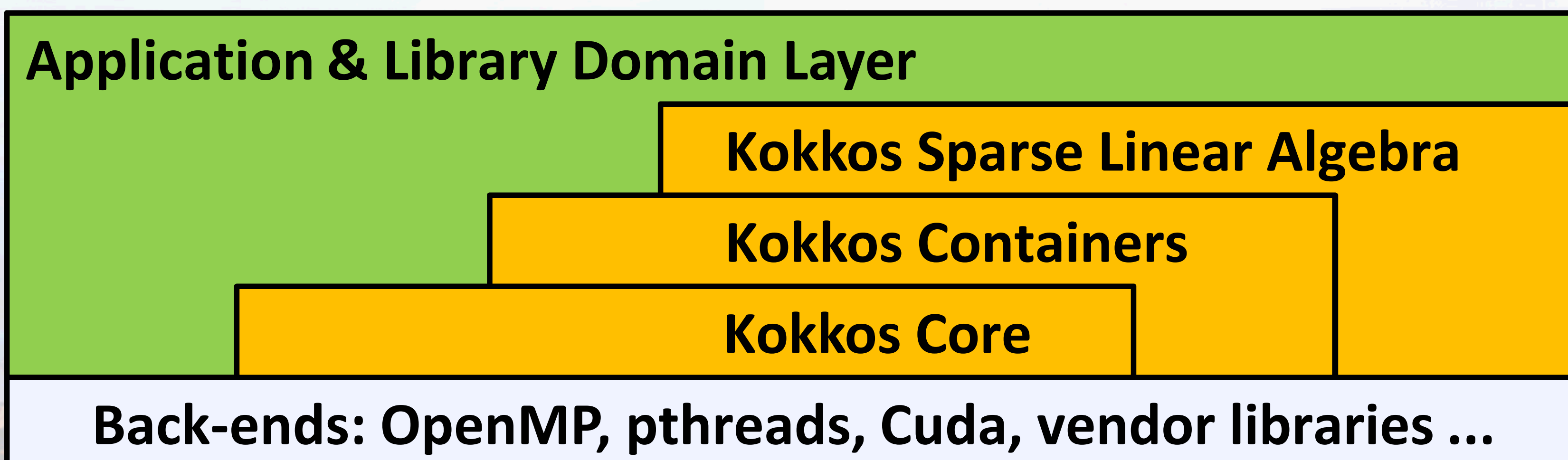


Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications*

- **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.sandia.gov

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



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**
  - 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

Trilinos

http://trilinos.sandia.gov

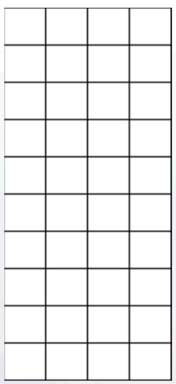
*M. Heroux, M. Hoemmen, *et al* (SNL)

Kokkos Integration

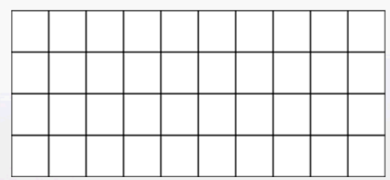
- **Kokkos views of UQ scalar type internally stored as views of 1-higher rank**
  - **UQ dimension is always contiguous, regardless of layout**

- **Facilitates**
  - **Fine-grained parallelism over UQ dimension**
  - **Efficient allocation and initialization**
  - **Specialization of kernels**
  - **Transfering data between host and device and MPI communication**

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

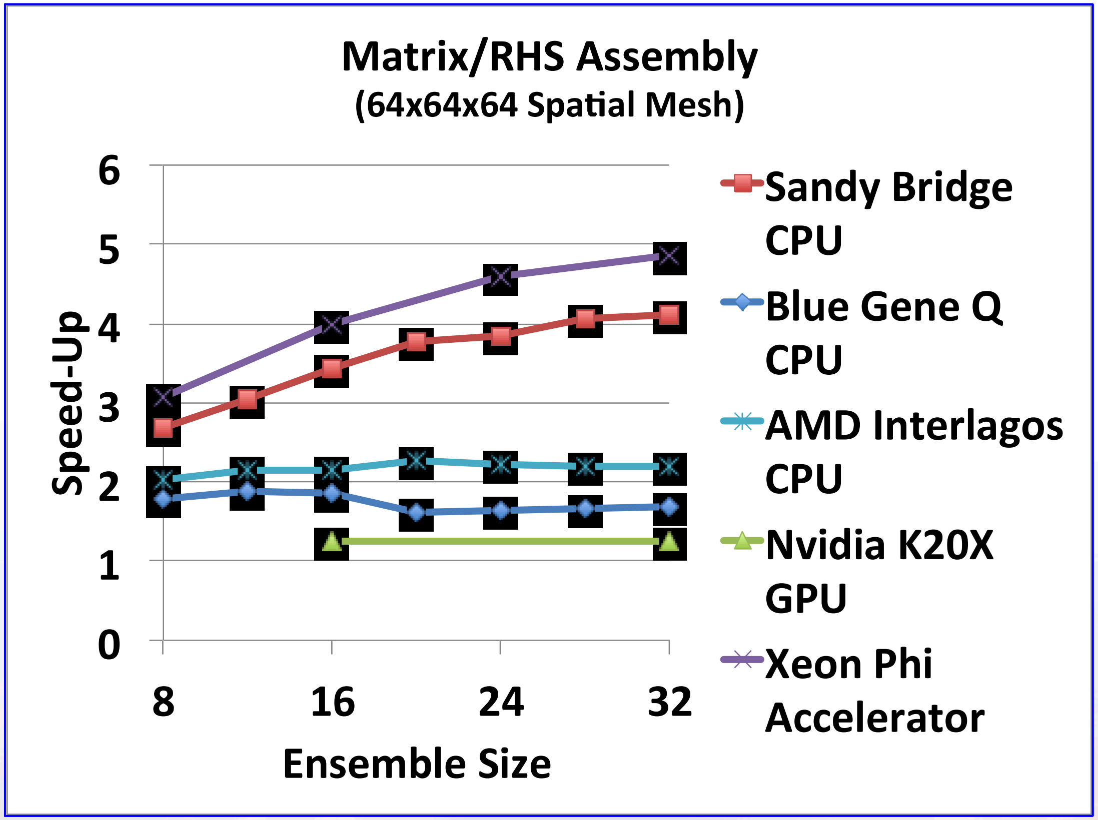


```
Kokkos::View< Ensemble<double,4>*, LayoutLeft, Device > view(“v”, 10);
```



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

PDE Matrix/RHS Assembly



Embedded Ensemble Scalar Type for PDE "Assembly"

- Evaluation of discrete SG residual/Jacobian entries is a significant challenge for nonlinear problems
- 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 ("non-intrusive")
  - Requires only two additional assembly kernels: PCE evaluation and quadrature
  - Use ensemble scalar type for evaluating residual/Jacobian at multiple quadrature points simultaneously

Stochastic Galerkin Assembly