Exploring Emerging Manycore Architectures for Uncertainty Quantification Through Embedded Stochastic Galerkin Methods

Eric Phipps (etphipp@sandia.gov), H. Carter Edwards and Jonathan Hu, Jakob Ostien
Sandia National Laboratories

SIAM Conference on Uncertainty Quantification

March 31-April 3, 2014

SAND 2014-1143C
Forward UQ

- 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

- UQ approaches usually implemented as an outer loop
  - Repeated calls of deterministic solver

- Single-point forward simulations use very little available node compute power (unstructured, implicit)
  - 3-5% of peak FLOPS on multi-core CPUs (P. Lin, Charon, RedSky)
  - 2-3% on contemporary GPUs (Bell & Garland, 2008)

- Emerging architectures leading to dramatically increased on-node compute power
  - Not likely to translate into commensurate improvement in forward simulation
  - Many simulations/solvers don’t contain enough fine-grained parallelism

- Can this be remedied by inverting the outer UQ/inner solver loop?
  - Add new dimensions of parallelism through embedded approaches
Outline

• Polynomial chaos-based UQ approaches
  – Non-intrusive spectral projection (NISP/NIPC)
  – Stochastic Galerkin (SG)

• Tools for implementing SG methods in large-scale PDE codes

• Application to model 3-D mechanics problems

• Reordering SG mat-vecs for contemporary multicore architectures
Polynomial Chaos Expansions (PCE)

- Steady-state finite dimensional model problem:
  
  $$u(\xi) \text{ such that } f(u, \xi) = 0, \xi : \Omega \rightarrow \Gamma \subset \mathbb{R}^M, \text{ density } \rho$$

- (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
  - Need to be careful to ensure quadrature rule preserves discrete orthogonality
    - SPAM (Constantine, Eldred, Phipps, CMAME, 2012)
    - Method is equivalent to stochastic collocation
Embedded Stochastic Galerkin UQ Methods

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

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

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

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

- Advantages:
  - Many 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 Linear Systems

- Stochastic Galerkin Jacobian:
  \[
  \frac{\partial F}{\partial U} \approx A = \sum_{k=0}^{P} G_k \otimes A_k, \quad A_k = \frac{1}{\langle \psi_k^2 \rangle} \int_{\Gamma} \frac{\partial f}{\partial u}(\hat{u}(y), y) \psi_k(y) \rho(y) dy, \quad G_k(i, j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}
  \]

- 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:

  \[
  Y = AX \implies \sum_{i=0}^{P} e_i \otimes y_i = \left( \sum_{k=0}^{P} G_k \otimes A_k \right) \left( \sum_{j=0}^{P} e_j \otimes x_j \right)
  \]

  \[
  \implies y_i = \sum_{j=0}^{P} \sum_{k=0}^{P} A_k x_j C_{ijk}, \quad C_{ijk} = G_k(i, j) = \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_i^2 \rangle}
  \]

- Sparsity determined by triple product tensor
- Only requires operator-apply for each operator PCE coefficient
- Organize algorithm to minimize operator-vector applies
Stokhos: Trilinos Tools for Embedded Stochastic Galerkin UQ Methods

- Eric Phipps, Chris Miller, Habib Najm, Bert Debusschere, Omar Knio

- Tools for describing SG discretization
  - Stochastic bases, quadrature rules, etc...

- C++ operator overloading library for automatically evaluating SG residuals and Jacobians
  - Replace low-level scalar type with orthogonal polynomial expansions
  - Leverages Trilinos Sacado automatic differentiation library
    \[
    a = \sum_{i=0}^{P} a_i \psi_i, \quad b = \sum_{j=0}^{P} b_j \psi_j, \quad c = ab \approx \sum_{k=0}^{P} c_k \psi_k, \quad c_k = \sum_{i,j=0}^{P} a_i b_j \frac{\langle \psi_i \psi_j \psi_k \rangle}{\langle \psi_k^2 \rangle}
    \]

- 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, ...
3-D Linear & Nonlinear Elasticity Model Problems

- 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

Displacement (Mean)

Displacement (Std. Dev.)

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
Structure of Galerkin Operator

- Operator traditionally organized with outer-stochastic, inner-spatial structure
  - Allows reuse of deterministic solver data structures and preconditioners
  - Makes sense for sparse stochastic discretizations

\[ A^{\text{trad}} = \sum_{k=0}^{P} G_k \otimes A_k \]

\[ A^{\text{com}} = \sum_{k=0}^{P} A_k \otimes G_k \]

- For nonlinear problems, makes sense to commute this layout to outer-spatial, inner-stochastic
  - Leverage emerging architectures to handle denser stochastic blocks
Commuted SG Matrix Multiply

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

- Two level algorithm
  - Outer: sparse (CRS) matrix-vector multiply algorithm

\[ \mathcal{N}_A(l) = \{ m \mid A_0(l, m) \neq 0 \} \]

\[ y( , l) = \sum_{m \in \mathcal{N}_A(l)} A( , l, m)x( , m) \]
Committed SG Matrix Multiply

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

- Two level algorithm
  - Outer: sparse (CRS) matrix-vector multiply algorithm

\[ \mathcal{N}_A(l) = \{ m \mid A_0(l, m) \neq 0 \} \]

\[ y(\ , l) = \sum_{m \in \mathcal{N}_A(l)} \quad \text{FEM basis} \]

\[ A(\ , l, m) x(\ , m) \quad \text{FEM bases sum} \]

\[ \quad \text{FEM basis} \quad \text{FEM basis} \quad \text{FEM basis} \]
Commuting SG Matrix Multiply

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

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

\[ \mathcal{A}(l) = \{ m \mid A_0(l, m) \neq 0 \} \quad \mathcal{C}(i) = \{ (j, k) \mid C(i, j, k) \neq 0 \} \]

\[ y(i, l) = \sum_{m \in \mathcal{A}(l)} \sum_{(j, k) \in \mathcal{C}(i)} A(k, l, m) x(j, m) C(i, j, k) \]

- Stochastic basis
- Stochastic bases sum
- Stochastic basis
- Stochastic basis
- Stochastic basis
- Triple product
- FEM basis
- FEM bases sum
- FEM basis
- FEM basis
- FEM basis
Multicore Architectures

- CPU – Dual-socket Intel Sandy Bridge
  - 8 cores/socket x 2 sockets x 2 threads/core = 32 threads, 256 bit ISA (4 doubles)
  - 2 x 4 FLOP/cycle/core x 16 cores x 2.6 G-cycle/s = 332 GFLOP/s
  - 32 kB L1, 256 kB L2 cache/core, 20 MB L3 cache/socket

- GPU – NVIDIA Kepler K40
  - 64 double precision cores/SM x 15 SM = 960 double precision scalar cores
  - 2 x 64 FLOP/cycle/SM x 15 SM x 0.745 G-cycle/s = 1430 GFLOP/s
  - Hierarchical thread structure:
    - Thread blocks must execute independently
    - Each block contains multiple warps (up to 32)
    - Each warp contains 32 threads which are synchronized (SIMT)
  - 12 GB global memory which all threads can access (slow)
  - 1.5 MB L2 cache for whole device (slow)
  - 48 kB shared (L1 cache) memory per SM (fast)
  - Hardware hides latency of global memory access by fast context switching of active warps

- Accelerator: Intel Xeon Phi 7120P
  - 61 cores x 4 threads/core = 244 threads, 512 bit vector ISA (8 doubles)
  - 2 x 8 FLOP/cycle/core x 61 cores x 1.238 G-cycle/s = 1208 GLFLOP/s
  - 32 kB L1, 512 kB L2 cache/core, no L3 cache
  - 12 GB global memory
Multicore-CPU: One-level Concurrency

\[ y(i, l) = \sum_{m \in N_A(l)} \sum_{(j,k) \in N_C(i)} A(k, l, m) x(j, m) C(i, j, k) \]

- Each FEM row “owned” by a CPU thread
  - 2 rows per core on Sandy Bridge
- Owning CPU thread computes \( y(:, l) \)
  - \((j,k)\) loop vectorized (auto-vectorization or intrinsics) for SIMD parallelism
  - Vector width = 4 (AVX) on Sandy Bridge
Intel Sandy Bridge CPU

- Simple 3D linear FEM matrix (size $n = 32 \times 32 \times 32$)
- $N =$ polynomial order (larger $N$, denser blocks)
- Significant speedup of polynomial approach over original algorithm
Multicore-CPU: One-level Concurrency

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

- Each FEM row “owned” by a CPU thread
  - 2 rows per core on Sandy Bridge
- Owning CPU thread computes \( y(:, l) \)
  - \((j,k)\) loop vectorized (auto-vectorization or intrinsics) for SIMD parallelism
  - Vector width = 4 (AVX) on Sandy Bridge
Simple 3D linear FEM matrix (size $n = 32 \times 32 \times 32$)

$N = $ polynomial order (larger $N$, denser blocks)

Significant speedup of polynomial approach over original algorithm
Manycore-GPU: Two-level Concurrency

\[ y(i, l) = \sum_{m \in N_A(l)} \sum_{(j, k) \in N_C(i)} A(k, l, m)x(j, m)C(i, j, k) \]

- **Multiple levels of concurrency:**
  - Each FEM row owned by a thread-block
  - Each warp within a thread-block owns an “i”
  - Warps within a thread perform SG multiply in parallel, executing FEM multiply loop serially
- **Sparse tensor stored in GPU global memory**
  - Reduce sparse tensor reads by blocking FEM column loop (“m” loop)
  - Heuristic to choose block size based on stochastic discretization size to balance shared memory usage (reduces occupancy) and tensor reads
  - Pack (ij) indices into single 32-bit integer
NVIDIA K40 GPU

- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Too much shared memory usage per CUDA block reduces occupancy
Manycore-GPU: Two-level Concurrency

- Multiple levels of concurrency:
  - Each FEM row owned by a thread-block
  - Each warp within a thread-block owns an “i”
  - Warps within a thread perform SG multiply in parallel, executing FEM multiply loop serially

- Sparse tensor stored in GPU global memory
  - Reduce sparse tensor reads by blocking FEM column loop ("m" loop)
  - Heuristic to choose block size based on stochastic discretization size to balance shared memory usage (reduces occupancy) and tensor reads
  - Pack (i,j) indices into single 32-bit integer
NVIDIA K40 GPU

- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Too much shared memory usage per CUDA block reduces occupancy
**Manycore-Accelerator: Two-level Concurrency**

- **core parallel**
  \[ y(i, l) = \sum_{m \in \mathbb{N}_A(l)} \sum_{(j,k) \in \mathbb{N}_C(i)} A(k, l, m) x(j, m) C(i, j, k) \]
- **hyperthread parallel**
- **vector parallel**
- **core L1 cache memory**
- **Accelerator global memory**

- **Map GPU to accelerator architecture:**
  - GPU thread -> vector lane
  - Thread warp -> hyperthread
  - Thread block -> core

- **Use essentially same algorithm as for GPU, except**
  - Automatic caching of A, x entries instead of shared-memory loads
  - Fixed block size for blocking of FEM column loop ("m" loop)
  - No packing of (i,j) indices
Intel Xeon Phi 7120P Accelerator

- Simple 3D linear FEM matrix (size n = 32x32x32)
- N = polynomial order (larger N, denser blocks)
- Significant speedup of polynomial approach except for larger stochastic discretizations
  - Calculation falls out of L1 cache
Concluding Remarks

- With proper software infrastructure stochastic Galerkin methods are feasible in large-scale codes
  - Stokhos/Trilinos solver tools

- Stochastic Galerkin method exhibits reasonable performance for small to moderate numbers of random variables

- Reordering layout improves performance of SG matrix-vector product\(^1\)
  - Significant improvement with commuted multiply
  - Significant additional levels of parallelism

- Ideas are being incorporated into Trilinos tools for next-generation architectures:
  - Kokkos (Edwards, Sunderland, Trott) and Tpetra (Heroux, Baker, Hoemmen)

- Mat-vecs are only a small part of the picture
  - Preconditioning
  - Interfacing this to “real” UQ algorithm (adaptivity, anisotropy, ...)