LA-UR-13-29322
Approved for public release; distribution is unlimited.

Title: Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx) Annual Report - Year 2

Author(s): Germann, Timothy C.
McPherson, Allen L.
Belak, James F.
Richards, David F.

Intended for: Report

Issued: 2013-12-09

Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx) Annual Report — Year 2

Los Alamos, Lawrence Livermore, Sandia, & Oak Ridge National Laboratories
Stanford University & California Institute of Technology
http://exmatex.org

November 21, 2013

Summary: All activities of the Exascale Co-design Center for Materials in Extreme Environments (ExMatEx) are focused on the two ultimate goals of the project: (1) demonstrating and delivering a prototype scale-bridging materials science application based upon adaptive physics refinement, and (2) identifying the requirements for the exascale ecosystem that are necessary to perform computational materials science simulations (both single- and multi-scale). During the first year of ExMatEx, our focus was on establishing how we do computational materials science, by developing an initial suite of flexible proxy applications. These "proxy apps" are the primary vehicle for the co-design process, involving assessments and tradeoff evaluations both within the ExMatEx team, and with the entire exascale ecosystem. These interactions have formed the basis of our second year activities. The set of artifacts from these co-design interactions are the lessons learned, that are used to re-express the applications and algorithms within the context of emerging architectures, programming models, and runtime systems.

1 ExMatEx Goals, Objectives, and Year 2 Milestones

Exascale computing presents an enormous opportunity for solving some of today's most pressing problems, including clean energy production, nuclear reactor lifetime extension, and nuclear stockpile aging. At their core, each of these problems requires the prediction of material response to extreme environments. Our Center's objective is to establish the interrelationship between software and hardware required for materials simulation at the exascale while developing a multiphysics simulation framework for modeling materials subjected to extreme mechanical and radiation environments.

When considering the exascale computing drivers and use cases within the context of materials in extremes, the increased computational power is increasingly needed to improve the *fidelity* of sub-scale physics models, such as the equation-of-state and strength models used in engineering-scale hydrocodes, rather than more traditional brute force increases in mesh size (and dimensionality) or timescale. While great effort has gone into developing transferable models valid across the entire range of parameter or phase space likely to be encountered, the large computational cost and often unknown accuracy of such models is unsatisfactory. Based upon these concerns about the physics fidelity, and the emerging architecture trends towards massive concurrency and asynchronous, dynamic approaches rather than traditional bulk synchronous parallel models, our science strategy is an adaptive physics refinement in which coarse-scale simulations dynamically spawn tightly coupled and self-consistent fine-scale simulations as needed. This strategy is crucial for capturing how the macroscale, bulk response is influenced by microstructural detail. Thus, in a high strain-rate loading problem, a finite element method calculation may spawn finer-scale crystal plasticity or atomistic models as needed when the available empirical constitutive model is inadequate.

As originally stated in our March 1, 2011 planning report [1], and kept unchanged during our annual reassessment of our plans, we targeted four Level-2 (L2) milestones during Year 2 (Y2):

1. ***Use SST simulation and GREMLIN interface layers to mimic exascale machine behavior on petascale platforms.*** As described in more detail in Section 2.3.1, a key accomplishment this year was the development and application of GREMLINs for power, performance, and resilience challenges at exascale. A conclusion from these studies, and from an SST hackathon we organized, was that there is a great need for performance modeling of advanced memory systems, including multilevel memory support and cache coherency protocols, e.g. to model interactions between OpenMP threads; such capabilities were the focus of this year's ExMatEx-supported SST work, as detailed in Section 2.3.2.

2. ***Identify critical features of programming models*** This was largely accomplished through the various hackathons with vendor and X-stack partners, another key accomplishment this year which is described more fully in Sections 2.1.2 and 2.2.1.

3. ***Assess and deliver data/resource sharing requirements, for both scale-bridging and in situ analysis/viz to exascale software partners.*** Our work on scale-bridging has followed two convergent paths: a top-down analysis and proxification of our target adaptive sampling scale-bridging application, as well as the bottom-up development of a flexible scale-bridging proxy to perform our initial assessments of programming models and runtimes, as described below in Section 2.2.3.

4. ***Release latest instantiation of Aspen/SST, GREMLIN, scalable tools used for evaluation and proxy apps to exascale ecosystem.*** Although we initially developed and applied these tools to ExMatEx proxy applications for our own co-design tradeoff analysis, they are broadly applicable by the wider community, including other application co-design centers and vendors. The 3-state cache coherency version and OpenMP support within SST has been released, as have the GREMLIN framework and individual GREMLINs, both described in Section 2.3. In addition, updated versions of the CoMD, VPFFT, CoGL, and ASPA proxy apps have been released on GitHub within the past year, as well as CoHMM's initial release and this month's updated CoMD within the Mantevo Suite Release 2.0. These proxy apps have provided the central mechanism for our co-design engagements, as described in Section 2.1.

These activities have led to **three key accomplishments in Y2**:

1. Multiple **deepdive hackathons** with our Fast Forward and X-stack partners using proxy applications has proven to be an extremely effective co-design engagement.

2. An initial evaluation of **runtime system requirements** for our scale-bridging workload was undertaken using our CoHMM proxy app.

3. The **GREMLIN emulation** infrastructure has proven to be effective to study power, performance, and resilience impacts at exascale, and has been released to the exascale community.

2 Project Accomplishments

2.1 Application & Algorithms

Applications and algorithms relate the computational demands of our materials science workload, both single-scale and scale-bridging, to the exascale ecosystem. Our proxy apps and the algorithms they contain are the primary vehicle for collaboration with external partners, particularly FastForward vendor projects

Table 1: Hackathons and other key engagements with external ecosystem partners during 2013.

Host	Location	Dates	Participants	Key Outcomes
IBM	Yorktown	Jan 21-22	Richards	Map key kernels to AMC using assembler, critique of architecture
Sandia SST	Albuquerque	April 10-12	Belak, Richards, McPherson, Mohd-Yusof	Put SST Toolkit in hands of co-design app developers, identified need for OpenMP support
Intel FF I	Santa Clara	June 4-6	Belak, Richards, Keasler, Karlin, Mohd-Yusof	Focus on CoMD, LULESH, debug infrastructure, used pthreads, need OpenMP, identified HW ops
IBM DCDC	Argonne	July 16-17	Richards	Improved simulator, AMC mods, compiler
Intel XStack	Hillsboro	Aug 6-8	Belak, Keasler, Mohd-Yusof, Mniszewski	EDT/OCR programming model, Roger Golliver's EDT implementation of LULESH
Nvidia FF	Santa Clara	Aug 13	Keasler	Focus on CUDA programming, Michael Garland engaged on RAJA and PHALANX
AMD FF	Austin	Sept 11-12	Belak, Laguna, McPherson, Mohd-Yusof, Mniszewski, Richards, Rountree	Focus on CoMD deep dive, resilience and power side engagements
Intel FF II	Santa Clara	Oct 22-24	Belak, Keasler, Karlin, Mohd-Yusof	OpenMP now supported, all CD centers invited, focus on EXaCT

within the past year, and are heavily used within our center. Our proxy applications and algorithms provide the context for these assessments by expressing the scientific requirements, and are the basis for the interrelationships with the Programming & Systemware and Hardware-Interfacing Tools areas.

2.1.1 Proxy applications status

During year 1 we created an initial suite of proxy apps to represent the materials science workload at a variety of length and time scales. This suite includes CoMD (classical molecular dynamics), VPFFT (crystal plasticity), CoGL (meso scale phase evolution), LULESH (Lagrangian hydrodynamics), ASPA (adaptive sampling), and CoHMM (hetergeneous multiscale model). During year 2, engagements with external partners and internal explorations have lead us to update and improve these proxies, as well as to produce multiple versions of some proxy apps in a variety of programming models. Because CoMD and LULESH were selected as part of the suite of apps used to assess progress of the Fast Forward projects, these two proxies have received the bulk of our attention for external engagements. Internally, CoHMM has been improved to facilitate evaluation of run time environments, and LULESH and ASPA have been combined to produce an integrated scale-bridging mini-app.

In response to vendor requests we released CoMD 1.1, which includes both MPI and OpenMP versions and removes features such as reading large files that created difficulties for architectural simulators. CoMD 1.1 was contributed to the Matevo project which won a 2013 R&D 100 award. Versions of CoMD have been created in OpenCL, CUDA, X10, and task-based OpenMP, and ports to specific hardware have been produced. LULESH has been extended to support more complicated constitutive models.

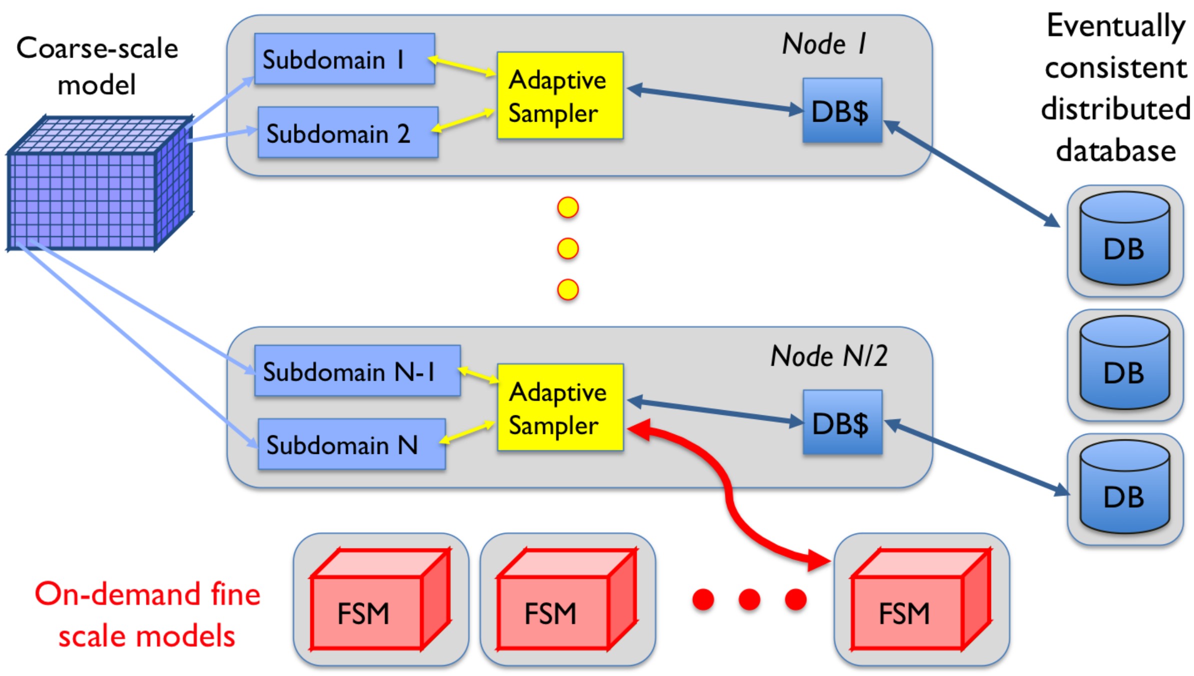


Figure 1: Components of a scale-briding application. A coarse-scale model runs across multiple nodes, each utilizing adaptive sampling to either spawn fine-scale models (FSM) as needed, or interpolate previous fine scale responses stored in a database (DB), consisting of an on-node cache and a distributed database.

2.1.2 Applications engage the ecosystem

Our most effective interactions with the Fast Forward and X-Stack communities have been in the form of hackathon or deep-dive meetings. Such meetings typically last 2–3 days, involve 10–30 participants, and include detailed presentations on hardware architecture, simulation tools, programming models, proxy app structure, and domain science, each by the developers in those areas. These presentations are usually highly interactive and result in all participants gaining a broader understanding of capabilities and trade-off potentials in other ecosystem components. The other important ingredient is at least 1 day of hands-on time for projects such as bringing up a proxy app in a vendor simulation environment. Table 1 lists our Y2 hackathon engagements.

The success of the hackathons is measured by the impact they have had on participants. The ExMatEx team has gained a much deeper understanding of future architectures and each of the Fast Forward vendors presented results from the hackathons in their semi-annual reviews. Hackathons have also identified specific hardware & system software features that are likely to have high impact, such as atomic operations in hardware, support for OpenMP in simulators, and compiler features.

2.1.3 CoHMM improvements

CoHMM presents the basic workflow requirements of a scale-bridging application as illustrated in Fig. 1. The coarse- and fine-scale models, and "glue" connecting the two, are intentionally kept simple to focus attention on the dataflow and workload requirements of a task-based scale-bridging model, enabling CoHMM to be rapidly reimplemented using various programming models, execution models, and runtime systems as described in Section 2.2.3. This year we have created CoHMM 2.0 which provides algorithmic improve-

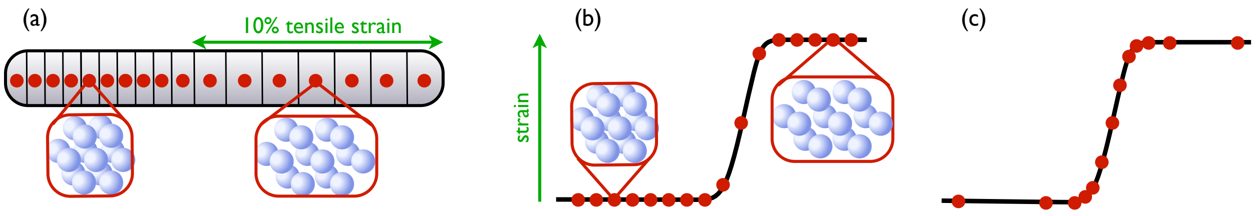


Figure 2: Illustration of spatial adaptive sampling applied to 1D elastodynamic shock propagation. (**a**) Red dots denote representative *microscopic* regions of the system, each of which is modeled by $\sim 10^3$ atoms. (**b**) The HMM obtains constitutive data from fine-scale molecular dynamics simulations, performed on a regular grid. (**c**) In spatial adaptive sampling, we dynamically adapt the location of fine-scale simulations to increase the efficiency of the HMM simulation.

ments compared to the original version. Instead of spawning a fine-scale simulation for each coarse scale element as in the original Heterogeneous Multiscale (HMM) method, we have developed a new, *spatial* approach to adaptive sampling. No database is required. Rather, at each macro-scale time step, we reconstruct the required constitutive data by interpolating fine-scale simulations on the $d \leq 3$-dimensional simulation domain. We use a *predictive* approach to determine the important sample points at the beginning of each macro-scale time-step (see Fig. 2). At the highest resolutions studied, spatial adaptive sampling yields three orders of magnitude speed-up relative to brute-force HMM, and more importantly, provides a dynamic workload that more closely represents that of our scale-bridging target application, described next.

2.1.4 Scale-bridging target application

A fundamental premise in ExMatEx is that scale-bridging performed in heterogeneous MPMD materials science simulations will place important demands upon the exascale ecosystem that need to be identified and quantified. As depicted in Fig. 1, we envision coarse "engineering" scale models coupled with concurrently executing "on demand" fine-scale models. The expense of performing fine-scale model evaluations, a large fraction of which will likely be similar to those previously performed, is mitigated by the use of adaptive sampling to "learn" the fine-scale response. In Y1 of our project, we focused on the single-scale components needed for a proxy implementation of this strategy, including LULESH, VPFFT and ASPA. In Y2, we have developed more of the "glue" needed to assemble a scale-bridging mini app that will eventually operate in the manner shown in Fig. 1 applied to a collection of demonstration problems. This will enable the quantification of problem-dependent "speeds and feeds" that are essential for the specification of job scheduling and database requirements.

To provide a concrete and fully documented scale-bridging proxy app, we created a design document describing an elastoviscoplasticity (EVP) model and time integration algorithm. Beginning with a Lagrangian coarse-scale model, the EVP model incorporates viscoplastic material response (*i.e.*, inelastic deformation that depends upon the rate at which loads are applied) using a fine-scale crystal plasticity model of the material microstructure in a representative volume element. The fundamental assumption coupling the coarse and fine-scale models is that the coarse-scale deformation gradient can be decomposed as $\mathbf{F} = \mathbf{V} \cdot \mathbf{R} \cdot \mathbf{F}^p$, where $\mathbf{V}$ is a symmetric, thermoelastic stretch tensor, $\mathbf{F}^p$ is the plastic deformation gradient determined by the fine-scale model, and $\mathbf{R}$ is a rotation between the fine and coarse-scale frames. This relation leads to a system of rate equations for the evolution of $\mathbf{V}$ and rotation $\mathbf{R}$ over a coarse time step

$$\frac{1}{a}\dot{\bar{\mathbf{V}}}' = \mathbf{R}^{\mathrm{T}} \cdot \mathbf{D}' \cdot \mathbf{R} - \bar{\mathbf{D}}', \tag{1}$$

$$\dot{J} = J\mathrm{tr}(\mathbf{D}), \tag{2}$$

$$\begin{aligned}\dot{\mathbf{R}} \cdot \mathbf{R}^{\mathrm{T}} &= \mathbf{W} - \mathbf{R} \cdot \bar{\mathbf{W}} \cdot \mathbf{R}^{\mathrm{T}} \\ &\quad - \frac{1}{a}\mathbf{R} \cdot \left[\bar{\mathbf{V}}' \cdot \left(\bar{\mathbf{D}}' + \frac{1}{2a}\dot{\bar{\mathbf{V}}}'\right) - \left(\bar{\mathbf{D}}' + \frac{1}{2a}\dot{\bar{\mathbf{V}}}'\right) \cdot \bar{\mathbf{V}}'\right] \cdot \mathbf{R}^{\mathrm{T}},\end{aligned} \tag{3}$$

where $\mathbf{D}$ and $\bar{\mathbf{D}}$ are the coarse and fine-scale strain rates, respectively, $\mathbf{W}$ and $\bar{\mathbf{W}}$ are the coarse and fine-scale spin rates, respectively, $J = \det(\mathbf{V})$, $a = J^{1/3}$, dots denote time derivatives and primes denote tensor deviators. Once $\mathbf{V}$ is known at a new coarse time step, its logarithm can be used as a measure of strain in an appropriate elasticity model to obtain the stress needed by the coarse scale model. Although seemingly a bit complicated, we nevertheless believe that this choice for a scale-bridging algorithm coupling a continuum mechanics model with a mesoscale crystal plasticity model balances the competing goals of materials science relevance, minimization of complexity in a proxy app that will be used for multiple co-design purposes, and flexibility with respect to exploring a large model and algorithm space.

In order to serve as the coarse-scale model in the EVP proxy, an extension of LULESH was necessary to support high-strain-rate solid-phase materials. This involved the replacement of LULESH's original ideal gas equation of state with a strength model expressed as a full stress tensor σ computed from an appropriate constitutive model. The divergence of the stress tensor yields the momentum equation force term used to accelerate element nodes,

$$\rho\dot{\mathbf{v}} = \nabla \cdot \sigma + \rho\mathbf{f}, \tag{4}$$

and the inner product of its deviator σ' with the velocity gradient $\mathbf{L} \equiv \nabla \otimes \mathbf{v}$ provides an additional term in the energy equation

$$\rho_0\dot{e} = \eta\sigma'{:}\mathbf{L} - (p+q)\,\dot{\eta}. \tag{5}$$

Since this extension to LULESH had always been anticipated, the modifications are highly localized and cause no significant modification to the overall code structure.

Using this extension of LULESH, as well as the existing VPFFT and ASPA proxy apps, we have created an initial implementation of the specified EVP algorithm. This primarily required the addition of the rate equations (1)-(3) and some refactoring of ASPA to enable a constitutive model library to be created, which, at least for the time being, is statically linked with LULESH. The new proxy app is currently being tested on a Taylor cylinder impact problem, which has been selected because it represents a moderate strain rate problem in which the effects of anisotropy in the fine-scale microstructure are readily observable in the coarse-scale material behavior. We will also soon begin applying the EVP proxy to a high-strain-rate shock problem in preparation for our Y4 demo.

2.2 Programming & Systemware

As explained in Section 2.1.4, our target multi-scale application is composed of various pieces, or components, that must interact with each other in a dynamic and adaptive fashion as the code runs (Fig. 1). These components include a coarse scale computation (e.g. LULESH) that adaptively launches a dynamically varying number of fine-scale computations (e.g. VPFFT or CoMD) as the simulation progresses through time. In addition, the application may cache previously computed results in a database to reduce or eliminate duplicate computation. Unfortunately, it seems clear that no single programming language will emerge that supports all of this required functionality—multiple languages and systems must be used in concert to build the final application. This realization has focused our research efforts on selecting and developing the best combination of programming models and systemware that will enable us to efficiently build and run these multi-scale computations. Our selection of this subset, tested and validated using proxy apps, will be driven by our scale-bridging target application (Section 2.1) and our analysis, modeling and simulation capabilities (Section 2.3).

Our research targets three specific areas of functionality that address different aspects of our applications. First, we investigate component- or node-level programming models. Second, we are exploring domain-specific languages as a way to insulate the application developer from the complexities and diversity of multiple programming models. Third, we look at programming in the large—how we couple the individual components of our applications and execute them using runtime systems and system services. In the following three sections we describe our efforts and results in each of these three areas.

2.2.1 Component-level Programming

Each individual, single-scale, component of our target application must execute efficiently on the available hardware. Using this requirement as a driver, we have used our proxy apps to evaluate and assess various programming models and languages. Our goals are twofold: to identify programming models best suited to particular algorithms and hardware, and to provide feedback to vendors and standards bodies based on our results and experience. In year 2 of the project we have examined a wide variety of programming approaches including CUDA, OpenCL, OpenMP, X10, Chapel, and Open Community Runtime (OCR).

Based on our experiences with this variety of models, it is clear that no single programming model has emerged as a universal soultion to exascale challenges, and we believe that programmers will continue to use a variety of programming models. It is also clear that practically all programming models still require significant expertise in hand tuning, often machine specific, in order to obtain high performance. It is critical that programming models continue to evolve to provide a more favorable balance between programmer productivity, code maintainability, and application performance. Features that provide more flexible and effective ways to express concurrency, as well as more powerful mechanisms for expressing data locality, will be critical. Support for legacy code, composability with other models, and abilities to control execution scheduling and express data dependencies are also important ingredients for any programming model to be sucessful at exascale.

2.2.2 Domain-specific Languages

To insulate application scientists and programmers from low-level language details, we have embarked on a high-risk, high-payoff project of developing a domain-specific language (DSL) infrastructure. DSLs are concise programming languages that are designed to naturally express the semantics of a narrow problem domain. The use of DSLs can provide significant gains in the productivity and creativity of application scientists, the portability of applications, and enhanced application performance.

The Liszt DSL for unstructured meshes is our vehicle of choice, providing language-level primitives for traversing and operating on the elements of a mesh (e.g. faces, vertices, edges, etc.). In Y1, we implemented LULESH in a version of Liszt that used a Scala[1]-based compiler framework. In the process we identified two major issues with the design of Liszt. First, the ordering of mesh elements was inconsistent in Liszt, while LULESH required ordered vertex access. This demonstrates that DSLs must be designed, as part of a language co-design process, *in direct collaboration with the domain scientists that will use them.* Second, although the Liszt version of LULESH used half the lines of code that the serial implementation of LULESH did, it ran only half as fast. This performance penalty is mitigated by the feature that *the same Liszt source code* compiled to multiple compute platforms (serial, multi-core, and GPU) as well the fact that minor changes to the Liszt generated C-code improved performance to within a factor of 1.2 of the

[1]Scala is a multi-paradigm language built on top of the Java virtual machine.

serial version of LULESH. In addition to non-optimal code generation, other issues were identified in the Scala-based compiler infrastructure, including the need to interoperate with legacy software written in C, C++ and Fortran; the ability to be more adaptive, both to processor and node architecture, and to internal data structures (e.g., for dynamic meshes); and the ability to support multiple, *interoperating* DSLs (e.g., for multiphysics applications).

For these reasons, we decided to re-implement the compiler infrastructure and have built Terra, a new low-level system programming language for building DSLs. Terra is designed to interoperate seamlessly with Lua, a high-level dynamically-typed programming language. We designed Terra to make it easy to write JIT-compiled DSLs, and integrate them with existing applications. Generated Terra code is fast. Terra programs use the same LLVM backend that Apple uses for its C compilers. This means that Terra code performs similarly to equivalent C code. Terra also includes built-in support for SIMD operations, and other low-level features like non-temporal writes and prefetches. Lua can be used to organize and configure the application, and then call into Terra code for controllable performance.

The Lua-Terra ecosystem is easy to integrate with existing applications. Lua was designed to be embedded in C programs. We leverage this design to make it easy to embed Lua-Terra programs in other software as a library. This design allows addition of a JIT-compiler, or integration of a DSL with existing high-performance code. The code that Terra generates works with code written in C, C++, or FORTRAN. Terra is available at `terralang.org` as open source, and has been used it to implement several domain-specific languages. We presented our design of Terra at PLDI [2] and along with several example applications.

We are converting the Liszt language to an implementation based on Terra. Using Terra will allow us to optimize Liszt kernels based on dynamically determined information, allowing us to remove many of the previous restrictions in the language. Mesh topology can change over the course of the program; kernels will be able to call already existing C, C++, or FORTRAN code; and features such as particles or custom mesh formats will be easier to add. The front-end of the Terra port of Liszt is now compiling programs and generating code for the single-core runtime that reads mesh files, runs iterative computations over per-element mesh values, and prints results. In order to port LULESH, we are implementing language features that allow programmers to invoke function calls from parallel Liszt code, and updating the code generation to use the updated runtime.

2.2.3 Runtimes and System Services

Many, if not most, of today's scientific simulation applications are developed using a fairly limited set of software technologies: a standard programming language such as C, C++, or FORTRAN, MPI for communication, and a static scheduler, such as Slurm or Moab, to execute the computation on the machine. Should a developer need to load balance their computation, they need to provide this functionality themselves. Similarly, fault-tolerance requires programmers to periodically write data to disk for later recovery. Likewise, communication patterns are generally fixed—dynamic communication patterns must be designed and implemented by the programmer. System services, or runtime systems, can provide much of this functionality. These services support "programming in the large"—coupling multiple diverse components of a dynamic multi-scale computation and orchestrating its execution on the system. These services, in category of functionality, are:

- Scheduling: concurrent, asynchronous, adaptively executing computational components, launched on-the-fly, and exiting when complete.
- Discovery: locating system resources based on application-supplied requirements (e.g. provide a list of all nodes with GPU accelerators).

Table 2: CoHMM implementations in various asynchronous task-parallel programming/execution models and runtime sytems, listed in chronological order.

System	Dimension	Adaptive	Database	Fault Tolerant Implementation?	Status
Scala	1D	No	No	No	Simple MD
Erlang	1D	No	No	Process	CoMD 1.0
"Cloud"	1D	No	Multiple	Process	CoMD 1.1
Swift	1D	No	No	Process	CoMD 1.0
HPX	*Bugs and poor documentation in v0.9.5, esp. for distributed systems*				Abandoned
Charm++	*Load balancing evaluated w/synthetic benchmarks; difficult to reconfigure*				Eval. only
Mesos	*Evaluated favorably, pursue with Spark running on top of Mesos*				Eval. only
Swarm	*Evaluated favorably, but early version limited to 128 child processes*				Eval. only
Pathos	1D	Yes	No	Process	CoMD 1.1
Scioto	1D, 2D	AMR, Kriging	redis	No	CoMD 1.1
Spark	1D, 2D	AMR, Kriging	redis	CoMD task	CoMD 1.1
CnC	2D	No	No	No *(requires Intel MPI)*	CoMD 1.1

- Messaging: setup and tear down, on-the-fly, dynamic, adaptive communication links between components of the calculation.

- Caching: services for temporarily storing data, perhaps in-memory, and retrieving it from anywhere in the computation. Caching can help prevent duplicate computation or store data to be used for recovery from faults. Caching can also be used for communication. Instead of sending messages, processes can store their data in the cache for retrieval by other processes.

- Fault detection: working with the application, operating system, and hardware, detect faults in the system and provide facilities for application notification or automated restart.

In general, there are two ways in which these services (or subsets of them) are implemented. First, there are the distributed monolithic systems that are closely tied to a programming language or model (e.g. Charm++, X10, Chapel, CnC, Erlang). These systems include a runtime[2] component that implements features of the programming models such as scheduling, communication, data distribution, etc. Second is a more loosely coupled approach that uses various single-function software, usually open source, to build an integrated, dynamic system. This approach is closer to what "industrial" developers use to build cloud- or web-based scalable systems (e.g. Netflix, Facebook, LinkedIn, Google, etc.).

In Y1, we began exploring this space with the initial CoHMM proxy app. As a monolithic approach, the Erlang programming language provides the desired features in a single language and runtime system. Our cloud-based approach used a more diverse set of tools including Apache ZooKeeper (for discovery, scheduling, and process tracking), `node.js` to manage the overall execution of the code, and multiple NoSQL databases (MongoDB, memcached, Riak, Couchbase) to cache data for fault tolerance. In both cases, the implementations simply managed the dynamic coupling between the coarse- and fine-scale components of CoHMM. All of the mathematical computation was done in the component-level APIs that those components used (e.g. MPI or OpenMP).

This year, we extended and expanded this work, primarily through the *Los Alamos IS&T Co-Design Summer School*. This team of talented students, with backgrounds in both computer science as well as physics, refined our CoHMM and CoMD proxy apps and used them to test a variety of runtime systems

[2]These system-level runtimes should not be confused with low-level runtimes provided by componet-level programming languages such as CUDA or OpenMP. They often provide the same conceptual features but at a much finer level of granularity.

from industry and academia. Our students evaluated software that acts as the "glue" between the coarse- and fine-scale components of CoHMM, whose simplicity allowed us to investigate a wide swath of software technologies—both monolithic and cloud-based. Our evaluations focused on some of the primitive features we describe above: scheduling, communication, caching, and fault tolerance. Various schedulers were tested against how well they supported dynamic and adaptive task scheduling. The students implemented multiple forms of adaptivity in both 1- and 2-D implementations—spatial sampling, which does not require a database, and kriging, which uses an in-memory database (`redis`) to cache previously computed results. In addition, fault tolerance was implemented using two approaches. In the first, process-level case, the runtime system detects the crash of a fine-scale CoMD process and the system restarts it from its initial conditions (potentially on another node) without crashing the entire application. In the second case an in-memory database (again, `redis`) is used to periodically cache particle positions from each CoMD process. If the runtime system detects a crash, the CoMD process is now restarted from the conditions encapsulated in the most recently cached particle positions—not from the initial conditions. These developments also enhanced our original test applications; fault tolerance was added to the Erlang version, and our cloud version now uses the proven `redis` database. Table 2 shows a condensed representation of the space we have explored in runtime systems and system services. This list is by no means definitive—there are many more potential systems to explore. It is clear that we must spend much of next year narrowing down the list of contenders to a small subset best suited to our particular problem domain and target application.

We can already draw some lessons learned on potential limitations or issues with this approach to programming in the large. Given that all of the computational time is spent in the CoMD calls (the coarse-scale computation, even at 2-D is quite fast, while each of the thousands of CoMD runs take over 20 seconds to converge) the overhead of scheduling and communication is minimal. If the granularity of our fine-scale calculations is small this overhead may become a concern. Similarly, at the scales at which we ran, database overhead was small (even using only a single node for the `redis` database). Scaling to larger numbers of nodes and cores, along with a reduction in fine-scale runtime, may drive this overhead higher. This is one of the prime motivations for building our EVP scale-bridging proxy—to generate realistic "speeds and feeds" for our target problem. We can use these to more accurately capture potential scaling issues with this approach.

2.3 Hardware-Interfacing Tools

Performance analysis and modeling is critical for almost any aspect of the exascale co-design process. It allows us to both understand and anticipate computation and communication needs in applications, it identifies current and future performance bottlenecks, it helps drive the architectural design process and provides mechanisms to evaluate new features, and it aids in the optimization and the evaluation of transformations of applications. No single analysis technique or methodology alone is capable of providing the necessary insight into application code behavior that is required to drive the co-design process. Instead, we require a wide range of techniques combining analytical models, architectural simulation, system emulation and empirical measurements to create a holistic picture of the behavior of a target application (see Fig. 3).

ExMatEx targets all four areas—using existing technology where available and developing new approaches, where necessary: we use empirical techniques to create baseline performance profiles for our proxy apps (as reported in year 1); we develop and apply the GREMLIN infrastructure to enable architectural emulation for power, performance, and resilience aspects (Section 2.3.1); we deploy architectural simulation using SST to improve our understanding of the impact of the memory system (Section 2.3.2); and we provide high-level and scalable models for our proxy applications using Aspen (Section 2.3.3). Combined, these techniques enable us to gain a deep insight into the performance of the applications relevant to ExMatEx, as well as to establish a comprehensive performance analysis infrastructure that provide the tools which can be used in any DOE co-design effort. In addition, we organized a deep-dive hackathon to

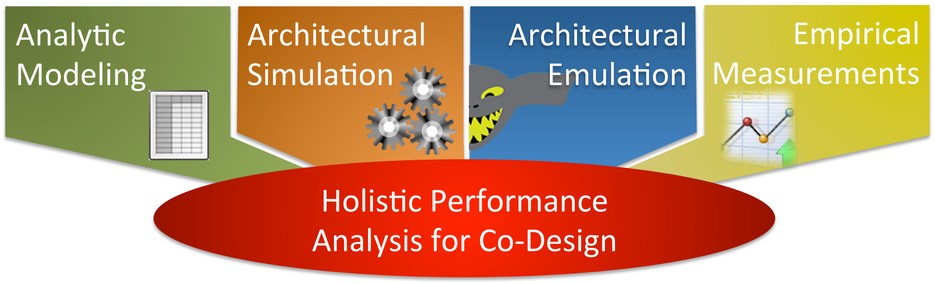


Figure 3: Techniques contributing to a holistic performance analysis.

expose the SST toolkit to the application developers and identified key aspects of the baseline programming environment SST needs to support, e.g. OpenMP.

2.3.1 GREMLINs: Emulating Exascale Conditions on Petascale Machines

The GREMLIN framework [3] enables architectural emulation. It allows us to go beyond measurements on existing machines, while still executing full applications, but within a controlled environment in which we can expose characteristics we expect of future extreme scale platforms. The GREMLIN framework provides this ability in a highly modular fashion. One or more modules, each responsible for emulating a particular aspect, are loaded into the execution environment of the target application. Using the P^NMPI infrastructure [4] we accomplish this transparently to the application for an arbitrary combination of GREMLINs, providing the illusion of a system that is modified in one or more different aspects.

This year, we have matured the concepts of the GREMLIN framework and have released[3] the base infrastructure as well as several fundamental GREMLINs: power GREMLINs that artificially reduce the operating power using DVFS or cap power on a per node basis [5, 6], memory GREMLINs that limit resources in the memory hierarchy such as cache size or memory bandwidth by running interference threads, and fault GREMLINs that inject faults into target applications and enable us to study the efficiency and correctness of recovery techniques. In the following we present some key results from all three areas. We are planning a GREMLIN hackathon during the coming year to expose the GREMLIN tools to the broader extreme-scale community.

Power GREMLINs: Emulating a Power Constrained World

Power will be one of the limiting resources on the road to exascale. Driven by both the cost to operate machines and the machine room infrastructure required, future systems will be limited in the amount of power they can use. This will likely lead to overprovisioned systems, on which we can no longer run each node at full power and on which we are faced with strict power caps at the node, rack and system levels [5].

To emulate the effects that such power caps have on application performance, we implemented a power capping GREMLIN using Intel's RAPL (Runtime Average Power Limits) interface. RAPL enables us to set per processor power caps through a set of Machine Specific Registers (MSRs), which are then enforced by the processor hardware through different voltage and frequency levels. The GREMLIN itself is therefore rather simple: we simply set the appropriate limit during initialization and then continue the execution under RAPL's hardware control.

Figure 4 shows the results of executing the CoMD proxy app on 128 processors under three different per processor power caps using a Sandy Bridge Infiniband DDR-3 cluster with 2 sockets/16 cores per node.

[3] https://computation-rnd.llnl.gov/gremlins/

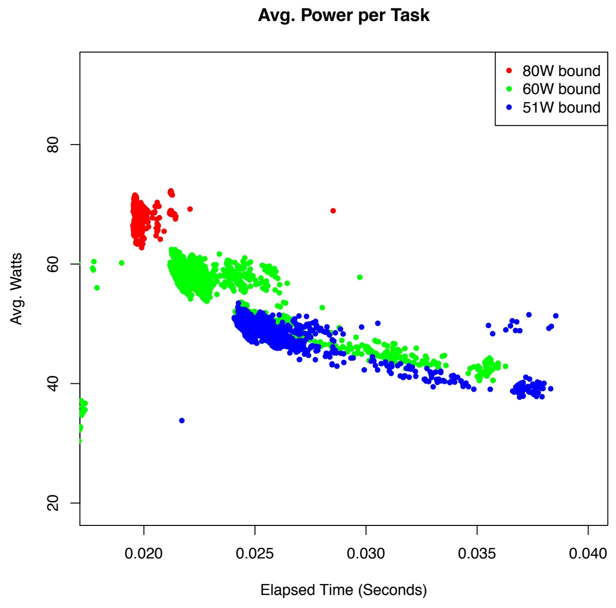


Figure 4: Impact of power cap on performance of CoMD.

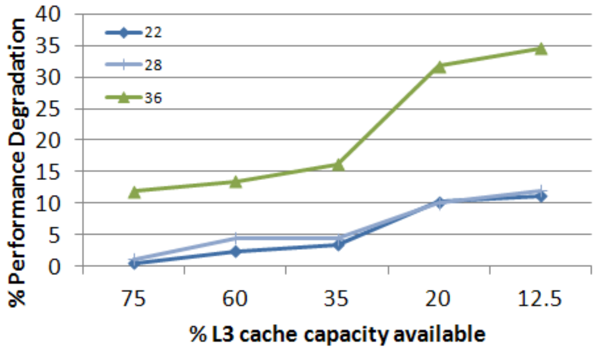


Figure 5: Impact of changes in cache size on LULESH performance.

Ignoring individual outliers, the graph shows that, as expected, lower power bounds lead to lower power consumption, while prolonging the execution of the application. However, we also see a second trend: while execution time is fairly uniform at the highest power bound of 80W, lower bounds lead to increasingly larger variations in execution time.

These experiments show that future power limitations will manifest themselves in performance variations directly visible to the application. *In other words, power caps can lead to imbalanced applications even if their workload is fully balanced.*

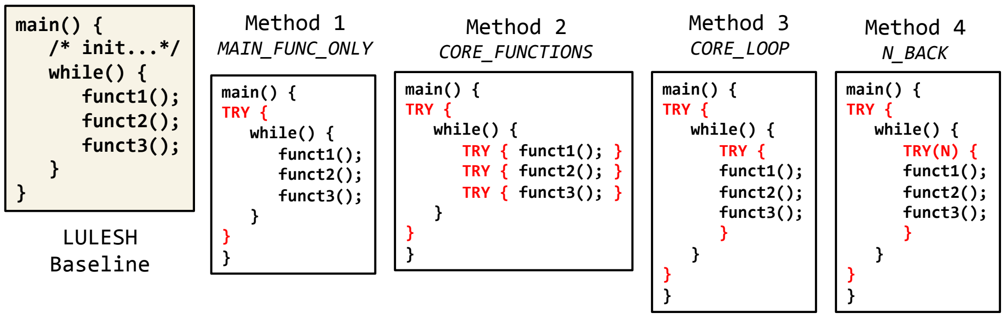
Memory GREMLINs: Emulating a System with Constraints in the Memory System

A second resource that is expected to be severely limited in future architectures is the memory system. This refers to both memory bandwidth, in particular off-chip bandwidth which is limited by physical constraints, and cache sizes. We can recreate these trends on today's platforms using carefully calibrated competing threads on each node, which either consume predefined amounts of bandwidth or cache storage and thereby "steal" the resource from the target application [7]. Fig. 5 shows the results of executing LULESH [8, 9] under the cache GREMLIN for multiple working set sizes. Reducing last level (L3) cache size to 35% only leads to a small impact on application performance, while *further reductions cause significant changes in application performance, especially for large working sets.*

Resilience GREMLINs: Emulating a World with More Faults

A third critical area for exascale system design is the expected rise of fault rates caused by larger component counts, reduced slack in architectural designs, as well as smaller feature sizes and possible near-threshold voltage operation. To study this effect, GREMLINs can be used to inject faults into an application's execution, enabling us to monitor effects on execution time and correctness as well as to study the impact of countermeasures implemented within applications.

As a simple first example, we instrumented LULESH with RETRY blocks, which create local mini-checkpoints at which the application stores its data and which can be used to roll back to, in case a fault



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

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

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

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

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

Figure 6: Instrumenting LULESH with RETRY blocks

is encountered within that block. Figure 6 shows the range of potential instrumentation locations, ranging from all of main() to wrapping the three functions called by main individually. The decision on where to place the RETRY annotations is critical or large overheads can occur.

We then implement a "fake" fault GREMLIN that periodically assumes non-correctable faults and informs the application through a fault interface (without actually causing a fault, but triggering the recovery mechanism in the application). We use this setup to measure the execution time of LULESH for varying fault rates. The results show a high, but nearly constant, overhead for protecting every function in the main loop, caused by frequent checkpoints but short rollbacks, while protecting all of main() leads to high overhead that rises sharply as we increase the error rate, since a fault causes the entire program to be re-executed. Protecting individual iterations shows a similar trend as protecting individual functions, but at much lower overheads, due to the reduced checkpointing. *Protecting every Nth iteration provides a good balance between the extremes, with 25 iterations between mini-checkpoints being a sweet-spot.*

2.3.2 SST: Detailed Performance Simulation

The above study of applying GREMLINs to the memory system shows the ability to run real applications in scenarios with reduced memory bandwidth, but also indicates some limitations. For instance, since caches are transparent for the application and any GREMLIN, any emulation has to be approximate. Further, more advanced approaches, which fundamentally change the memory system, cannot be emulated at all. For these tasks in our performance evaluation spectrum, we apply the Structured Simulation Toolkit (SST), albeit with significantly increased execution times and limited to smaller size applications and kernels. In addition, we have added SST support for multi level memory (e.g. NVRAM) to *enable the assessment of the heterogeneous memory emerging for exascale computing.* In the following, we focus on hardware-based coherency.

Obtaining information relating to hardware-based coherency events is extremely difficult and often requires vendor specific tools or access to simulation environments. We have utilized the processor core, cache and memory modeling capabilities in SST/Micro to evaluate the coherency events for a simple 8-core processor cluster which could be used as a 'tile' for a future many-core exascale class processor. In this study we employ a simple coherency protocol which approximately maps to MESI (modified, exclusive, shared, invalid)—a common coherency mechanism used in contemporary processors. A diagram of how a stream of instructions is captured by SST and fed into the model is shown in Fig. 7.

Our analysis of LULESH shows the cost of invalidations for varying OpenMP thread counts. These are essentially a first order approximation for the cost of coherency within a processor since they are commands to the cache to invalidate (mark a cache line as invalid), essentially flushing the data. Any energy expended migrating this data into the caches has been wasted if the data is invalidated. Whilst it is impossible to have

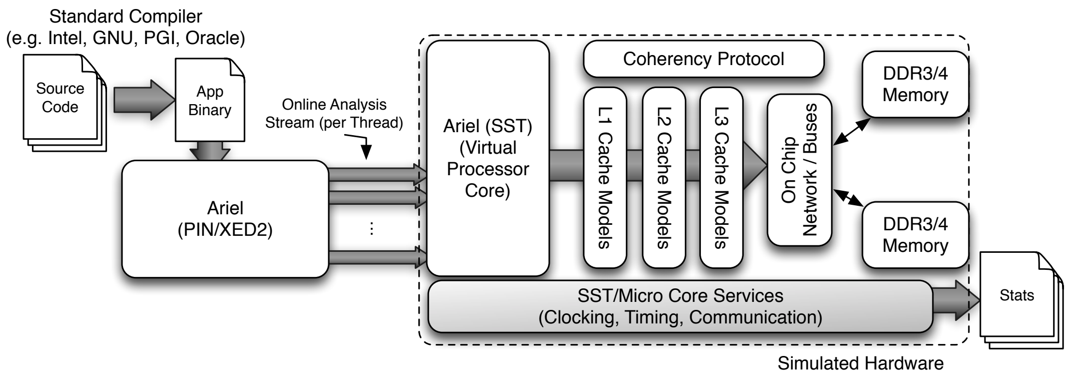


Figure 7: Representation of using SST to evaluate coherency mechanisms

a fully coherent cache system without invalidation events, a design goal for future processors will be reduce these through a richer coherency protocol or the appropriate selection of core count or cluster size. The data provided by SST is also able to show that by fixing the problem size but executing over a larger thread count, a greater number of memory requests are resolved by other caches at the first level (up to $\sim$5%). This can be used by algorithm designers and hardware architects and can be viewed through two perspectives: (1) a resolution in an alternative L1 cache requires that data to be moved to the requesting L1 or processor core forcing a data movement to take place; alternatively, (2), the overall capacity of cache on the chip is increased if all processor cores can make more effective use of the data stored with them, thus reducing the need to obtain data from memory, which is even higher in energy consumption and time.

2.3.3 Aspen: A DSL for Performance-Modeling

While both GREMLIN and SST provide valuable insights into the performance on expected architectures and applications, their application can be limited by their speed, scalability, and flexibility. In order to address these limitations in ExMatEx, Aspen has been designed as a prototype system for the analytical modeling of extreme-scale applications and architectures. Aspen (Abstract Scalable Performance Engineering Notation) [10] is a modeling framework that is based on a domain specific language (DSL). Scientists write structured models of their applications and architectures using Aspen's DSL. Aspen specifies a formal grammar for describing two types of models. This approach has several benefits including formal checks on semantics, composability, modularity, reusability, flexibility, and speed.

During the past year, we made significant progress on Aspen in terms of new language features, model development, and user-facing infrastructure, and have created new Aspen models for CoMD, LULESH, and VPFFT. With these models in hand, we can run any of our tools on these models.

The language itself has received major new features. First, the current version of the Aspen modeling language now supports conditionals, where parameters in the model define the execution paths or resource requirements. Second, the language now supports probabilistic execution, which can be used to model branches, load imbalance, and other certain classes of nondeterministic behavior, frequently found in scientific applications. Third, one of the most significant changes is the unification of control flows and execution kernels.

In terms of use of these Aspen models, we have developed several tools. First, we designed a full web-driven collaborative user interface and analysis tool, AspenLab. AspenLab hosts application and machine models, grouped by user. Once models have been uploaded or created in AspenLab, users can run a model checker and suite of analysis tools, including tables and interactive charts showing resource usage by kernel, over parameterized variables, kernel data usage, and computational intensity. Second, we created a general-purpose performance model design space exploration engine. Using non-linear optimization, modelers can select machine or application constraints and request the tool to minimize or maximize certain parameters. For example, the user might request the optimizer to find the largest problem size for a given application model which fits in a given machine's RAM and runs in less than a given time limit. This tool is flexible and supports a wide range of optimization tasks. Finally, we have created an interface from Aspen to SST/Macro, where the Aspen model is used to generate a parameterized synthetic workload for interconnect simulation. We have successfully tested several models end-to-end through this new framework.

3 Summary and Ongoing Activities

Our emphasis this past year has been on exposing our application workload to the exascale ecosystem, in particular via deep engagements with all of the FastForward vendors (e.g., via hackathons), with programming models from Xstack projects, both node-level and task-based execution models and runtime system; and establishing the tools for quantifying the metrics for tradeoff analysis. As detailed in this report, three key accomplishments resulted from these activities:

1. Multiple **deepdive hackathons** with our Fast Forward and X-stack partners using proxy applications has proven to be an extremely effective co-design engagement.
2. An initial evaluation of **runtime system requirements** for our scale-bridging workload was undertaken using our CoHMM proxy app.
3. The **GREMLIN emulation** infrastructure has proven to be effective to study power, performance, and resilience impacts at exascale, and has been released to the exascale community.

As envisioned in our original project plan [1], the agile management process involves a continual re-evaluation and rewriting of the project milestones. Based on this year's assessment, and the key Y4 milestone to demonstrate task-based scale-bridging on 10+ PF platforms, our revised Y3 milestones are:

3.1	Define petascale (Y4) and exascale high strain-rate scale-bridging target problems, and a working smaller-scale prototype app.
3.2	Establish and document requirements, and initial implementation, of single-physics and scale-bridging programming models.
3.3	Assess uncertainty requirements for scale-bridging and implement within initial prototype scale-bridging app.
3.4	Use power and resilience analysis to inform programming models and runtime services.
3.5	Develop Aspen model for scale-bridging app, and use Aspen/SST coupling to assess scalability.
3.6	Release updated proxy applications and analysis tools/extensions.

4 Acknowledgements

This work is supported by the DOE Office of Science, Advanced Scientific Computing Research Program, and was performed under the auspices of the U.S. Department of Energy by Los Alamos, Lawrence Livermore, Sandia, and Oak Ridge National Laboratories under Contract Nos. DE-AC52-06NA25396, DE-AC52-07NA27344, DE-AC04-94AL85000, and DE-AC05-00OR22725, respectively.

References

[1] ExMatEx: Exascale Co-Design Center for Materials in Extreme Environments. Planning activity report, March 1, 2011.

[2] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In Hans-Juergen Boehm and Cormac Flanagan, editors, *PLDI*, pages 105–116. ACM, 2013.

[3] M. Schulz, J. Belak, A. Bhatele, P.-T. Bremer, G. Bronevetsky, M. Casas, T. Gamblin, K. Isaacs, I. Laguna, J. Levine, V. Pascucci, D. Richards, and B. Rountree. Performance Analysis Techniques for the Exascale Co-Design Process. In *Proceedings of PARCO 2013*, October 2013.

[4] Martin Schulz and Bronis R. de Supinski. P^NMPI Tools: a Whole Lot Greater than the Sum of their Parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[5] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *ICS*, pages 173–182, 2013.

[6] B. Rountree, D. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz. Beyond DVFS: A first look at performance under a hardware-enforced power bound. In *8th Workshop on High-Performance, Power-Aware Computing (HPPAC)*, May 2012.

[7] Marc Casas and Greg Bronevetsky. Active Measurement of Memory Resource Consumption. Technical Report LLNL-TR-636044, Lawrence Livermore National Laboratory, May 2013.

[8] Ian Karlin, Abhinav Bhatele, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, Edward Luke, Scott Lloyd, Jim McGraw, Rob Neely, David Richards, Martin Schulz, Charles H. Still, Felix Wang, and Daniel Wong. Lulesh programming model and performance ports overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory, December 2012.

[9] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '13, May 2013.

[10] Kyle Spafford and Jeffrey S. Vetter. Aspen: A domain specific language for performance modeling. In *SC12: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2012.