SAND2016-2545C

Embedded Ensemble Propagation for Improving Performance, Portability and Scalability of Uncertainty Quantification on Emerging Computational Architectures

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

14th Copper Mountain Conference on Iterative Methods
March 20-25, 2016

SAND2016-XXXX C

U.S. DEPARTMENT OF ENERGY | Office of Science

Equinox

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

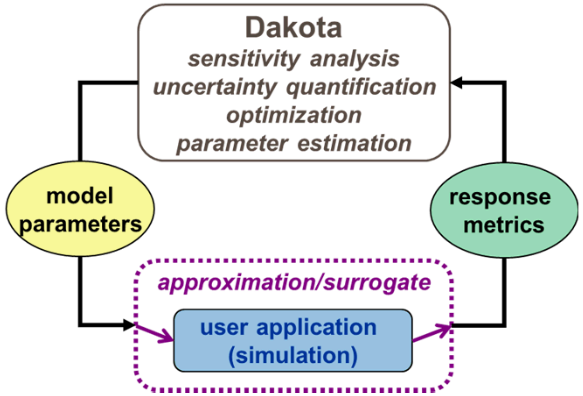
Sandia National Laboratories

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 $O(10^4\text{-}10^6)$ expensive forward simulations
  - Often can only afford $O(10^2)$ on today's petascale machines

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;
   }
}
```

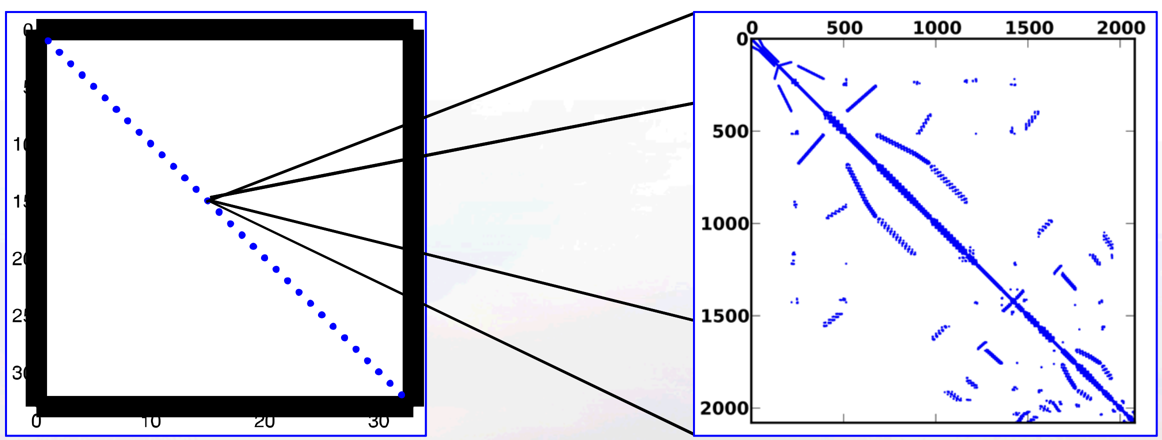
Simultaneous ensemble propagation

- **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**

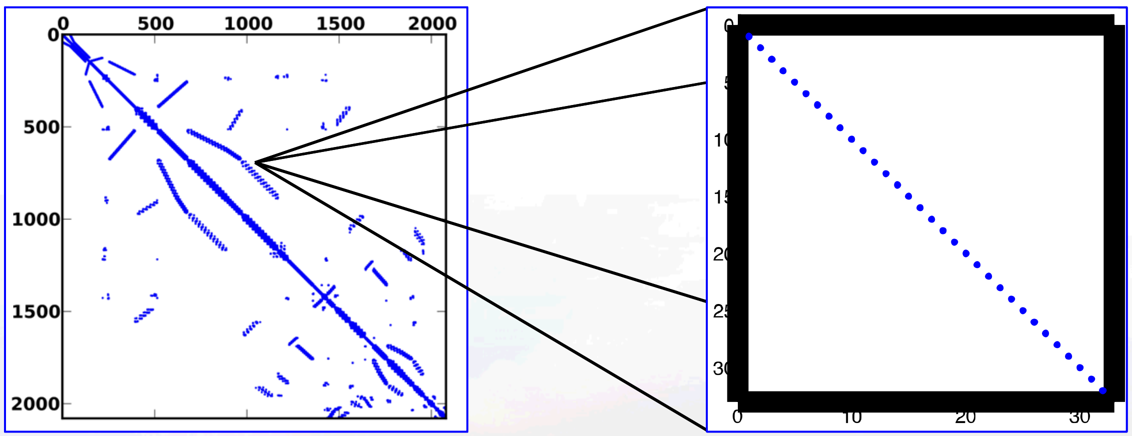
Ensemble Matrix-Vector Product

```
// 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**

Commuted, Ensemble Matrix-Vector Product

```
// Ensemble matrix-vector product using commuted layout
template <typename T, int m>
void ensemble_commuted_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

Sandia National Laboratories

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[m] = v; }
  Ensemble& operator=(const Ensemble& a) {
    for (int e=0; e<m; ++e) val[m] = a.val[m];
    return *this;
  }
  Ensemble& operator+=(const Ensemble& a) {
    for (int e=0; e<m; ++e) val[m] += a.val[m];
    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 = Ensemble<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;
  }
}
```

Stokhos: Trilinos Tools for Embedded UQ Methods

http://trilinos.sandia.gov

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

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

- **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

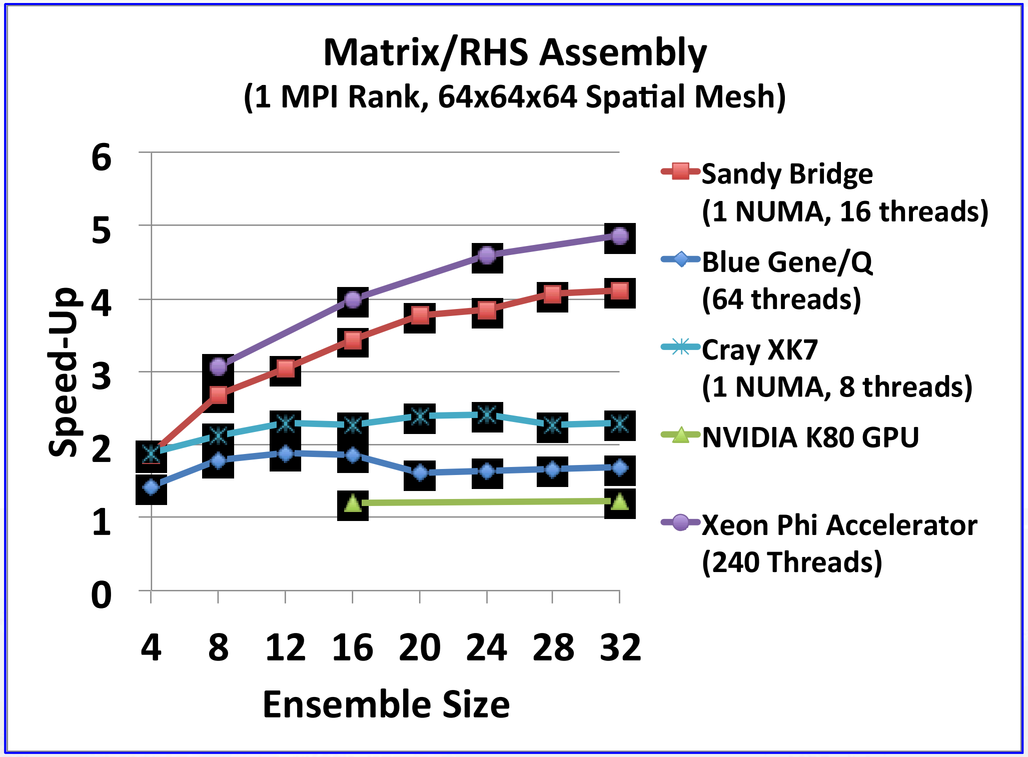
http://trilinos.sandia.gov

- 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 grids

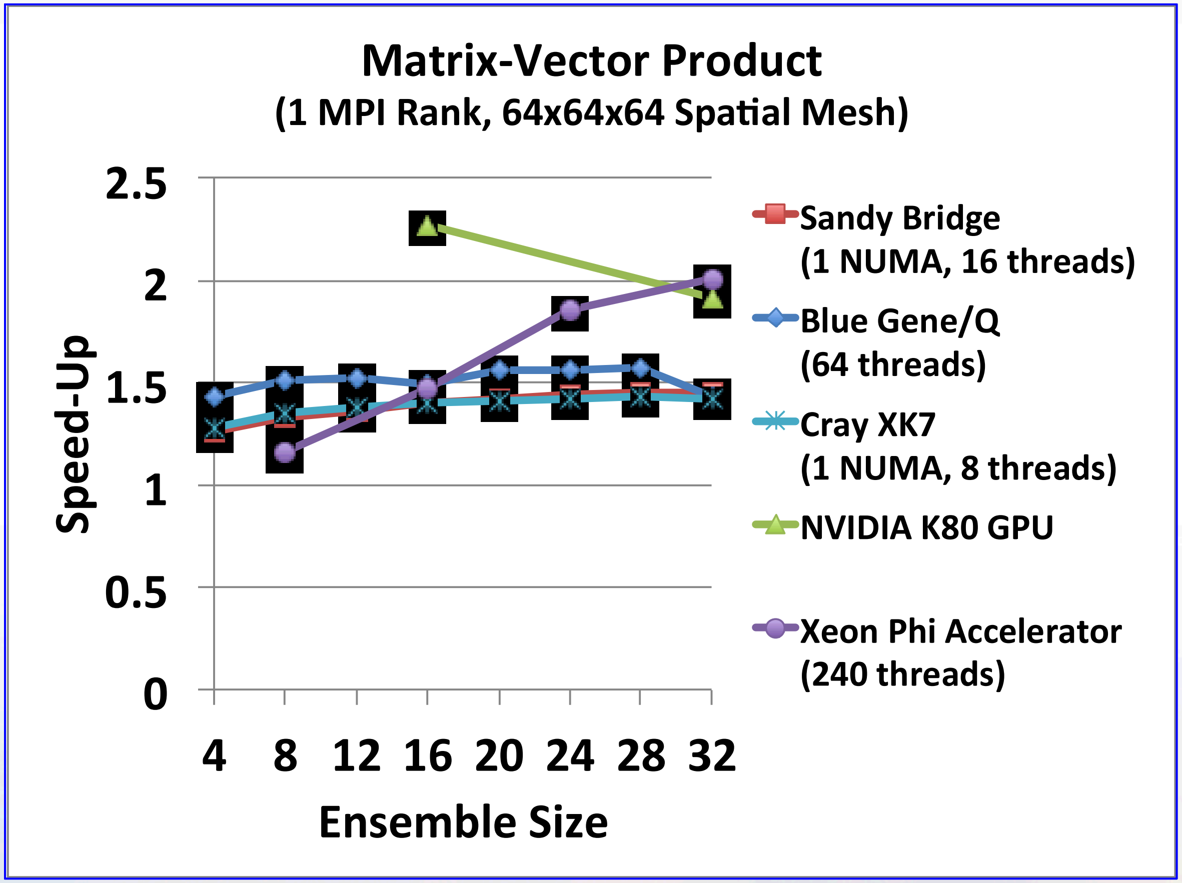
Ensemble PDE Matrix/RHS Assembly Speed-Up



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

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

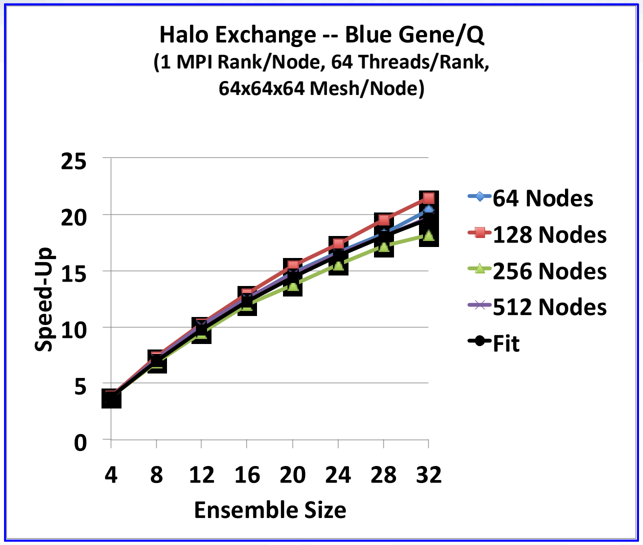
Ensemble Sparse Matrix-Vector Product Speed-Up

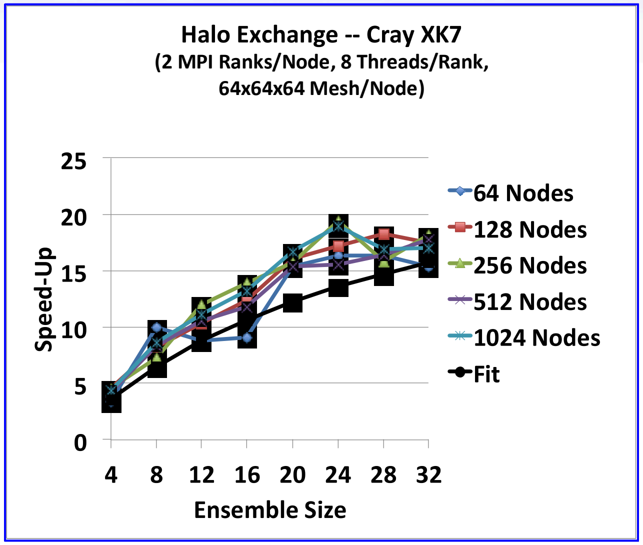


- **Speed-up results from**
  - **Reuse of matrix graph (20%)**
  - **Replacement of sparse gather with contiguous load**
  - **Perfect vectorization of multiply-add**

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

Interprocessor Halo Exchange



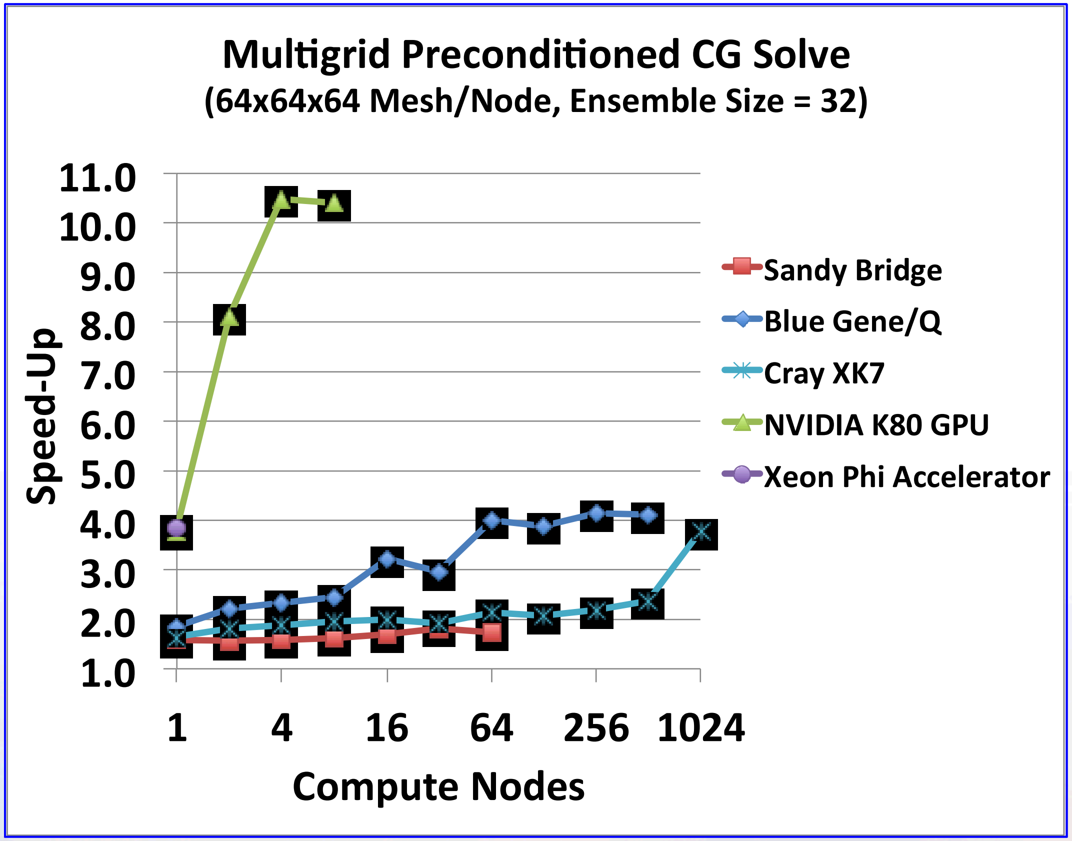


$$\text{Time} \approx a + bm$$

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

- **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 repartioning (Zoltan2)

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

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!

Ensemble Grouping for Anisotropic Diffusion Problems

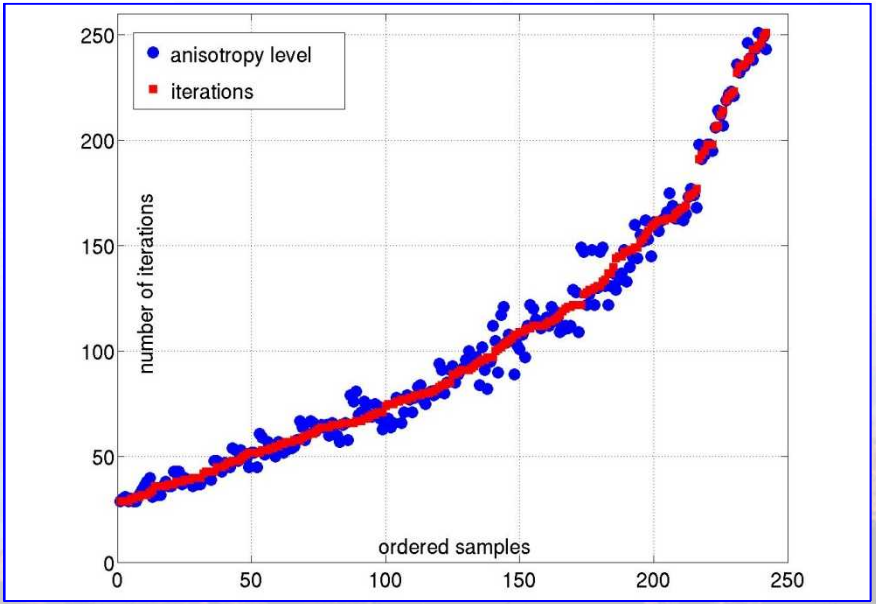
- **Model problem:**

$$-\nabla \cdot (A(\mathbf{x},\mathbf{y}))\nabla u) = f \qquad A(\mathbf{x},\mathbf{y}) = diag(a(\mathbf{x},\mathbf{y}), a_2, a_3)$$

$$a(\mathbf{x},\mathbf{y}) = \check{a} + \hat{a} \exp\left\{ \sum_{n=1}^{N} \sqrt{\lambda_n} b_n(\mathbf{x}) y_n \right\}$$

- **Strong anisotropy induces large variation of PCG iterations from sample to sample**
  - **Want to group samples with similar #iterations**

- **Anisotropy level strongly correlated with #iterations:**



Anisotropy level: $\left\| \dfrac{\lambda_{\max}(A(\mathbf{x},\tilde{\mathbf{y}}))}{\lambda_{\min}(A(\mathbf{x},\tilde{\mathbf{y}}))} \right\|_\infty$

64x64 finite element mesh

AMG (ML) Preconditioned CG

Ensemble Grouping for Anisotropic Diffusion Problems

- **Strategy: Order based on anisotropy level, group into ensembles of size *S***

$$R = \frac{S \sum_{i=1}^{\overline{S}} \mathrm{ITS}_i}{\sum_{k=1}^{M} \mathrm{its}_k}$$

ITS: #its for the i^{th} ensemble
its: #its for the k^{th} sample

RF covariance	*S*	*R (anisotropy ordering)*	*R* (no-ordering)
Gaussian	8	1.374	1.793
Gaussian	16	1.469	2.197
Gaussian	32	1.652	2.852
Exponential	8	1.274	1.448
Exponential	16	1.337	1.673
Exponential	32	1.427	1.847
γ-Exponential	8	1.217	1.503
γ-Exponential	16	1.272	1.794
γ-Exponential	32	1.384	2.223

Summary

- Embedded sampling approach improves aggregate UQ performance by
  - Eliminating sparse memory accesses
  - Amortizing communication/access latency
  - Perfect fine-grained vector/Cuda-thread parallelism

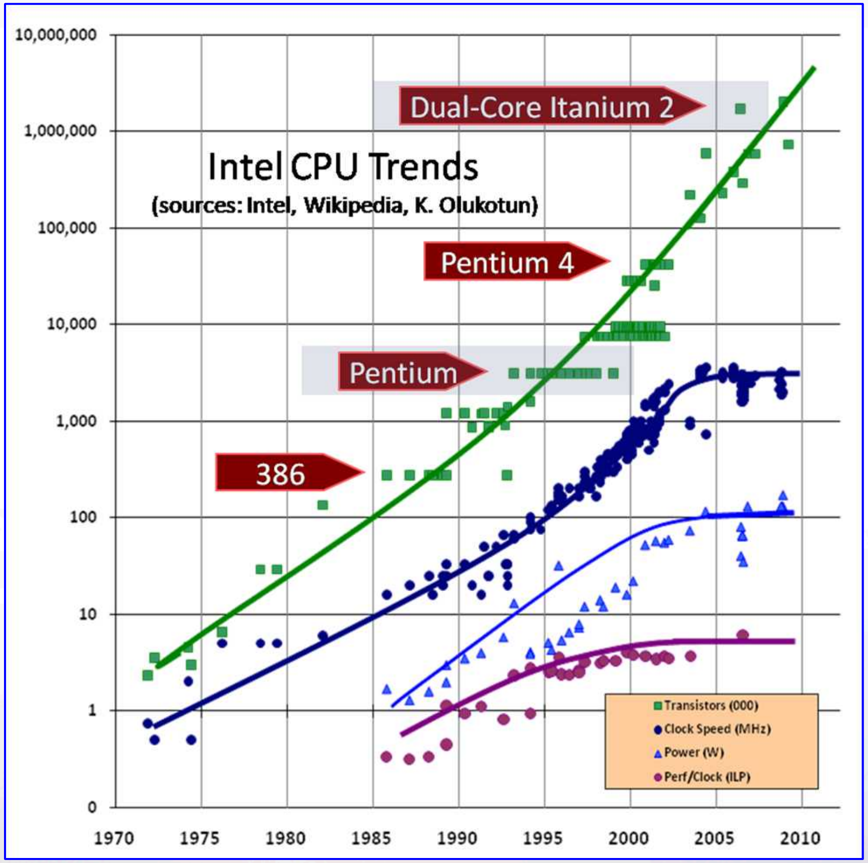
- Applying technique through C++ templates greatly facilitates implementation
  - Alleviate code developers from having to worry about UQ

- Smart grouping of samples into ensembles required for more challenging problems:
  - Effective strategies require understanding dependence of computational cost on uncertain parameter variations
  - Can we do this more generally?
    - A variety of approaches to consider

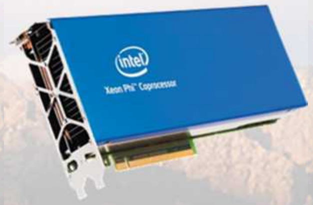
Extra Slides

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. Dobb’s Journal



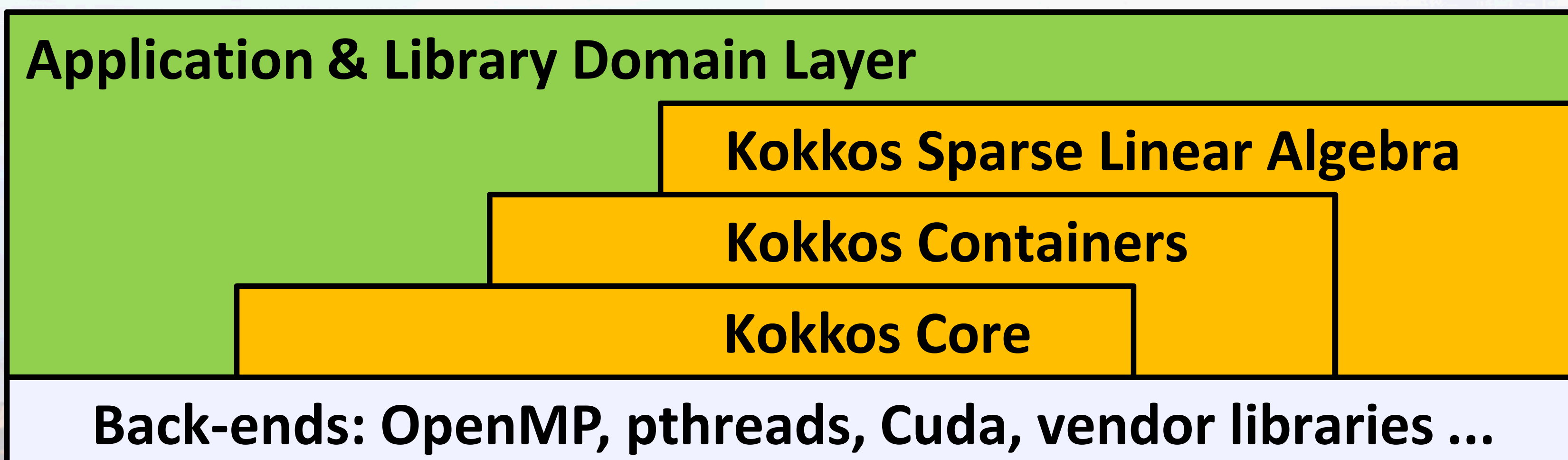
Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications*

- **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

Trilinos

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, …

Trilinos

http://trilinos.sandia.gov

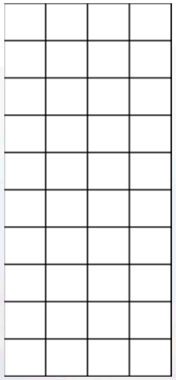
- Higher level solver libraries built on Tpetra
  - Preconditioned iterative algorithms (Belos)
  - Incomplete factorization preconditioners (Ifpack2, ShyLU)
  - Multigrid solvers (MueLu)
  - All templated on the scalar type

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

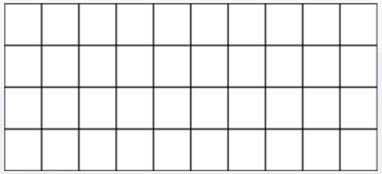
Kokkos Integration

- 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