Preparing our Multi-Physics Applications for Advanced/Future Architectures

Charm++ Workshop

May 7, 2012

Rob Neely

LLNL-PRES-556396
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

Overview

- **A bit on my background**
- **Some ASC perspective on exascale planning**
- **Multi-physics applications, and the challenges they present**
- **Co-design and proxy applications**
- **Efforts ongoing at LLNL in tackling exascale challenges**
- **Programming models survey**

Office of Science and NNSA have been planning an exascale initiative since at least 2009

Sustained joint government and industry research and development is needed to revolutionize processors, power, and programming.

Technical issues

- System power
- Memory
- Programming model
- Operating system
- Reliability and resiliency

Given the magnitude of the proposed investments, the novelty and challenges of a Science-NNSA joint effort and the lack of broad government consensus on the requirements for exascale, building this program has been extraordinarily difficult….

Exascale ramp up planning has been lengthy…

- **CY2008-2009**
  - Science drives Scientific Grand Challenge Workshops
- **CY2009**
  - ASC and ASCR charter laboratories to develop a Exascale Roadmap (E7 Group)
- **CY2010**
  - HQ Briefings to Koonin and D'Agostino
  - Decadal Cost Est: <= $6B
  - NNSA = $3B ($2B+) & Science = $3B ($1B+)
  - Presentations to OMB by ASCR and ASC
  - NNSA workshop on SW requirements for exascale
- **CY2011**
  - OMB pass back for FY12 forces slow start $126M
    - ~$40M Science and $6M ASC is "new"
  - Science codesign effort launched
  - Senate Letter "cannot cede leadership" to Obama
  - Kusnezov "what if we do nothing?" exercise
  - Congress requests a Plan of HQ – public on March 21. Will focus more on research first, platforms later as opposed to *ab initio* integrated effort
  - HQ disbands E7 'planning group' and replaces with E7 "exascale" execs – focused on 'execution'

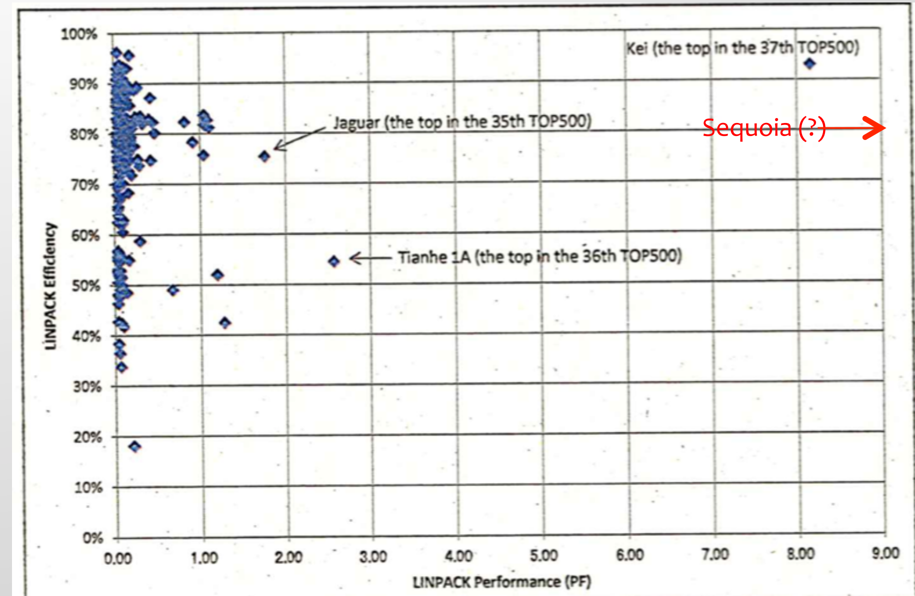
Scientific Grand Challenges Workshops

- Climate Science (11/08)
- High Energy Physics (12/08)
- Nuclear Physics (1/09)
- Fusion Energy (3/09)
- Nuclear Energy (5/09)
- Biology (8/09)
- Material Science and Chemistry (8/09)
- National Security (10/09)
- Cross-cutting technologies (2/10)

If we don't make aggressive changes to our ASC apps to account for fine-grained parallelism, and we end up with bandwidth and capacity memory limitations – the impact is that effective utilization of machines remains largely flat, even as they become > 100x faster in peak performance.

The challenge from China (and Japan and Russia and France and Germany and India) is real and will not go away

- China has three (3) architectural tag teams
  - Have recently held #1 position
  - Largely US technology today, but….
  - Exascale by 2018/19 (?)
    - — increasingly indigenous technology
  - Have told Intel they will hold all the top ten spots by 2015
  - Next: a concerted effort on apps: defense, industrial applications and science



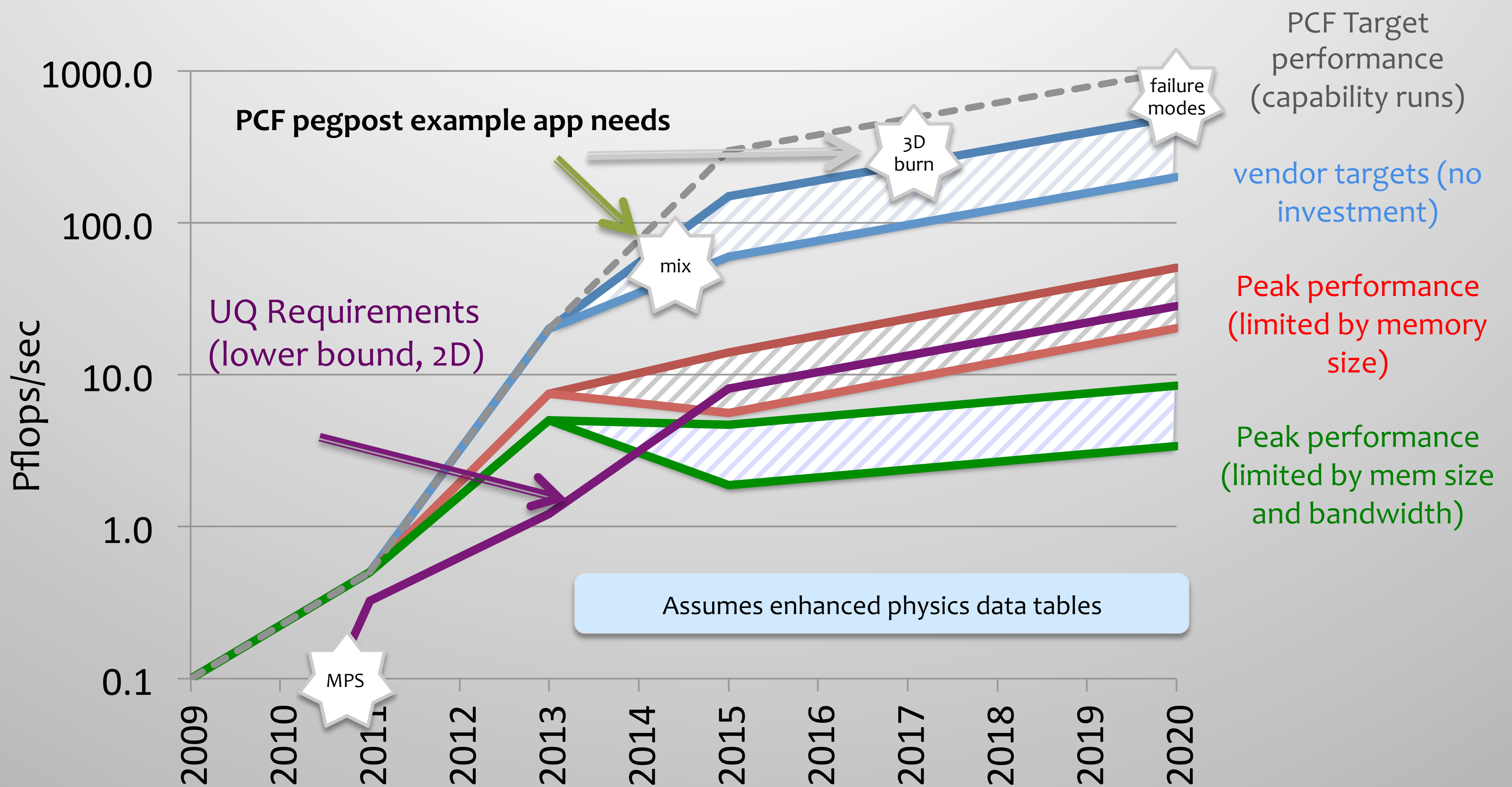
Minoru Nomura – Science and Technology Trends Quarterly review No. 21 Jan 2012

- Leadership is another word for control
  - Control the arc of high end IT innovation for the coming decades
  - Compete effectively in energy economy
  - Out compute in nuclear design and in assessment of adversary's devices?

"China is developing three new members of its home-grown Godson family of microprocessors. The most powerful new member of the family, Godson-3C, will have 16 CPU cores."
*3GHz * 16 * 8 = 384 GF/s/Processor*

Futuristic Chinese Center planned for Exascale

The impact of no exascale initiative on ASC application performance is potentially dire



ASC apps require major work to avail fine-grained parallelism. Vendor roadmaps currently incur memory bandwidth and capacity limitations. Thus, effective utilization of machines remains largely flat (bottom curve), even with > 100x peak performance. ASC programmatic demands continue rising (PCF pegposts and UQ curves above)

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

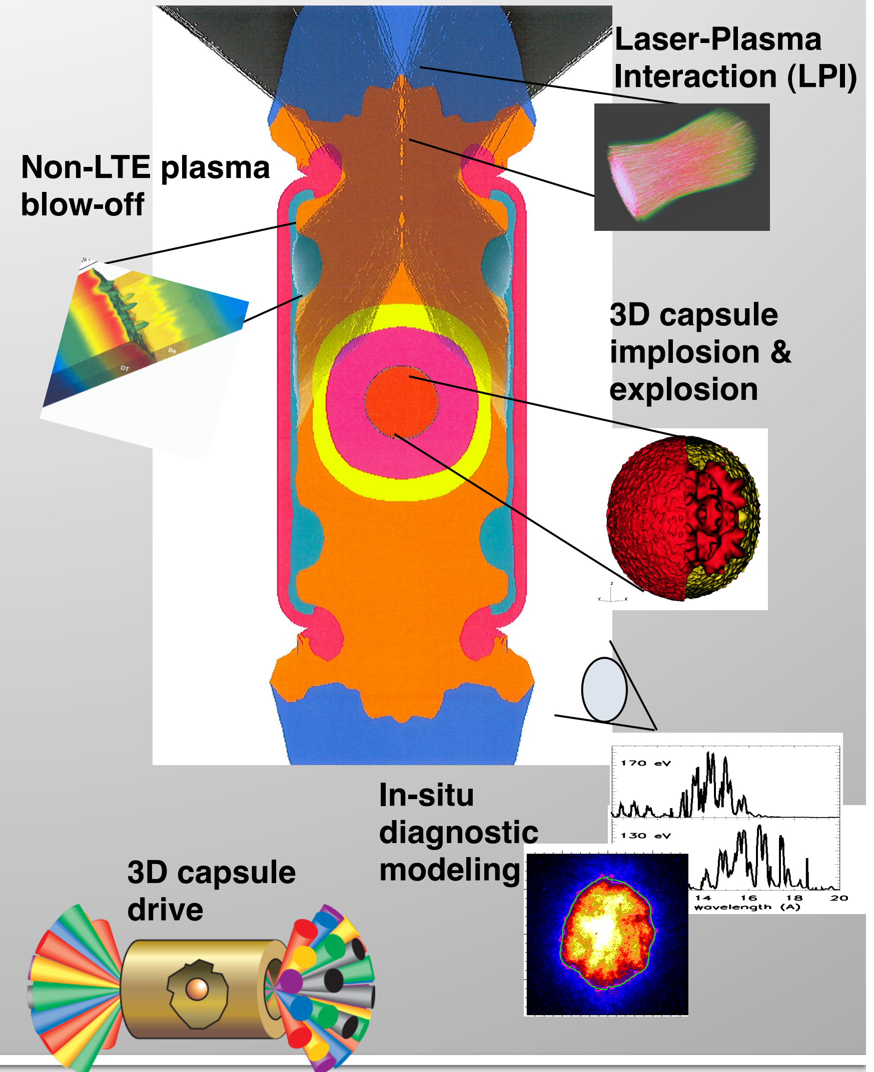
Improved Physics

- 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)

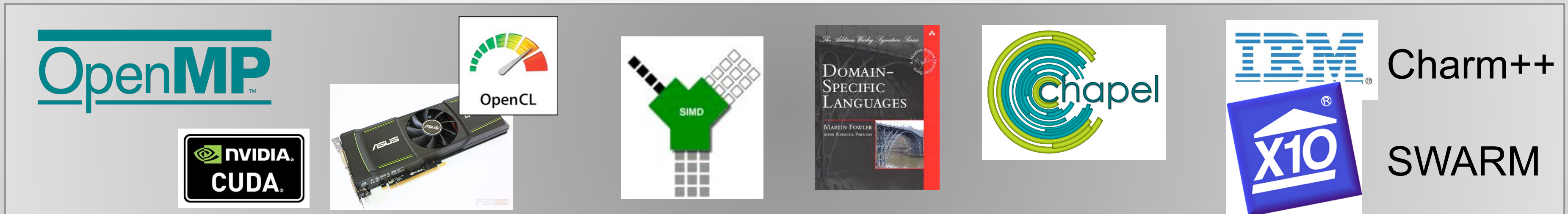


Our physics packages have differing computational requirements, making generalizations difficult

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

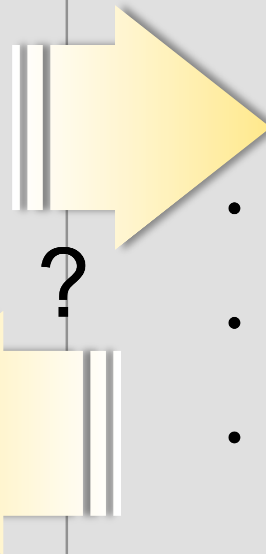
Typical Characteristics	Hydrodynamics	Deterministic Transport	Monte Carlo Transport	Diffusion
Memory needs	0.1 - 1 KB/zone	40 - 240 KB/zone	3 - 30 KB/zone	0.1 - 1 KB/zone
Memory access pattern	Regular with modest spatial and temporal locality	Regular, low spatial but high temporal locality	Irregular, low spatial and temporal locality	Regular, good spatial and temporal locality
Communication pattern	Point to point, surface communication	Point to point, some volume	Point to point, some volume	Collective communications and point to point
Mflops per zone per cycle	0.02 – 0.1 (10X for iterative schemes)	2 – 12	.03 - .07	0.1 - 3
I/O (startup data)	20-160 MB (EOS)	0.3 - 12 MB (Nuclear)	100 - 300 MB (Nuclear)	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)
- Implement fine-grained threading
- Application-controlled resilience
- GPU directives
- Leverage validated code base



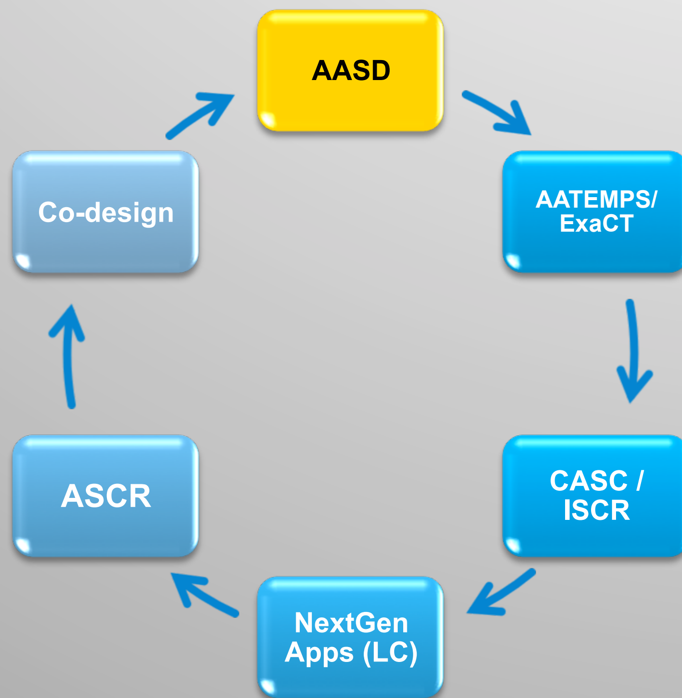
Undertake new "from scratch" rewrite

- 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

Advanced Architectures Software Development (AASD) Project

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

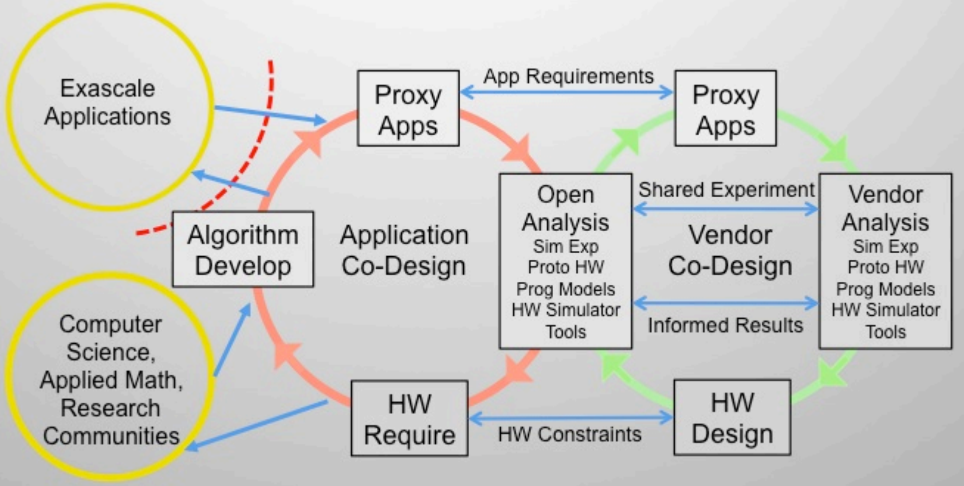


- Work with research and vendor community to identify promising and applicable technologies
- Inform programmatic funding of key technologies before they end due to lack of research funding

Current and projected AASD projects in the first 6-8 months of the effort

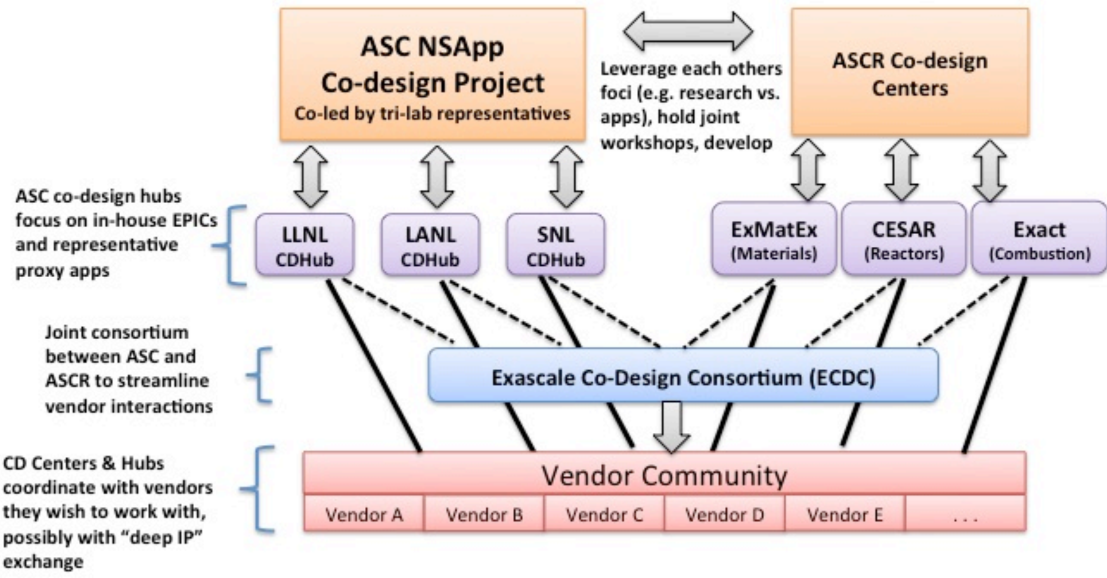
	Project	Goals
On-node concurrency	**Hybrid IndexSets**	Build general (DSL-like) abstractions for loop traversal over unstructured lists
	Exploiting SIMD in IndexSets	Automated ways to develop alternate loop bodies to exploit vectorization when available
	Threading Building Blocks	Explore the applicability of Intel TBB to Kull
	GPU programming – CUDA and directives	Exploring use of OpenACC style directives to extract performance on GPUs, with performance comparisons to hand-written CUDA
Memory models	**EOS data table sharing**	Share (read-only) EOS tables between MPI tasks in shared mem space
	SCR and NVRAM	Explore SCR (Scalable Checkpoint Restart) in a real application, attempt use of NVRAM storage for “burst buffers”
Proxy app	**Steering proxy app**	Build framework to explore combinations of front end (python, LUA, basis) and back end (C++, C, F90) code steering technologies
	Material library threading and vectorization	Explore threading of existing materials library, and what it will take to extract SIMD vectorization
Collab-oration	**Embed IC staff in other Co-design efforts**	ExMatEx: Learn and apply GREMLIN and ASPEN models CodEx: Deeper understanding of metrics and SST
	Proxy App relevance	Work with Heroux @ SNL to understand and apply results of their L2
Prog. Models	**Chapel 5-year plan**	Establish co-design relationship with Chapel with a goal of establishing it as basis for future application design
	Dynamic run-time systems	Explore dynamic PM’s using Charm++ as proxy for HPX, SWARM, etc…

NNSA/ASC is developing a co-design strategy in partnership with Office of Science



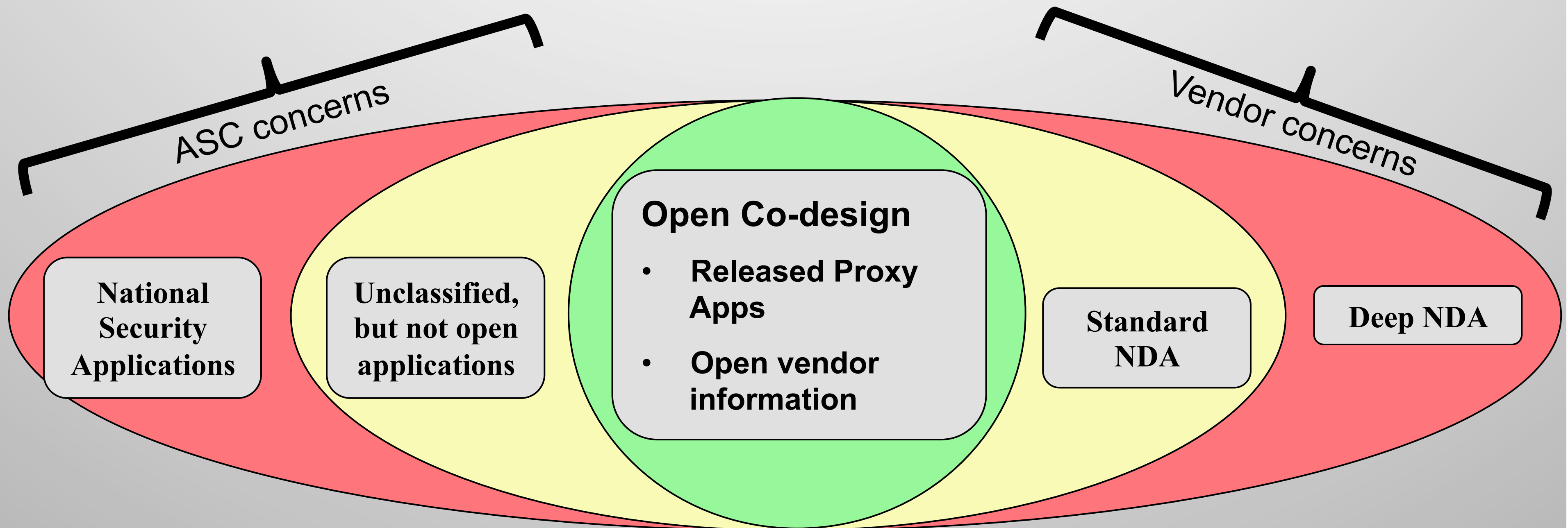
Application teams collaborating closely with hardware and system software designers to inform and influence architectural trade-offs

Proposed DOE co-design ecosystem (in progress)



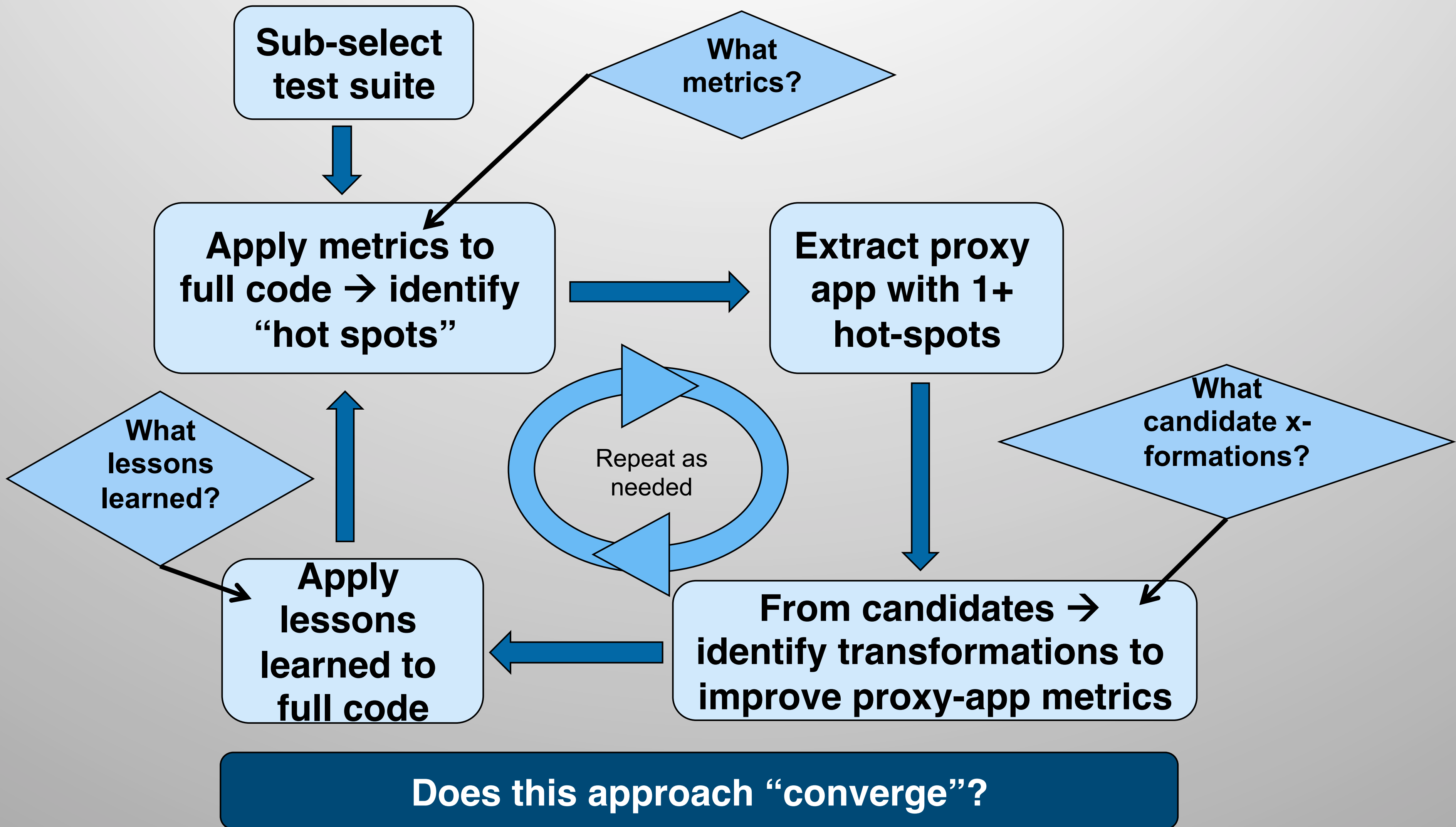
(One of) the difficulties of co-design

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**

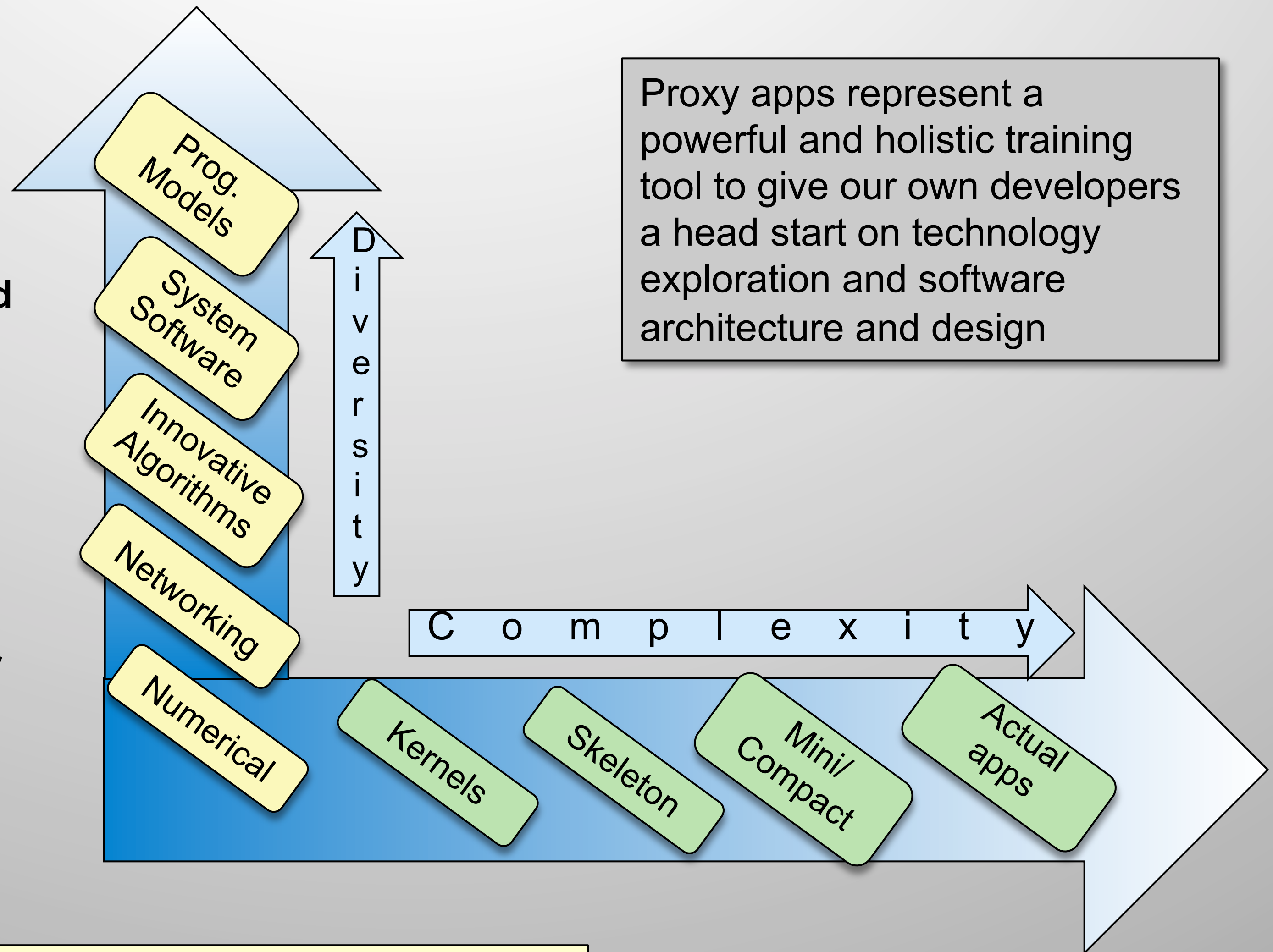
Proxy applications are a core strategy for co-design



Does this approach "converge"?

Proxy apps development is being pursued strategically along two axes

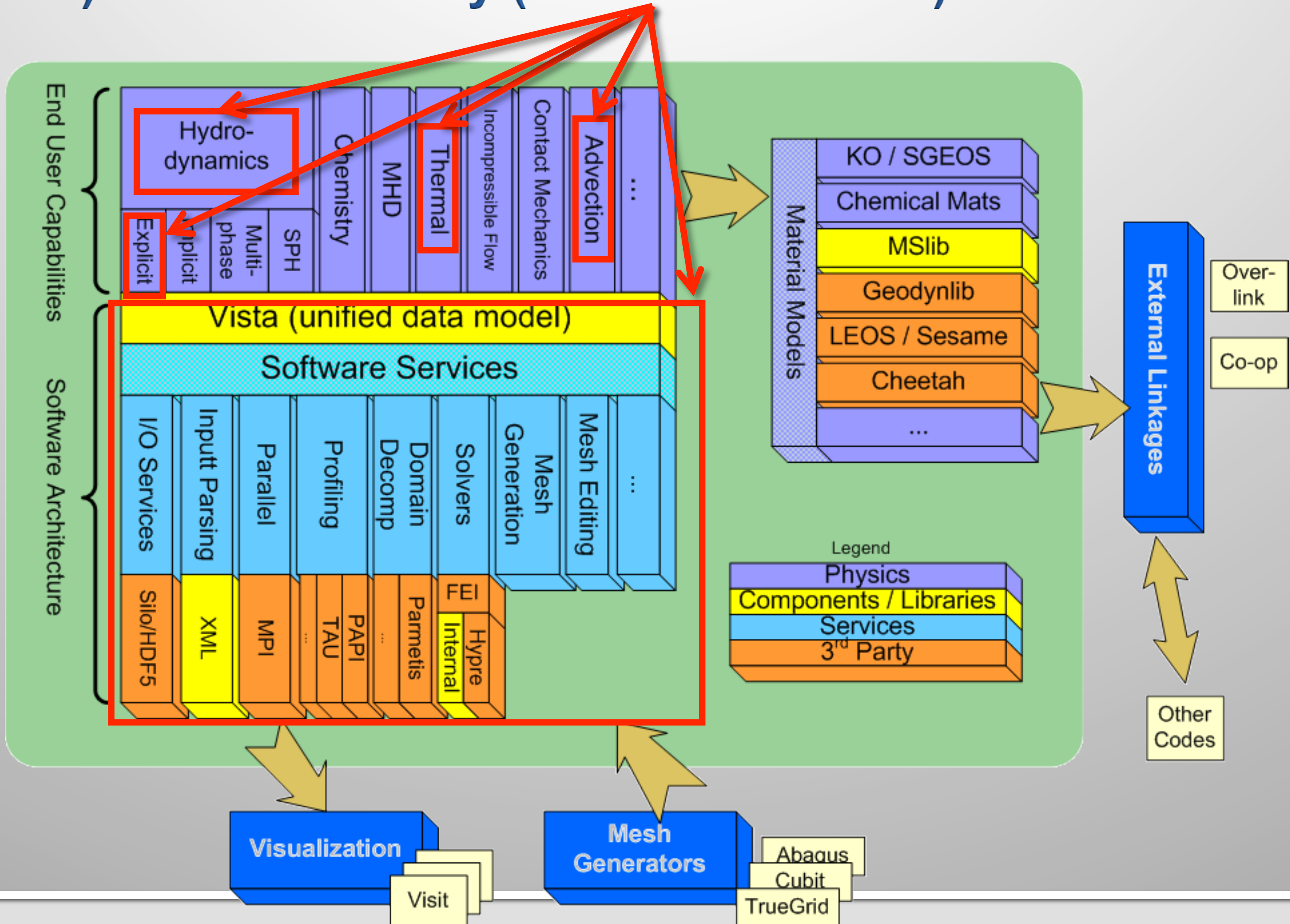
- **Simple, open, and easy to pick up and explore**
- **Must accurately represent original applications**
- **The collection should account for more than just fast numerical performance**



Proxy apps represent a powerful and holistic training tool to give our own developers a head start on technology exploration and software architecture and design

These are more than just a benchmark

LLNLis developing a large multi-physics compact app (xALE) to use for study (based on ALE3D)



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

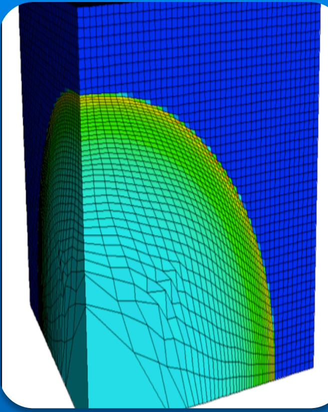
Sequoia Benchmark	Exists / released	Exists / unreleased	Under development	Undeveloped

Current list (with download links) will be available at http://codesign.llnl.gov

* May be restricted

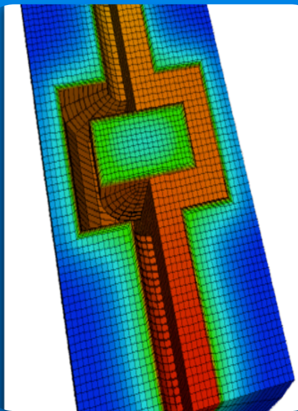
Co-design

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

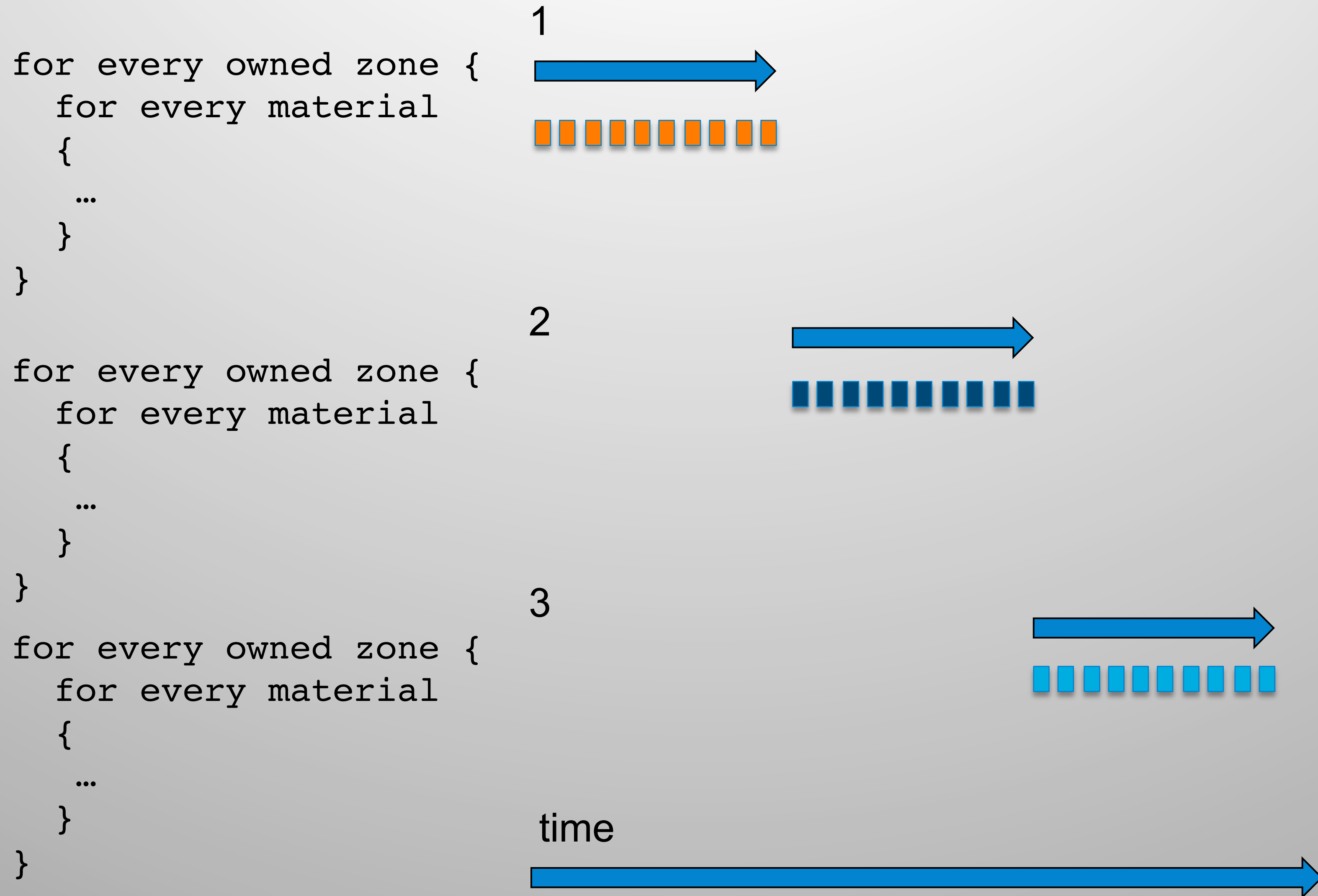
Experience on existing platforms is giving us insight into scalability for upcoming petascale architectures

- **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**

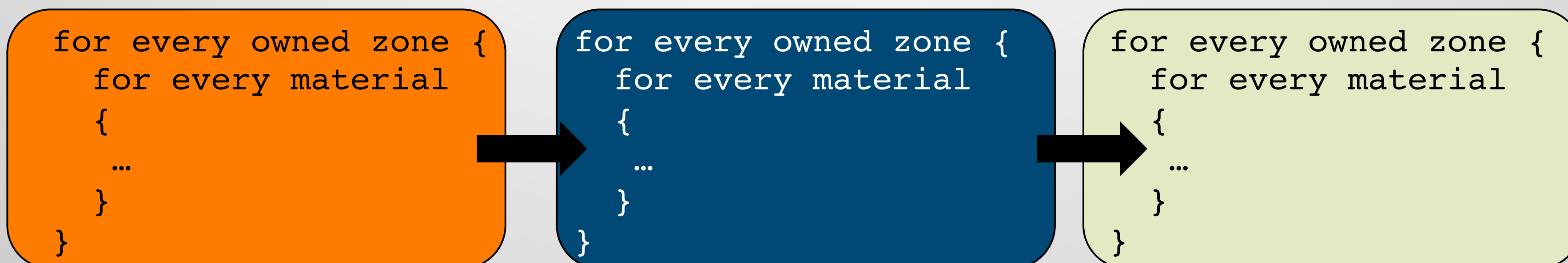
We're dusting off our OpenMP books (and learning some new tricks, too)

- **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

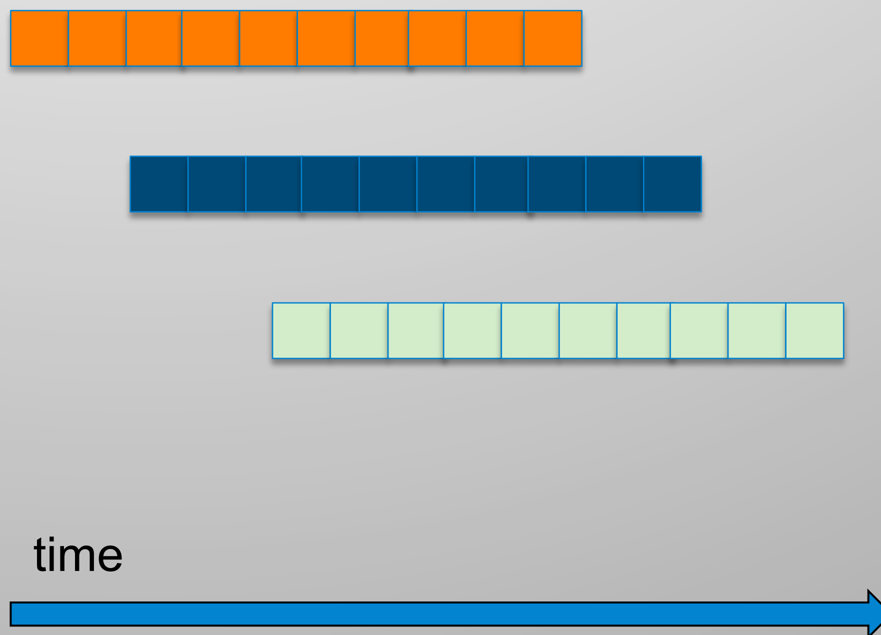


We're exploring the use of the TBB pipeline construct to expose more parallelism (at the cost of additional complexity)



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"

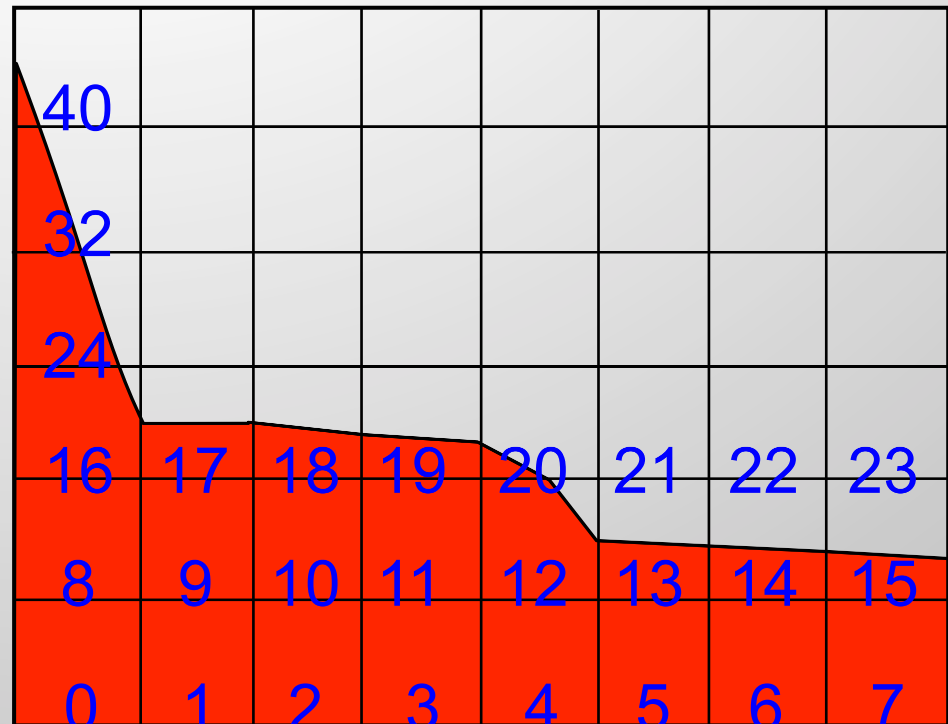


Index Sets are a common data structure for managing subsets

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 ] ]"
}
```

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



$$Z_M = \{ 0 – 20 , 24 , 32 , 40 \}$$

Index Set types and tradeoffs

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

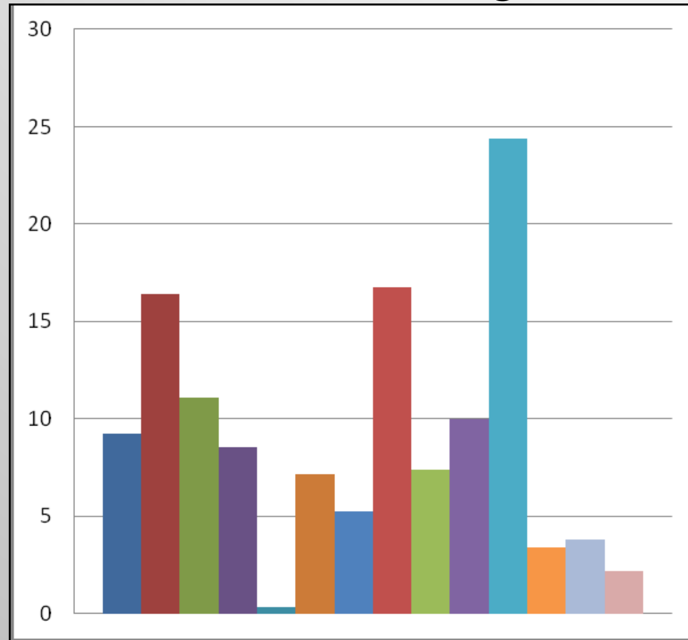
```
for ( int i = 0 ; i < len ; ++i ) {

   // expression with "data[ index[ i ] ]"
}
```

Unstructured

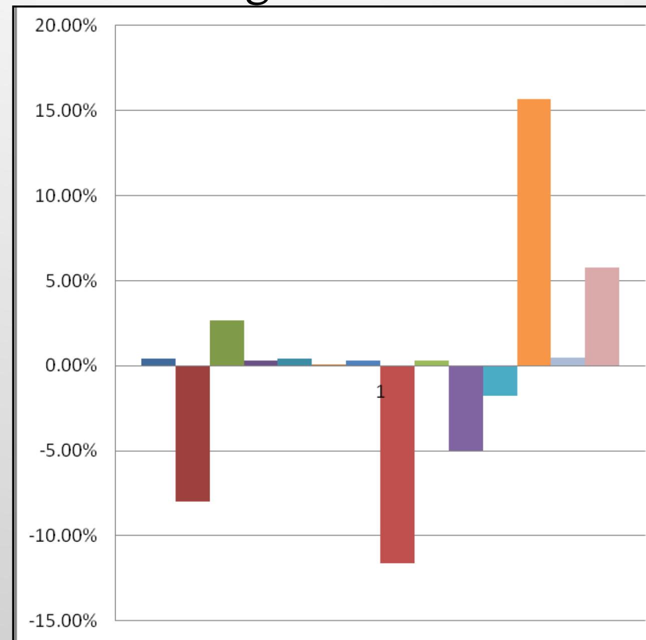
+ Allows detailed optimizations within each loop
– Hybrid traversal requires multiple loops & loop bodies
– Modification & specialization for platform-specific traversals requires changing loops throughout code

GPU explorations on LULESH mini-app
Current double-precision speedup is 9.8X (16.0X single)

Speedup by stage

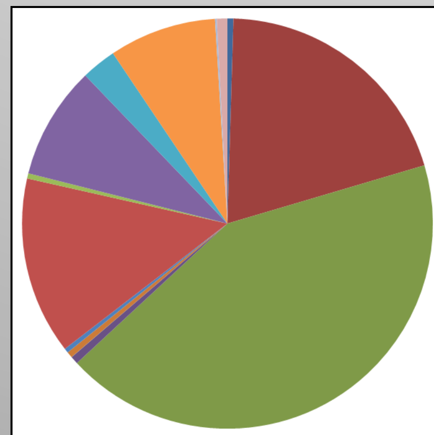


Change in run-time %

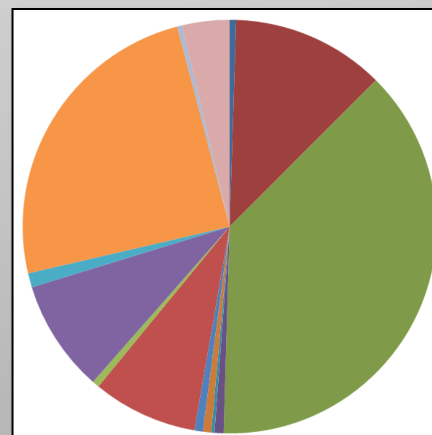


- ElemInitStress
- ElemIntegrateStress
- ElemCalcHourglass
- NodeAcceleration
- NodeAccelBC
- NodeVelocity
- NodePosition
- ElemKinematics
- ElemLagrangePt2
- ElemMonotonicQGrad
- ElemMonotonicQ
- ElemMaterialProp
- ElemVolume
- ElemTimeConstraints

CPU run-time %

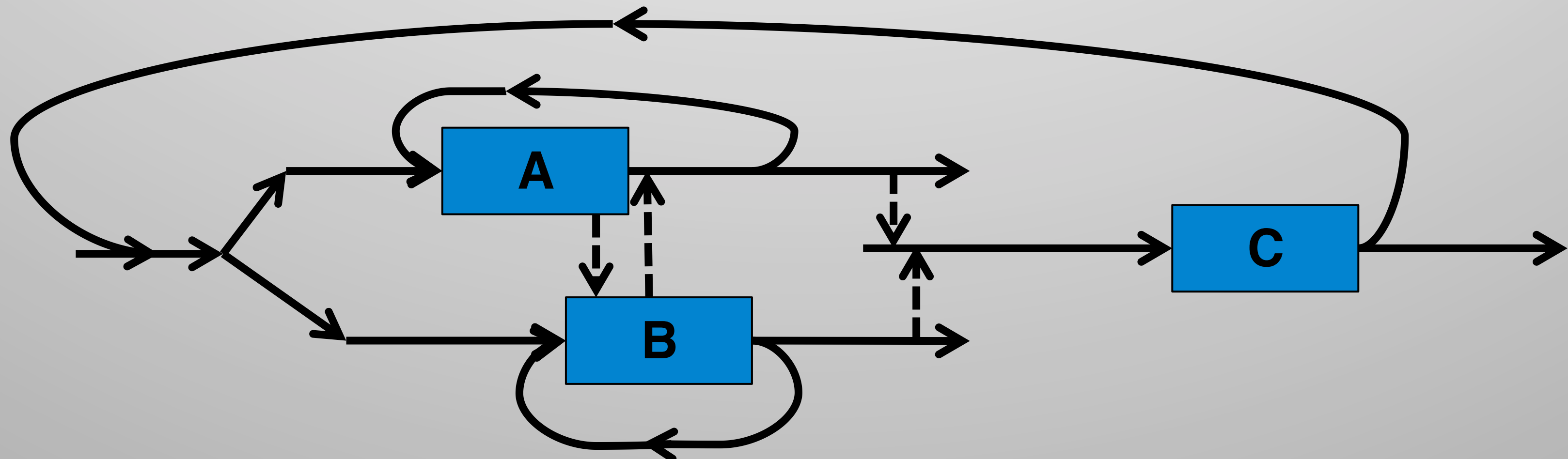


GPU run-time %



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

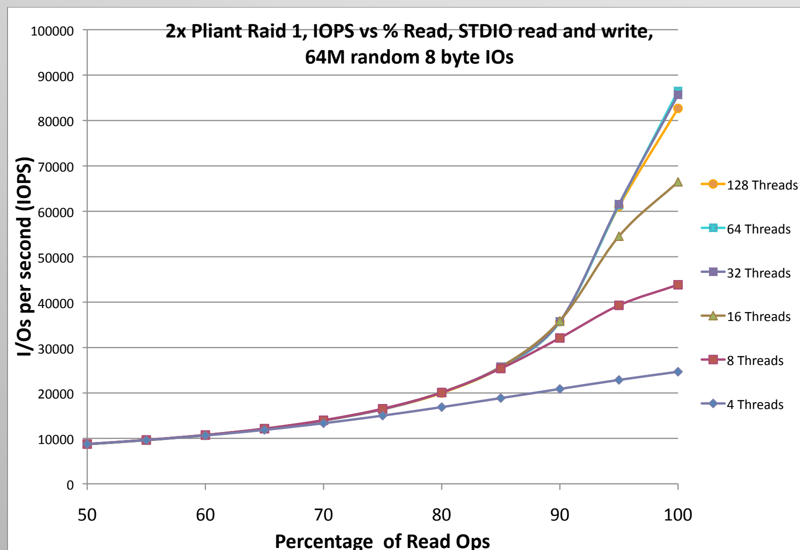
- **Current codes process physics packages in a mostly serial fashion**
- **Future architecture challenge:**
  - ***Can physics packages be run simultaneously on different sets of processors?***
  - ***What are the communication and accuracy constraints?***



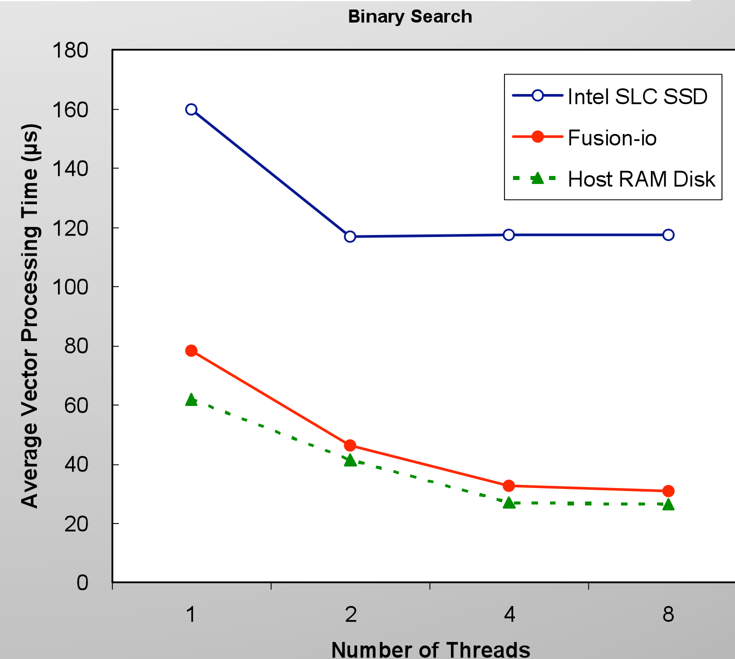
Package A and B run simultaneously on different sets of processors and feed results to package C

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

Disk	Persistent Memory
Random access is bad	Random access is good
Reading and 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 ;
…
}
```

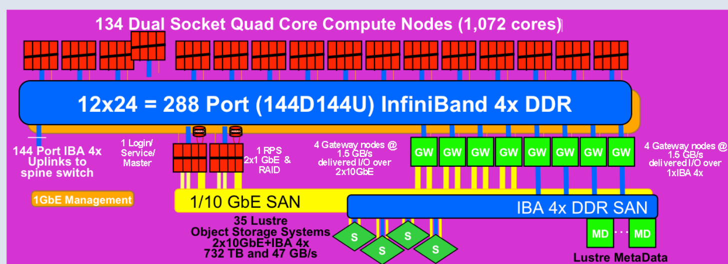
```
while(domain.time() < domain.stoptime() ) {
    if(ready_to_write){
     backup();  /* Persistent memory library call */
     ready_to_write = false;
    }
     TimeIncrement() ;
     LagrangeLeapFrog() ;
 if (domain.cycle() >= checkpoint_iter) break;
  }
```

- 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

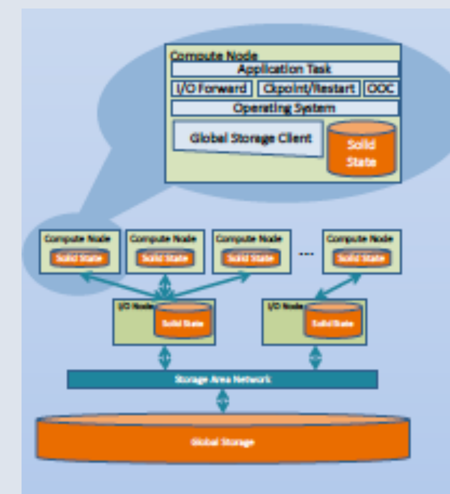
- Checkpoint files are created in a common format that a library manages.
- The application copies program variables to the checkpoint file using library calls.
- The checkpoint file is written to a global storage area network.



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

Source-to-source Compiler Resiliency Transformations for *Processor* Soft Errors

Original Source Code

```
void relax ()
   {
#pragma resiliency elemental
      for (int i = 1; i < arraySize-1; i++)
            array[i] = (array[i-1] + array[i+1]) / 2.0;
   }
```

Transformation

Generated Source Code

```
void relax_tmr_elemental ()
   {
     for (int i = 1; i < arraySize-1; i++)
        {
          register float var1a = array[i];
          register float var2a = array[i-1];
          register float var3a = array[i+1];

          register float var1b = array[i];
          register float var2b = array[i-1];
          register float var3b = array[i+1];

          register float var1c = array[i];
          register float var2c = array[i-1];
          register float var3c = array[i+1];

          var1a = (var2a + var3a) / 2.0;
          var1b = (var2b + var3b) / 2.0;
          var1c = (var2c + var3c) / 2.0;

          if (var1a != var1b || var1a != var1c)
            {
           // Handle arbitration by recomputing value.
              printf ("Detected an error...\n");
            }
        }
   }
```

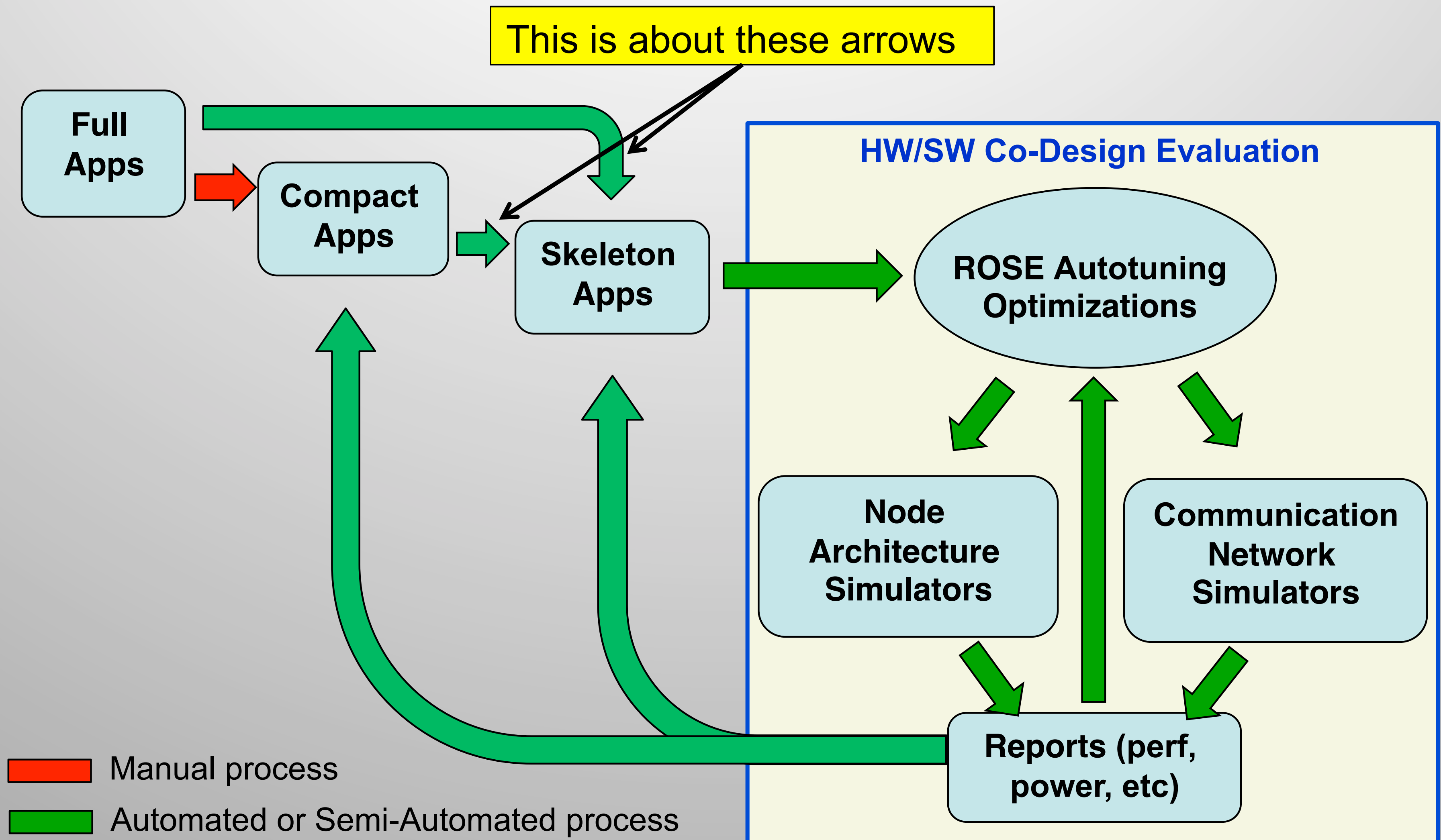
Work done 3 times

Test for same results

- *Triple Modular Redundancy* as a compiler transformation
- Leverages ROSE source-to-source compiler
- Targets soft errors in processor hardware
- Could be supported directly via pragmas in the code for semi-automated solution
- Compliments memory resiliency checking (previous slide)
- Optimizations for memory reuse
- Control over where separate computations could be done:
  - Same cores
  - Separate cores, processors, sockets, nodes … planets ☺
  - Threaded solutions …
- ***ROSE Compiler Work is now being released…***

CoDesign Tool Flow using ROSE

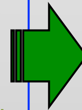
Automatic Generation of Skeletons for Rapid Analysis



Example of Automated Skeleton Code Generation: Before/After

Before

```
do {
    if (rank < size - 1)
        MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE,
                  rank + 1, 0, MPI_COMM_WORLD );
    if (rank > 0)
        MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
                  MPI_COMM_WORLD, &status );
    if (rank > 0)
        MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                  MPI_COMM_WORLD );
    if (rank < size - 1)
        MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE,
                  rank + 1, 1, MPI_COMM_WORLD, &status );
    itcnt ++;
    diffnorm = 0.0;
    for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++) {
            xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                         xlocal[i+1][j] + xlocal[i-1][j]) /
    4.0;
            diffnorm += (xnew[i][j] - xlocal[i][j]) *
                        (xnew[i][j] - xlocal[i][j]);
        }
    for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++)
            xlocal[i][j] = xnew[i][j];
    MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE,
                   MPI_SUM, MPI_COMM_WORLD );
    gdiffnorm = sqrt( gdiffnorm );
    if (rank == 0) printf( "At iteration %d, diff is %e\n",
                           itcnt, gdiffnorm );
} while (gdiffnorm > 1.0e-2 && itcnt < 100);
```

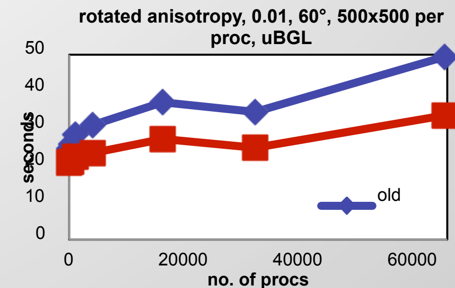


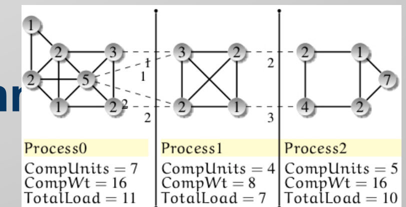
After

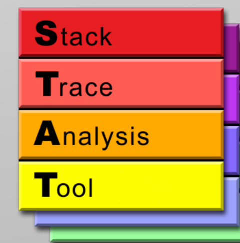
```
do {
   if (rank < size - 1)
       MPI_Send( xlocal[maxn / size], maxn, MPI_DOUBLE,
                 rank + 1, 0, MPI_COMM_WORLD )
   if (rank > 0)
       MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
                 MPI_COMM_WORLD, &status );
   if (rank > 0)
       MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                 MPI_COMM_WORLD );
   if (rank < size - 1)
       MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE,
                 rank + 1, 1, MPI_COMM_WORLD, &status );
   itcnt ++;

   MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD );

} while (gdiffnorm > 1.0e-2 && itcnt < 100);
```

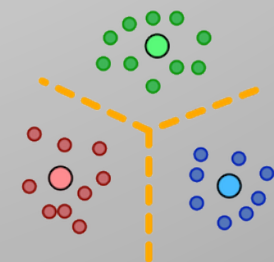
There are many research efforts ongoing under ExaCT (LDRD)

- **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





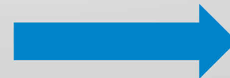


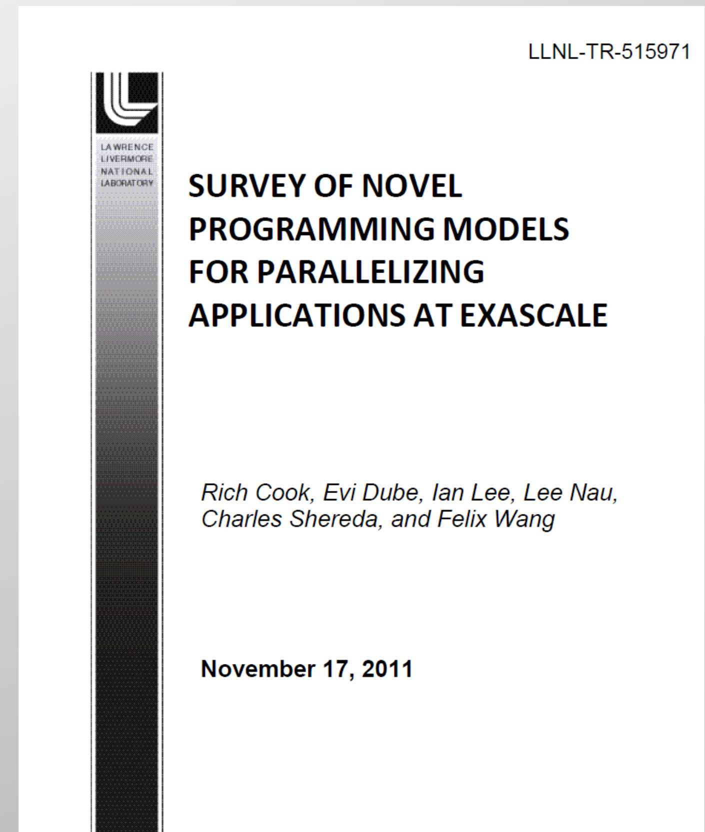


Goal of Survey to Characterize Novel Programming Models that might have Applicability for Exascale

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.




LLNL-TR-515971

LAWRENCE LIVERMORE NATIONAL LABORATORY

SURVEY OF NOVEL PROGRAMMING MODELS FOR PARALLELIZING APPLICATIONS AT EXASCALE

Rich Cook, Evi Dube, Ian Lee, Lee Nau, Charles Shereda, and Felix Wang

November 17, 2011

We characterized 10 systems spanning several data and control models

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

Characterizations provide basic overview to allow developer to determine if further investigation is warranted

Owner / Development Location	Cray Inc. (head of team is based in Seattle, WA)
Project Website	http://chapel.cray.com/index.html
Download Page	http://chapel.cray.com/download.html
Platforms Available	Most UNIX-based systems, Mac OS X, Windows. Works in conjunction with the GASNet library which works with various interconnects.

Each characterization starts with the information above and

- Overview
- Present State of the Model
- Tool Availability
- Performance
- Suitability to LLNL Application Codes
- Resources and Additional Information and/or Bibliography

A characterization leaves the developer with future reference

- Language Specs
- Tutorials
- Presentations and Videos
- Programmer's Assistance
- Wiki's
- Papers, Articles, Journals
- Downloads

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

Pros to a language:

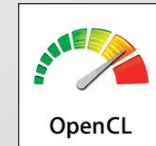
- Data structures allow for adaptive meshes and sparse matrices
- Programming ease and elegance
- Domains distributed across locales of clustered system
- Simplifies, enhances data distribution
- Code based on C++, Fortran, Java so easy to learn

Cons to a language:

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

Our research culminated with a set of suggestions regarding these models

- We recommend a further study of Chapel – specifically, an application port.
- We recommend monitoring X10 & Intel PBB.
- We recommend MPI support staff familiarize themselves with Charm++ /AMPI and to see if some of its innovations can be applied to issues such as fault tolerance at large scale.
- We recommend maintaining expertise in OpenCL and CUDA but caution against developing a significant codebase, especially in CUDA, which is proprietary.



Intel Cilk +

Charm++/ AMPI

X10

Fortress

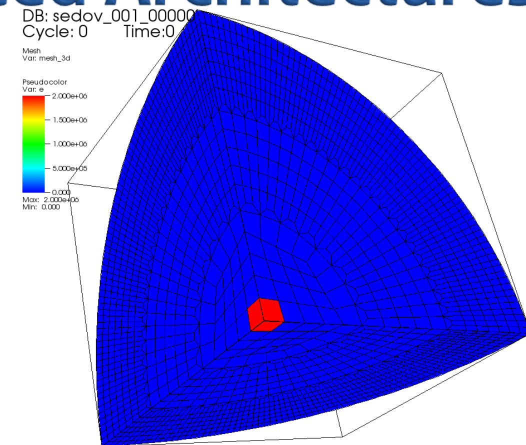
Intel ArBB/TBB

Report available at:
https://asc.llnl.gov/exascale/references.php (Under "Miscellaneous")

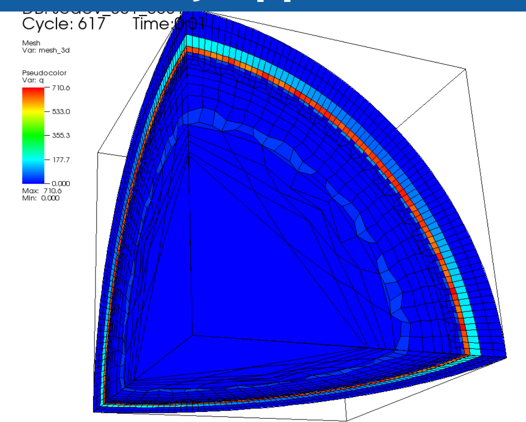
Excellent example of confluence of merging efforts to propel LLNL forward to Advanced Architectures

April 2011 …… March 2012

- Initial Development of Proxy App
- Programming Model Survey
- Invitation for Chapel lead to visit LLNL
- LLNL gaining basic familiarity
- 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

Charm++ offers an opportunity to explore dynamic run-time programming models

- Clearly, Charm++ has "staying power"!
- Until now, MPI (with occasional coarse-grained threading) has carried the day
  - No longer…
- Understanding the benefits of programming models such as Charm++ or AMPI on our algorithms is a desired goal
  - Need one or more proxy apps that demonstrate advantages
- Has Charm++'s "time to shine" arrived?
  - Let's find out together!
- Goal for next years' Charm++ workshop – LLNL/Charm++ success stories

Lawrence Livermore
National Laboratory