SAND2013-2091P

Sandia National Laboratories

Exceptional service in the national interest

Fault tolerant programming models

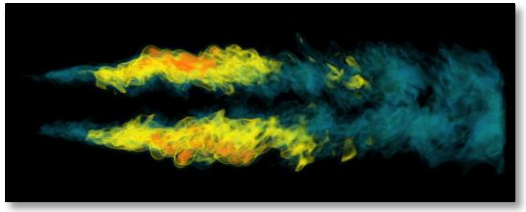
Work by Janine Bennett, John Floren, Nicole Slattengren, Yevgeniy Vorobeychik

January 25, 2013

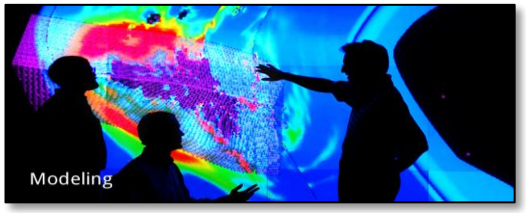
U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

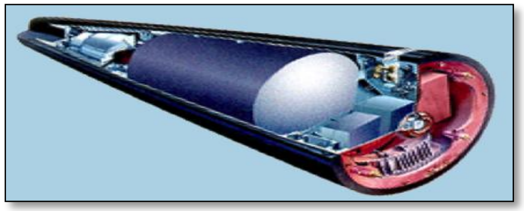
Leadership-class HPC compute capabilities are required for DOE policy and decision making



Energy: Reduce U.S. reliance on foreign energy, reduce carbon footprint



Climate change: Understand, mitigate, and adapt to the effects of global warming



National Nuclear Security: Maintain a safe, secure, and reliable nuclear stockpile

Exascale computing and beyond is required to simulate complex phenomena that characterize the DOE mission space

Resilience is one of the many research challenges posed by the shift to exascale computing

Exascale systems will experience errors/faults much more frequently than petascale systems*

Cause: There is a significant increase in the number of components with insufficient improvements in mean time to failure for each one.

Solution: True exascale resilience requires advancements in

- Fault detection, understanding, and propagation
- Fault recovery
- Fault-oblivious algorithms
- Stress testing of proposed fault-tolerance solutions

*Towards Exascale Resilience, Cappello et al., Intl. Journal of High Performance Computing Applications Nov 2009 vol. 23 no. 4 374-388

Our goal: Discover the right approach for extreme-scale, fault-resilient programming

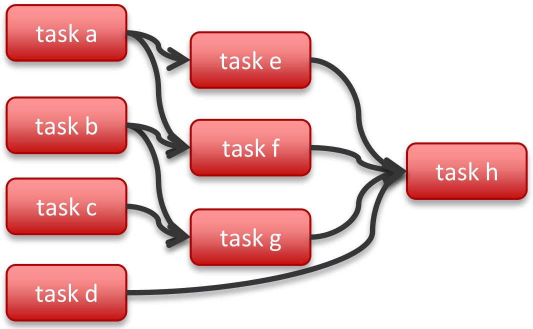
The community needs to understand:

- Can MPI+X offer high scalability at exascale even in the face of faults?
- If not, which programming models can reach which scales?
- If no programming model can reach scales of interest for a given application without algorithmic changes, how might algorithms be adapted?
- What are co-design implications for tradeoffs between memory, I/O, power, resilience, application performance, and development effort?

Existing programming models are inherently not fault-resilient	Asynchronous many-task (AMT) programming models can be fault-resilient
Single Program Multiple Data (SPMD), implicitly synchronous algorithms cannot recover from failure nor adapt well to node degradation	Asynchronous execution and redundancy minimize the impact of node degradation/failure and benefit scalability even without failure
Global check-points no longer feasible	Synergistic with local check-pointing

To achieve our goal, we must explore challenges impeding the use of AMT programming models

- How does one intuitively express tasks to achieve asynchronous execution?
  - Task-Directed Acyclic Graphs (task-DAGs) can depict data dependencies and flow; however may not be enumerable until run-time



- What is the best approach for resilient, decentralized scheduling of tasks?
  - How can missing task subgraphs be regenerated after failure using asynchronous local check-points?
  - Can our approach to resilience be leveraged to improve scalability and avoid additive cascading delays?

Challenges will be explored via tests on hardware simulators as well as tests on current system architectures

Deliverables for FY13

- Creation of initial metrics for development effort and performance profile analysis for non-MPI programming models
- Validation of simulator performance predictions with realistic application workloads using Cielo and/or other current ASC platforms
- Exploration of the programming model design space to include
  1. Applications that are difficult to load-balance
  2. New failure response strategies
  3. Models of failure histories
  4. Task-DAG scheduling frameworks

FY 13 RESULTS

Specific FY13 efforts

- Understand past and current PM and resilience efforts
- Develop metrics for fair comparisons between MPI and non-MPI programming models both with and without faults
- Validation of simulator performance predictions of the cellular automaton on Cielo
- Develop a shared-memory task-DAG API/runtime and add in-memory redundancy and local check-pointing capabilities
- Port a simple conjugate gradient mini-app to our model
- Start extending the task-DAG API/runtime to work in a distributed-memory environment
- Concurrently, starting porting more realistic mini-apps, starting with mini-FE

Some alternative programming models efforts that involve Task-DAGs

- DAGuE (Bosilca et al. 2012)
  - Task-DAG scheduling framework for distributed, many-core arch
  - Custom build tools pre-compile a compact, problem-size independent representation of the DAG
  - Uses dynamic, fully-distributed asynchronous scheduler based on cache awareness, data locality, and task priority
- Intel Threading Building Blocks
  - C++ template library for dynamic, task-based thread parallelism
- Chunks and Tasks (Rubensson & Rudberg, preprint 2012)
  - Task-DAG library for distributed, many-core architechtures
  - Scheduling based on relationships between data and work
  - Able to handle hierarchical, recursive algorithms
  - Resilience by redundant storage and replaying of failed tasks

Some alternative programming models efforts that do not involve Task-DAGs

- Chapel/Fortress/X10: new programming languages; no direct path forward from legacy codes
- UPC: SPMD, PGAS language extension of C; number of threads fixed at execution time
- Global Arrays: library-based PGAS model that supports incremental checkpointing and a pool of spare nodes
- Global Futures: library-based APGAS model built on top of GA
- Charm++: object-oriented, message-driven; supports adaptive load balancing and automatic checkpointing based on migratable objects
- ParalleX/HPX: object-oriented, message-driven, AGAS; scheduler and AGAS server are single points of failure

Specific FY13 efforts

- Understand past and current PM and resilience efforts
- Develop metrics for fair comparisons between MPI and non-MPI programming models both with and without faults
- Validation of simulator performance predictions of the cellular automaton on Cielo
- Develop a shared-memory task-DAG API/runtime and add in-memory redundancy and local check-pointing capabilities
- Port a simple conjugate gradient mini-app to our model
- Start extending the task-DAG API/runtime to work in a distributed-memory environment
- Concurrently, starting porting more realistic mini-apps, starting with mini-FE

Performance metrics for comparisons between MPI and non-MPI programming models

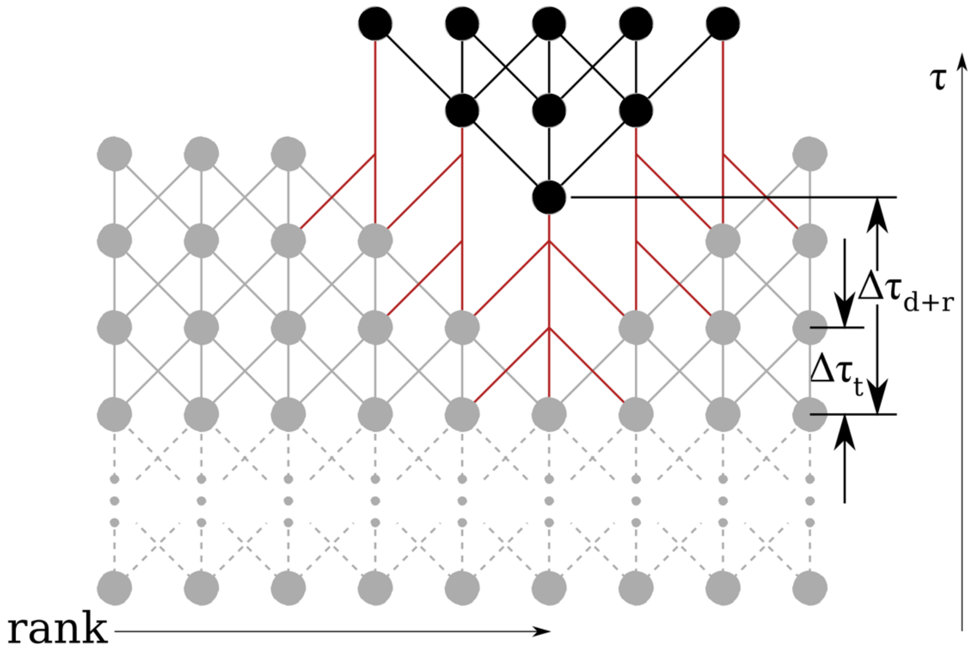
- Perform comparisons both in the absence of failures and under different rates of failure
- Use failure models where failed nodes either leave the computation permanently (fail-stop) or rejoin after some delay (fail-restart)
- For different classes of applications (e.g., CPU-bound, memory-bound, I/O-bound), measure scalability and performance in terms of:
  - Time to completion (including checkpoint/restart)
  - Progress made (e.g. iterations completed) in a period of time
  - Processor utilization
  - Communication cost

Specific FY13 efforts

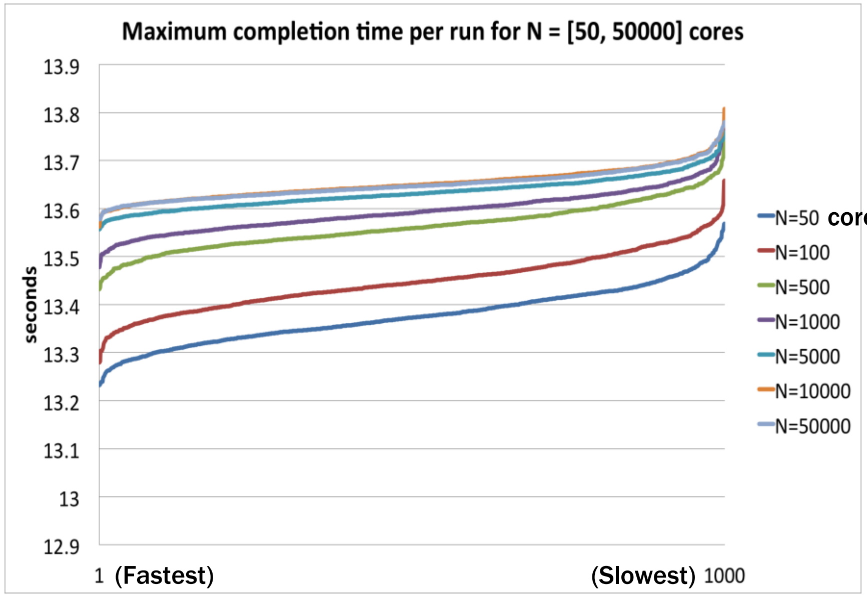
- Understand past and current PM and resilience efforts
- Develop metrics for fair comparisons between MPI and non-MPI programming models both with and without faults
- Validation of simulator performance predictions of the cellular automaton on Cielo
- Develop a shared-memory task-DAG API/runtime and add in-memory redundancy and local check-pointing capabilities
- Port a simple conjugate gradient mini-app to our model
- Start extending the task-DAG API/runtime to work in a distributed-memory environment
- Concurrently, starting porting more realistic mini-apps, starting with mini-FE

We track the propagation of delays due to failures through the 1D cellular automaton

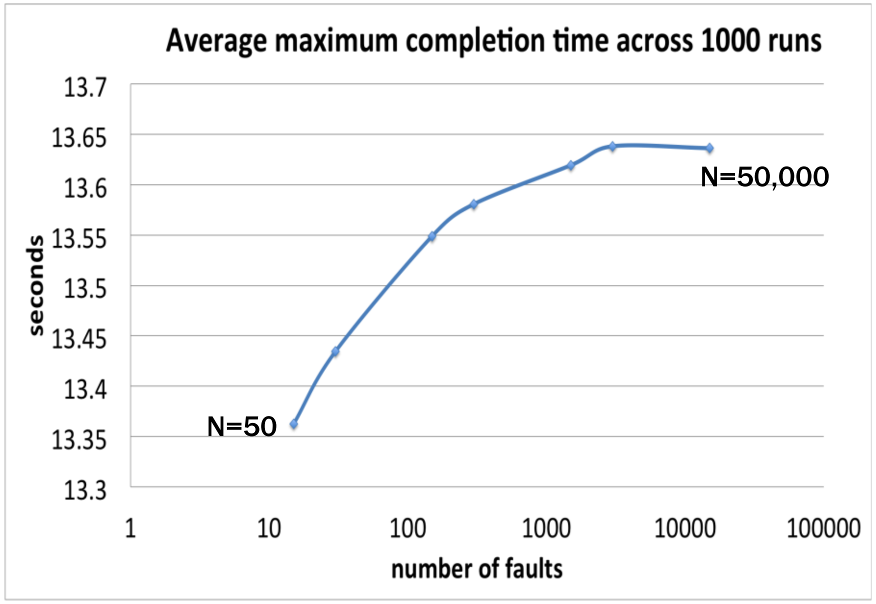
- State of a cell at step k depends on the state of that cell and its two neighbors at step k-1
- Inject fault-induced delays using a Poisson fault model
- Measure the time until all tasks have completed (“maximum completion time”)



Validation of simulator performance predictions for the 1D cellular automaton on Cielo



Run number (sorted by maximum completion time)



Average over runs

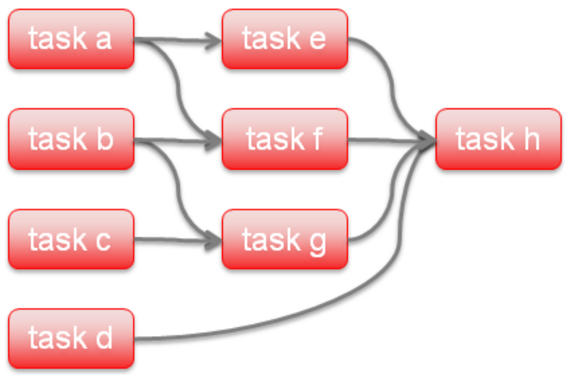
Poisson fault model: # of faults increases with number of cores

- Task-driven automaton code scales up to 50,000 cores on Cielo
- We will use Cielo results to validate the SST/macro simulator performance predictions that we are in the process of collecting

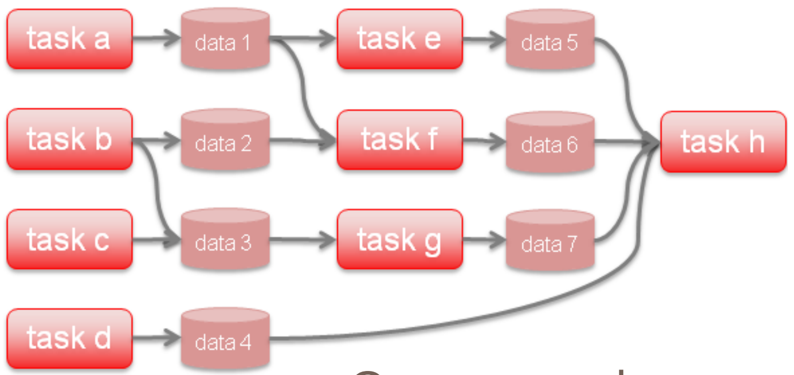
Specific FY13 efforts

- Understand past and current PM and resilience efforts
- Develop metrics for fair comparisons between MPI and non-MPI programming models both with and without faults
- Validation of simulator performance predictions of the cellular automaton on Cielo
- Develop a shared-memory task-DAG API/runtime and add in-memory redundancy and local check-pointing capabilities
- Port a simple conjugate gradient mini-app to our model
- Start extending the task-DAG API/runtime to work in a distributed-memory environment
- Concurrently, starting porting more realistic mini-apps, starting with mini-FE

Our approach: tasks depend only on data; dependencies among tasks are implicit



Typical approach:
DAG of only tasks



Our approach:
Bipartite DAG of
tasks and data

- Explicit dependencies of tasks on data allows automation of data replication, check-pointing, and other FT mechanisms
  - Differs from approaches in DAGuE and TBB; also differs from Tasks and Chunks in that we do not allow tasks to depend on other tasks
  - Leverage resilience work on local check-pointing (ASC)

Our approach: tasks depend only on data; dependencies among tasks are implicit

- Transaction-like semantics of tasks allow them to safely be replayed even when we don't know the exact point of failure
  - Tasks can modify state *only* by producing defined, write-once results
  - If data exists in more than one location, guaranteed to be coherent
  - Similar to the read-only chunk approach of Tasks and Chunks; differs from ParalleX/HPX
  - Leveraging the resilience API from Bob Lucas's group would allow us to respond to failures instead of abort

- The task-DAG API/runtime can be provided as a library, potentially allowing it to integrate with legacy codes
  - Not an entirely new language like Chapel, X10, Fortress

When we move to a distributed-memory environment…

- Dynamic scheduling may allow us to make more intelligent scheduling decisions based on the time needed to retrieve a dependency from a remote node
  - Don’t ignore the cost of data movement
  - Leverage data stores from FOX (DOE ASCR X-Stack) and/or Nessie
- Fully dynamic scheduling may allow us to make most efficient use of resources
  - Resources not fixed at execution time like in MPI and UPC
  - Adapt when nodes drop out due to failure
  - Increase allocation when more resources become available
- We can evaluate our approach at exascale both with and without faults using SST/macro simulation

Specific FY13 efforts

- Understand past and current PM and resilience efforts
- Develop metrics for fair comparisons between MPI and non-MPI programming models both with and without faults
- Validation of simulator performance predictions of the cellular automaton on Cielo
- Develop a shared-memory task-DAG API/runtime and add in-memory redundancy and local check-pointing capabilities
- Port a simple conjugate gradient mini-app to our model
- Start extending the task-DAG API/runtime to work in a distributed-memory environment
- Concurrently, starting porting more realistic mini-apps, starting with mini-FE

Summary

- Exascale systems will experience errors/faults much more frequently than petascale systems
- Our goal is to understand how task-DAG programming models can address both scalability and fault-resilience at exascale
- We believe that the use of a bipartite DAG, with explicit dependencies between tasks and data, opens up opportunities to automate fault-recovery
- We are creating an API/runtime that can be used to experiment with various approaches and demonstrate their effectiveness on problems of interest to ASC

Questions?