SAND2014-15555PE

Improving Performance of Sampling-Based Uncertainty Quantification on Advanced Computing Architectures Through Embedded Ensemble Propagation

Eric Phipps (etphipp@sandia.gov)
& H. Carter Edwards
Sandia National Laboratories

SIAM Annual Meeting

July 7-11, 2014

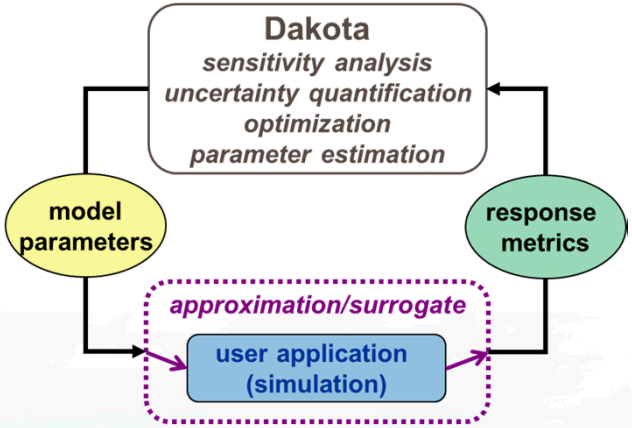
SAND 2014-xxxx C

Forward Uncertainty Quantification

- Uncertainty Quantification (UQ) means many things
  - Best estimate + uncertainty
  - Model validation
  - Model calibration
  - Reliability analysis
  - Robust design/optimization, …
- 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
  - NISP/NIPC
  - Regression PCE (“point/probabilistic collocation”)
  - Stochastic Galerkin, …
- Key challenges:
  - Achieving good accuracy
  - High dimensional uncertain spaces
  - Expensive forward simulations

Emerging Architectures Motivate New Approaches to Uncertainty Quantification

- UQ is a significant driver for exascale computing:
  - 1000x increase in performance would make many UQ problems tractable
- UQ approaches traditionally implemented as an outer loop:

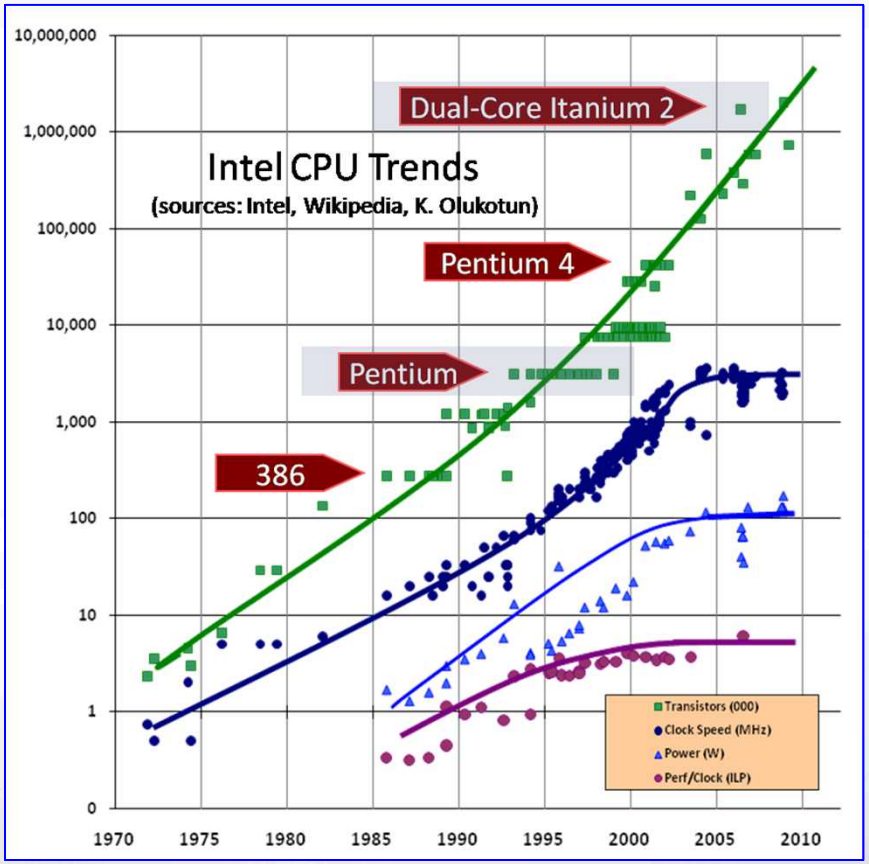


http://dakota.sandia.gov

  - Aggregate UQ performance limited to that of underlying deterministic simulation
- Emerging architectures leading to dramatically increased on-node compute power
  - Will require very good strong scalability to very high thread-counts
- Achieving this is difficult for many PDE simulation problems
  - Poor memory access patterns
  - Inability to expose sufficient fine-grained parallelism
- Can this be remedied by inverting the outer UQ/inner solver loop?
  - Expose new dimensions of parallelism through *embedded* approaches

Computer Architectures Are Changing Dramatically

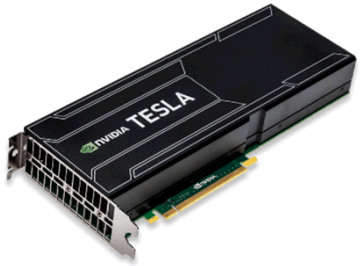
- Historically (super)computers have gotten faster by
  - Decreasing transistor size
  - Increasing clock frequency
  - Making memory appear faster (hiding latency) by
    - Executing multiple instructions simultaneously
    - Reordering instructions on-the-fly
  - Adding more compute nodes that communicate through an interconnect
- These techniques have hit a wall
  - Nearing physical limits on transistor sizes
  - Pumping up frequency makes the chips run hotter, which requires too much power to cool them
  - Adding more compute nodes increases power usage, failure rate



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

Performance increases are instead being achieved by increasing fine-grained parallelism

- Lots of *node-level* parallelism at lower clock frequencies
  - NVIDIA K40:
    - 896 double precision scalar cores @ 0.7 GHz
    - Scheduler issues instructions to groups of 32 cores (1 warp) at a time (SIMT)
  - Xeon Phi:
    - 61 8-wide double precision cores @ 1.2 GHZ
    - Compiler/programmer issues instructions to operate on all components of the vector simultaneously (SIMD/vectorization)



- Lots of independent threads of execution that can easily be swapped in and out to hide latency
  - No out-of-order execution and limited instruction-level parallelism

- Restrictive memory access patterns
  - Access memory chunks commensurate with SIMD/SIMT size
  - Limited fastest cache size shared amongst many threads

Keys to Achieving Maximal Algorithmic Performance on Modern Architectures

- All modern architectures perform arithmetic on short vectors
  - Intel Sandy/Ivy Bridge CPU: 4-wide double precision (AVX)
  - Blue Gene/Q CPU: 4-wide double precision (QPX)
  - Intel Xeon Phi Accelerator: 8-wide double precision (MIC)
  - NVIDIA GPU: 32-wide single/double precision

- To achieve maximum performance, an algorithm must vectorize well:
  - CPU/MIC:
    - Auto-vectorization by compiler
    - Explicit vector intrisics
  - GPU:
    - Each “vector lane” (actually GPU thread) programmed explicitly

- Load/store contiguous regions of memory
  - Architectures always load whole cache lines when accessing any data from memory
    - CPU/MIC: 64 Bytes, GPU 128 Bytes
  - When values are stored contiguously, loading a full vector can have cost as low as accessing a single scalar value
    - MIC: 8 doubles in one instruction
    - GPU: 32 floats, 16 doubles in one instruction

- As architectures evolve, these features will become more and more important
  - No increase in scalar floating-point throughput
  - No decrease in memory latency

UQ in this environment

- Solving PDEs on complex domains is a challenge in this environment
  - Unstructured meshes
  - Sparse linear algebra
- And that makes UQ for these problems a challenge
  - Traditional approach is to run independent simulations on disjoint subsets of compute nodes
  - Therefore aggregate performance is limited to any single simulation
- Take a holistic view of the entire UQ workflow
  - Single-point forward simulation is no longer the end-point
  - Codes are being rewritten for these architectures, why not treat UQ as the basic unit of calculation?
- Uncertainty propagation is often a better structured calculation than the original simulation
  - Lots of reuse of data from simulation to simulation
    - e.g., spatial mesh, matrix graph
  - Many UQ methods rely on (local) smoothness, so data generated by solution process is often similar across samples
    - E.g., preconditioner, Krylov space
- What if we propagated a collection of samples (ensemble) simultaneously?
  - See Giering and Vossbeck, 2012.

Simultaneous ensemble propagation

- **PDE:**

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

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

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

- **Commute Kronecker products (just a reordering of DoFs):**

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

- **Each sample-dependent scalar replaced by length-*m* array**
  - **Automatically reuse non-sample dependent data**
  - **Sparse accesses amortized across ensemble**
  - **Math on ensemble naturally maps to vector arithmetic**

Potential Speed-up for PDE Assembly

```
import(u) // halo exchange

for e = 0 to N_elem do

  // Sparse gather of global solution
  for i = 0 to N_node do
    I = NodeIndex(e,i)
    u_e(i) = u(I)
  end for

  // Evaluate element residual/Jacobian
  f_e = local_residual(u_e)
  J_e = local_jacobian(u_e)

  // Sparse scatter into global residual/Jacobian
  for i = 0 to N_node do
    I = NodeIndex(e,i)
    atomic_add(f(I), f_e(i))
    for j = 0 to N_node do
      J = ElemGraph(e,i,j)
      atomic_add(J(I,J), J_e(i,j))
    end for
  end for
end for
```

- **Halo exchange**
  - Amortize MPI latency across ensemble
- **Gather**
  - Reuse node-index map (mesh)
  - Replace sparse with contiguous loads
- **Local residual/Jacobian**
  - Vectorized math
- **Scatter**
  - Reuse node-index map and element graph (mesh)
  - Replace sparse with contiguous stores

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)

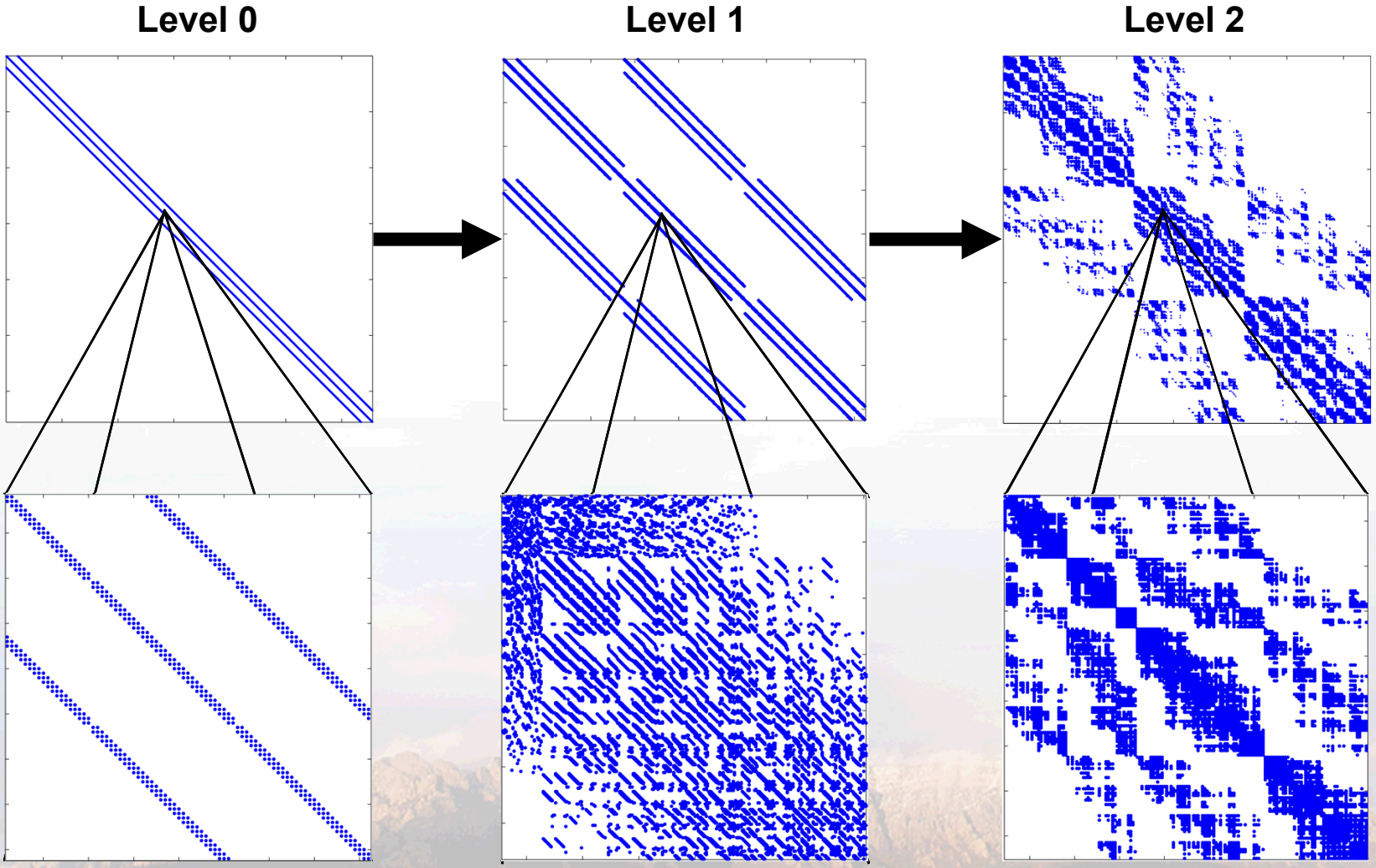
- Sparse matrix-vector products
  - Amortize MPI latency in halo exchange
  - Reuse matrix graph
  - Replace sparse with contiguous loads
  - Vector arithmetic

- Dot-products
  - Amortize MPI latency

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

Algebraic Multigrid Generates Unstructured Matrices*

Linear FEM discretization of 3-D Laplacian $\nabla \cdot \nabla u$



*Matrices courtesy of J. Hu (SNL)

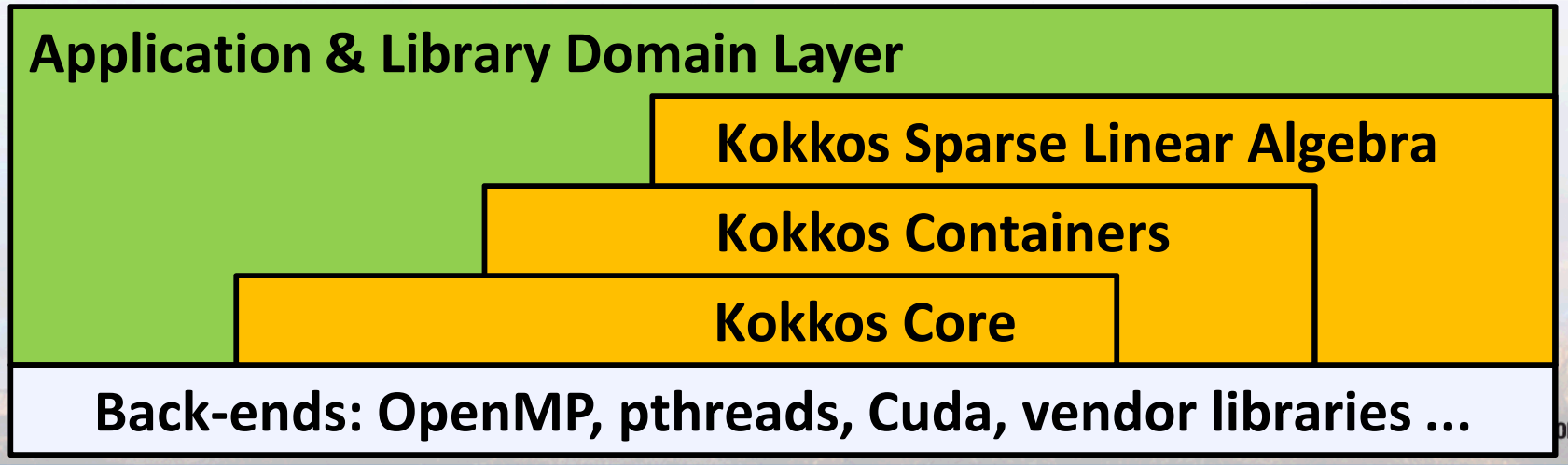
Kokkos: A Manycore Device Performance Portability Library for C++ HPC Applications*

- 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)



Stokhos: Trilinos Tools for Embedded UQ Methods

- Trilinos tool originally developed for stochastic Galerkin methods

http://trilinos.sandia.gov

- Provides "ensemble scalar type"
  - C++ class containing an array with length fixed at compile-time
  - Overloads all math operations by mapping operation across array

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

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

- Integrated with Kokkos for many-core parallelism
  - Specializes Kokkos data-structures, execution policies to map vectorization parallelism across ensemble
  - For CUDA, currently requires manual modification of parallel launch to use customized execution policies

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**

http://trilinos.sandia.gov

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

Techniques Prototyped in FENL Mini-App

- Simple nonlinear diffusion equation

$$-\kappa \nabla^2 u + u^2 = 0$$

  - 3-D, linear FEM discretization
  - 1x1x1 cube, unstructured mesh
  - KL-like random field model for diffusion coefficient

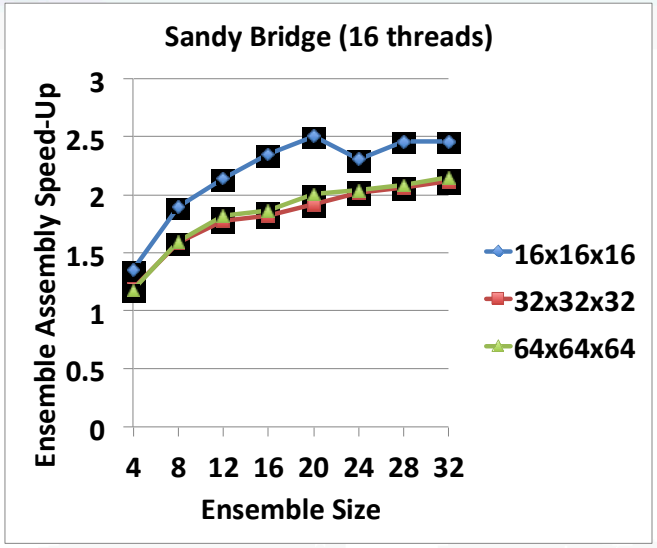
http://trilinos.sandia.gov

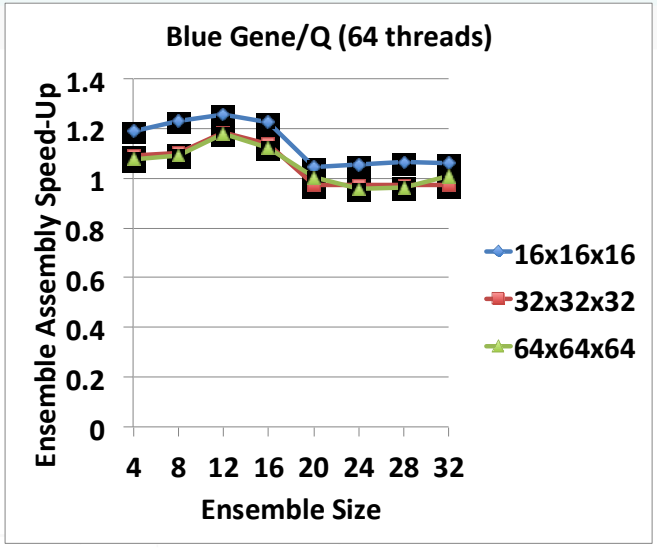
- Hybrid MPI+X parallelism
  - Traditional MPI domain decomposition using threads within each domain
- Employs Kokkos for thread-scalable
  - Graph construction
  - PDE assembly
- Employs Tpetra for distributed linear algebra
  - CG iterative solver (Belos package)
  - Smoothed Aggregation AMG preconditioning (MueLu)
- Supports embedded ensemble propagation via Stokhos through entire assembly and solve
  - Samples generated via tensor product & Smolyak sparse grid quadrature

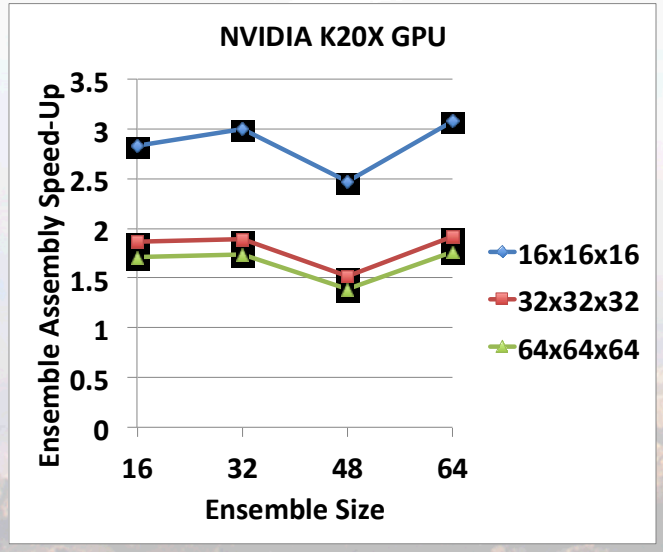
Multicore Architectures Studied

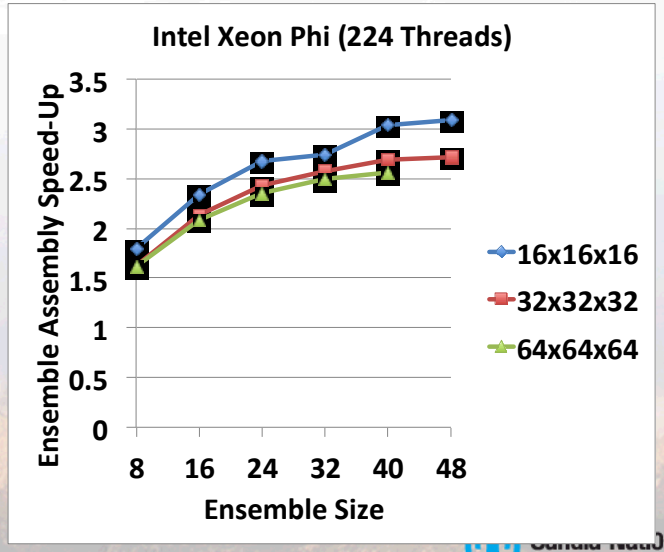
- CPU – Dual-socket Intel Sandy Bridge
  - 16 cores/threads (hyperthreads disabled)
  - 4-wide (double precision) vector ISA (AVX)
- CPU – Blue Gene/Q
  - 16 cores x 4 threads/core = 64 threads
  - 4-wide (double precision) vector ISA (QPX)
  - Vector instructions not supported by GNU compiler (IBM compiler is not C++ standard compliant)
- GPU – NVIDIA Kepler K20X
  - 1430 GFLOPS peak floating point
  - 12 GB global memory
  - 288 GB/s global memory bandwidth
  - 1.5 MB L2 cache, 48 kB shared (L1 cache) memory
- Accelerator: Intel Xeon Phi
  - Pre-production hardware
  - 56 (usable) cores x 4 threads/core = 224 threads
  - 8-wide (double precision) vector ISA, no scalar ISA

Ensemble Assembly Speed-Up

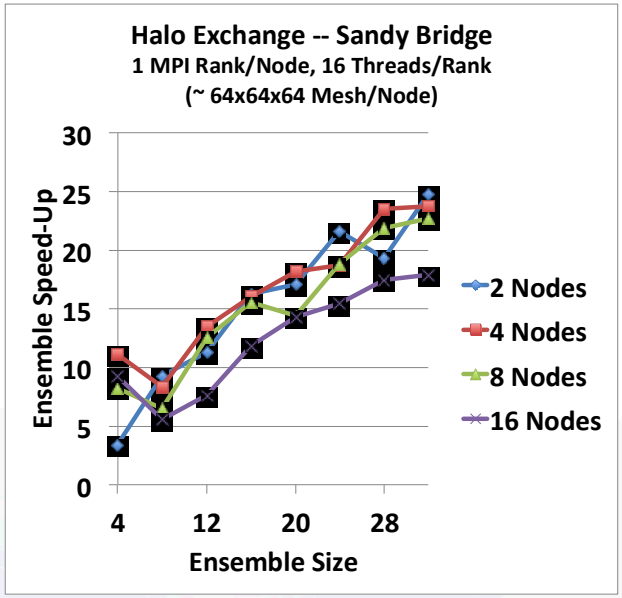


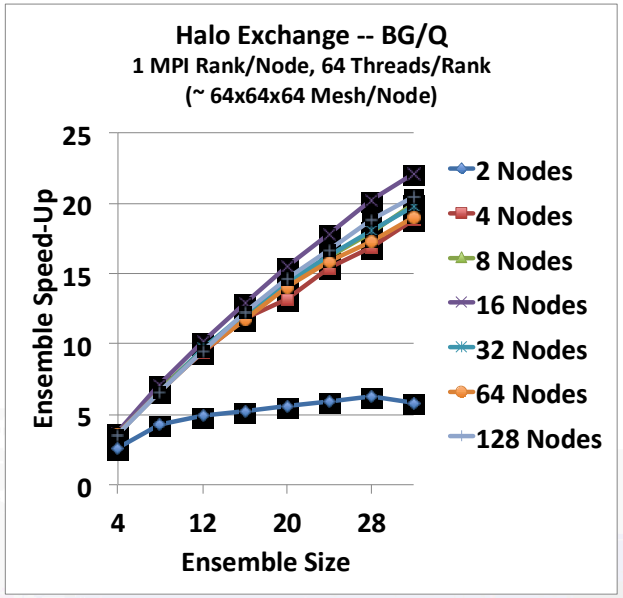




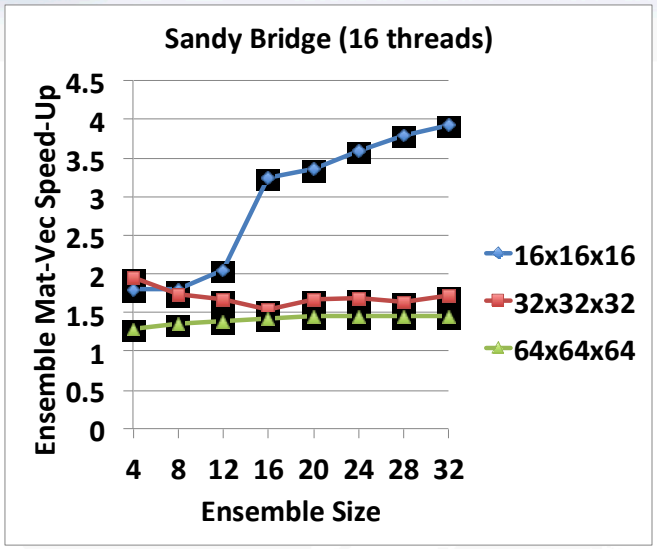


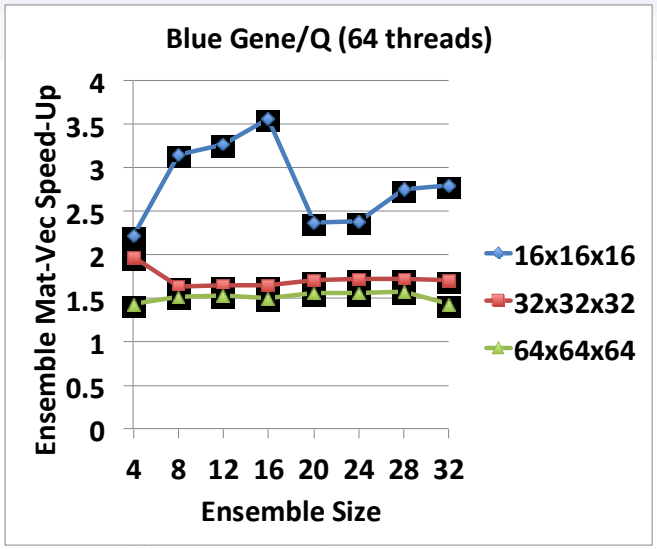
Ensemble MPI Halo-Exchange Speed-Up

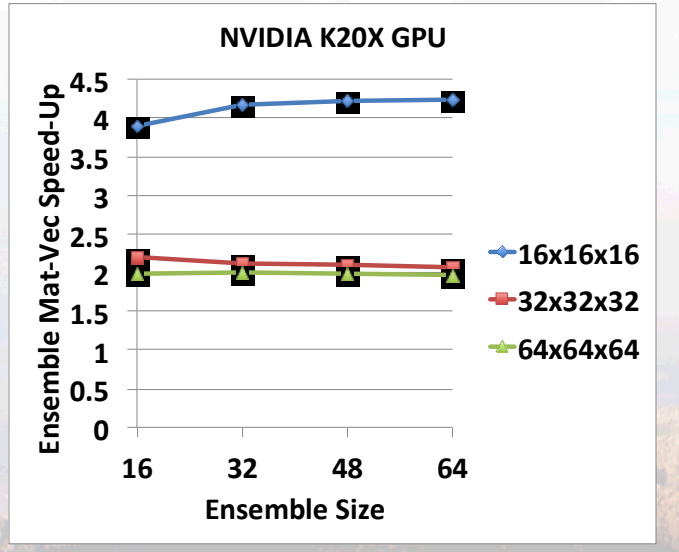


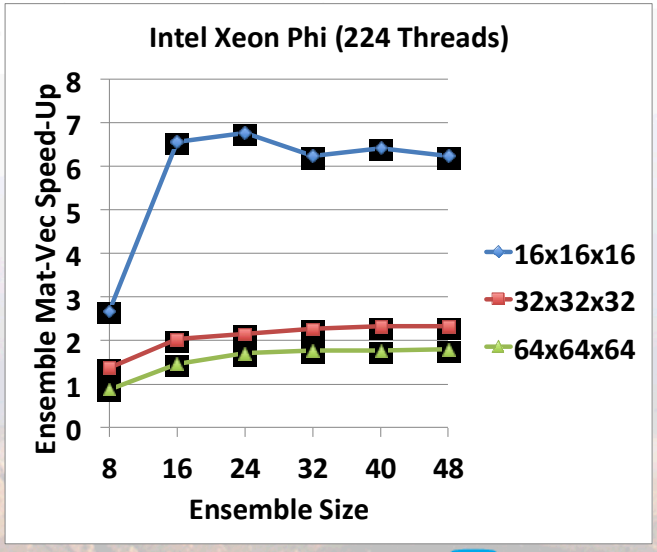


Ensemble Matrix-Vector Product Speed-Up

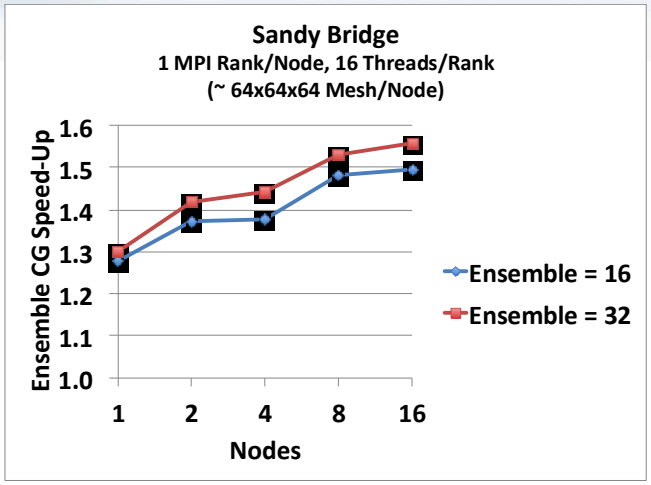


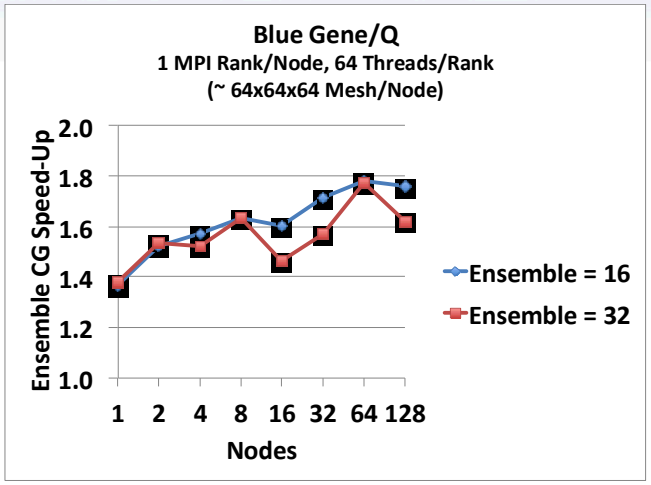


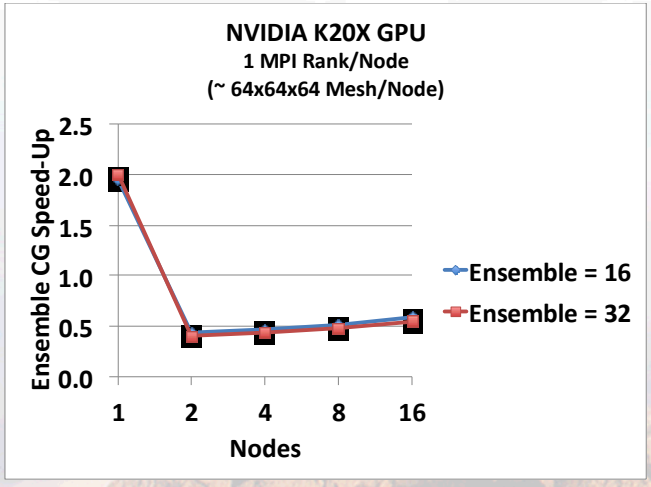


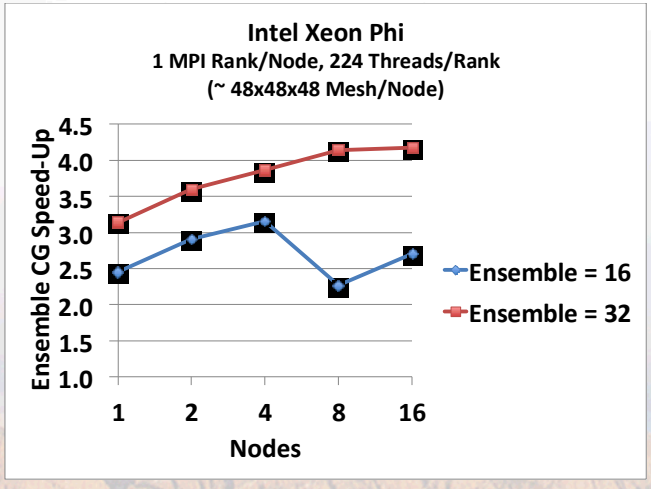


Ensemble CG Speed-Up



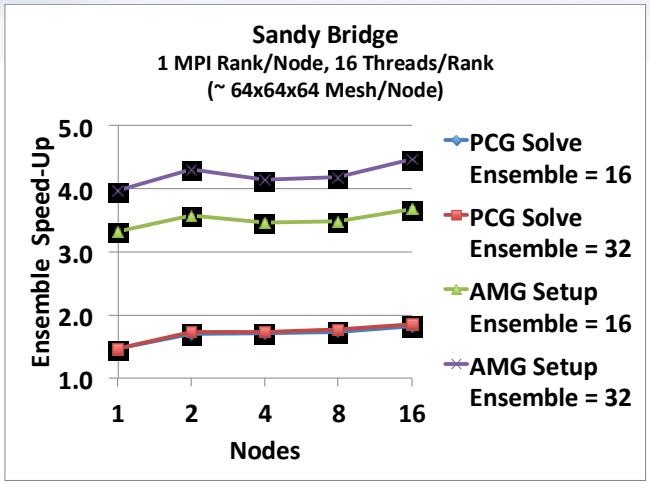


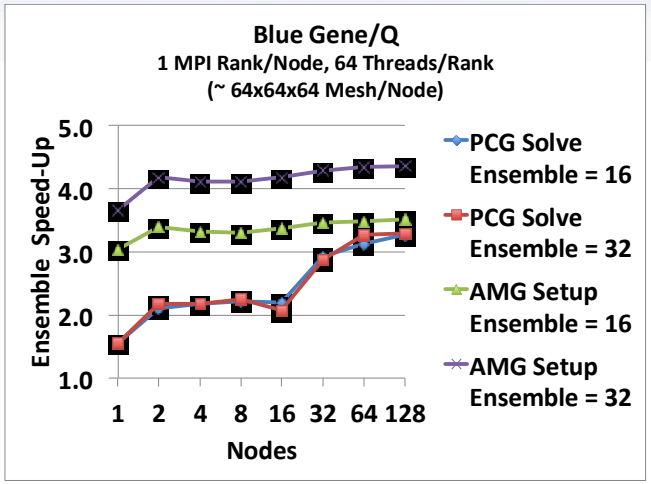


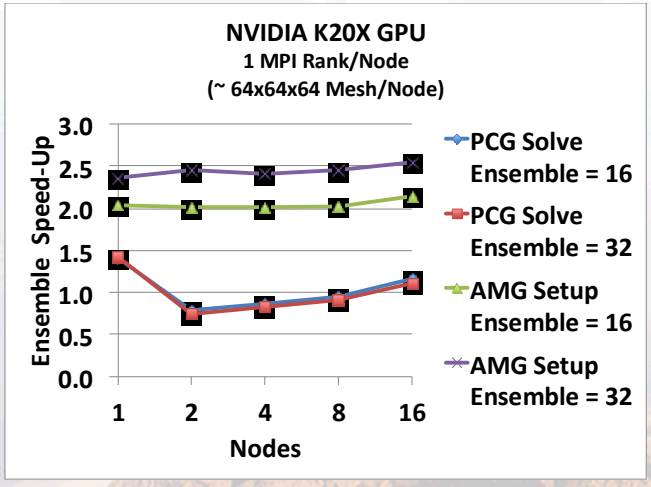


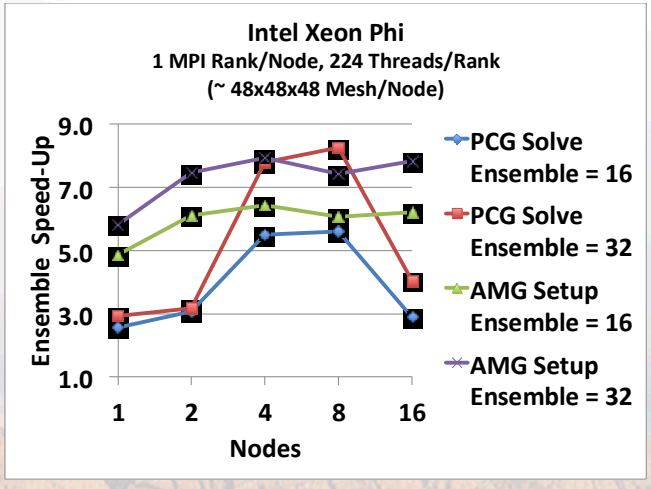
Problem with current implementation that will be fixed soon

Ensemble AMG-Preconditioned CG Speed-Up









Problem with current implementation that will be fixed soon

Concluding Remarks

- Results demonstrate substantial improvements in performance are possible by propagating multiple samples together
  - However these are “fake” UQ problems where all samples are the same
  - But results provide an upper bound on performance

- What happens when this is applied to real UQ samples?
  - Will number of CG iterations increase substantially?
  - What is the effectiveness of propagating ensembles directly through preconditioners?
  - What about other approaches, e.g., applying a single-point preconditioner across an ensemble?
  - See J. Hu talk in Part II of this session, MS45 for answers!

Future Work

- How do we decide when/which samples to propagate together
  - You're not going to do all of them
  - Some things really do change dramatically between some samples
    - Bifurcations
    - Discontinuities
    - Adaptivity (time, spatial)
    - Branches in the simulation code
- Software implementation
  - Fix remaining CUDA kernels not optimized for ensembles
  - Investigate approaches that don't require modifying parallel launch