Co-Design Center – Xstack Post-docs

Jim Belak (with Andrew Siegel and John Bell)

Lawrence Livermore National Laboratory

CS14-018

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Join us

for the

Lawrence Livermore National Lab

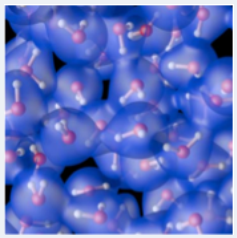
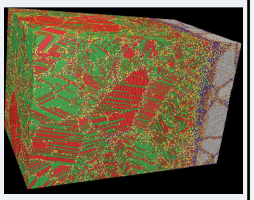
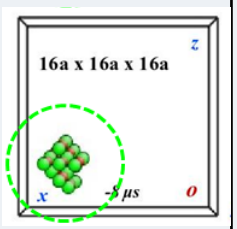
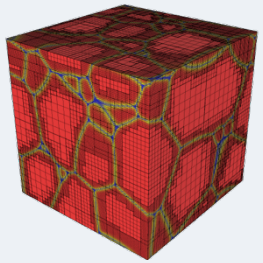
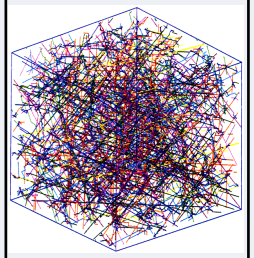
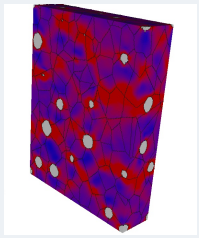
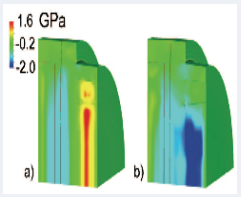
2014 Summer Co-design School

The Computational Co-design Summer School offers graduate students and qualified undergraduates the opportunity to engage in practical research experience to further their educational goals. The focus of the 2014 summer school will be programming for exascale computing. Qualified applicants will recreate scientific proxy application codes in several emerging programming models to assess applicability for exascale computing.

Contact:
James Belak | **belak1@llnl.gov**
Scott Futral | **futral2@llnl.gov**
David Richards | **richards12@llnl.gov**
Martin Schulz | **schulz6@llnl.gov**

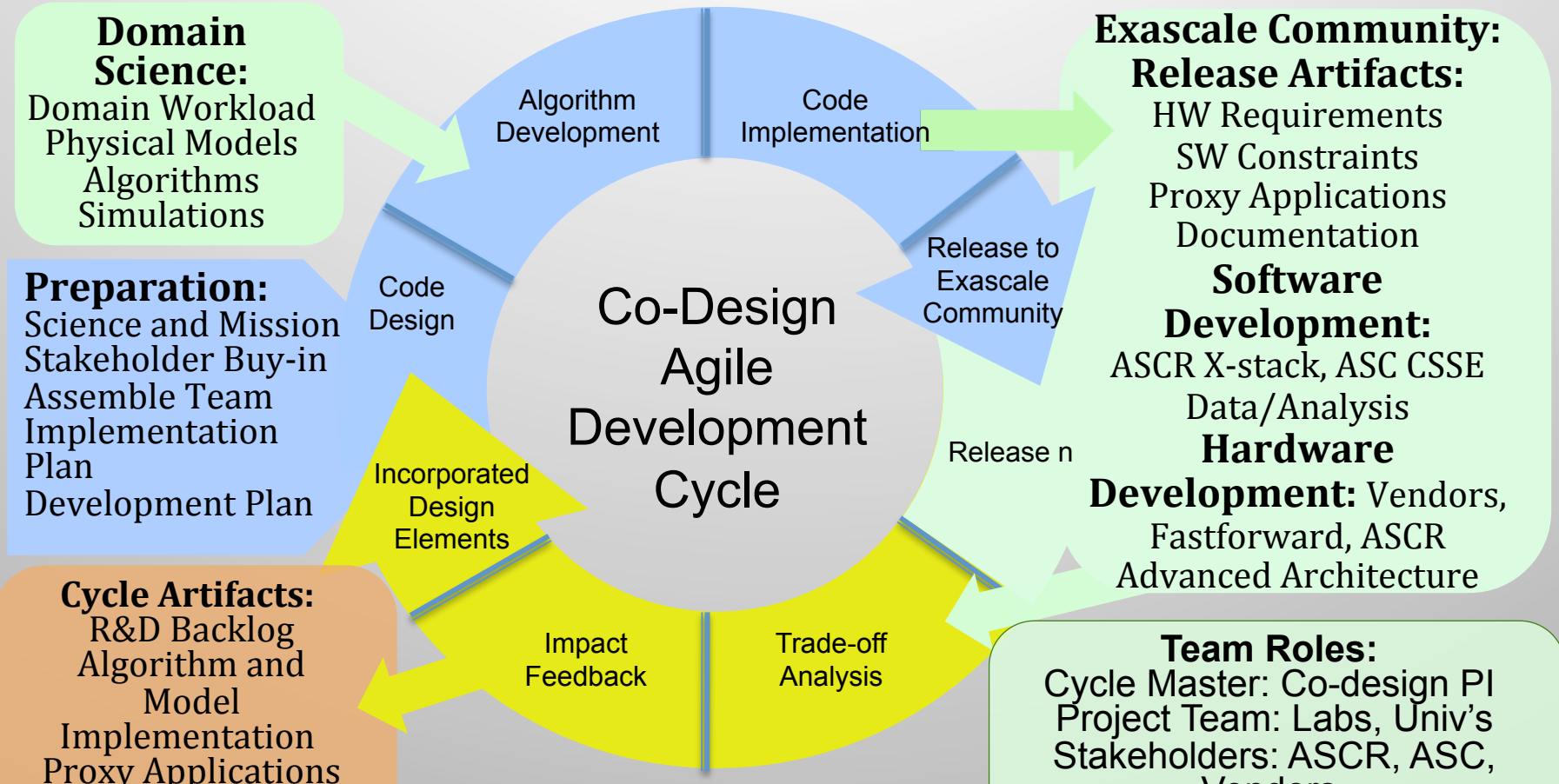
codesign.llnl.gov/exmatex.php
computation.llnl.gov/jobs

The Seven Pillars of Computational Materials Science

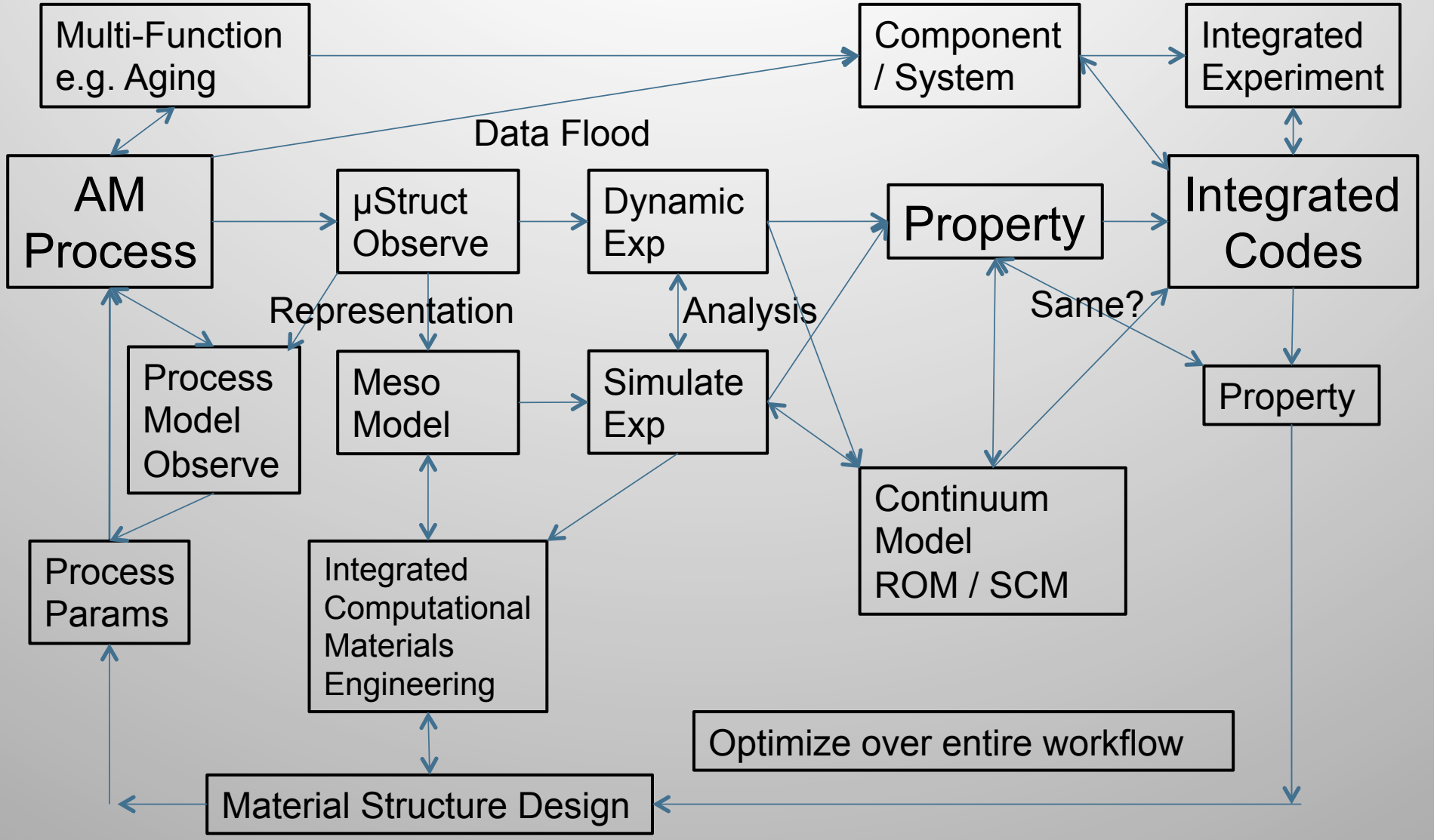
Ab-initio	Atoms	Long-time	Microstructure	Dislocation	Crystal	Continuum
Inter-atomic forces, EOS, excited states	**Defects and interfaces, nucleation**	**Defects and defect structures**	**Meso-scale multi-phase, multi-grain evolution**	**Meso-scale strength**	**Meso-scale material response**	**Macro-scale material response**
						
Code: Qbox/ LATTE Motif: Particles and wavefunctions, plane wave DFT, ScaLAPACK, BLACS, and custom parallel 3D FFTs Prog. Model: MPI + CUBLAS/CUDA	Code:SPaSM/ ddcMD/CoMD Motif: Particles, explicit time integration, neighbor and linked lists, dynamic load balancing, parity error recovery, and *in situ* visualization Prog. Model: MPI + Threads	Code: SEAKMC Motif: Particles and defects, explicit time integration, neighbor and linked lists, and *in situ* visualization Prog. Model: MPI + Threads	Code: AMPE/GL Motif: Regular and adaptive grids, implicit time integration, real-space and spectral methods, complex order parameter Prog. Model: MPI	Code: ParaDiS Motif: "segments" Regular mesh, implicit time integration, fast multipole method Prog. Model: MPI	Code: VP-FFT Motif: Regular grids, tensor arithmetic, meshless image processing, implicit time integration, 3D FFTs. Prog. Model: MPI + Threads	Code: ALE3D/ LULESH Motif: Regular and irregular grids, explicit and implicit time integration. Prog. Model: MPI + Threads

How do we get exascale lessons learned into quotidian science applications (VASP, LAMMPs, …)?

Each co-design project is using *proxy apps* to capture the requirements of their application and reformulating these *proxy apps* from the lessons learned from co-design trade-off analysis.



Use Case / Workflow for Materials Genome Initiative / Advanced Manufacturing (Usage Models, Jeopardy)



General Comments:

- Post-docs have research themes, resilience, data access patterns, data abstraction (tiling), data layout and network sensitivity
- Post-docs most effective interactions with Xstack projects for which there is already a collaboration
- Post-docs most effective when the theme of research is commensurate with theme of Xstack project, e.g. resilience
- Consensus: Post-docs not appropriate for “ambassador” to all Xstack projects, rather, shared deep-dives (hack-a-thons) proved far more effective.
- The problem is not going away and we need other mechanisms, e.g. Summer Schools, Use Cases / Workflows (beyond Proxy Apps)
- Side Question: We’ve been Single-Program Multiple-Data for 20 years. Do we need to go beyond SPMD?
- Side Question 2: Most of our simulation codes are time synchronous. Where can we utilize asynchronous thinking?

PD <-> Theme Mapping

- Ignacio Laguna (ExMatEx) <-> Resilience
- Nan Dun (CESAR) <-> Resilience
- Huiwei Lu (CESAR) <-> data layout, memory access patterns
- Didem Unat (ExaCT) <-> durable data abstraction (tiling)
- Cy Chan (ExaCT) <-> data layout and network sensitivity

Resilient Programing Abstractions for Exascale Computing

ExMatEx (POC: Jim Belak)

May 5, 2014

Post-Doc: Ignacio Laguna

Lawrence Livermore National Laboratory

CS14-018

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Programing Abstractions for Resilient Computing

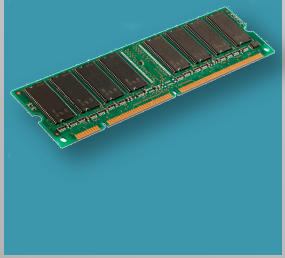
Research Goals:
Develop composable resilient programing abstractions/models
Understand errors better by data gathering, analysis and modeling

Node recovery

* Process & node failures
* **MPI fault tolerance**
* Modeling shrinking recovery
* Application: *ddcMD*

Local recovery

* Recoverable hardware errors
* Example: instruction errors
* **Retry code code region**
* Application: *LULESH*
* Similar: *Containment Domains*



Error data analysis

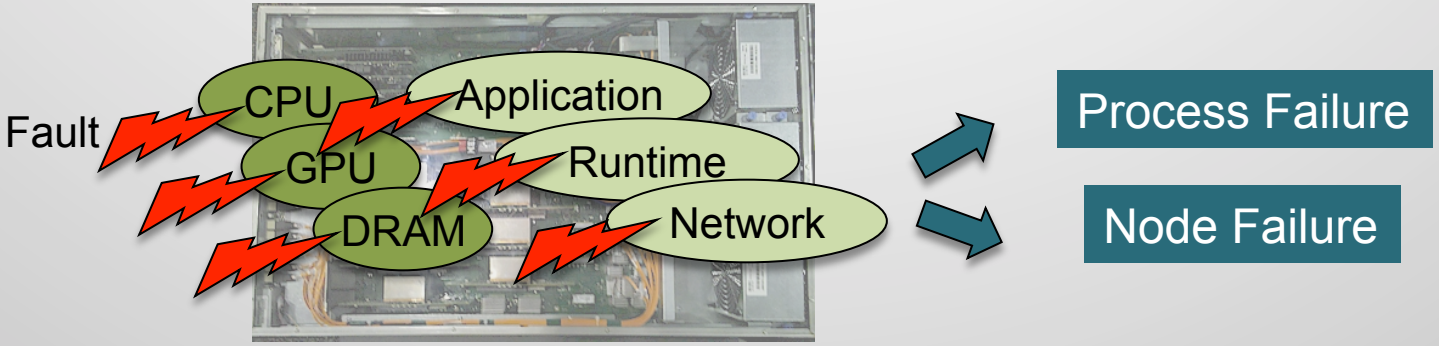
* Gathering DRAM error data
* Modeling errors in current sys.
* Extrapolate error rates
* Analysis of error propagation

Network errors

* Arise from bad circuit / cables
* Recovered by retransmissions
* Alternate interpolation techniques
* Application: *FFT*

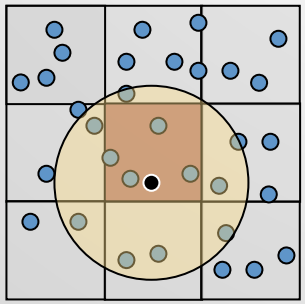
Abstractions for Node/Process Failure Recovery

- Many (if not most) hardware/software errors result in process failures



- Failure rates will increate at Exascale
  - Larger number of hardware/software components
- MPI is the most widely used programing model in HPC
  - *As an example*: 80% of tier-1 CORAL benchmarks use MPI
- MPI provides little (or null) support for process/node failures
  - Little can be done to react to failures (other than aborting)

How are forces calculated in a parallel MD code?



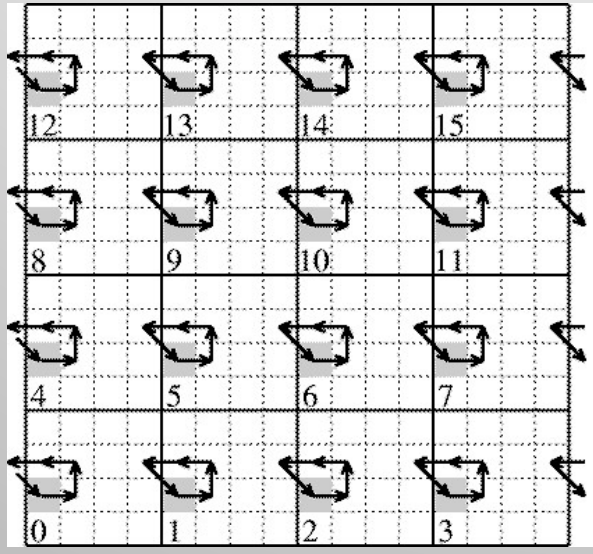
~ 20 atoms in each box

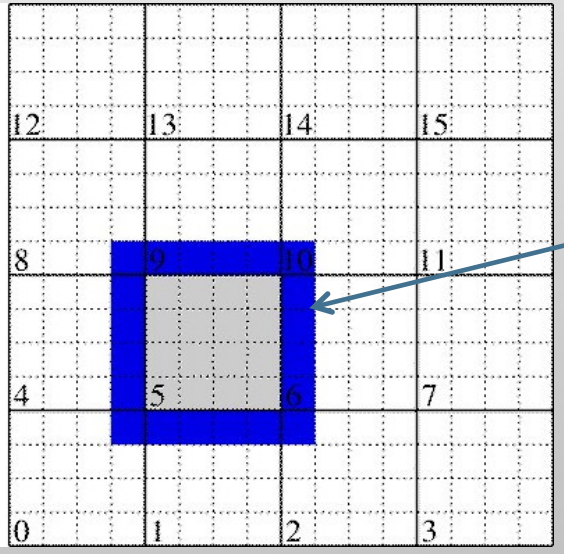
⇒ each atom interacts with 540 other atoms

⇒ However, only ~70 atoms lie within cutoff

⇒ Lots of wasted work

⇒ We need a means of rejecting atoms efficiently even within this reduced set



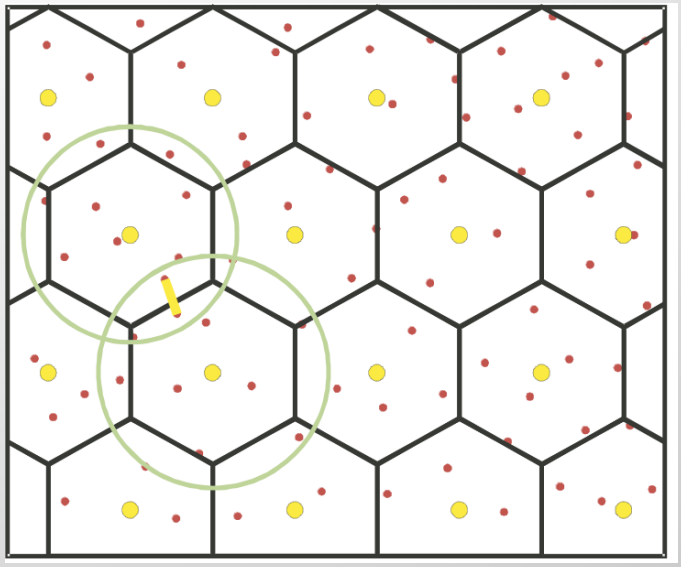


Fixed geometric domain decomposition limits scalability for any heterogeneous problem. Furthermore, statistical fluctuations in the force calculation between processors leads to an effective scalar term that also limits scaling (Amdahl's law).

Domain decomposition strategy for ddcMD (Dave Richards and Jim Glosli)

Design requirements:

- Run efficiently on arbitrary number of processors
- Excellent weak scaling to extend size of simulation
- Excellent strong scaling to extend MD time scale



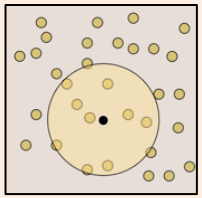
Solution:

- Particle-based domain decomposition - processors own particles, not regions - allows decomposition to persist through atom movement
- Maintain minimum communication list for given decomposition - allows extended range of "interaction"
- Arbitrary domain shape - allows minimal surface to volume ratio for communication

Modeling of MPI Failure Recovery

- Shrinking recovery
  - If a node dies, continue with the remaining N-1 nodes (resources are *shrunk*)
  - Reasonable model from hardware perspective (simply discard failed nodes)
  - In Non-shrinking recovery failed nodes are replaced (or rebooted)
  - Recent proposal to MPI standard (ULFM) allows shrinking recovery

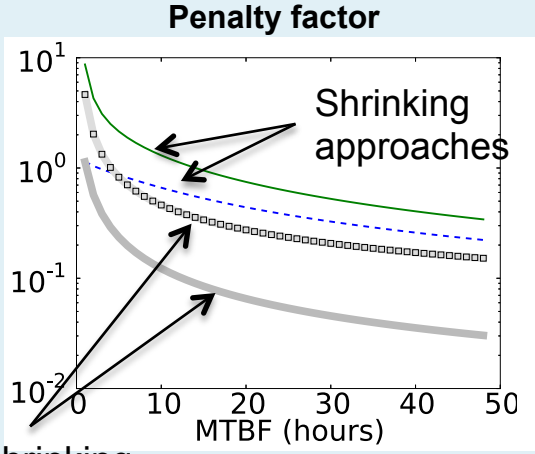
Shrinking recovery in ddcMD



Modeled efficiency based on future system configurations:
Machine size
Memory usage
Filesystem bandwidth, …

Lessons Learned

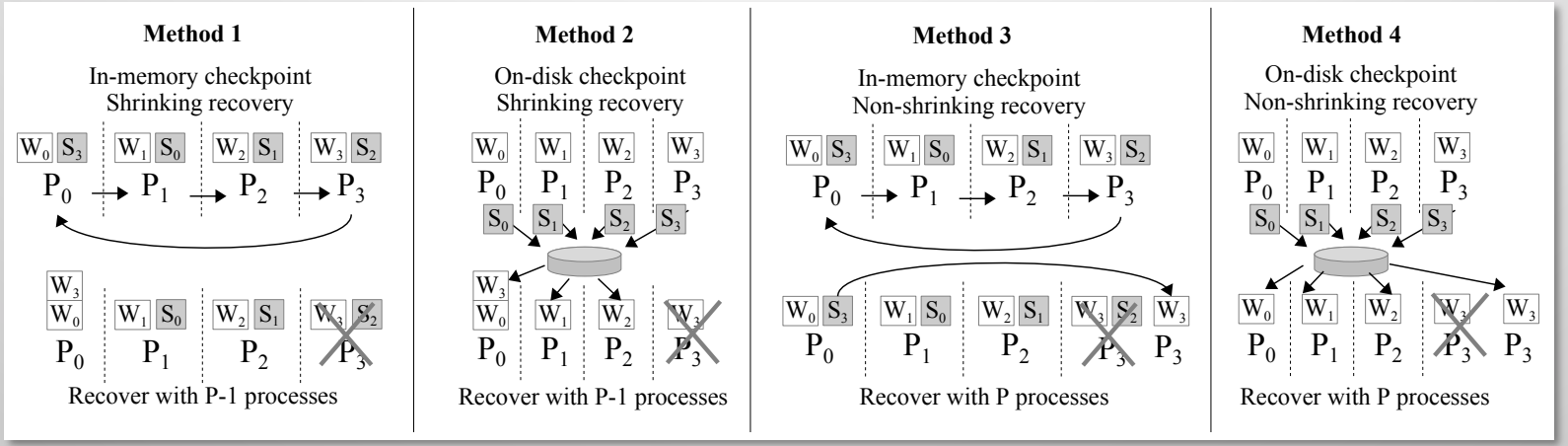
- Non-Shrinking recovery is the most efficient in most cases
- Shrinking recovery works well only in a few scenarios (e.g., if loads are balanced after failures)



Evaluation of MPI Shrinking Recovery in ddcMD

Shrinking recovery:
Continue running with alive nodes after a node failure

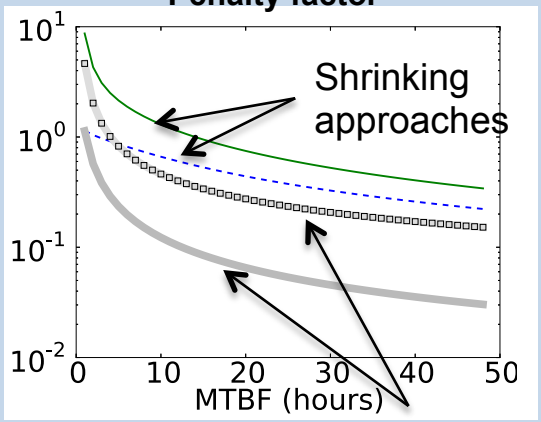
Checkpointing:
On-disk & In-memory



Code to repair MPI state

```
Input: error code
Output: failed rank, new communicator
MPI error handler:
   If error code == process failure:
      Set rank of the failed process
      Revoke communicator
      Shrink communicator
      Call application recovery
```

Penalty factor



Lessons Learned

Non-Shrinking recovery is the most efficient in most cases

Shrinking recovery works well only in a few scenarios (e.g., if loads are balanced after failures)

Local Recovery via Retry Code Blocks

- Simple *retry* code blocks
  - Programmer annotates (or protect) code block
  - If error occurs, code block is re-executed
  - Retry until block terminates without errors

- Fault model
  - Errors detected by hardware
  - Notification through OS that triggers RETRY block

Original code

```
void function(double *array) {
   for (...)
      array[i] = ...
}
```

Annotated code

```
void function(double *array) {
   RETRY{
   for (...)
      array[i] = ...
   }
}
```

Annotations can be done by developer or automatically by a compiler

How to annotate LULESH?

```
main() {
   /* init...*/
   while() {
      funct1();
      funct2();
      funct3();
   }
}
```

Method 1 *MAIN_FUNC_ONLY*

```
main() {
TRY {
  while() {
    funct1();
    funct2();
    funct3();
  }
}
}
```

Method 2 *CORE_FUNCTIONS*

```
main() {
TRY {
  while() {
    TRY { funct1(); }
    TRY { funct2(); }
    TRY { funct3(); }
  }
}
}
```

Method 3 *CORE_LOOP*

```
main() {
TRY {
  while() {
    TRY {
      funct1();
      funct2();
      funct3();
    }
  }
}
}
```

Method 4 *N_ITERATIONS_BACK*

```
main() {
TRY {
  while() {
    TRY(N) {
      funct1();
      funct2();
      funct3();
    }
  }
}
}
```

Evaluation of Retry Blocks in LULESH

How to annotate LULESH?

```
main() {
   /* init...*/
   while() {
      funct1();
      funct2();
      funct3();
   }
}
```

Method 1 *MAIN_FUNC_ONLY*

```
main() {
TRY {
  while() {
    funct1();
    funct2();
    funct3();
  }
}
}
```

Method 2 *CORE_FUNCTIONS*

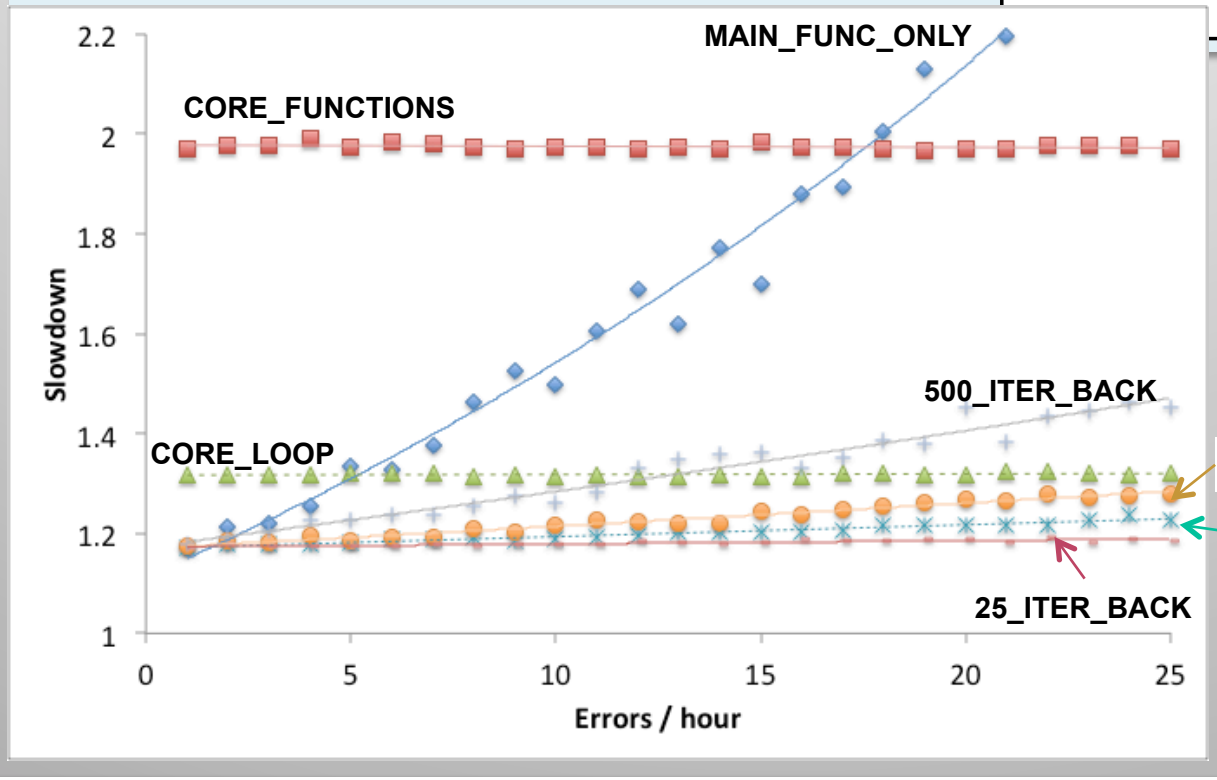
```
main() {
TRY {
  while() {
    TRY { funct1(); }
    TRY { funct2(); }
    TRY { funct3(); }
  }
}
}
```

Method 3 *CORE_LOOP*

```
main() {
TRY {
  while() {
    TRY {
      funct1();
      funct2();
      funct3();
    }
  }
}
```

Method 4 *N_ITERATIONS_BACK*

```
main() {
TRY {
  while() {
    TRY(N) {
      funct1();
      funct2();
      funct3();
    }
  }
}
}
```



Rolling back LULESH to the 25th loop iteration is the optimal strategy

CESAR Post-docs

- POC: Andrew Siegel
- Post-docs: Nan Dun and Huiwei Lu
- put effort into an area where otherwise collaborations would have been thin, and it made me very aware of what was going on in the x-stack research world, and some good work was done.

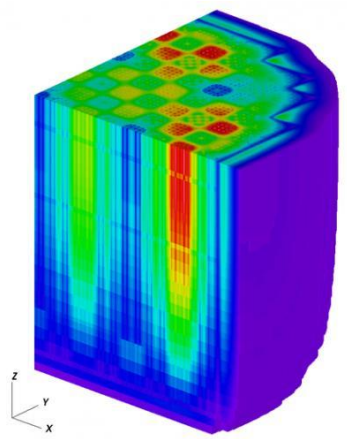
Project Introduction

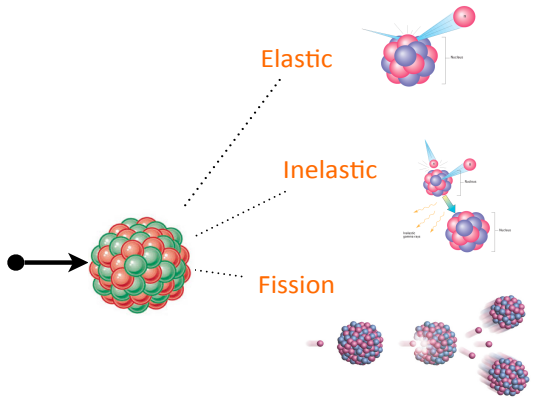
- A joint work of GVR + CESAR
  - The Global View Resilience (GVR) Project
    - An X-Stack center project focusing on resilience
    - PI Andrew A. Chien
  - OpenMC Monte Carlo Transport Simulation
    - A CESAR application for nuclear reactor simulation
  - A co-design: Both OpenMC and GVR are changing during this work
  - Exploiting future opportunities with other CESAR applications

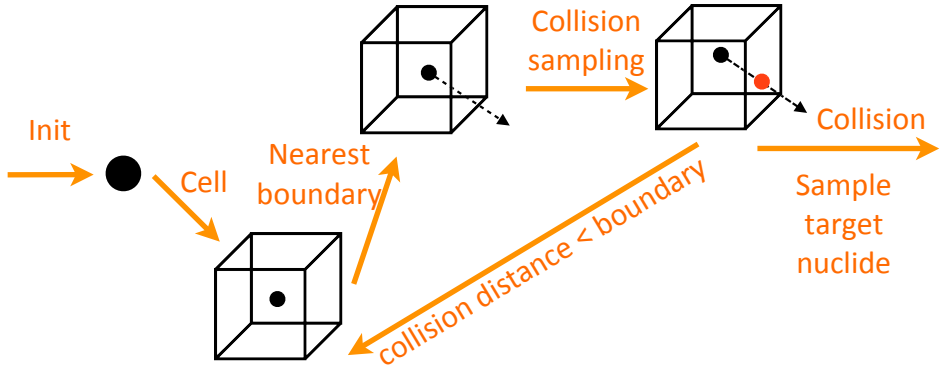
Motivation

- Fight against errors in scientific applications
  - Errors are increasing because of more complex software/hardware, systems, etc.
  - Recovery from errors are expensive because the amount of computation is large
- To make applications resilient to errors
  - OpenMC: A large-scale (both computation and data) scientific application
  - GVR: Our approach (i.e., library) to inject flexible resilience into applications

Monte Carlo Transport Simulation (OpenMC)







Tally: The events happened during the particle transport, e.g., fission, scatter, etc. For each particle, the tallies are accumulated during its lifetime.

Batch: A bag of particles to simulate, may up to millions. We do many batches to make the simulation realistic.

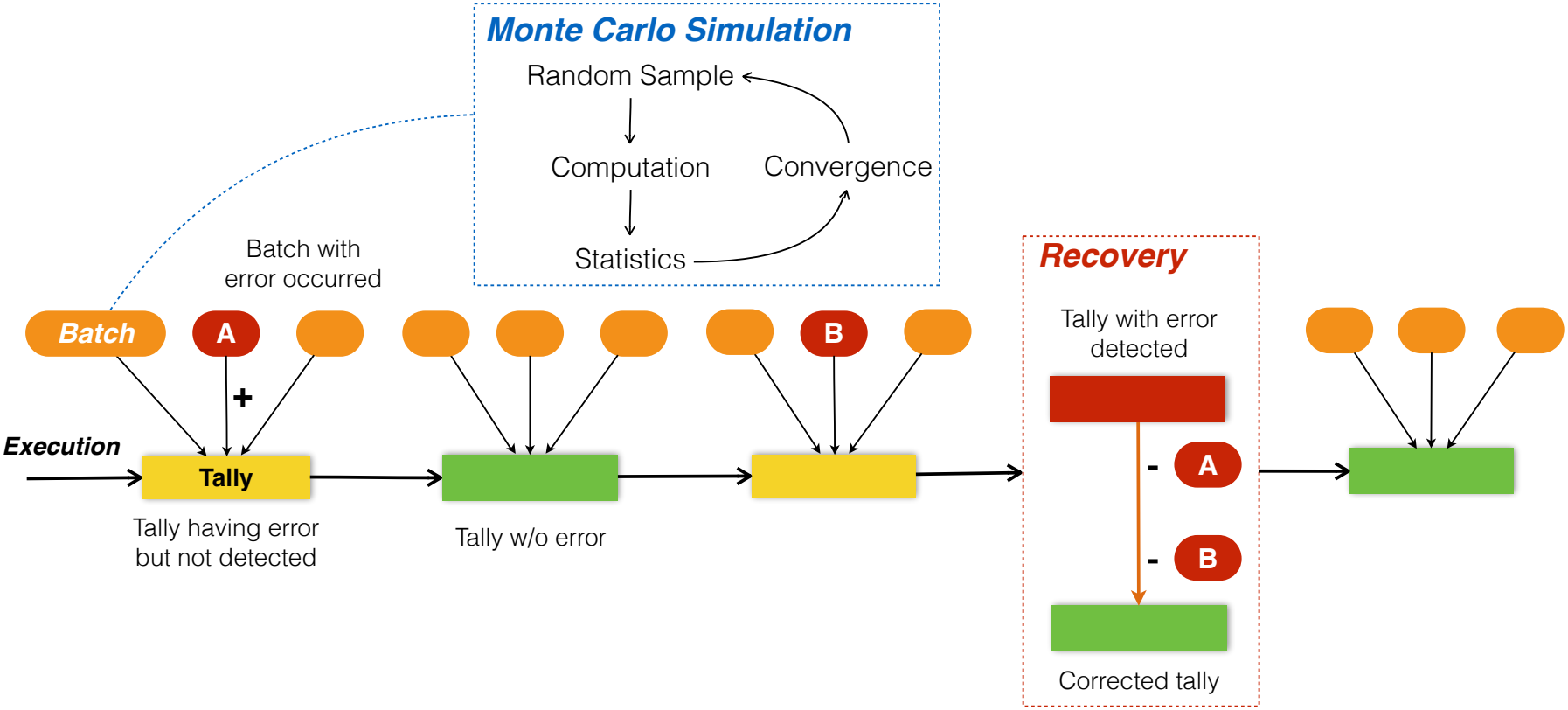
Don't forget errors in OpenMC!

Global View Resilience (GVR)

- A way of recovering errors in software
  - Keep a series of snapshots/version (history) of data
  - Figure out where/what is wrong by examining the history
  - Recover the error by removing the bad part and compensating with correct re-computation
- How GVR helps OpenMC?
  - Use GVR-versioned memory to store tally data and conduct recovery in OpenMC
  - Simply by injecting GVR code into OpenMC (does not change OpenMC structure)
  - Improve performance and scalability of tally accumulation
  - Provide a natural way to program MC simulations

Illustration of OpenMC Recovery

On-line correction, no rollback, without losing previous correct computation effort

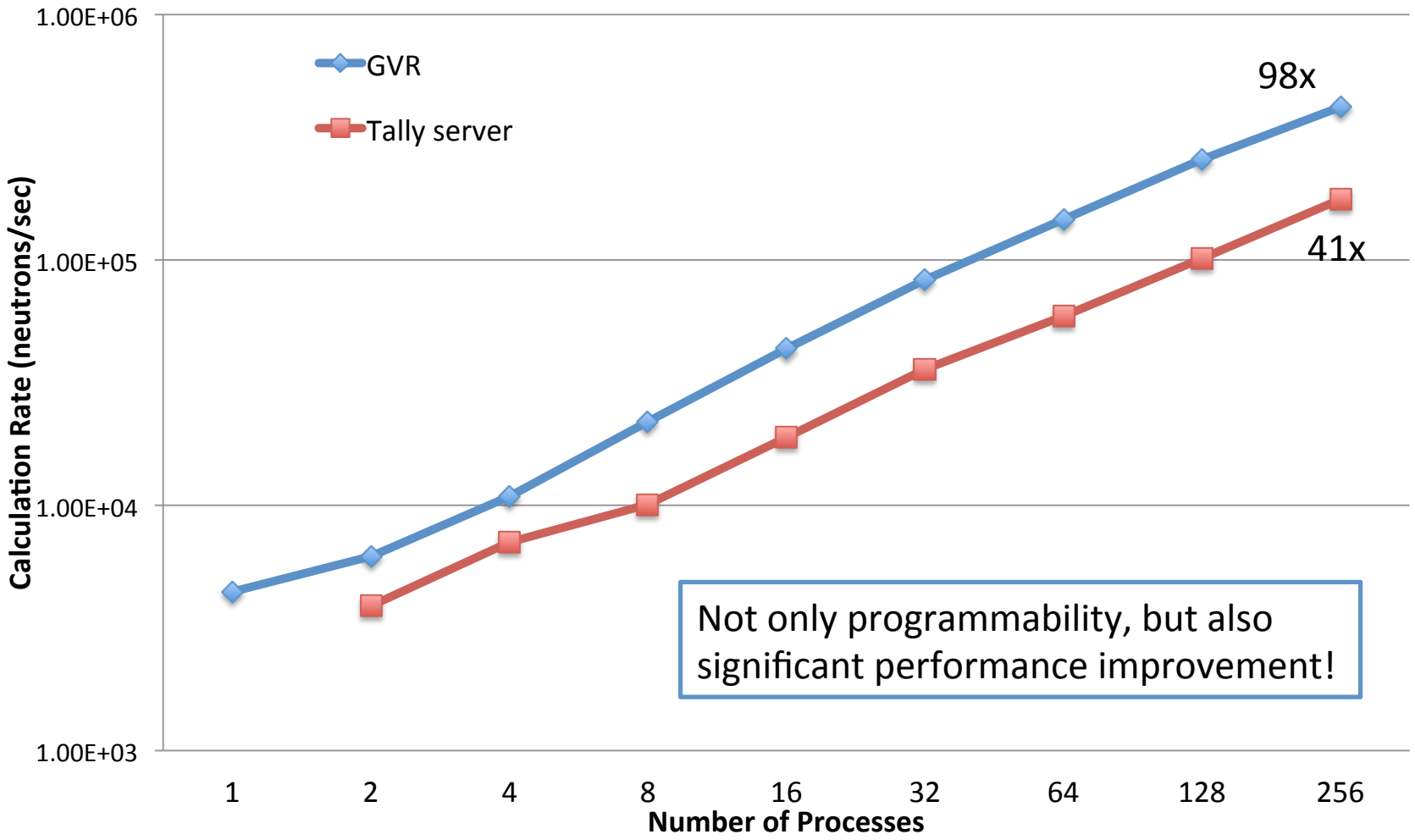


Programmability

```
Initialize initial neutron positions
GDS_create(tally & source_site); //Create global tally array and source sites
for each batch
  for each particle in batch
     while (not absorbed)
         move particle and sample next interaction
         if fission
           GDS_acc(score, tally) // tally, add score asynchronously
           add new source sites
      end
      if tally contains error
         GDS_move_to_prev(err_tally, err_ver);
         GDS_move_to_prev(err_prev_tally, err_prev_ver)
         GDS_move_to_prev(err_prev_src_site, err_prev_ver);
         tally = tally + (err_prev_tally – err_tally)
         Redo batch computation with err_prev_source_site
      else
         GDS_fence() // Synchronize outstanding operations
         resample source sites  & estimate eigenvalue
         GDS_ver_inc(tally) // Increment version
         GDS_ver_inc(source_site) // Increment version
   end
end
```

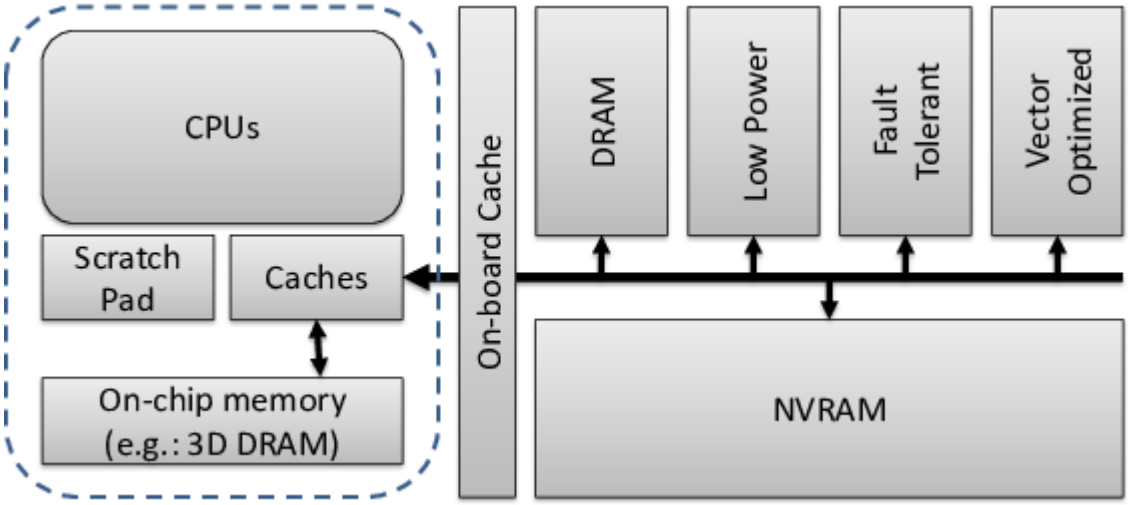
Less than 1% LOC changes

Non-versioning Performance



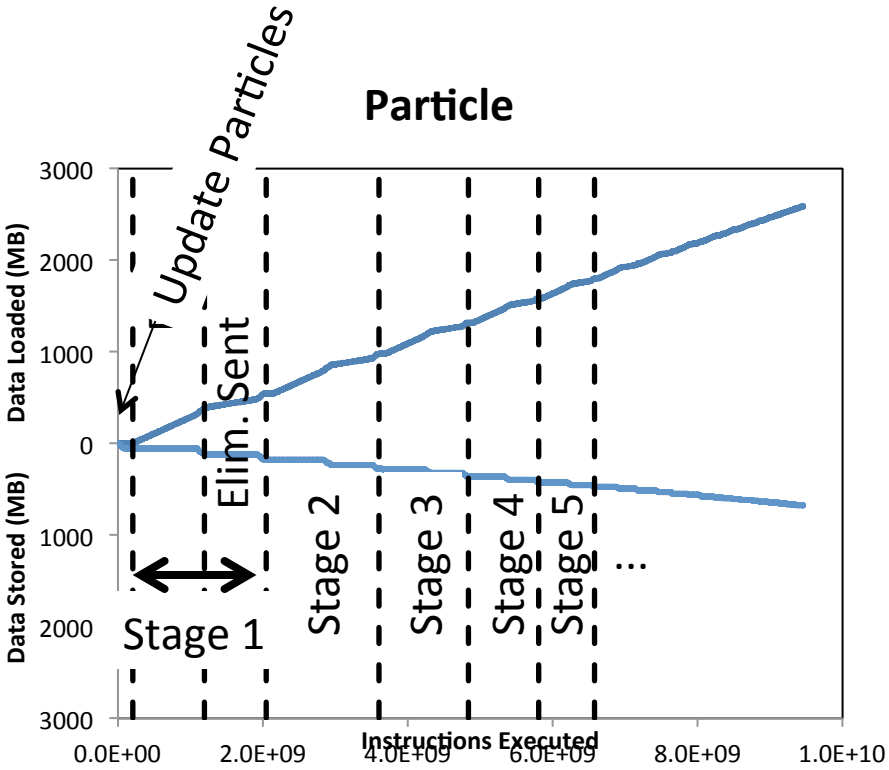
Understanding Data Access Patterns using Object-Differentiated Memory Profiling

- Objective:
  - To distribute application's data among **memory subsystems** in upcoming **heterogeneous memory systems** at **object-level granularity**
- First step:
  - Need to find out how applications make use of their different memory objects
- Approach:
  - Dynamic profiling on emulated hardware: Valgrind
  - Incorporate object-differentiated capabilities into Valgrind's core and extend tools



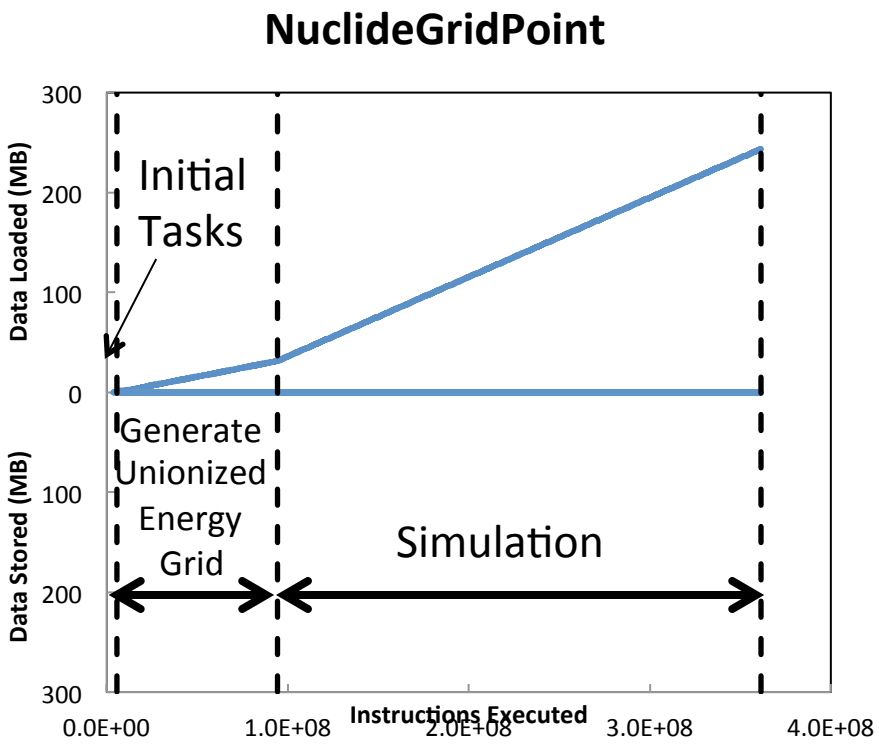
MCCK

- "Particle" object covering most of the data accesses
  - Read and written along all the execution
  - Array of:
    - double x, y, z;
    - double energy, angle;
    - int absorbed;
- Stages:
  - *Update Particles* – W*rite only, reduced rate*
  - *Pack* – 8 bytes read per each written
  - *Exchange* – Short task, high ratio of read and write
  - *Eliminate Sent* – 3 bytes read per each written



XSBench

- "NuclideGridPoint" large object covering many data accesses
  - Array of six doubles
  - Extensively used on read accesses
  - Could benefit from memory highly optimized for reads, and optionally fairly bad at writes
  - Access rate markedly varies between the two main stages (memory migration?)
- *Initial Tasks* – Short stage without many accesses
- *Generate Unionized Energy Grid* – 0.4 bytes read per instruction
- *Simulation* – 0.8 bytes read per instruction



Summary

- Largely different access patterns across applications
- Access patterns for a particular object present marked differences among the different stages of the execution.
  - MCCK:
    - *Particles* object accessed mostly for sorting purposes
    - Only writes during the last merge stage
    - The first of the two sortings reveals a much higher access rate (more fields involved)
  - XSBench:
    - Read-only
- Co-design purposes:
  - Memory usage and bandwidth requirements is a major concern
  - Access patterns may provide insight into unexpected behaviors

[1] A. J. Peña and P. Balaji. "A framework for tracking memory accesses in scientific applications", in The Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), Minneapolis, MN, USA, Sep. 2014. Accepted.

Work in Progress

- Aim:
  - Use object-differentiated profiling data to propose optimized object distribution among memory subsystems in heterogeneous memory systems
- Problem:
  - CESAR miniapps store their data of interest into a single big object
  - Prevents object granularity approach
  - Our research using miniapps from other projects that don't concentrate all their data in a single object reveals the feasibility and interesting potential of our approach
- Looking into:
  - Divide the memory objects into smaller logical pieces to enable our approach

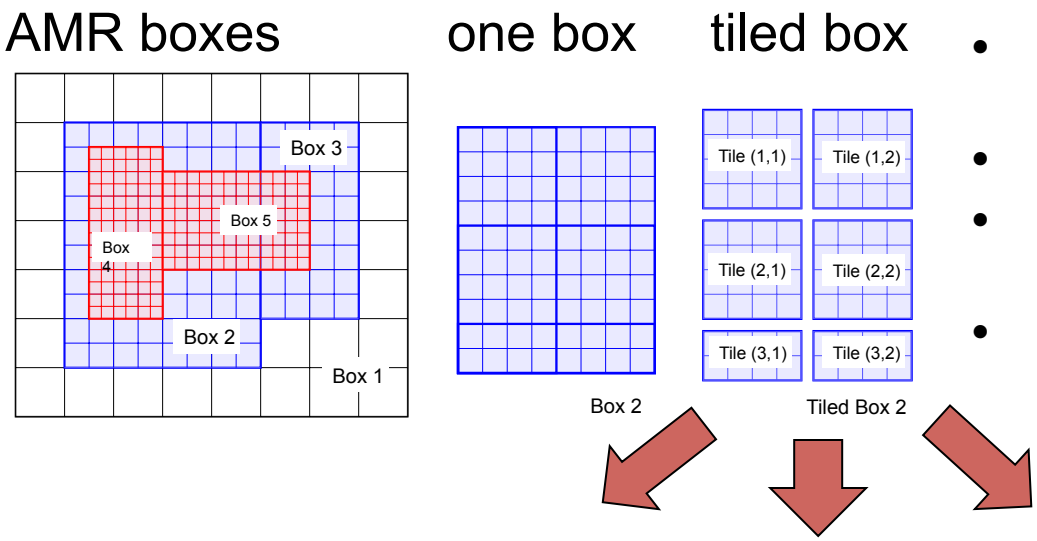
[2] A. J. Peña and P. Balaji. "Understanding data access patterns using object-differentiated memory profiling", in International Conference on Parallel Processing (ICPP), Minneapolis, MN, USA, Sep. 2014. Submitted.

[3] A. J. Peña and P. Balaji. "Toward the efficient use of multiple explicitly managed memory subsystems", in IEEE Cluster 2014, Madrid, Spain, Sep. 2014. Submitted.

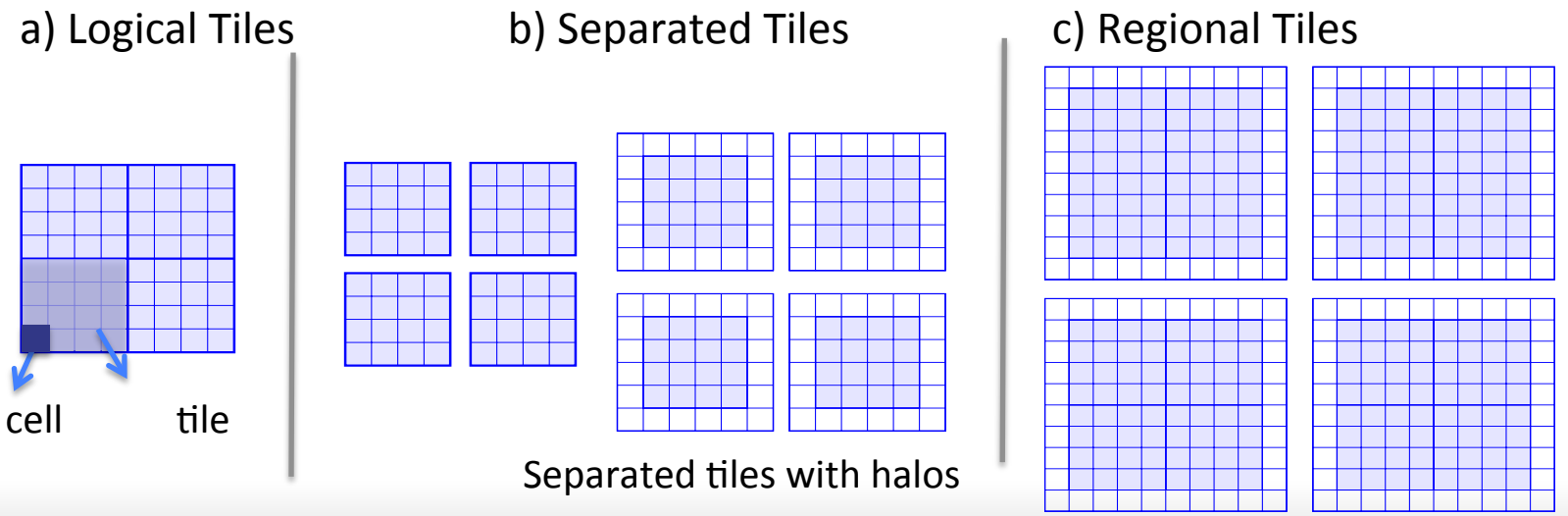
Post-docs
Didem Unat and Cy Chan
Primary Interactions: Degas and Xtune

TiDA: Tiling as a Durable Abstraction

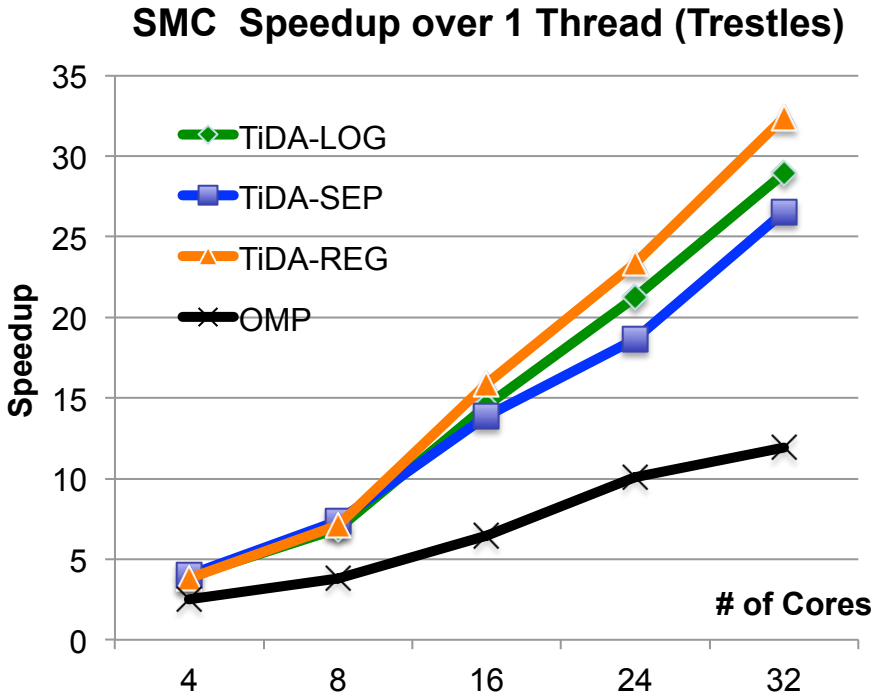
by Didem Unat, Weiqun Zhang, John Bell and John Shalf

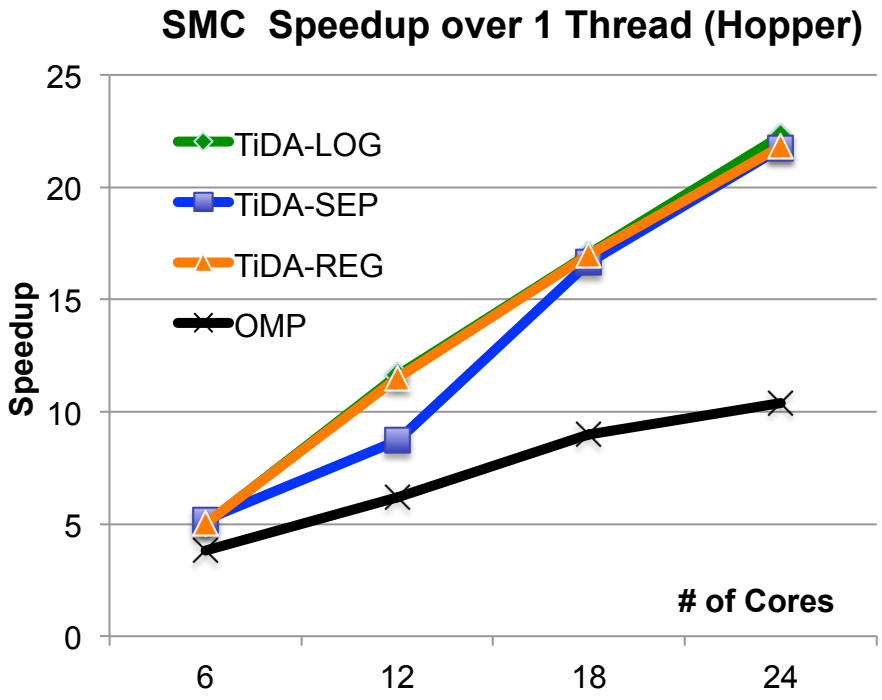


- Provides a simple API to naturally express **data layout and tiling**
- Focuses on **data decomposition**
- Supports mechanisms to deal with various **cache scenarios**
- Enables **parameterized tiling** and **layout** changes without or minimal code modifications



SMC: Combustion Proxy App

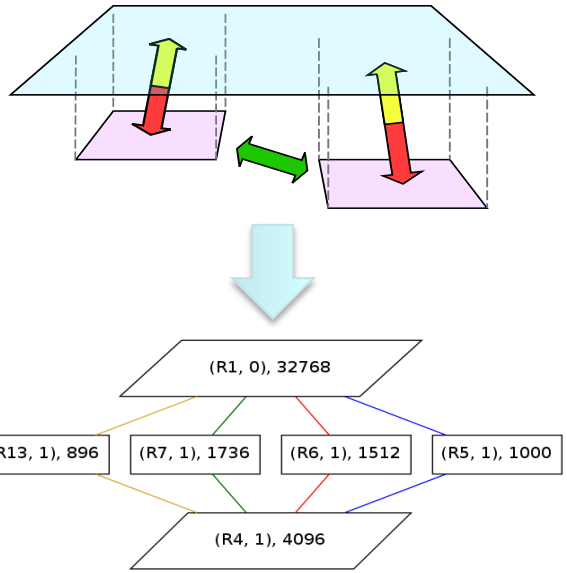


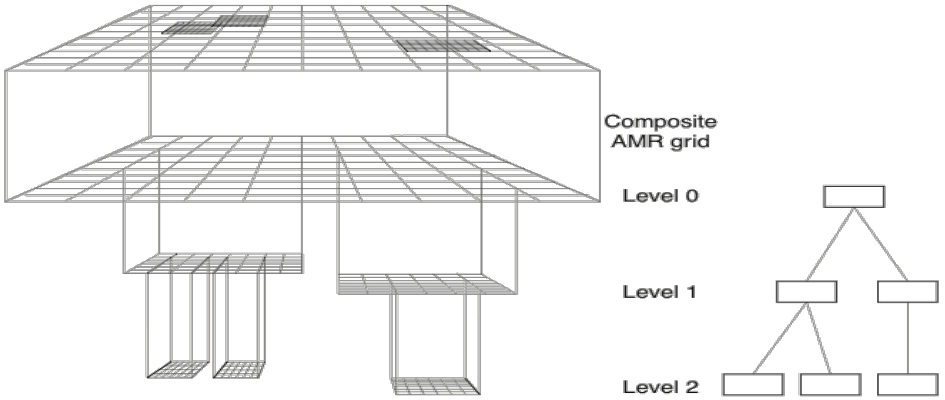


- TiDA achieves 32x and 22x speedups over single OMP thread on Trestles and Hopper
- Usually it is not recommended to tile in X dim
  - Z partitioning is for NUMA nodes and Y partitioning is for cache reuse
- Tiling in X dimension is necessary for SMC because of the large working set
  - About 256 MB for N=256 and #species=9

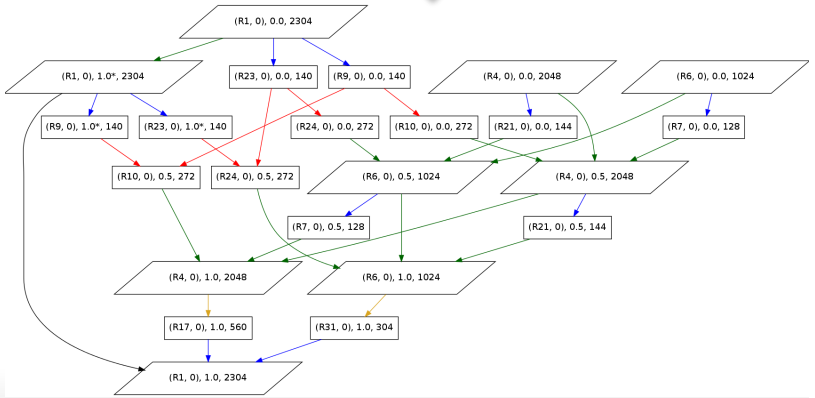
Adaptive Mesh Refinement (AMR) Dependency Analysis and Network Simulation Tool

- Multi-level AMR grid hierarchy with multigrid
- Derives dependency graph from box geometry
- Utilizes compact graph representation that can be unfurled dynamically
- Goal: scale up to 100k simulated nodes for various network topologies and data layouts

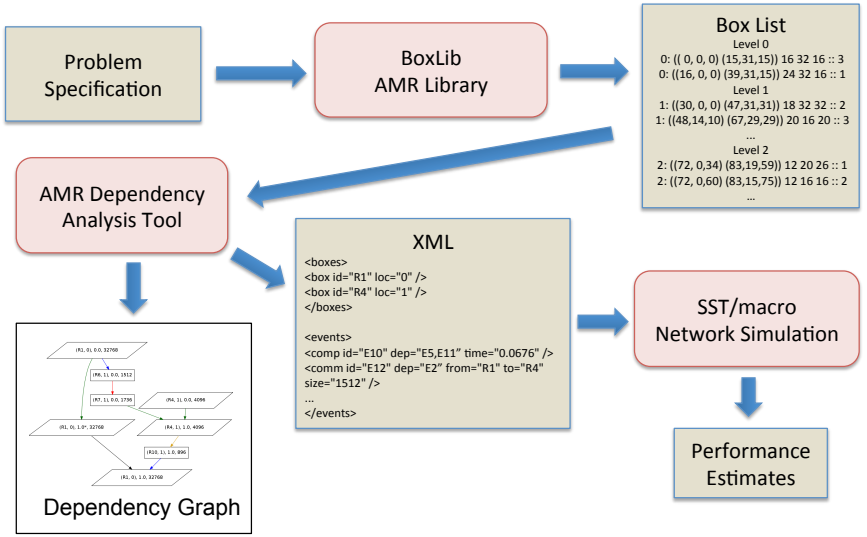


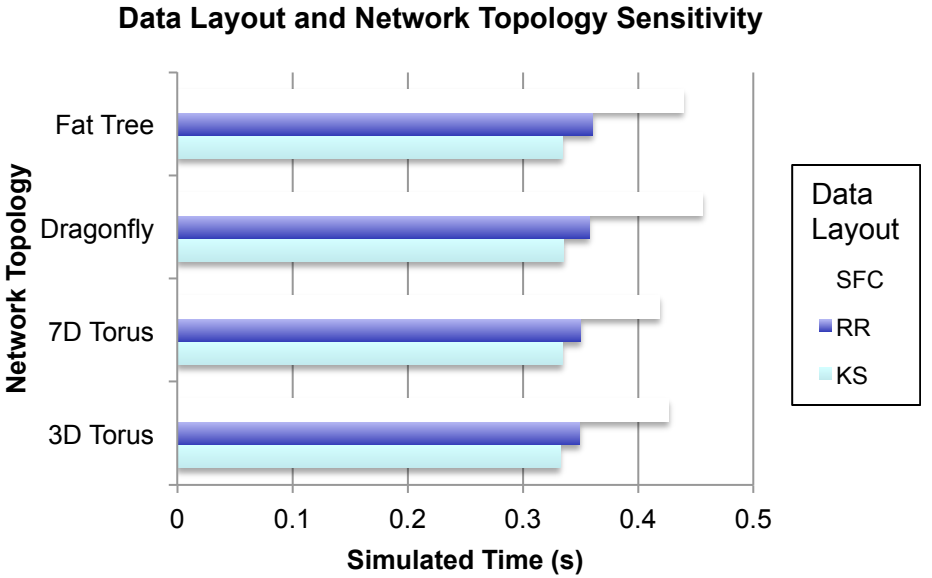


AMR Hierarchy



Analysis Toolchain and SST/macro Simulation Results

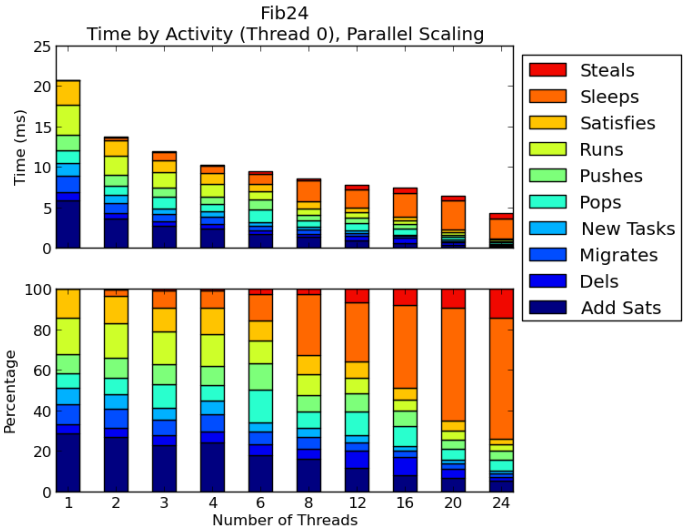


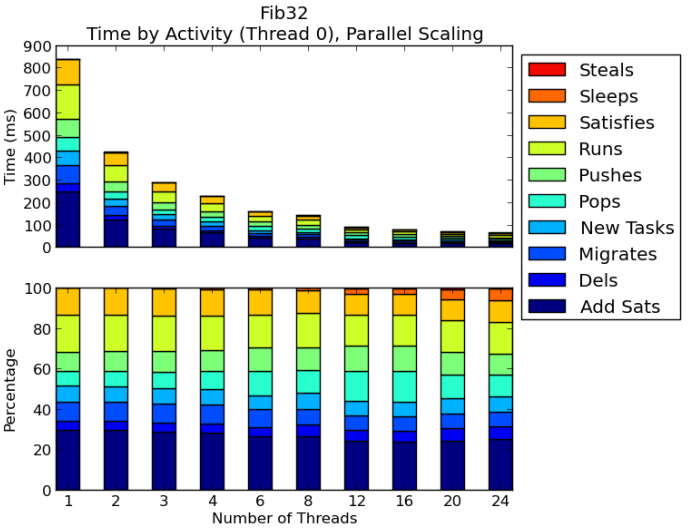


- AMR box list derived from combustion problem (BoxLib)
- Analysis tool generates XML specifying computation and communication dependency graph
- SST/macro simulates execution on parameterized network
- Evaluation for various data distributions and network topologies

Rambutan: Adaptive Runtime Modeling/Simulation Tool

Type	Operation	Description
Task Allocation	New	Create new task
	Delete	Delete task
Queue Operations	Push	Push task onto queue
	Pop	Pop task off of queue
	Migrate	Migrate task between queues
Dependency Management	Add Satisfaction	Add dependency between tasks
	Satisfy	Satisfy task dependency
Scheduling and Load Balance	Steal	Steal chunk of tasks from another process
	Sleep	Sleep waiting for work





- Compact and lightweight runtime model: analysis instrumentation built in from the start
- Instrumentation captures dynamic runtime metrics
- Modeling tool to explore the impact of design choices:
  - Compare asynchronous execution vs. bulk-synchronous
  - Model heterogeneous OS and hardware performance (e.g. jitter, faults, throttling)
  - Support for hardware task management operations
  - Explore task decomposition, granularity, and scheduling