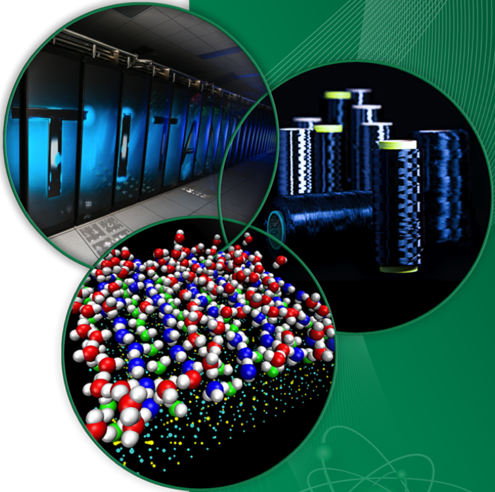
# Parallel Algorithms for Monte Carlo Linear Solvers




Stuart Slattery
Steven Hamilton
Tom Evans (PI)

Oak Ridge National Laboratory

March 17, 2015

# Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research program.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

OAK RIDGE National Laboratory

# Motivation

- As we move towards exascale computing, the rate of errors is expected to increase dramatically
  - The probability that a compute node will fail during the course of a large scale calculation may be near 1
- Algorithms need to not only have increased concurrency/scalability but have the ability to recover from hardware faults
  - Lightweight machines
  - Heterogeneous machines
  - Both characterized by low power and high concurrency

# Towards Exascale Concurrency and Resiliency

- Two basic strategies:
  1. Start with current "state of the art" methods and make incremental modifications to improve scalability and fault tolerance
     - Many efforts are heading in this direction, attempting to find additional concurrency to exploit
  2. Start with methods having natural scalability and resiliency aspects and work at improving performance (e.g. Monte Carlo)
     - Soft failures introduce an additional stochastic error component
     - Hard failures potentially mitigated by replication
     - Concurrency enabled by several levels of parallelism

# Outline

- Monte Carlo Linear Solvers
- Domain Decomposition and Replication
- Scaling Studies
- Algorithm Variations
- Current/Future Work and Conclusions

# Monte Carlo Methods

# Monte Carlo for Linear Systems

- Suppose we want to solve $\mathbf{A}\mathbf{x} = \mathbf{b}$
- If $\rho(\mathbf{I} - \mathbf{A}) < 1$, we can write the solution using the Neumann series

$$\mathbf{x} = \sum_{n=0}^{\infty} (\mathbf{I} - \mathbf{A})^n \mathbf{b} = \sum_{n=0}^{\infty} \mathbf{H}^n \mathbf{b}$$

  where $\mathbf{H} \equiv (\mathbf{I} - \mathbf{A})$ is the Richardson iteration matrix
- Build the Neumann series stochastically

$$x_i = \sum_{k=0}^{\infty} \sum_{i_1}^{N} \sum_{i_2}^{N} \dots \sum_{i_k}^{N} h_{i,i_1} h_{i_1,i_2} \dots h_{i_{k-1},i_k} b_{i_k}$$

- Define a sequence of state transitions
$$\nu = i \rightarrow i_1 \rightarrow \cdots \rightarrow i_{k-1} \rightarrow i_k$$

# Forward Monte Carlo

- Choose a row-stochastic matrix $\mathbf{P}$ and weight matrix $\mathbf{W}$ such that $\mathbf{H} = \mathbf{P} \circ \mathbf{W}$
- Typical choice (Monte Carlo Almost-Optimal):

$$\mathbf{P}_{ij} = \frac{|\mathbf{H}_{ij}|}{\sum_{j=1}^{N} |\mathbf{H}_{ij}|}$$

- To compute solution component $\mathbf{x}_i$:
  - Start a history in state $i$ (with initial weight of 1)
  - Transition to new state $j$ based probabilities determined by $\mathbf{P}_i$
  - Modify history weight based on corresponding entry in $\mathbf{W}_{ij}$
  - Add contribution to $\mathbf{x}_i$ based on current history weight and value of $\mathbf{b}_j$
- A given random walk can only contribute to a single component of the solution vector with $\mathbf{x} \approx \mathbf{M_{MC}}\mathbf{b}$

# Sampling Example (Forward Monte Carlo)

- Suppose

$$\mathbf{A} = \begin{bmatrix} 1.0 & -0.2 & -0.6 \\ -0.4 & 1.0 & -0.4 \\ -0.1 & -0.4 & 1.0 \end{bmatrix} \rightarrow \mathbf{H} \equiv (\mathbf{I} - \mathbf{A}) = \begin{bmatrix} 0.0 & 0.2 & 0.6 \\ 0.4 & 0.0 & 0.4 \\ 0.1 & 0.4 & 0.0 \end{bmatrix}$$

then

$$\mathbf{P} = \begin{bmatrix} 0.0 & 0.25 & 0.75 \\ 0.5 & 0.0 & 0.5 \\ 0.2 & 0.8 & 0.0 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} 0.0 & 0.8 & 0.8 \\ 0.8 & 0.0 & 0.8 \\ 0.5 & 0.5 & 0.0 \end{bmatrix}$$

- If a history is started in state $3$, there is a $20\%$ chance of it transitioning to state $1$ and an $80\%$ chance of moving to state $2$

# Solving the Heat Equation: Forward Method

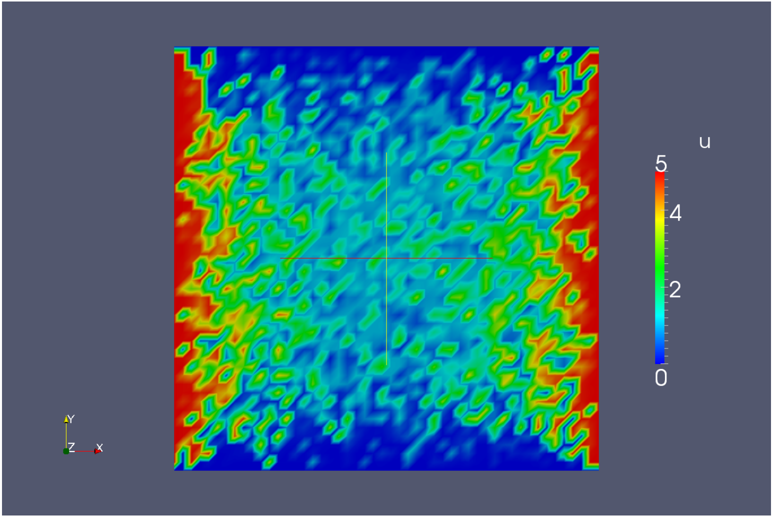



Figure : **Forward solution.** $2.5 \times 10^3$ *total histories.*

OAK RIDGE National Laboratory

# Solving the Heat Equation: Forward Method

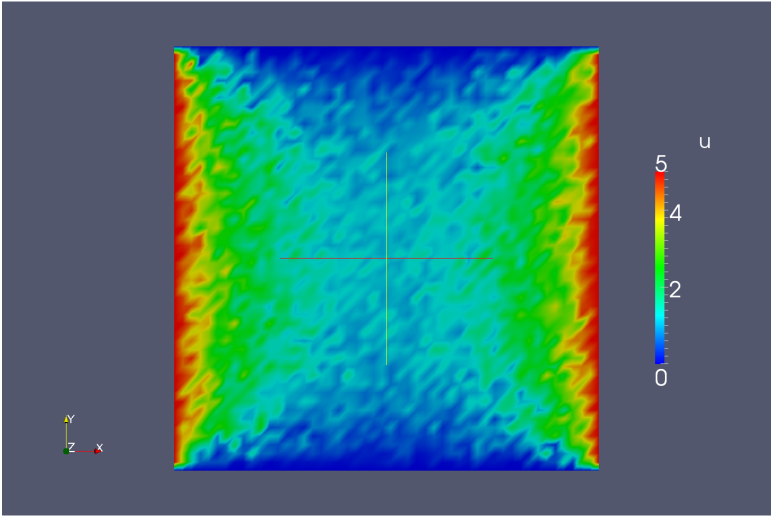



Figure : **Forward solution.** $2.5 \times 10^4$ *total histories.*

OAK RIDGE National Laboratory

# Solving the Heat Equation: Forward Method

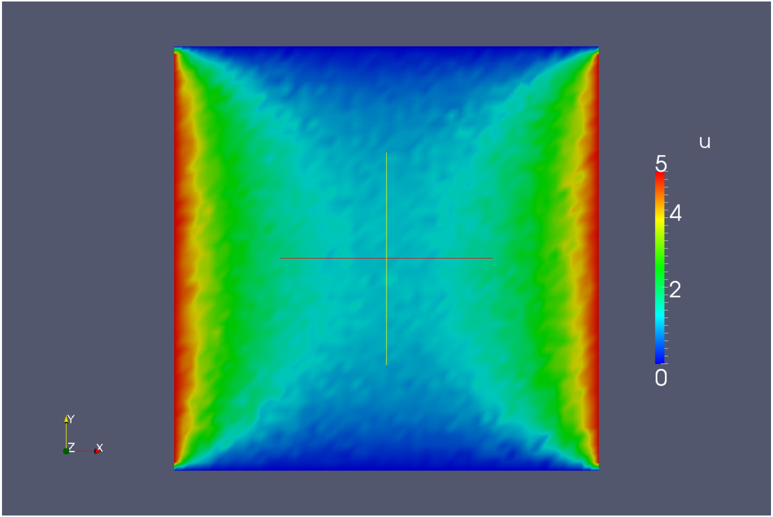



Figure : **Forward solution.** $2.5 \times 10^5$ *total histories.*

# Solving the Heat Equation: Forward Method

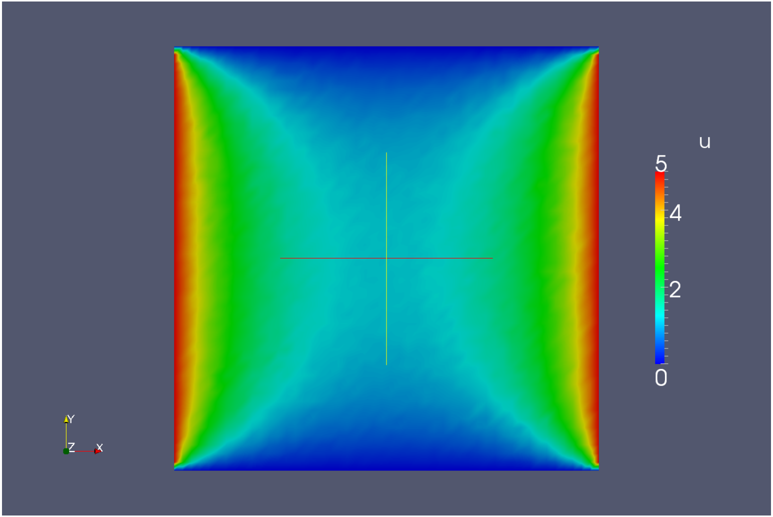



Figure : **Forward solution.** $2.5 \times 10^6$ *total histories.*

Domain Decomposition and Replication

# Domain Decomposed Monte Carlo

- Each parallel process owns a piece of the domain (linear system)
- Random walks must be transported between adjacent domains through parallel communication
- Domain decomposition determined by the input system
- Load balancing not yet addressed

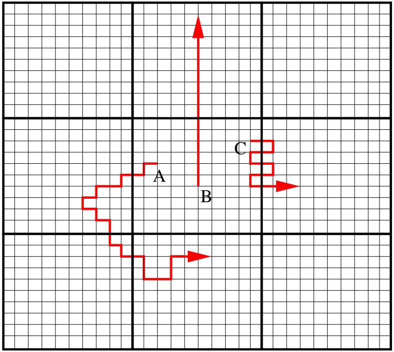
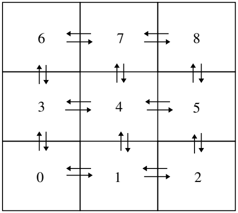
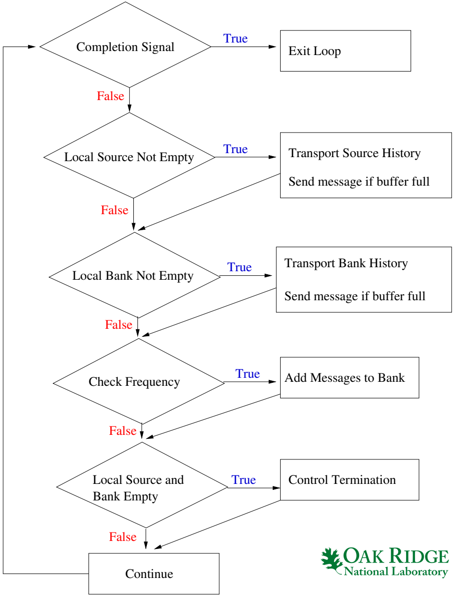



Figure : Domain decomposition example illustrating how domain-to-domain transport creates communication costs.

# Asynchronous Monte Carlo Transport Kernel

- General extension of the Milagro algorithm (LANL)
- Asynchronous nearest neighbor communication of histories
- System-tunable communication parameters of buffer size and check frequency (performance impact)
- Need an asynchronous strategy for exiting the transport loop without a collective (running sum)



# Exiting the Transport Loop without Collectives

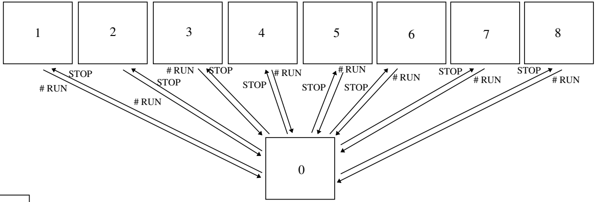



Figure : Master/Slave

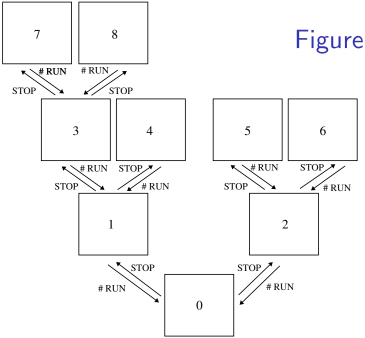



Figure : Binary Tree

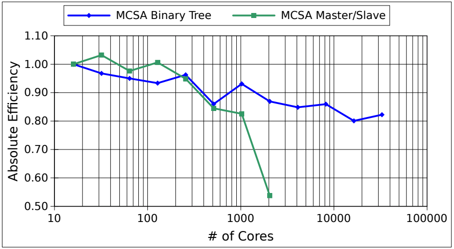



Figure : Weak scaling absolute efficiency

# Replication

Different batches of Monte Carlo samples can be combined in summation via superposition if they have different random number streams. For two different batches:

$$\mathbf{M_{MC}x} = \frac{1}{2}(\mathbf{M_1} + \mathbf{M_2})\mathbf{x}$$

Consider each of these batches independent *subsets* of a Monte Carlo operator where now the operator can be formed as a general additive decomposition of $N_S$ subsets:

$$\mathbf{M_{MC}} = \frac{1}{N_S}\sum_{n=1}^{N_s}\mathbf{M_n}$$

We replicate the linear problem and form each subset on a different group of parallel processes. Applying the subsets to a vector requires an AllReduce to form the sum. Each subset is domain decomposed.

Scaling Studies

# Parallel Test - Simplified $P_N$ $(SP_N)$ Assembly Problem

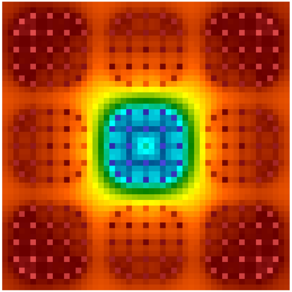


Figure : $SP_N$ solution example

The $(SP_N)$ equations are an approximation to the Boltzmann neutron transport equation used to simulate nuclear reactors

$$-\nabla \cdot \mathbb{D}_n \nabla \mathbb{U}_n + \sum_{m=1}^{4} \mathbb{A}_{nm} \mathbb{U}_m = \frac{1}{k} \sum_{m=1}^{4} \mathbb{F}_{nm} \mathbb{U}_m$$

Scaling problem – $1 \times 1$ to $17 \times 17$ array of fuel assemblies with 289 pins each resolved by a $2 \times 2$ spatial mesh and 200 axial zones

- 7 energy groups, 1 angular moment, 1.6M to 273.5M degrees of freedom
- 64 to 10,800 computational cores via domain decomposition
- We are usually interested in solving generalized eigenvalue problem - we use the operator from these problems to test the kernel scaling

OAK RIDGE National Laboratory

# Monte Carlo Communication Parameters

| | | Message Check Frequency | | | |
|---|---|---|---|---|---|
| | | **128** | **256** | **512** | **1 024** |
| Message Buffer Size | **256** | 1.054 | 1.061 | 1.076 | 1.076 |
| | **512** | 1.103 | 1.146 | 1.211 | $\boxed{1.270}$ |
| | **1 024** | 1.062 | 1.088 | 1.133 | 1.176 |
| | **2 048** | 1.030 | 1.042 | 1.072 | 1.107 |
| | **4 096** | 1.010 | 1.012 | 1.025 | 1.050 |
| | **8 192** | 1.001 | $\boxed{1.000}$ | 1.008 | 1.018 |
| | **16 384** | 1.017 | 1.003 | 1.010 | 1.009 |

- OLCF Eos: 736-node Cray XC30, Intel Xeon E5-2670, 11,776 cores, 47 TB memory, Cray Aries interconnect
- 64 cores, 1.6M DOFs, history length of 15, 3 histories per DOF
- 27% decrease in runtime observed for bad parameter choices
- Worth the time to do this parameter study when running on new hardware

OAK RIDGE National Laboratory

# Monte Carlo Scaling

| Cores | DOFs | DOFs/Core | Time Min (s) | Time Max (s) | Time Ave (s) | Efficiency |
|---|---|---|---|---|---|---|
| 256 | 273 509 600 | 1 068 397 | 260.53 | 260.54 | 260.54 | 1.00 |
| 1 024 | 273 509 600 | 267 099 | 61.92 | 61.92 | 61.92 | 1.05 |
| 4 096 | 273 509 600 | 66 775 | | | | |
| 7 744 | 273 509 600 | 35 319 | | | | |
| 10 816 | 273 509 600 | 25 288 | | | | |

Table : Strong Scaling

| Cores | DOFs | DOFs/Core | Time Min (s) | Time Max (s) | Time Ave (s) | Efficiency |
|---|---|---|---|---|---|---|
| 64 | 1 618 400 | 25 288 | 6.432 | 6.432 | 6.432 | 1.00 |
| 256 | 6 473 600 | 25 288 | 6.493 | 6.493 | 6.493 | 0.99 |
| 1 024 | | 25 288 | | | | |
| 4 096 | | 25 288 | | | | |
| 7 744 | | 25 288 | | | | |
| 10 816 | | 25 288 | | | | |

Table : Weak Scaling

| Subsets | Cores | DOFs | Time Min (s) | Time Max (s) | Time Ave (s) | Efficiency |
|---|---|---|---|---|---|---|
| 1 | 256 | 6 473 600 | 6.493 | 6.493 | 6.493 | 1.00 |
| 2 | 512 | 6 473 600 | | | | |
| 3 | 768 | 6 473 600 | | | | |
| 4 | 1 024 | 6 473 600 | | | | |

Table : Replication Scaling. 256 cores per subset.

## Algorithm Variations

# Monte Carlo Synthetic Acceleration (MCSA)

- Devised by Evans and Mosher in the 2000's as an acceleration scheme for radiation diffusion problems (LANL)
- Can be abstracted as a general linear solver with Monte Carlo as a preconditioner
- First Richardson step hits the high frequency error modes and second Monte Carlo step hits the low frequency error modes

$$\begin{aligned}
\mathbf{r}^k &= \mathbf{b} - \mathbf{A}\mathbf{x}^k \\
\mathbf{x}^{k+1/2} &= \mathbf{x}^k + \mathbf{r}^k \\
\mathbf{r}^{k+1/2} &= \mathbf{b} - \mathbf{A}\mathbf{x}^{k+1/2} \\
\mathbf{x}^{k+1} &= \mathbf{x}^{k+1/2} + \mathbf{M_{MC}}\mathbf{r}^{k+1/2}
\end{aligned}$$

# Matrix-Free Algorithm

- At each application of $\mathbf{M_{MC}}$, execute the Monte Carlo process
- Must perform Monte Carlo every time you want to apply with a better approximation requiring more time and more operations
- Variations in random number streams are amortized over iterations
- Vast majority of solve time spent doing Monte Carlo

| $L$ | $N_S$ | MC Time (s) | MC Fraction | MCSA Iters |
|---|---|---|---|---|
| 3 | 1 | 30.885 | 0.96 | 266 |
| 3 | 2 | 60.869 | 0.98 | 261 |
| 5 | 1 | 27.422 | 0.97 | 180 |
| 5 | 2 | 54.319 | 0.98 | 175 |
| 10 | 1 | 23.871 | 0.98 | 102 |
| 10 | 2 | 45.551 | 0.99 | 97 |
| 15 | 1 | 50.395 | 0.98 | 164 |
| 15 | 2 | 42.951 | 0.99 | 69 |
| 15 | 3 | 65.292 | 0.99 | 68 |
| 25 | 1 | - | - | - |
| 25 | 2 | 70.505 | 0.99 | 78 |
| 25 | 3 | 63.677 | 1.00 | 47 |

Table : MCSA performance. $A$ had $115\,600$ rows and $1\,186\,464$ non-zero entries.

Oak Ridge National Laboratory

# Stochastic Approximate Inverse Algorithm

- Construct $\mathbf{M_{MC}}$ as a sparse matrix by executing the Monte Carlo process once and tallying the row entries
- Use this operator as a stochastic approximation to the inverse
- A better approximation to the inverse requires more setup time and more storage
- We will investigate a drop tolerance strategy to control sparsity

| $L$ | $N_S$ | $NNZ$ | $NNZ$ Ratio | MC Time (s) | Setup Time (s) | MCSA Iters |
|---|---|---|---|---|---|---|
| 3 | 2 | 484 714 | 0.41 | 0.104 | 0.671 | 255 |
| 3 | 3 | 622 123 | 0.52 | 0.145 | 0.705 | 255 |
| 5 | 2 | 783 153 | 0.66 | 0.158 | 0.737 | 185 |
| 5 | 3 | 1 032 573 | 0.87 | 0.237 | 0.831 | 171 |
| 5 | 4 | 1 241 442 | 1.05 | 0.302 | 0.906 | 171 |
| 10 | 3 | 1 969 540 | 1.66 | 0.433 | 1.061 | 95 |
| 10 | 4 | 2 416 572 | 2.04 | 0.570 | 1.214 | 95 |
| 15 | 3 | 2 867 005 | 2.42 | 0.645 | 1.317 | 132 |
| 15 | 4 | 3 544 181 | 2.99 | 0.833 | 1.534 | 67 |
| 15 | 5 | 4 157 269 | 3.50 | 1.029 | 1.765 | 66 |

Table : MCSA Performance. $A$ had $115\,600$ rows and $1\,186\,464$ non-zero entries.

# Unpreconditioned Algorithm Comparison

- No preconditioning, serial computation, fastest MCSA times reported
- GMRES easier to precondition - performance here only indicates Monte Carlo potential
- These results indicate good stochastic approximate inverse performance for traditional CPU architectures
- Matrix-free approach may be more effective when vectorized for new architectures by favoring operations over storage - 95%+ of the runtime spent in Monte Carlo

| Solver | Setup Time (s) | Solve Time (s) | Total Time (s) | Iters |
|---|---|---|---|---|
| Richardson | 2.098 | 1.6709 | 3.769 | 1 017 |
| MCSA Matrix-Free | 2.104 | 24.389 | 26.493 | 102 |
| MCSA Approximate Inverse | 2.953 | 0.779 | 3.731 | 95 |
| Belos GMRES | 1.791 | 1.021 | 2.812 | 81 |

Table : $A$ had $115\,600$ rows and $1\,186\,464$ non-zero entries.

OAK RIDGE National Laboratory

# Current/Future Work and Conclusions

# Current Work - Vectorization and Threading

- We have implemented a Monte Carlo kernel using the Kokkos threading model (Trilinos)
  - The kernel supports multi-threaded CPU, GPU, and Xeon Phi architectures with a single implementation
- Vectorization an area of active research with a focus on heterogeneous architectures (Titan and Summit)
  - Currently investigating event-based algorithms to enable vectorization
  - Event-based algorithm concepts are based on vectorized Monte Carlo algorithms from particle transport
- We are exploring an additive-Schwarz formulation to eliminate parallel communication in the Monte Carlo kernel
- Other threading models will be considered (e.g. HPX)

# Conclusions and Future Work

- Monte Carlo methods offer great potential for both resilient and highly parallel solvers
  - A fully asynchronous algorithm provides scalability without collectives
  - Replication potentially offers resiliency with some overhead
- Matrix-free and stochastic approximate inverse algorithms are complementary - trade operations for storage
- Extending methods to broader problem areas is a significant algorithmic challenge and an attractive area for continued research
  - Explicit preconditioners are required for all problems
- Performance modeling and resiliency simulations this FY
  - Fault injection studies using the xSim simulator

OAK RIDGE National Laboratory