

# *Steering the Ship in the Face of a Tidal Wave: Co-design Strategy at LLNL*

**Salishan Conference on High Speed Computing**

April 25, 2012

Rob Neely

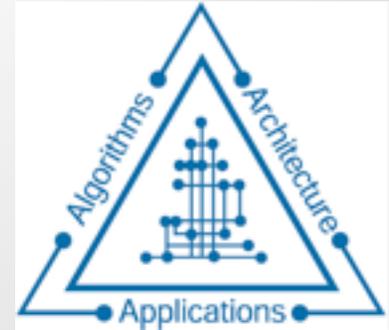
 Lawrence Livermore  
National Laboratory

LLNL-PRES-551777

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

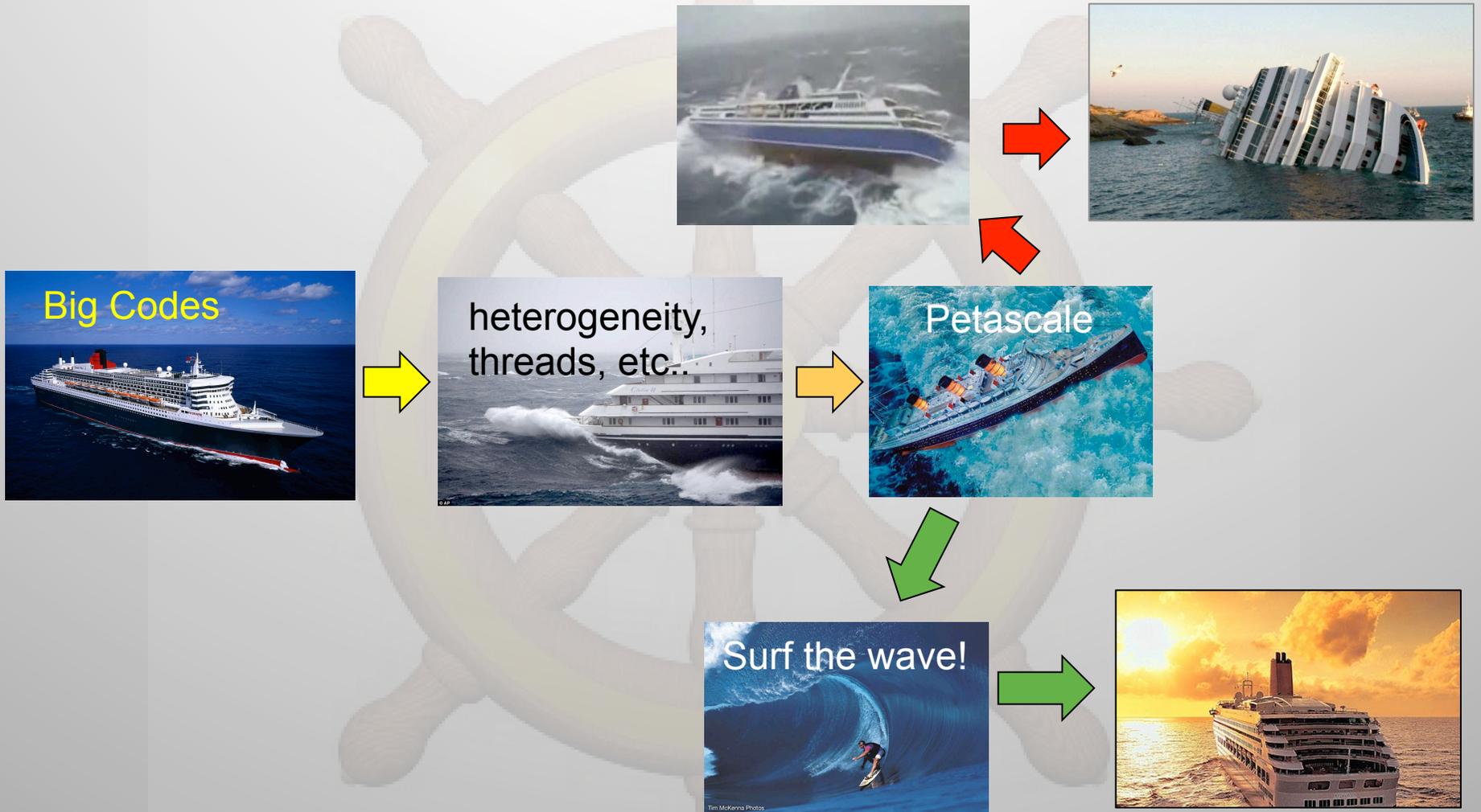


# Salishan Conference themes



- **Conference theme: how to more effectively use current or emerging advanced architectures.**
- **Session theme: Application/Machine hierarchy**
  - **What is our experience dealing with many levels of memory?**
  - **What have we learned about memory hierarchy that will enable effective use of 3D DRAM and on-node NVRAM in this timeframe?**
  - **What experiences do we have with “new” programming models ?**
  - **What is the effectiveness of current languages versus new languages?**

# Big Ships, Rough Seas, and Unseen Hazards



# We're facing an enormous challenge of how to move our multi-physics apps to exascale machines.

- Often > 10 physics packages
- 10 to ~30 third party libraries
- Long life-time projects with >1 million lines of code
- 15+ years of development by large teams (10 – 20+ FTEs)
- Many different spatial, temporal scales
- Variety of parallelism approaches
- Steerable / interactive interfaces
- Multi-language (C++, C, Fortran90, Python)
- End users are typically not developers (no ability to just fix and recompile)
- All have adapted excellent SQA processes for major evolutionary restructuring
- Algorithms tuned for minimal turn-around time instead of maximal computational efficiency



**We must continue to deliver our programmatic mission while addressing the needs of next generation advanced architectures.**

# Exascale computing presents unique challenges to multi-physics integrated codes

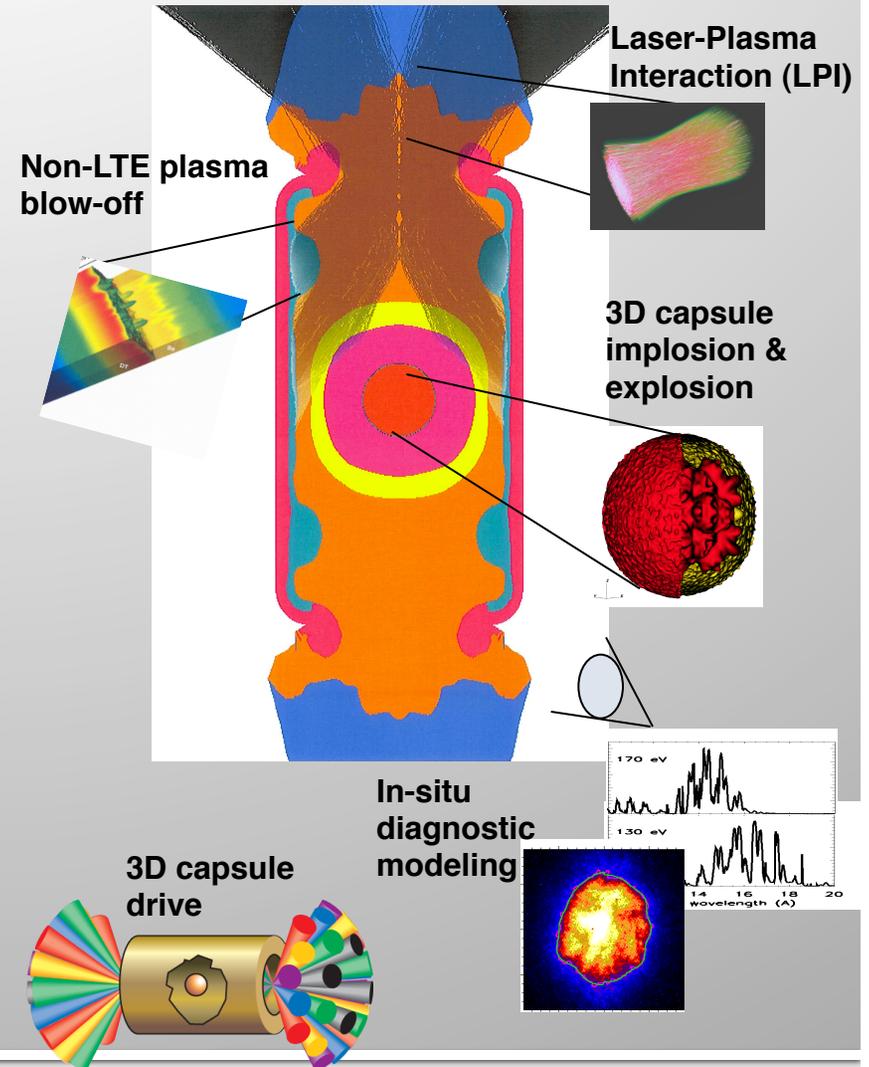
## Improved Physics

- Laser beam effects
- Plasma blow-off and effect on drive, symmetry
- Capsule implosion details
- Explosion symmetry
- Atomic physics
- Line radiation transport

## Improved Resolution (multi-scale, time/space)

**Improved Understanding  
(predictive capability)**

## HEDP Example

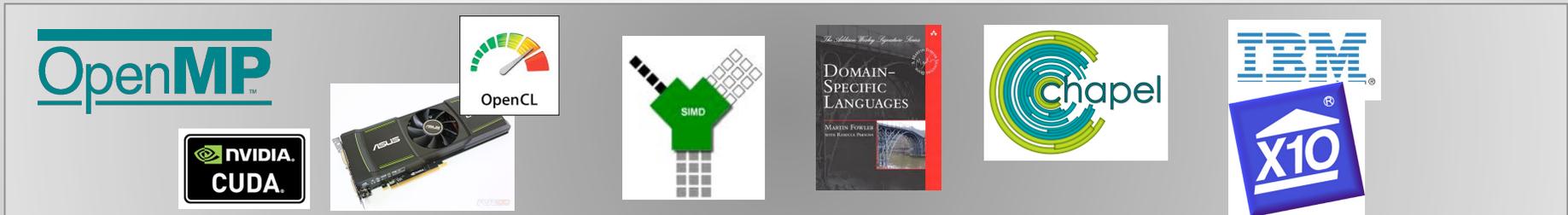


# Our physics packages have differing computational requirements, making generalizations difficult

- Below are examples of some common physics packages
- Typical characteristics of each package are listed, with those that typically limit performance listed in red

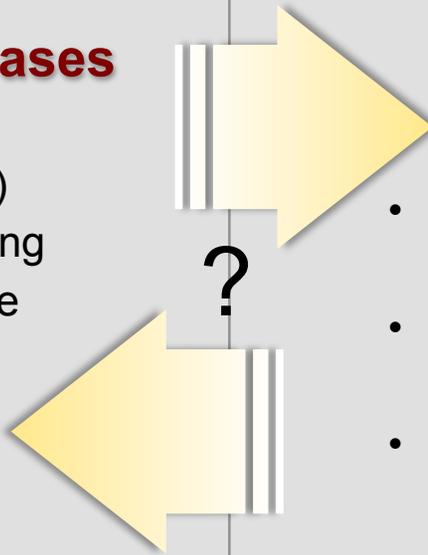
Typical Characteristics	Hydrodynamics	Deterministic Transport	Monte Carlo Transport	Diffusion
Memory needs	0.1 - 1 KB/zone	<b>40 - 240 KB/zone</b>	3 - 30 KB/zone	0.1 - 1 KB/zone
Memory access pattern	Regular with modest spatial and temporal locality	Regular, low spatial but high temporal locality	<b>Irregular, low spatial and temporal locality</b>	Regular, good spatial and temporal locality
Communication pattern	<b>Point to point, surface communication</b>	Point to point, some volume	Point to point, some volume	<b>Collective communications and point to point</b>
Mflops per zone per cycle	0.02 – 0.1 (10X for iterative schemes)	2 – 12	<b>.03 - .07</b>	0.1 - 3
Databases	<b>20-160 MB (EOS)</b>	0.3 - 12 MB (Nuclear)	<b>100 - 300 MB (Nuclear)</b>	0.1 - 1 KB/zone

# Evolve or Rewrite? This is a fundamental question we're addressing



## Evolve existing code bases

- Gain experience with massive scaling (Sequoia / BlueGeneQ)
- Implement fine-grained threading
- Application-controlled resilience
- GPU directives
- Leverage validated code base



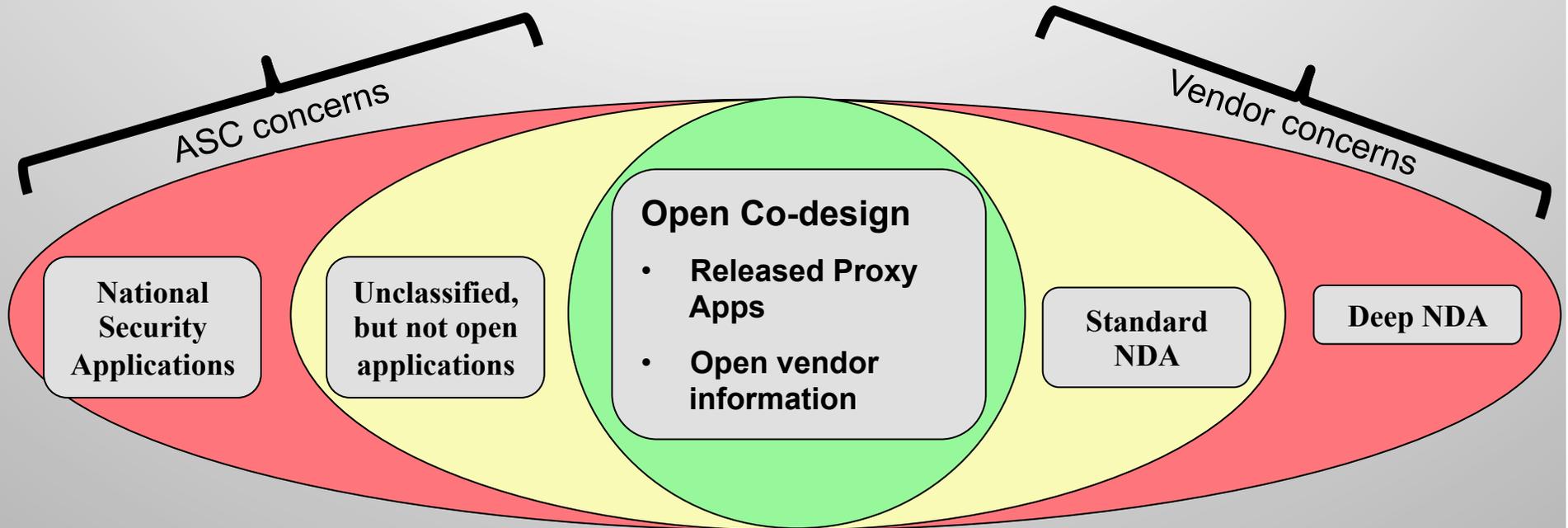
## Undertake new “from scratch” rewrite

- Evaluate and gain experience with new programming models
- Develop proxy applications to streamline explorations
- Determine degree of rewrite needed (if any)

**It's too early to choose a technology to rewrite our applications  
HOWEVER  
It's never too early to explore and influence promising technologies**

# (One of) the difficulties of co-design

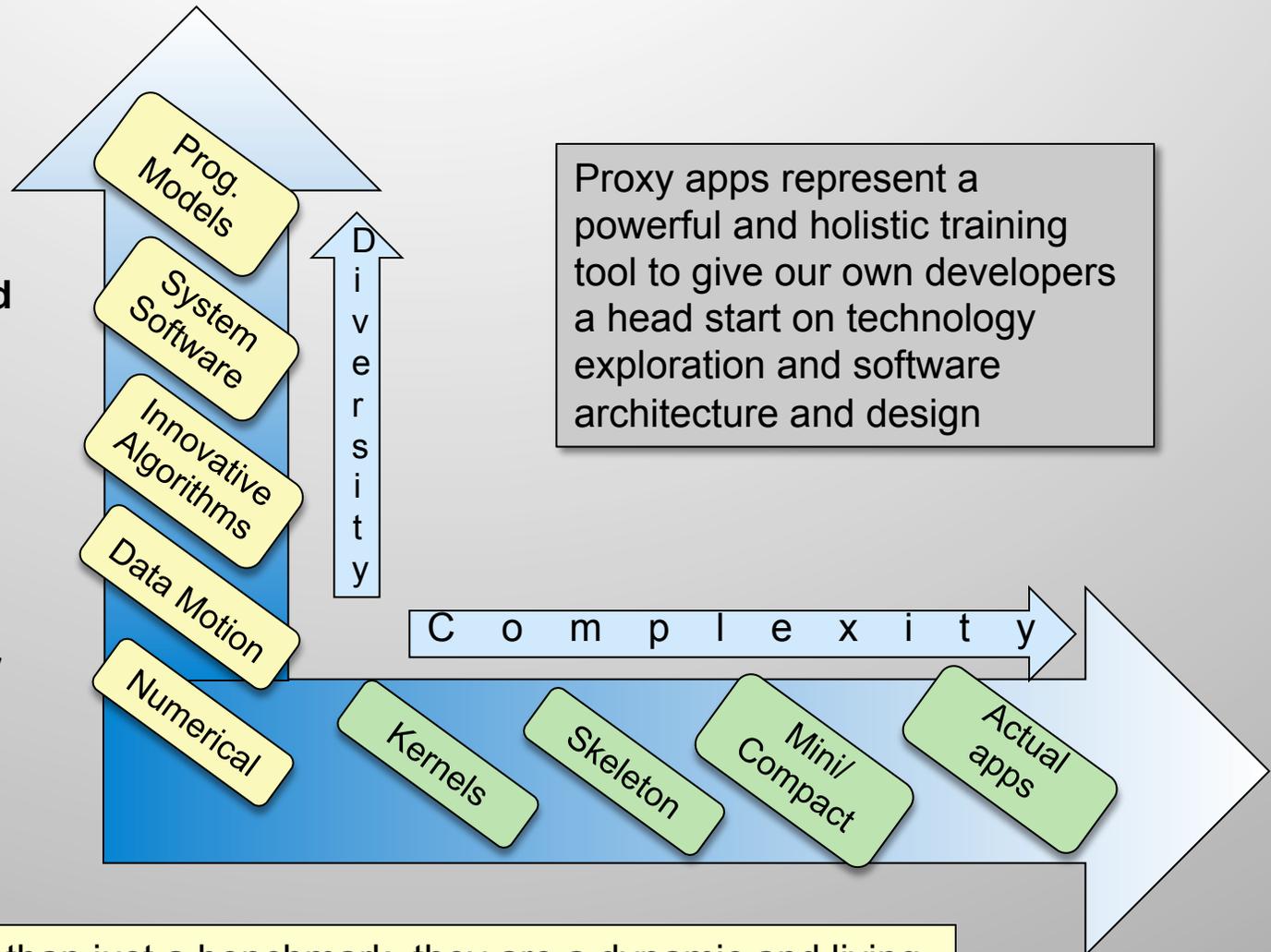
Co-design gets more difficult the further you get from open collaboration and the closer you get to the “truth”



- **ASC** : Involve staff with clearances in co-design efforts
- **Vendor** : Firewalling of lab staff from engaging in multiple “deep NDA” involvements

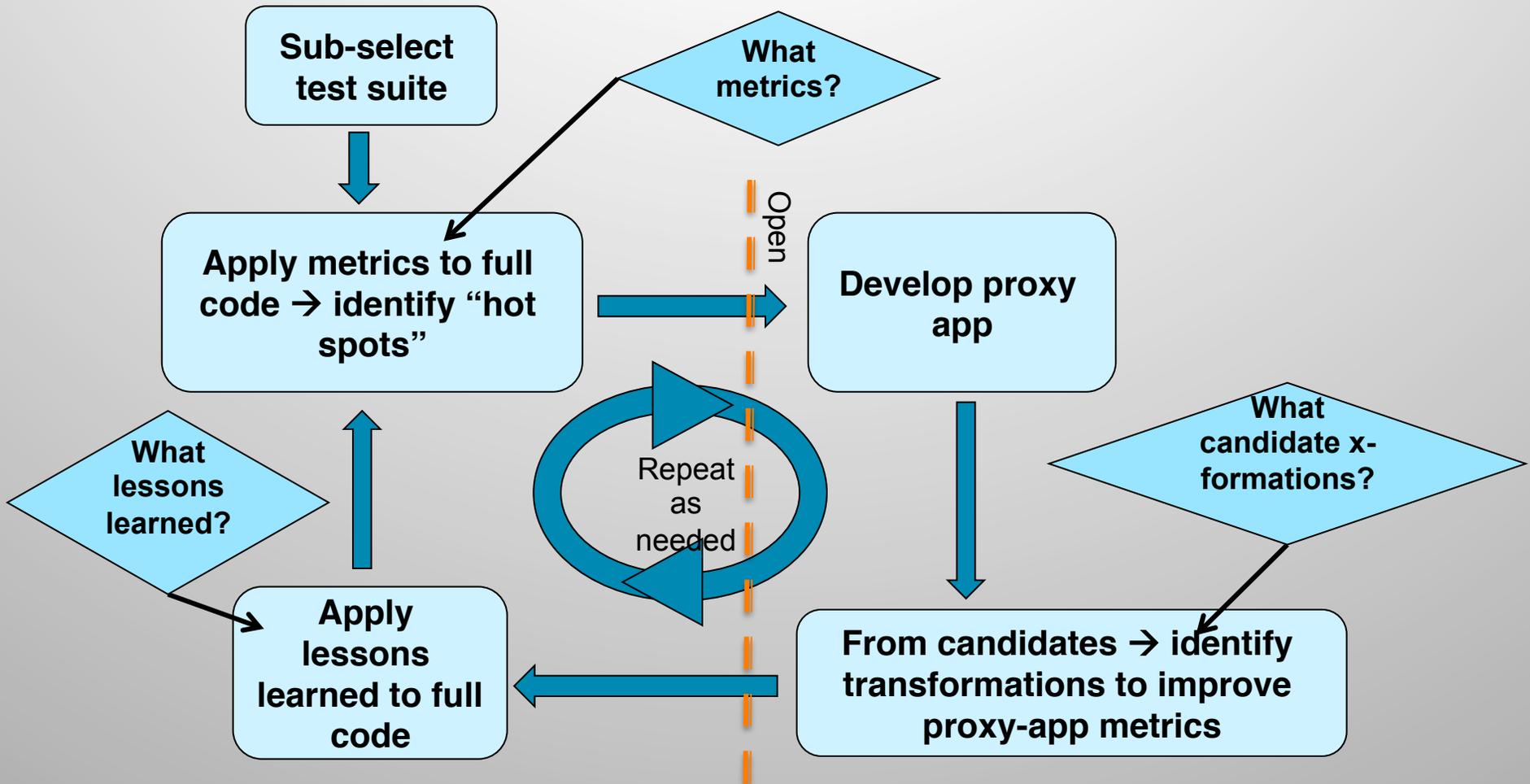
# Proxy apps development is being pursued strategically along two axes

- Simple, open, and easy to pick up and explore
- Must accurately represent original applications
- The collection should account for more than just fast numerical performance



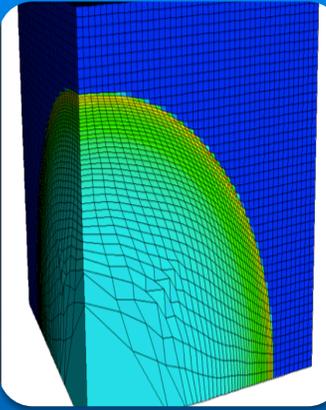
Proxy apps are more than just a benchmark, they are a dynamic and living representation of the current state of how typical applications are written

# Proxy app explorations must tie back to the full applications they are representing



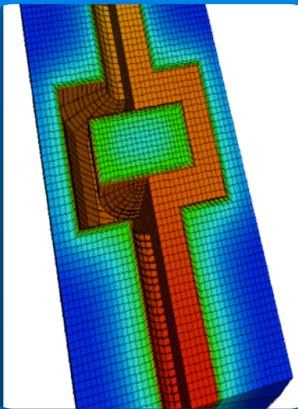
**When does this approach “converge”?**

# LULESH and Mulard are two new proxy apps developed in the past year



## LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics

- Representative of data structures and numerics of a major ASC application
- Performs a Sedov (blast wave) calculation
- 3D *unstructured* hex mesh
- 8 different versions (and counting)



## Mulard: multigroup radiation diffusion

- 10-100 coupled diffusion equations transport radiation
- Many, large scale linear solves
- Lots of data, complicated setup
- Each group matrix has similar structures
- Can assemble all groups at once
- Can solve groups independently or together

# Current proposed set of LLNL Proxy Apps

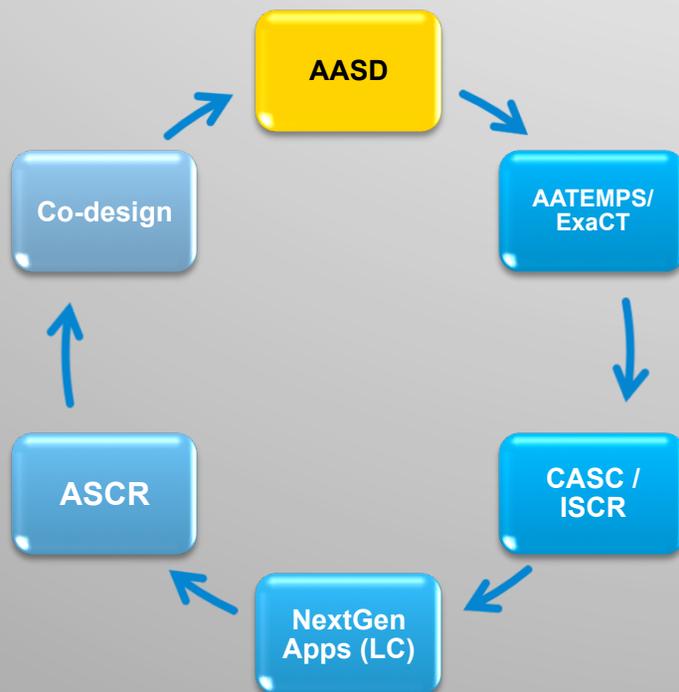
Name	Description	Language	Type
UMT	Unstructured Mesh Transport	Ftn, py, C, C++, MPI, OMP	Compact
AMG (hypre)	Algebraic Multigrid	C, MPI, OMP	Mini
CLOMP	OpenMP, TM/SE performance & overheads	C, OMP	Mini
MCB	Monte Carlo transport	C++, MPI, OMP	Skeleton
Lulesh	Explicit Lagrange shock hydro on unstructured mesh	C++, MPI, OMP	Mini
f3d kernels	Single precision vectorization, complex arithmetic	C, OMP, (yorick)	Mini
Mulard*	High order diffusion (MFEM based)	C++, MPI	Compact
LIP	Livermore Interpolation Package (used by LEOS)	C	Mini
Blast*	High order hydrodynamics (MFEM based)	C++, MPI	Compact
HEART	Vectorization	C, OMP	Kernel
EOS_fm4	Gruneisen analytic equation of state	C	Kernel
MIAVAS	Array-of-structs vs struct-of-arrays	C	Kernel
AdvB	Advection	C++, MPI	Mini
ioperf	HDF5 LLNL benchmark	C	Skeleton
Steer	OS support for code steering	Py,	Mini
LLNLLoops 2	SIMD vectorization	C	Kernel
AMR	Adaptive Mesh Refinement	?	Compact
Contact	Slide surfaces, contact (LDEC-based?)	?	Mini
Mslib*	Element by element material models	C	Compact

Sequoia Benchmark	Exists / released	Exists / unreleased	Under development	Undeveloped
-------------------	-------------------	---------------------	-------------------	-------------

\* May be restricted

# Advanced Architectures Software Development (AASD) Project

- Launched in Sept 2011 to coordinate activities in multiphysics integrated code teams aimed at next gen architecture app development
- Provide developers much-needed “free energy” to explore new technologies



- Work with research and vendor community to identify promising and applicable technologies
- Inform programmatic funding of key technologies before they end due to lack of research funding

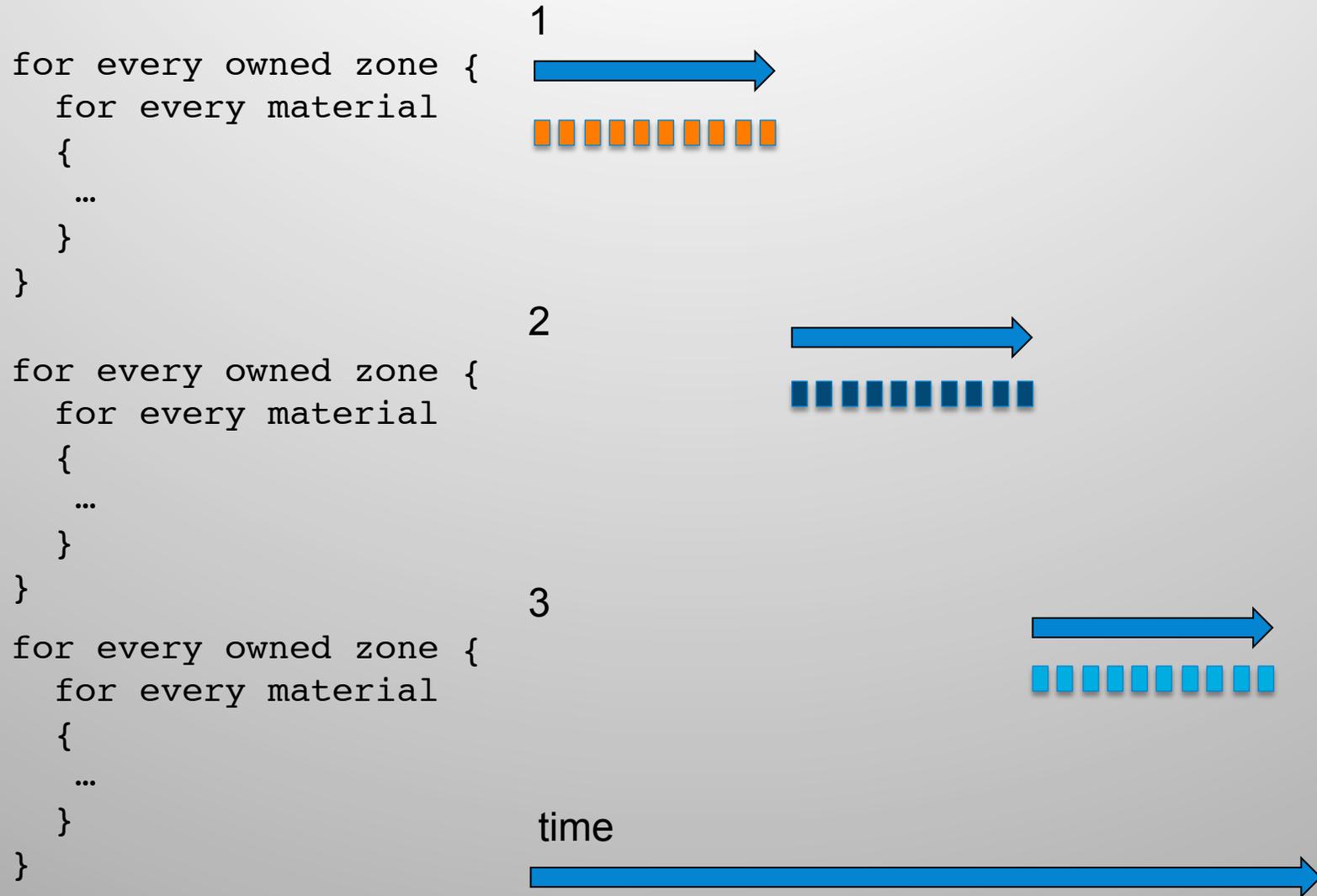
## Experience on existing platforms is giving us insight into scalability for upcoming petascale architectures

- **Existing petascale platforms at LLNL:**
  - Dawn (BlueGene/P) – 147k cores (.5 Pf)
  - Zin (Linux TLCC2) – 45k cores (.97 Pf)
- **O(P) data structures quickly rear their heads**
- **Threading is a requirement for performance on Sequoia (BG/Q) for best performance**
- **SCR (Scalable Checkpoint-Restart) intercepts file I/O to main memory, and is in direct response to:**
  - **Increased file I/O times**
  - **Resilience issues at scale**

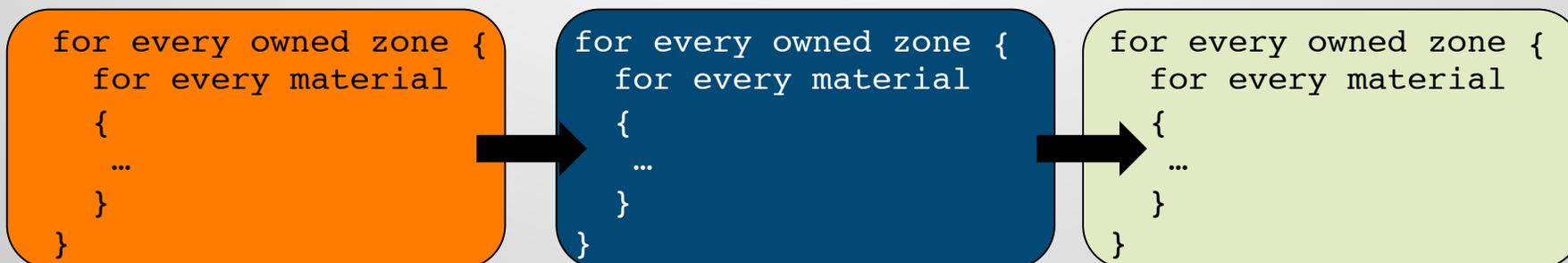
# We're dusting off our OpenMP books (and learning some new tricks, too)

- Too little work relative to the Overhead
  - Make sure time saved with parallelism exceeds overhead spent
- Shared Memory: Ensure all have latest data values (flushed)
- Data Race Conditions – Tricky & random, use tools to find!
  - Multiple threads updating data simultaneously
- Private variables, critical sections, & other restrictions
  - Unnecessary or excessive restrictions slows threads down
- Thread Scheduling / Chunking / Affinity (Multi-Socket)
  - Where will related thread run? Near data? Cache preload?
- Amdahl's Law still applies! Don't sequentialize unnecessarily
  - Time dominated by sequential sections as parallelism scaled up
- Plus, **Transactional Memory** (via compiler directives) is available on BlueGene/Q– early results are encouraging

# Sequential work loops are common in parallel applications

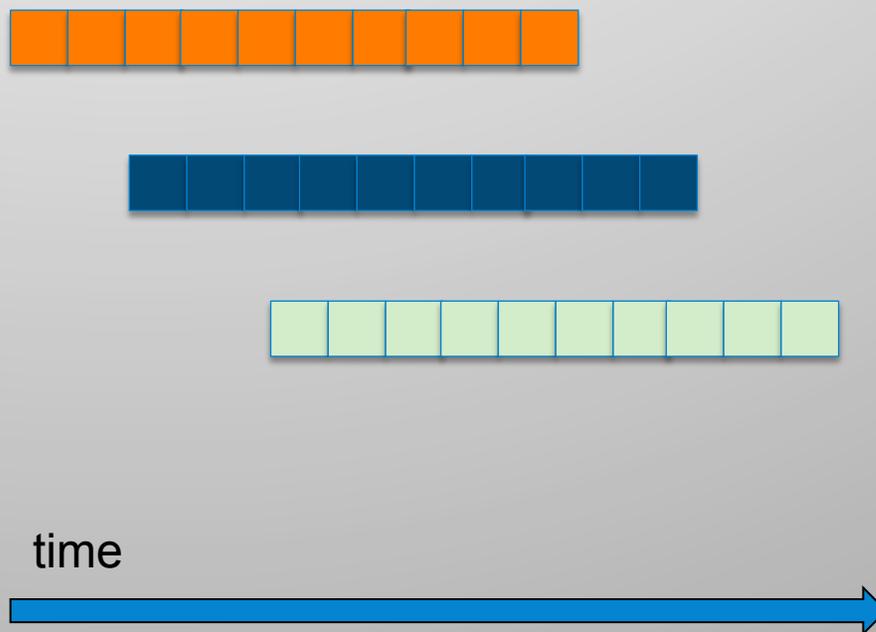


# We're exploring the use of the TBB pipeline construct to expose more parallelism (at the cost of additional complexity)



Once a segment of work from one loop is completed, its output becomes available as input to the next loop.

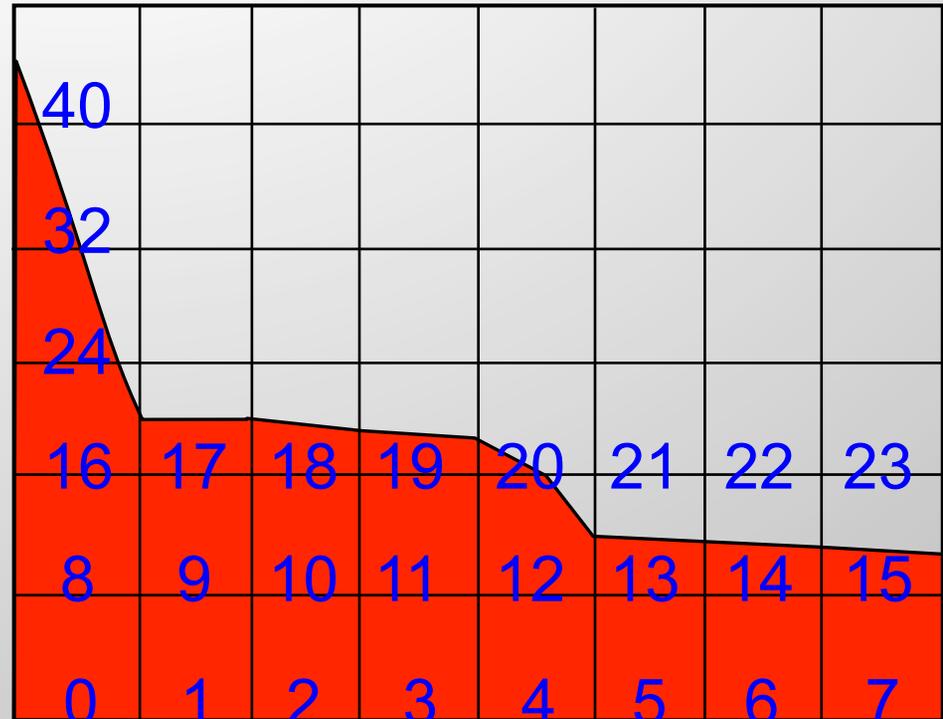
The syntax is a bit “disruptive”



# Index Sets are a common data structure for managing subsets

An index set defines a traversal over a subset of items in an ordered collection.

```
for ( int i = 0 ; i < len ; ++i ) {  
    // expression with  
    // "data[ index[ i ] ]"  
}
```



$$Z_M = \{ 0 - 20 , 24 , 32 , 40 \}$$

Indirection makes SIMD vectorization difficult or impossible (without gather/scatter)

# Index Set types and tradeoffs

$$\text{Recall } Z_M = \{ 0 - 20, 24, 32, 40 \}$$

## ■ Structured Range

- Consists of contiguous range (or IJK), possibly with stride
- High performance, but limited iteration patterns
- Traversal can vectorize well at compile time

## ■ Unstructured List

- Consists of a set of arbitrary index values
- Lower performance, but very flexible iteration patterns
- Not directly vectorizable, streams more data through cache

## ■ Hybrid

- Binds structured & unstructured sets in a single traversal construct
- Can yield best of both types, but normally requires add'l compiler support, source-to-source translation, or manual loop splitting

# Using hybrid “range” abstractions allows for multiple versions of the same loop

```
for ( int i = begin ; i < end ; ++i ) {  
    // expression with “data[ i ]”  
}
```

Structured

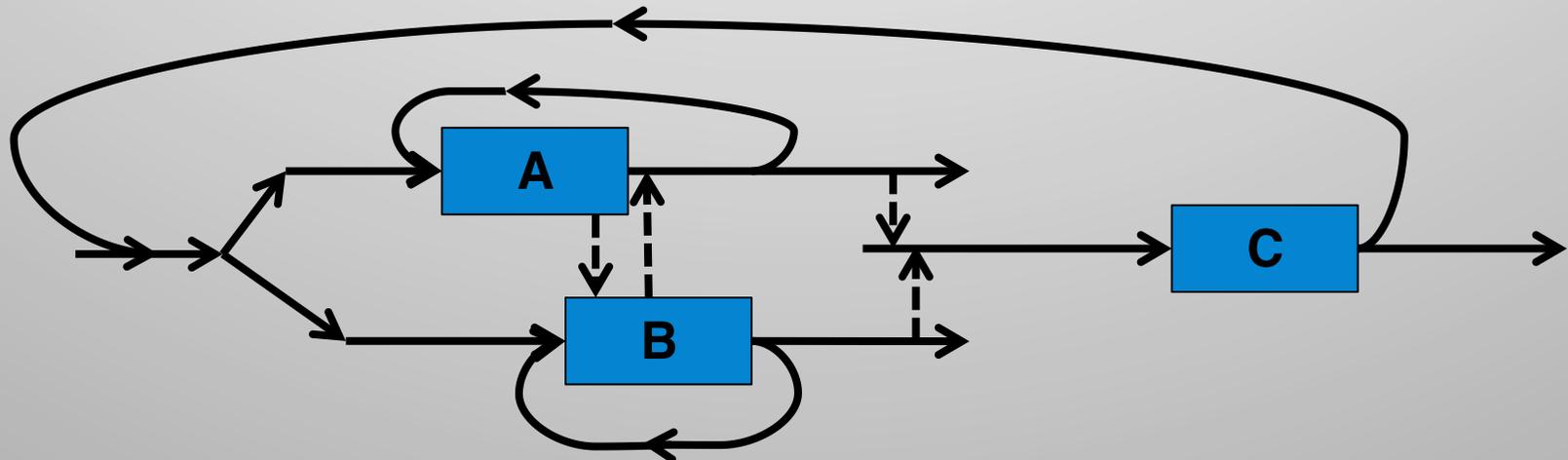
```
for ( int i = 0 ; i < len ; ++i ) {  
    // expression with “data[ index[ i ] ]”  
}
```

Unstructured

- + Allows detailed optimizations within each loop
- Hybrid traversal requires multiple loops & loop bodies
- Modification & specialization for platform-specific traversals requires changing loops throughout code

## Moving beyond the software pipeline provides a mechanism for exploiting additional concurrency

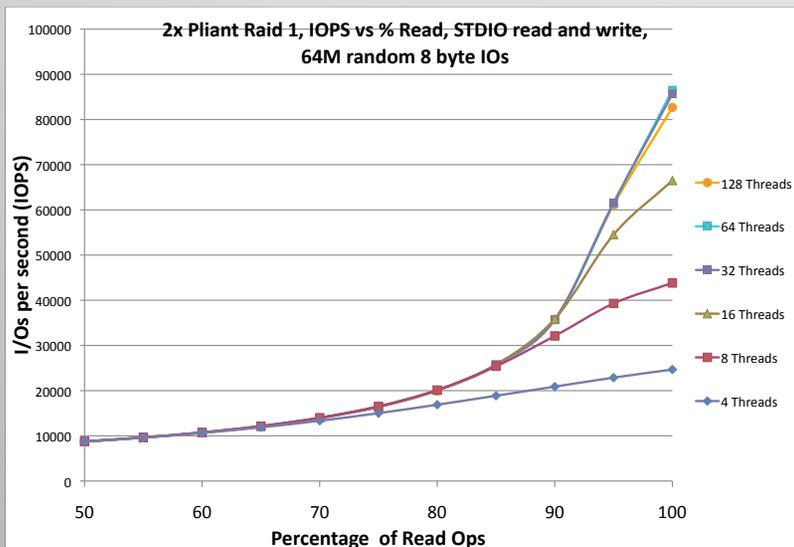
- Current codes process physics packages in a mostly serial fashion
- Future architecture challenge:
  - *Can physics packages be run simultaneously on different sets of processors?*
  - *What are the communication and accuracy constraints?*



Package A and B run simultaneously on different sets of processors and feed results to package C

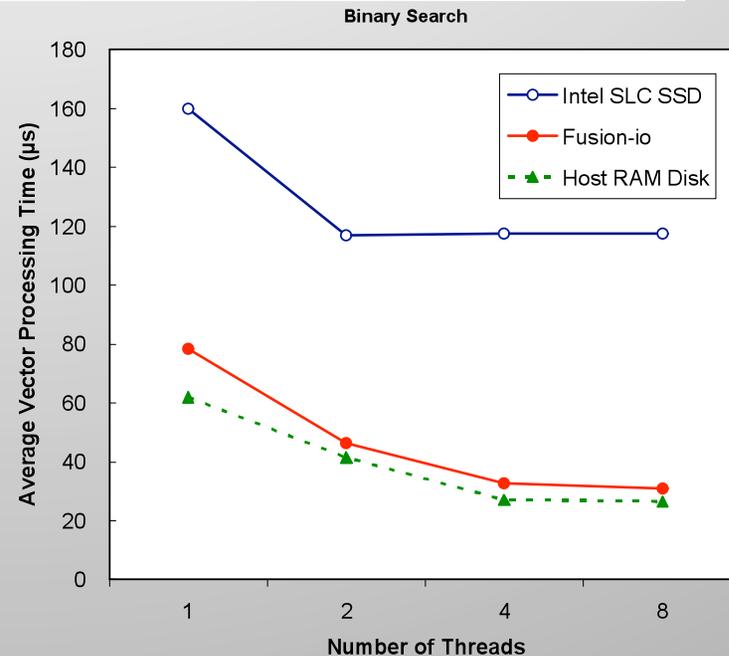
# We are studying the effects of persistent memory characteristics on our algorithms

Disk	Persistent Memory
Random access is bad	Random access is good
Reading <u>and</u> writing good	Reading is better than writing
Concurrent requests are bad	Concurrent requests are good



Courtesy: Maya Gokhale

There is a factor of 9× increase in number of I/Os per second for read-only access



Interconnect bandwidth impacts application run time by 2–3×

# Persistent variables are synchronized to persistent memory during a low latency checkpoint

```
template<class T>
struct PersistentType
{
    typedef std::vector<T,PERM_NS::allocator<T> >
    vector;

};
PERM struct Domain { ...
    PersistentType<Real_t>::vector m_x; /* coordinates */
    PersistentType<Real_t>::vector m_y;
    PersistentType<Real_t>::vector m_z;
... }
```

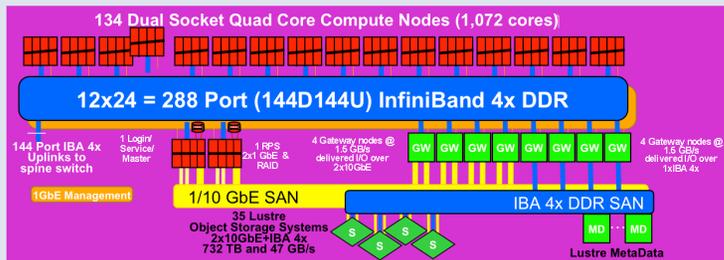
```
while(domain.time() < domain.stoptime() ) {
    if(ready_to_write){
        backup(); /* Persistent memory library call */
        ready_to_write = false;
    }
    TimeIncrement();
    LagrangeLeapFrog();
    if (domain.cycle() >= checkpoint_iter) break;
}
```

- The programmer designates certain variables as permanent
- These variables are allocated into the persistent memory and used normally in the program
- Checkpoints, at program points specified by programmer, copy the persistent memory region to a file
- Restart initializes persistent variables from the file

# One approach to checkpointing targets future exascale architectures

## Today: Explicit copying, global files

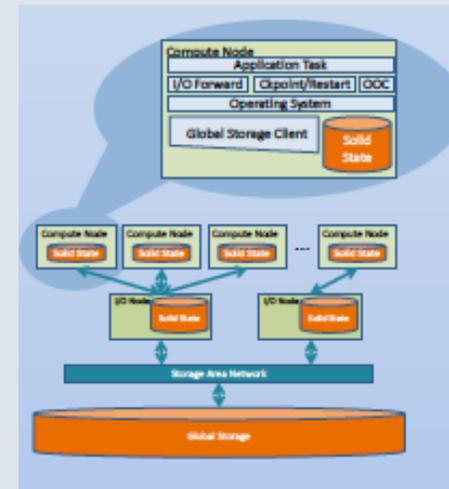
- Checkpoint files are created in a common format that a library manages.
- The application copies program variables to the checkpoint file using library calls.
- The checkpoint file is written to a global storage area network.



Today's clusters separate storage from compute

## Exascale: Implicit copy, local files

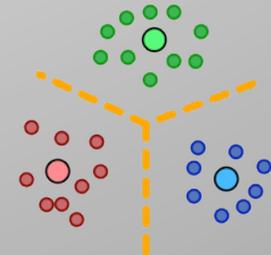
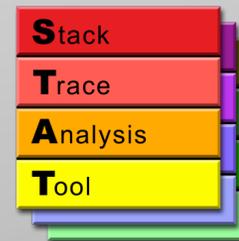
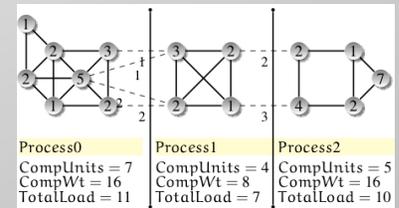
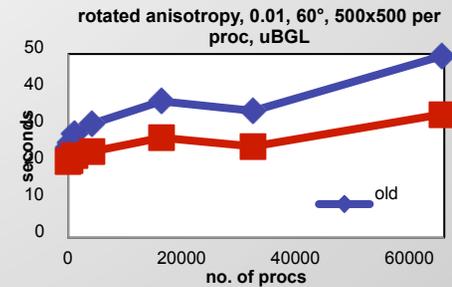
- The checkpoint file format is application specific.
- The application does not need to do explicit copy of individual variables.
- The checkpoint file is written to local persistent memory.



At exascale storage is in the compute cluster

# There are many research efforts ongoing under ExaCT

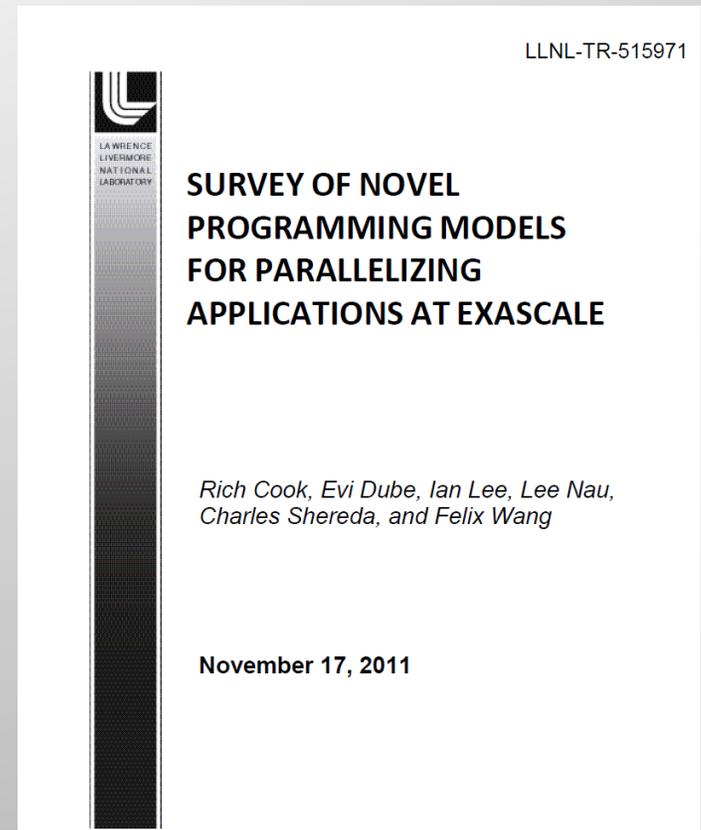
- **Algebraic Multigrid (AMG) Solvers**
  - Scalability, Performance Modeling
- **Resilience**
  - Scalable Checkpoint-Restart (SCR)
  - Algorithmic Fault Tolerance
- **Load Balance Analysis**
  - Evaluating the Effectiveness of Load Balance Algorithms
- **Multicore**
  - Memory Sharing with SBLLMalloc
- **Debugging**
  - Stack Trace Analysis Tool (STAT)
  - AutomaDeD & CAPEK



# Goal of Survey to Characterize Novel Programming Models that might have Applicability for Exascale

Characterization includes:

- The ease in learning and adopting these languages.
- The specific benefits to switching to the new language paradigm.
- The robustness of the model.
- The potential of this model to meet programming needs in the future, regardless of its present state.



# We characterized 10 systems spanning several data and control models

System (a)	Programming Model (b)	Data Model	Control Model
<b>Chapel</b>	Partitioned Global Address Space (PGAS)	Global memory view	Global view
<b>X10</b>	Asynchronous PGAS	Global memory view	Global view
<b>Fortress</b>	PGAS	Global memory view	Global view
<b>Cilk Plus</b>	Multithreaded	Global memory view (single node only)	Global view (single node)
<b>Intel Parallel Building Blocks</b>	Multithreaded	Global memory view (single node only)	Global view (single node)
<b>UPC</b>	PGAS	Global memory view	Global view
<b>Charm++</b>	Object-oriented	Local memory view	?
<b>AMPI</b>	Message passing	Local memory view	Local view
<b>OpenCL</b>	GPU language	GPU memory view (data is transferred to and from GPU memory)	Global view (single node)
<b>CUDA</b>	GPU language	GPU memory view (data is transferred to and from GPU memory)	Global view (single node)

The Appendix mentions Titanium, Global Arrays, ParallelX and High Performance ParallelX, writing Domain Specific Languages, and OpenMP Advancement

# Metrics included flexibility, data compatibility, ease of use, evolutionary shift to measure suitability to LLNL Apps

## Pros to a language:

- Data structures allow for adaptive meshes and sparse matrices
- Programming ease and elegance
- Domains distributed across locales of clustered system
- Simplifies, enhances data distribution
- Code based on C++, Fortran, Java so easy to learn

## Cons to a language:

- Dramatic change in approach
- Inability to exist as secondary language
- Not heavily tested as scientific app code
- Limited functionality

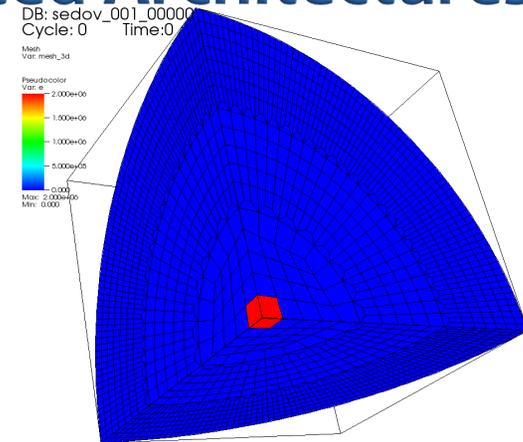
# Excellent example of confluence of merging efforts to propel LLNL forward to Advanced Architectures

April 2011

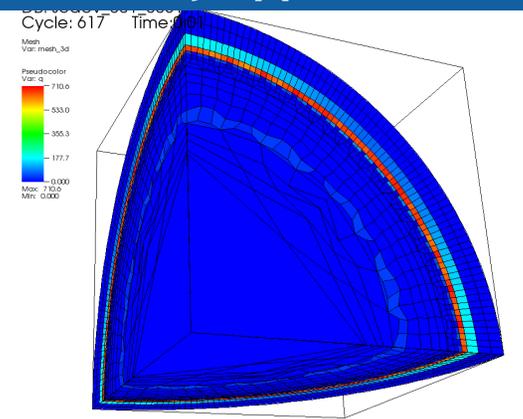
.....

March 2012

- Initial Development of LULESH Proxy App at Cray
- Programming Model Survey
- Invitation for Chapel lead to visit LLNL
  - Tutorial on Chapel basics
  - General enthusiasm from app developers
- LLNL gains basic familiarity, learns from initial LULESH port
- Reciprocated visit to Seattle
  - Block Coding -> Unstructured Coding ~ 6 hours
  - 25 extra lines of code!



## Proxy App Lulesh



# The message to our application developers must be clear

- **We cannot stand still**
  - **Concurrency, memory restrictions, memory bandwidth, vectorization, scaling, accelerators, resilience...**
  - **Programming models abound: languages, run-time systems, power and resilience management, ...**
  - **Even commodity clusters will be “advanced architectures” in coming years**
- **We can't do this alone - collaboration is more important than ever**
  - **Between code teams, internal lab efforts, labs, and NNSA and ASCR**
- **Despite the lack of well-funded post-petascale strategy, DOE is making significant progress**
  - **Three funded co-design centers**
  - **ASCR funded projects (e.g. X-stack)**
  - **FastForward RFP out**

# This is an *Evolutionary Revolution*



We are the 1% - and proud of it!

