Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application

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[†], Charles H. Still[†]

[†]Lawrence Livermore National Laboratory, P. O. Box 808, Livermore, California 94551 USA
[‡‡]Cray Inc., Seattle, Washington 98164 USA
[¶]Stanford University, Palo Alto, California 94305 USA
[‡]University of California, Los Angeles, California 90095 USA
[*]Mississippi State University, Mississippi State, Mississippi 39762 USA
[§]University of Illinois at Urbana-Champaign, Urbana, Illinois 61801 USA

E-mail: karlin1@llnl.gov, bhatele@llnl.gov, keasler@llnl.gov

***Abstract*—Parallel machines are becoming more complex with increasing core counts and more heterogeneous architectures. However, the commonly used parallel programming models, C/C++ with MPI and/or OpenMP, make it difficult to write source code that is easily tuned for many targets. Newer language approaches attempt to ease this burden by providing optimization features such as automatic load balancing, overlap of computation and communication, message-driven execution, and implicit data layout optimizations. In this paper, we compare several implementations of LULESH, a proxy application for shock hydrodynamics, to determine strengths and weaknesses of different programming models for parallel computation. We focus on four traditional (OpenMP, MPI, MPI+OpenMP, CUDA) and four emerging (Chapel, Charm++, Liszt, Loci) programming models. In evaluating these models, we focus on programmer productivity, performance and ease of applying optimizations.**

***Keywords*-parallel programming models, productivity, performance, co-design, proxy application**

I. Introduction

We face a variety of challenges on the path to extreme scale computing. Solving them will require substantial changes in how we design and implement parallel programs. One frequently mentioned area of change is the programming model used for writing parallel applications. Traditionally, most scientific applications are implemented using the Message Passing Interface (MPI) [1], coupled in some cases, with a second model for threaded execution on-node, typically referred to as MPI+X. Examples of X include OpenMP [2] for multi-core architectures or CUDA and OpenACC for GPU architectures. In order to address future systems and applications, however, a wide range of new programming models have been proposed with the goal of improving performance, productivity or both. In particular, domain specific languages (DSLs) [3] are attracting significant attention as one possible technique to write applications easily and efficiently.

Programming approaches vary in many aspects, including but not limited to, how parallelism is expressed, how data are organized, how execution is scheduled and how communication and synchronization are handled – all of which determine the amount of control the programmer has over various aspects of the program execution. Additionally, they differ in their generality (in terms of application domains that can be covered), portability (which architectures they can run on), as well as productivity (the amount of work a programmer has to invest to design, implement, optimize, and maintain an application). In the rest of the paper, we refer to the collective set of parallel programming models, languages, runtimes, and specific implementations as "programming models" for ease of reference. Programming models can range from purely task based models like Intel's TBB [4], Cilk [5], or PLASMA [6], to globally synchronous approaches, such as BSP [7]. Furthermore, they can be based on a common memory paradigm like Global Arrays [8] or HPF [9] or be data-driven, such as Charm++ [10]. In contrast to these application domain independent systems, DSLs seek to create a language for a specific problem domain, such as MATLAB [11] for linear algebra, Liszt [12] for partial differential equations and SEQUEL [13] for databases.

In this paper, we explore both traditional and emerging programming models using the LULESH (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics) proxy application (proxy app). LULESH [14] is a shock hydrodynamics code developed at Lawrence Livermore National Laboratory (LLNL) as part of the DARPA UHPC effort and now used in DOE's co-design efforts. LULESH is large enough (more than 4,000 lines of code for the parallel MPI implementation) to be more complex than traditional benchmarks, yet compact enough to allow a large number of implementations. It is also exhibits characteristics similar to typical problems in both the DOE and DOD application space, making it an ideal candidate for such a study.

The base version of LULESH is available as serial, OpenMP and MPI code [15]. Further, LULESH can be executed as hybrid MPI+OpenMP code with threading introduced within each MPI process. A version of LULESH exists for GPUs

with an implementation in CUDA [16]. Additionally, we have ported LULESH to four new and emerging programming models: Chapel [17], CHARM++ [10], Liszt [12] and Loci [18]. The Liszt port can generate both a multiprocessor version and a GPU-enabled version. The models and languages were chosen based on their widespread use or their potential for extreme scale (for performance or productivity). We think that the chosen models span a large design space covering DSLs, partitioned global address space (PGAS), message-driven and functional/relational models.

Comparison between programming models is not straightforward and must be done using a holistic approach across a range of aspects and metrics. Further, in order to provide a fair comparison and to take the different stages of development into account for the various models, we need more than just metrics that measure performance. We therefore take a different approach, which includes:

- **Productivity:** In addition to easily derivable metrics, like the number of source lines of code (SLOC), we provide subjective impressions on the productivity we experienced when porting LULESH to different models.
- **Performance:** We compare the performance of the implementations on an Intel Sandy Bridge cluster, Cab (uses an Infiniband interconnect) and an IBM Blue Gene/Q (BG/Q) machine, Sequoia, both at LLNL. In some cases, we manually adjust the implementations as described in the text, in order to do an apples-to-apples comparison.
- **Ease of Optimization:** We compare the models with respect to the ease of applying possible manual and automatic optimizations in LULESH.

Our analysis shows that the newer programming approaches, such as Loci and Chapel result in programs that are up to 80% smaller than the MPI implementation (based on SLOC). These models also contain many features that enable portable application performance with less programmer effort. Models like CHARM++ perform comparably to the MPI implementation, despite LULESH not benefiting from the adaptivity and asynchrony in CHARM++. Other newer models, such as Liszt, show high levels of portability and the potential for high performance, once its compiler back end improves. Overall, our paper makes the following novel contributions:

- We provide implementations of a single application – the shock hydrodynamics proxy app, LULESH, in a wide range of traditional and emerging parallel programming models/languages.
- We discuss the differences in implementation of LULESH in the various models.
- We explore the productivity and performance potential of the various models.
- We discuss the suitability of particular models for the given problem and discuss design criteria for future programming models.

The remainder of this paper is structured as follows. We introduce LULESH in Section III and discuss its implementation in the traditional models in Section IV. We introduce the emerging models used in this study in Section V together with a description of the implementation of LULESH in these models. In Section VI, we describe optimizations we have found profitable for LULESH performance and then describe how the new languages make it easier for the programmer to apply these optimizations. This is followed by an evaluation of the implementations in Section VII and we conclude the paper with lessons learned in Section VIII.

II. RELATED WORK

Several studies have investigated the benefits of individual models, described motivations, or compared models in the same language family. Coarfa et al. [19] compare two PGAS languages, Co-Array Fortran and Unified Parallel C and show that on four NAS parallel benchmarks, both languages can yield scalable performance when using bulk synchronous communication. Podobas et al. [20] look at task-based models and compiler implementations of OpenMP using micro-benchmarks and small kernels. Janssen et al. [21] use skeleton applications and simulations to explore how different message exchange schemes will perform on future hardware. Appeltauer et al. [22] compare eleven context-oriented languages using micro-benchmarks and show that they often have high execution overhead.

Chamberlain [17] and Saha [23] present multiple models as a theoretical foundation to point out the benefits of the authors' own approach. However, no study exists in which a single, realistic application is used to compare a wide range of traditional and emerging programming models or approaches in a systematic way, covering a wide range of aspects, including expressiveness, performance, productivity and manageability. Such information would be highly valuable to help direct research into future programming models and would contribute to current co-design efforts to design an efficient software and hardware ecosystem for extreme scale.

III. THE LULESH PROXY APPLICATION

The Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) [14] proxy app was originally developed as one of the five challenge problems in the DARPA Ubiquitous High Performance Computing (UHPC) program. Hydrodynamics was chosen for inclusion because it accounts for about 27% of data center utilization at DOD. LULESH solves one octant of the spherical Sedov problem using Lagrange hydrodynamics. In this section, we describe the baseline serial implementation of LULESH and the parallelism available within the code.

A. Serial Implementation

The serial reference implementation of LULESH is a hexahedral mesh-based physics code with two *centerings*. *Element centering* (at the center of each hexahedron) stores thermodynamic variables, such as energy and pressure. *Nodal centering* (where the corners of hexahedra intersect) stores kinematics values, such as positions and velocities.

The program flow involves a setup and initialization phase where the spatial coordinates of the domain are defined. Then an index set is defined for the single material. The index set is used to mimic multi-material problems and their associated algorithmic costs. Next, the initial problem state and boundary conditions are defined. The simulation is run via time stepping using a Lagrange leapfrog algorithm followed by a time constraint calculation. Algorithm 1 shows the three main computational phases in the serial LULESH code. The operations in italics denote communication and are absent in the serial version (they occur only in the parallel implementations and are shown for completeness).

Algorithm 1 Phases in a single LULESH time step

Advance node quantities
  Calculate forces (stress and perform hourglass correction)
  Calculate acceleration, velocity and update positions
Communicate positions
Advance element quantities
  Calculate kinematics
  Calculate artificial viscosity
    Communicate velocity gradients
  Apply material properties
  Update volumes
Calculate time constraints
  Calc Courant constraint
  Calc hydro constraint
  Reduce global minimum constraint

The Lagrange leapfrog algorithm consists of two major steps: advancing the node quantities followed by advancing the element quantities. Advancement of the node quantities requires calculating the nodal forces, which is the most compute-intensive part of the simulation. First, the volume force contribution of each mesh element is computed, followed by the stress values for each element. Then the contributions of each element are summed to its eight surrounding nodes. After a diagnostic check for negative volumes, the hourglass contribution is applied to each node's forces on an element basis. The forces obtained are then used to calculate accelerations via $F = ma$ with appropriate symmetry boundary conditions applied to the acceleration. Subsequently, the nodal velocities are advanced to the next time step using the accelerations and then the positions are advanced using the velocities.

The second part of the leapfrog phase involves advancing the element quantities. This entails calculating kinematic values for the elements based on the new nodal positions and velocities. Next artificial viscosities are calculated in two phases. First, element-based values are computed followed by the region-based values. Note, though, that these two loops are over the same space, since LULESH is a single material problem. However, for multi-material problems, a different loop structure would be used for the region-based loop. Next, material properties are applied to each element and the equation of state is evaluated. Finally, the new volume calculated in the kinematics is stored for use in the first phase of the leapfrog algorithm at the next time step.

Time constraints are used to limit how far in time the simulation advances at the next time step. Two functions, each which is only applied to elements whose volume is changing, are used to limit time stepping. The first calculates the Courant constraint, which is the characteristic length of an element divided by its change in volume. However, when the element is being compressed additional terms are added. The second calculates the hydro constraint, which is a prescribed maximum allowable volume change divided by the change in volume in the previous time step. The minimum value of these constraints for all elements in the mesh limits the time step that can be taken in the next leapfrog phase. The simulation can be run without any constraints if a small enough value is chosen as a fixed time step.

B. Available Degrees of Concurrency

LULESH contains large amounts of potential concurrency by design. The problem can be weak scaled to millions of domains by increasing the resolution of the problem being solved. Also, all loops in the reference serial implementation of LULESH, except for the one that checks the boundary conditions, perform work over either 45^3 elements or 46^3 nodes per time step. Finally, since each element can usually be represented as an independent work unit, SIMD vectorization can be performed on most loops. Therefore, parallel implementations have many options for exploiting multiple nodes and threads on modern machines.

IV. Traditional Programming Models

Model	Single-node	Multi-node
Serial	☑	
OpenMP	☑	
MPI		☑
CUDA	☑	
Chapel	☑	☑
CHARM++	☑	☑
Liszt	☑	☑
Loci	☑	☑

TABLE I
LIST OF THE PROGRAMMING MODELS LULESH IS IMPLEMENTED IN AND PARALLELISM AVAILABLE IN EACH

Table I lists the programming models in which we have implemented LULESH. The first column identifies if the model provides constructs for on-node parallelism and the second column denotes models that work across nodes. Models that provide cross-node parallelism along with specific code optimized for single node environments within a unified model are included in both columns. When the model is a library usable with C/C++/Fortran, we use the model along with C++. All single-node only models presented here can be used as multi-node models by using MPI to communicate between

nodes. In this section, we provide an overview of the commonly used models (MPI, OpenMP, hybrid MPI+OpenMP, and CUDA) along with a description of how the parallel implementation of LULESH differs from the serial code.

A. OpenMP

OpenMP uses pragma directives that are added to C, C++ and Fortran programs [2]. These directives can specify regions and loops to be parallelized by the OpenMP compiler using threads. Further, directives can be used to mark critical or atomic sections within the parallel regions. Through information added to the compiler directives, a programmer can specify which variables are shared or private in order to prevent false sharing and to isolate effects from multiple threads. Additionally, the pragmas allow specification of the number of threads per loop as well as reductions. Finally, OpenMP allows for nested parallelism with each thread capable of spawning child threads.

The OpenMP implementation of LULESH adds thread level parallelism by placing `#pragma omp parallel for` directives around the 45 loops over elements or nodes. In some cases multiple loops are wrapped in the same parallel region, resulting in 30 parallel regions. There are two places in LULESH where multiple threads can write data to the same node simultaneously. The resulting race conditions occur in the stress and hourglass routines where values are calculated on a per element basis and written to the nodes. To remove the race condition, all eight values computed for each node are placed in a temporary array as they are calculated and then a second loop sums the values to the nodes. The other changes necessary to support OpenMP in LULESH were in the hydro and Courant constraints where a reduction to find the minimum and maximum values is performed. Each thread finds its own local minimum and maximum and then the main thread uses the local values to determine the global value. In all cases, the techniques used were faster than using critical or atomic statements, and faster than transactional memory on Blue Gene/Q.

B. MPI

The Message Passing Interface (MPI) provides a comprehensive messaging API that can be used to communicate between processes that reside in separate address spaces [1]. The MPI standard specifies the functionality of the routines and a library writer is then free to implement them as seen fit for a particular hardware. Because each process has its own address space, a copy of the program and data tables is required for each MPI task.

A consequence of this distributed memory approach is that no process has immediate access to all of the data and access to data on neighboring processes is only possible through explicit messaging. The MPI implementation of LULESH adds communication of ghost fields (that represent copies of boundary data on neighboring processes) in two places. After the force calculation, the force values, f_x, f_y and f_z are communicated so that neighboring elements have the same values at their shared mesh nodes. Then, a second exchange of velocity gradients, Δv_ξ, Δv_η and Δv_ζ is needed between the two phases on the `monotonicq` calculation. A final all-to-all communication is performed to find the minimum Courant constraint and the maximum hydro constraint across all tasks.

C. Hybrid MPI plus OpenMP

A hybrid MPI plus OpenMP programming model typically refers to the situation where MPI handles coarse-grained domain-level parallelism and OpenMP is used for more fine-grained parallelism within each MPI process. Further, in most applications, OpenMP parallel regions are kept separate from MPI communication, since this simplifies synchronization both for the user code and the MPI implementation.

The hybrid MPI+OpenMP LULESH implementation uses MPI between nodes and OpenMP for cores on a node. OpenMP directives are put on the same loops as the pure OpenMP code to obtain strong scaling within an MPI task. The reductions of the hydro and Courant constraints are done within the threads on a task before messages are sent between tasks. The location of message passing constructs is unchanged from the pure MPI code.

D. CUDA

CUDA is set of C++ language extensions plus an accompanying runtime API for programming NVIDIA GPUs [16]. A computational kernel is programmed essentially as a C++ function that is run for every thread. Threads are grouped hierarchically into warps, blocks, and grids. The finest group, a warp (currently 32 threads), runs the same set of instructions in SIMD fashion with support for diverging execution paths. Threads in the same block are active at the same time and have access to fast, on-chip shared memory and local synchronization primitives. Finally, the various thread-blocks in a grid are executed completely independently and in arbitrary order, allowing for execution of problems too large to fit on the hardware simultaneously.

Porting LULESH to CUDA is similar to porting the code to OpenMP (and in fact the initial CUDA port was used as a basis for the OpenMP port). Kernels generally either map threads to mesh elements or mesh nodes. As in the OpenMP port, two element-centered kernels need to accumulate force values into the adjacent nodes, causing potential parallel write conflicts. The same solution to that problem is applied here: first writing the data to temporary arrays, then accumulating in a second node-centered kernel.

When optimizing the CUDA variant of LULESH, we successfully applied two types of GPU-specific optimizations. The first involves data layout. Hexahedral meshes are unstructured with respect to their nodes, so each element requires 8 indices or pointers to the adjacent node data. The serial code favors storing this information by element and then by corner node, whereas the GPU code favors the transpose. Thus when all the threads in an element-centered kernel request corner *i* for their particular element, the memory references are all adjacent in linear memory. Our mesh structure stores element data, node

data, and corner indices in arrays to be accessed by the CUDA kernels. If the optimal layout of these arrays changes from architecture to architecture, no GPU code requires rewriting, only the initial mesh setup code does.

The second useful optimization is in the mapping of sequential loops to threads. For some particularly complex numerical kernels, the performance is limited by the large number of registers required (which in turn reduces our utilization of the hardware). In some of these cases, we have achieved speedups by reducing the loop body granularity, i.e., by parallelizing over element corners instead of elements. After some per-corner computation, final results for each element are accumulated using the on-chip shared memory. Unlike the first type of optimization, this level of code optimization requires significant human effort to rewrite the relevant CUDA kernel, and could vary between generations of GPUs.

V. Emerging Parallel Programming Approaches

In addition to the commonly used, traditional HPC programming models described in the previous section, we have implemented LULESH in four new and emerging models: Chapel, CHARM++, Liszt and Loci. In this section, we describe these programming models as well as the implementation and optimization of LULESH using these models in more detail. We also discuss how our porting process was able to improve the design of some of these models.

A. Chapel

Chapel is an emerging parallel language initiated under the DARPA HPCS program with the goal of improving programmer productivity [17]. Chapel is designed using a block-imperative syntax with optional support for object-oriented programming, type inference, and other productivity-oriented features. Chapel supports both task- and data-parallel styles of programming, and permits these styles to be mixed arbitrarily. Task-parallelism is supported by creating abstract concurrent tasks that coordinate through shared synchronization and atomic variables. Data-parallelism is expressed via loops and operations on data aggregates—most notably, first-class index sets called *domains* and arrays defined via domains. Chapel supports reasoning about locality on node via the concept of a *locale*; for example, locales are often used to represent compute nodes on large-scale systems. Domains and arrays can be distributed across sets of locales in a high-level manner using the concept of *user-defined domain maps* [24].

The Chapel version of LULESH was initially created by transliterating the OpenMP version into Chapel; for example, OpenMP's `parallel for` loops were rewritten as data-parallel *forall* loops in Chapel. Chapel domains were introduced to represent the sets of nodes and elements, and their fields were stored using arrays defined by those domains. From this initial version, further localized rewrites were applied to make better use of Chapel features and improve performance. The code was converted from shared to distributed memory by applying the *Block* distribution to the node and element domains. Over the course of the code's evolution, we also changed the code from using regular 3D domains to an irregular 1D domain representation, and from using dense representations of the material elements to sparse ones. These changes demonstrate Chapel's ability to separate data structure implementation details from the computations that operate on that data in a rank-independent manner. In the specific case of LULESH, only the domain declarations and some supporting iterators had to change, while all of the physics computation, i.e., the core of the code, did not.

B. CHARM++

CHARM++ is a parallel programming system based on message-driven migratable objects [10], [25]. It is implemented as additions to the C++ language coupled with an adaptive runtime system. Parallelism in CHARM++ is created by over-decomposing an application into its logical work and data units, referred to as *chares*; the number of chares is typically more than the number of processors. The programmer expresses application flow, computation and communication as operations performed by chares. The distribution of chares to processors and scheduling of their execution is handled by the CHARM++ runtime system. Communication between chares is performed through remote method invocations and is also handled by the runtime system. Communication is asynchronous with respect to other chares which provides the benefit of adaptive overlap with computation. One optional language feature is that the parallel control flow can be specified by the user through a structured directed acyclic graph (SDAG) which can lead to more elegant code.

CHARM++ introduces new language extensions and syntax to allow the programmer to specify and use chares with respect to the runtime system, without modifying the standard C++ syntax. Therefore, the CHARM++ port is able to leverage the original C++ code to implement the physics computations of LULESH. The work in porting the application lies in expressing LULESH in terms of chares. This expression was handled by discretizing the problem into sub-domains, with appropriate ghost regions, such that two neighboring domains could share the relevant data necessary for computation.

The CHARM++ version of the port includes the same three communication phases as the MPI implementation. Therefore, the execution flow of a single chare is performed in three stages. Upon receiving the time step value for the next iteration, the chare executes the force calculation, which it then sends to its neighbors. After a chare receives all the information it is expecting from its neighbors for the force calculation, it begins executing again until it has sent the results of its viscosity calculation. In the same manner as the previous stages, the third execution stage begins with the receipt of the final viscosity message and ends with the computation and sending of the local minimum time step for the sub-domain the chare covers.

C. Liszt

Liszt is a Scala-based domain-specific language for solving partial-differential equations on meshes [12]. The language is

designed for code portability across heterogeneous platforms. The problem domain is represented as a three-dimensional mesh whose elements can be accessed only through mesh-based topological functions as immutable first-class values. The mesh is initialized at program start time and its topology does not change over the program's lifetime. Fields are abstracted as unordered maps indexed only using mesh elements. Liszt provides three features for parallelism: a parallel *for-comprehension* on sets of mesh elements, atomic *reduction operators* on field data, and *field phases*, i.e. read/write restrictions on field data inside a for-comprehension. Moreover, Liszt does not support recursion. These semantic constraints ensure that the Liszt compiler can infer data dependencies automatically, enabling it to generate a parallel implementation for code written in a serial style. One drawback is that Liszt provides no high-level abstraction for load balancing and mesh decomposition. Lack of direct programmer control on these aspects has performance implications for certain back ends.

The Liszt implementation of LULESH is almost identical to the serial C++ version, modulo syntactic differences. Race conditions in the stress and hourglass routines are handled implicitly due to the atomicity of reductions, thus requiring no programmer intervention. Furthermore, Liszt also supports dense vector and matrix operations, which significantly reduces the overall number of lines of code. The current Liszt implementation does not preserve consistent planar orientation for mesh elements; hence for each iteration the alignment of a cell's vertices along the x, y and z planes needs to be recalculated in the `monotonicq` gradient routine incurring additional overhead.

From the Liszt source, the compiler generates equivalent C++ code that can then be compiled for the desired execution platform. This output C++ code is fairly modular itself, and computationally intensive sections of it can be hand-optimized, or completely replaced with relatively little effort. Currently, Liszt supports back ends that produce either MPI or GPU code.

D. Loci

Loci is a C++ framework that implements a declarative logic-relational programming model [18]. The programming model is implicitly parallel and uses relational abstractions to describe distributed irregular data structures. A logic programming abstraction similar to Datalog [26] is used to facilitate composition of transformation rules. The programming model exploits a notational similarity to mathematical descriptions found in papers and texts of numerical methods for the solution of partial differential equations [27]. In addition, the programming model facilitates partial verification by exploiting the logic programming model to provide runtime detection of inconsistent or incomplete program specification. Parallel execution is achieved using loosely synchronized SPMD approach that exploits the data-parallelism that naturally emerges from the distribution of relations to processors. Communication costs in the generated parallel schedule are controlled through message vectorization and work replication optimizations [28].

In the Loci code, computations are defined at a fine-grained per-entity level. For example, the computation for a single node is provided and the system coordinates the identification of sets of entities that require the same computation and schedules loops over these entities. For the Loci implementation of LULESH, we utilize the built-in data-types for 3D vectors to simplify the algorithm description. For example, the time-step advance of the mesh coordinates are specified as:

```
// Update nodal coordinates using computed nodal
// velocity. Inform Loci that it is permitted to
// destroy old coordinate values in the update.
$rule pointwise
(coord{n+1}<-coord{n}, vel{n+1}, dt{n}),
  inplace(coord{n+1}|coord{n}) {
    $coord{n+1} = $coord{n} + $vel{n+1} * $dt{n};
}
```

Note, the above computation is for a given node of the computation. The Loci preprocessor generates the loop over entities and the runtime system uses set inferences to compute loop bounds automatically. Reductions are described in Loci using a map-reduce formalism. For example, element contributions to nodal forces are computed and then combined using the `hexnodes` relation that contains the indexes of the nodes that form an element. Loci schedules the communication such that the reductions are applied consistently in parallel. Thus the force computation is represented by the Loci rule:

```
// Compute force on element nodes due to
// element stresses
rule apply(hexnodes->force<-elemNodes,
  p_next,q_next) [Loci::Summation] {
  double P = -$p_next - $q_next ;
  // pressure contribution to stress
  vect3d Stress(P, P, P) ;
  // element node normals
  Array<vect3d, 8> B ;
  ElemNodeNormals(B, $elemNodes) ;
  // Integrate stress to nodes
  Array<vect3d, 8> nf ;
  ElemStressToNodeForces(nf, B, Stress) ;
  // Here we join (sum) the element
  // forces to nodes
  for(int i=0; i<8; ++i)
    join($hexnodes[i]->$force, nf[i]) ;
}
```

VI. Optimizations to LULESH

In previous work [29], we have shown four optimizations that significantly improve the performance of the OpenMP version of LULESH. In this section, we present these optimizations along with others, followed by a discussion of the ease of applying these optimizations in various programming models to achieve portable performance of large codes.

A. Optimization Possibilities in LULESH

Loop fusion is the combination of multiple (often adjacent) loops into a single loop, eliminating redundant data motion. Temporary arrays produced by one loop and consumed by another can be eliminated as well [30].

Model	Loop fusion	Global allocation	Data struct. trans.	Vectorization	Blocking	C-C Overlap
Chapel			☑		☑	☑
CHARM++					☑	☑
Liszt	☑	☑	☑	*	*	
Loci		☑	☑	*	☑	☑

TABLE II
OPTIMIZATIONS MADE EASIER BY EACH MODEL

Global data allocation involves moving all malloc and free statements of temporary variables outside of the time step loop. Doing this reduces the number of system calls and also the number of times a local to global TLB mapping is made.

Data structure transformations change the typical struct of arrays representation to an array of structs. This transformation has two performance advantages. First, it reduces the number of data streams, thereby using the hardware prefetch units more effectively. Second, it reduces the number of indirect accesses and cache misses required for codes like LULESH that read unstructured data across multiple centerings. For example, if x, y and z are typically accessed together, storing them together generally improves performance. While there are ways to add support for this in C and C++ [31], doing so requires using that approach during initial development. Without such a design, time consuming and error prone changes would be required for each architecture.

Vectorization using SIMD (Single Instruction Multiple Data) instructions increases the performance of the compute heavy portions of LULESH, such as the hourglass calculation and part of the `monotonicq` calculation. These instructions improve performance by evaluating multiple compute operations in a single cycle. For most codes, programmers rely on the compiler to issue these. However, codes written in lower level languages like C lose higher level information about the calculation being performed. Without this information, the compiler must be conservative. Due to these and other difficulties at the compilation level, it is often possible to get performance gains by manually adding intrinsics and directives that are typically not portable.

Blocking is a decomposition technique that breaks up a large data computation into a set of memory-coherent blocks. The size of these blocks is chosen to fit within a level of the memory hierarchy (registers, cache, TLB). This increases temporal data reuse as the full computation is performed by iterating over the blocks. Because the ideal block size is dependent on both the problem and the target hardware, writing portable blocked code in a language like C is difficult. Although possible for small kernels such as those found in the BLAS library, this is impractical for larger codes [32].

Communication and Computation Overlap allows for better hardware utilization by keeping both the processor and the network busy at the same time. However, modifications for overlap in a complex code often create maintainability and portability issues.

B. Applicability to Each Model

Some programming models reduce the amount of work needed to optimize code, effectively increasing the portable performance of the code for a given programmer effort. Table II lists programming models that allow easier expression or portability of optimizations relative to C++ code using MPI and OpenMP for parallelism. We use checkmarks (☑) to denote places where the model makes it easier for a programmer to perform optimizations and asterisks (*) where the model makes a compiler writer's job easier to perform the static analysis to optimize the code automatically.

Chapel's domain maps can be used to implement many high-level tuning techniques. These maps are used both to distribute data among nodes and to specify the memory layout, parallelization strategies and iteration order within a node. By applying appropriate domain maps, blocking optimizations can be achieved. Common domain maps such as block and cyclic are provided within the standard library but users can also define their own. By changing the domain map of a domain, all operations on its indices and arrays are rewritten to use the specified strategy with no further modification to the source needed. Zippered iterators [33] perform what we define as data layout transformations. Chapel also has asynchronous communication constructs that make it easier to overlap computation and communication.

CHARM++ leverages over-decomposition of an application into chares to achieve a number of optimizations. Blocking is enabled by choosing the number/size of chares so that the data can fit in cache. Communication and computation overlap occurs naturally by scheduling multiple chares per processor – when one chare is waiting on communication, another can perform its computation. In addition, CHARM++ eases load balancing via transparent chare migration by the runtime system. CHARM++ still allows any optimization that can be performed to a C++ application, such as loop fusion or data structure transformations, however it does not provide any support to make these easier than in C++.

As a domain specific language for PDEs on meshes, Liszt allows a higher level expression of mesh information and its associated calculations. This high-level, domain-specific information about the problem makes it easier for a compiler to optimize the application. In this setting, the static analysis needed to determine profitability and safety of vectorization and blocking is less complex. It is also easier to determine if and when to perform optimizations such as loop fusion. By

moving this tuning to the compiler, portability is increased. Finally, since all data are allocated in Liszt globally it performs this optimization already, though this can be a drawback when memory is tight.

The Loci programming model performs many optimizations for the user such as automatically generating loops over element and node sets. While the system does not presently implement loop fusion optimizations, this is not a fundamental limitation and such optimizations can be implemented in the future. Loci utilizes a blocking strategy to minimize memory allocation, improve cache performance and access costs involved in transferring information between loops that are potential candidates for loop fusion. The model supports global data allocation, but defaults to the opposite approach of minimizing memory footprint through variable lifetime reduction and maximizing memory recycling through a randomized greedy scheduler. Loci also utilizes aliasing directives when synthesizing loops over sets of elements to allow the compiler to better utilize SIMD instructions. Finally, a work-replication optimization eliminates communication by re-computing values on the local processor. Although overlapping of communication and computation is not implemented, it can be added to the runtime system without changing the program specification.

VII. Comparative Evaluation

Evaluating the various strengths and weaknesses of programming languages requires a holistic approach as there are many factors that impact programmability, productivity and performance. In this section, we look at the productivity of the languages, the performance they currently achieve, and the ease of performance tuning on various architectures. For emerging languages and programming models, we report their current state along with an analysis of what the model is capable of with further implementation work.

A. Productivity

Programmer productivity is difficult to measure in a controlled manner due to the differing strengths and weaknesses of each programmer and the fact that some languages allow certain applications to be more effectively expressed than others. However, source lines of code (SLOC) is a quantitative metric used often and one that does not require a carefully controlled experiment. SLOC is a measure of the number of lines of code not counting blank lines and comments. The SLOC metric has limitations, such as the implicit assumption that each line of code requires the same effort and thought in each language, but we believe that in the case of the LULESH proxy app, it is a reasonable way to quantitatively compare programming models and to derive overall trends. We measure SLOC using sloccount [34] and use it to compare all investigated versions of LULESH. Table III shows the results.

From the SLOC counts, we can see a clear advantage of newer programming models in reducing source code size. The Chapel, Loci and Liszt versions are about one quarter the size of the MPI program or smaller, while providing both on- and off-node parallelism in a single code. In addition, Liszt handles multiple target architectures from this single source compounding its advantage. As we explain later in Section VII-C, another advantage of these approaches is that fewer lines of code need to be changed to implement an optimization, i.e., the code becomes more maintainable.

Model	**SLOC**
Serial	2183
OpenMP	2403
MPI	4291
MPI + OpenMP	4476
CUDA	2990
Chapel	1108
CHARM++	3922
Liszt	1026
Loci	742

TABLE III
SOURCE LINES OF CODE FOR LULESH IN EACH PROGRAMMING MODEL

B. Performance

Our goal with presenting performance results is to provide an intuition regarding trade-offs of using different programming models on existing systems. For the non-GPU versions we measured performance on two systems: a cluster of Intel Sandy Bridge processors named Cab, and an IBM Blue Gene/Q (BG/Q) system called Sequoia. Strong scalability tests were run on a single-node with a problem size of 81^3 and 121^3 on the node. Weak scalability tests were run at a problem size of 32^3 and 48^3 per core. We compared the GPU versions (CUDA, Liszt) to the OpenMP version on a single node containing Intel Westmere CPUs and NVIDIA Tesla M2050 GPUs at problem sizes of 45^3, 55^3, 65^3, 75^3, 85^3 and 96^3. For all experiments, each programming model was allowed to find its optimal workload division between tasks, threads, or other units. Optimal on-node configurations were utilized in the weak scaling studies on the same architectures.

All tests were run ten times for 500 iterations and we report the execution time per iteration here. The time does not include mesh generation and load time because these costs are a small fraction of total runtime for large simulations. For the non-CUDA variants, except Liszt, we used icc version 12.1.339 with the -O3 -mavx options on the Sandy Bridge cluster, and xlc 12.1 with -O3 -qhot=novector -qsimd=noauto -qarch=qp on the BG/Q system. For Liszt we used Clang on the Intel machines because Liszt-generated code makes excessive use of the *memcpy* function to access mesh fields and Clang++, in contrast to other compilers that we tested, was able to optimize away these constructs leading to a significantly improved performance. We do not report Chapel and Loci performance on BG/Q because they are not yet ported to this new architecture.

Version	Blue Gene/Q 32^3	Blue Gene/Q 48^3	Sandy Bridge 32^3	Sandy Bridge 48^3
Serial	0.179	0.617	0.027	0.099

TABLE IV
SERIAL PERFORMANCE (TIME PER ITERATION IN SECONDS) OF LULESH ON BLUE GENE/Q AND THE SANDY BRIDGE CLUSTER

1) Baseline Performance for the Serial, OpenMP and MPI versions: Table IV shows the performance of the serial code on the problem size that is weak scaled in order to provide a baseline cost per iteration. Figure 1 shows that the OpenMP variant exhibits good strong-scaling behavior, indicating that there is ample parallelism to be harvested in the code. Figure 2 (top) shows that the MPI version of the code exhibits near perfect weak scaling of the serial code, which is not surprising since there is not much communication and most of what is present is local.

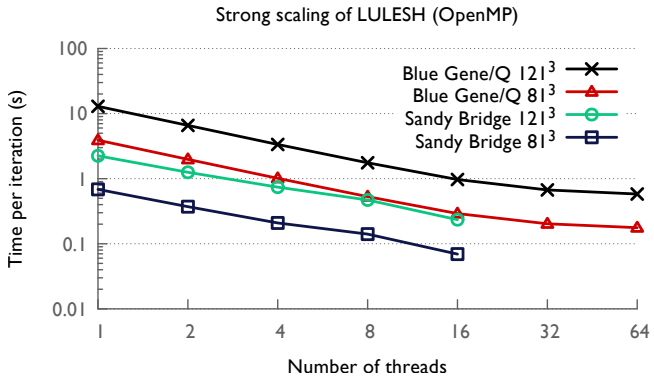

Fig. 1. Strong scaling performance of the OpenMP implementation

Figure 2 (bottom) shows that the hybrid MPI plus OpenMP version displays similar scaling characteristics, but significantly worse performance. OpenMP code contains extra data motion to handle the race condition when summing to the nodes in the hourglass and stress computations. Also, different experiments on both machines show that for low processor counts, the OpenMP overhead is higher than the MPI overhead and can cause hybrid codes to run slower. However, there are other advantages to having threaded code, including a smaller memory footprint due to shared data structures, and a better surface to volume ratio for domains (in cases unlike LULESH) that need to be subdivided into smaller pieces in order to fit within memory.

2) Performance of Emerging Programming Models: When evaluating emerging technologies, one must keep in mind the focus of effort in developing them and their relative maturity. In addition, they often compile to intermediates like C to avoid the need to develop and constantly update their own compiler back ends. Therefore, performance is not always as good as commercially supported technologies.

Figure 3 shows Chapel's single node performance. Chapel achieves more than 80% efficiency at 16 cores although its

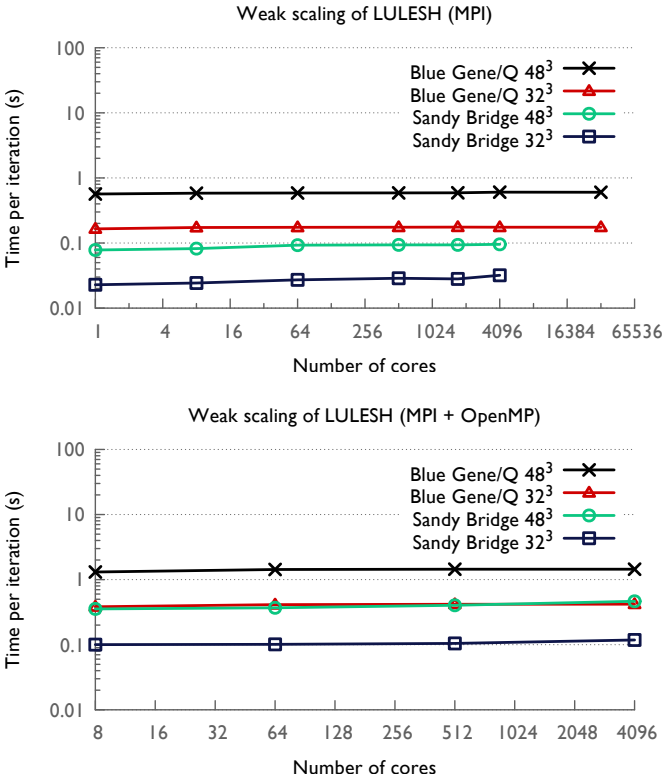

Fig. 2. Weak scaling performance of the MPI and MPI+OpenMP ports

single-core performance is significantly worse compared to OpenMP. Part of the difference comes from the fact that in computing the cube input set, Chapel linearizes the global set of elements/nodes and block-distributes the 1D linearization of them rather than block-distributing the conceptual 3D set. The result of this is that the surface-to-volume ratio in the Chapel implementation is much worse than OpenMP's and undoubtedly results in a lot more communication.

These results do show that Chapel has the machinery in place to be a high performance parallel language. However, like many other new languages, performance is limited by the large resources needed to create efficient compilers and runtime technology. Currently Chapel relies on converting its code to C99 and most of the Chapel literature is focused on expanding the feature base to broaden its usability. Therefore, it is not surprising that its performance is currently behind highly-optimized native code languages and compilers.

CHARM++ also weak scales extremely well on both machines as shown in Figure 4. Its performance is compara-

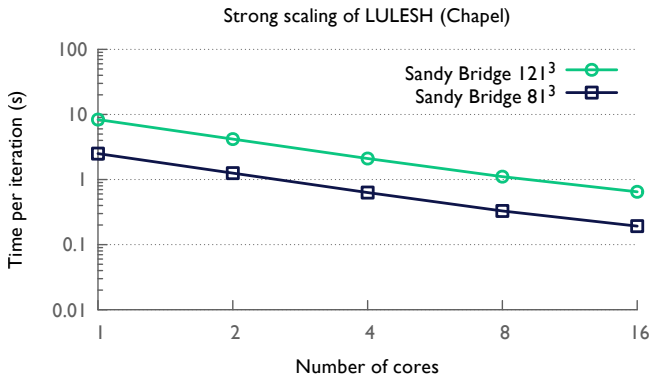

Fig. 3. Strong scaling performance of the Chapel implementation

ble to the MPI implementation that it was based on and it out-performs that implementation on one core of BG/Q. LULESH is not a good fit for porting to CHARM++ because it does not allow the exploitation of two of CHARM++'s strongest features: load balancing and asynchronous execution. These features are not very useful for LULESH because it is perfectly load balanced and does not leave much room for computation-communication overlap. Thus, obtaining comparable performance to MPI on LULESH indicates a strength of the CHARM++ approach.

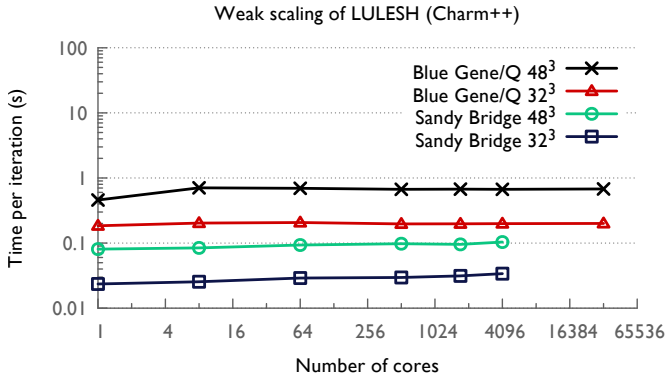

Fig. 4. Weak scaling performance of the CHARM++ implementation

Figure 5 shows the results of a weak scaling experiment on the Liszt code. Liszt scalability is significantly impacted for small problem sizes once the calculation is run on more than one node (and hence requires cross-node communication). From profiling various parts of the physics, we saw that performance issues were occurring in `courantconstraint` where a global reduction occurs, `positionupdate` and in `monotonicqgradient` where ghost zone exchanges occur. Another limiting factor with the current implementation (but not the language itself) is that the mesh file, which scales with problem size, is read on one processor and then distributed. Also, on BG/Q, each processor can only access a fraction of a node's memory, further restricting the sizes that can be run.

The Liszt MPI back end's on-node performance was about 50% worse when compared to the native MPI implementation, which was caused by the function call semantics in the Liszt runtime system. Liszt performance has been focused on memory bound codes and LULESH was the first significant compute intense application ported to the language. By working with the Liszt developers we were able to insert native C++ code into our Liszt program for two compute intense kernels, hourglass and `monotonicqgradient`, which brought the runtime of the Liszt version to within 10% of the MPI implementation. This optimization can be integrated in the the Liszt compiler and hence does not demonstrate an inherent disadvantage of this programming model.

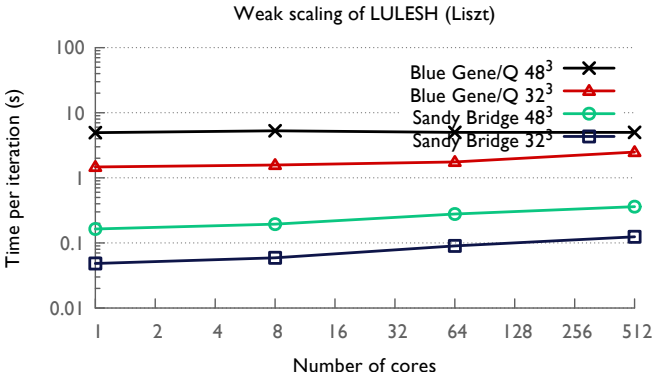

Fig. 5. Weak scaling performance of the Liszt implementation

Loci is the one language that was able to outperform a comparable approach. Figure 6 shows its strong scaling on a node where it outperforms OpenMP by up to 15%. Overall, its scaling is a bit worse than OpenMP, but good serial performance often limits overall scalability.

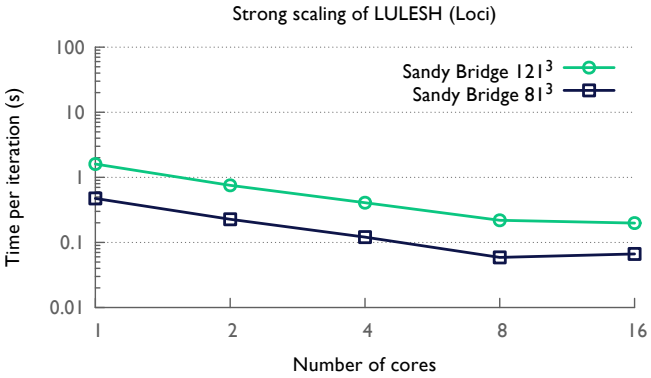

Fig. 6. Strong scaling performance of the Loci implementation

Figure 7 shows the weak scaling performance of the Loci implementation on the Sandy Bridge cluster. In this case also, the performance of the Loci port comes close to that of its counterpart, the MPI implementation. This shows the potential of Loci as a programming model for extreme scale - with the fewest number of lines of code, it performs comparably to the OpenMP and MPI implementations.

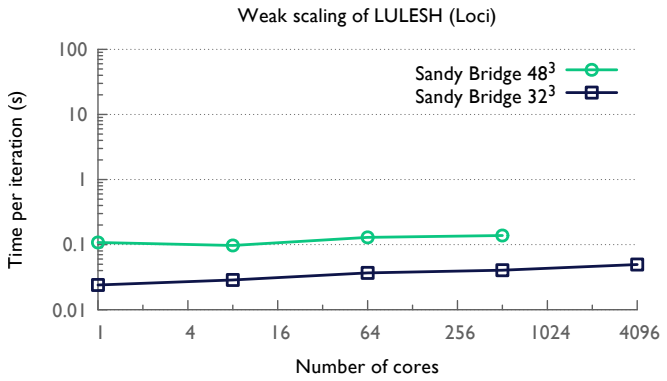

Fig. 7. Weak scaling performance of the Loci implementation

3) Comparison of GPU and OpenMP Variants: Table V shows a performance comparison of LULESH OpenMP, CUDA and Liszt GPU variants. In this test our goal was to grow the problem size in order to explore the performance characteristics of the OpenMP and CUDA versions, which share many similarities in terms of the style of coding. The Liszt-produced GPU code was about a factor of two slower than the original CUDA code. In this case even replacing Liszt code with native CUDA code did not help performance because the compiler's default (and the only available) thread-scheduling strategy, *coloring*, proved to be sub-optimal for LULESH. Our assessment is that the language and approach are not at fault, but that the current implementation requires further optimization.

Version	45^3	55^3	65^3	75^3	85^3	96^3
CUDA	0.008	0.014	0.023	0.035	0.052	0.069
Liszt	0.016	0.029	0.047	0.071	0.103	0.147
OpenMP	0.017	0.032	0.053	0.086	0.128	0.182

TABLE V
GPU AND OPENMP PERFORMANCE OF LULESH. A SINGLE NVIDIA FERMI M2050 (CUDA AND LISZT) IS COMPARED AGAINST DUAL INTEL WESTMERE HEX-CORE CPUS (OPENMP), SHOWING A 2-2.6X PERFORMANCE ADVANTAGE (NOTE: A SECOND AVAILABLE GPU IS NOT EXERCISED IN THIS TEST).

C. Ease of Optimizations

Aside from the productivity in implementing a proxy app in a particular language and its performance, we also explore how easy it is to apply optimizations to the code. This has an impact not only on the ease of improving performance of a code, but also provides a good indicator of performance portability across platforms, where each platform may need a different set of optimizations, as well as code maintainability.

We start by looking at applying blocking to Chapel, which can be achieved using a high level mapping construct:

```
var x, y, z: [1..n] real;
var xd, yd, zd: [1..n] real;

x += xd * dt;
y += yd * dt;
z += zd * dt;
```

By adding a domain map we can change the underlying data structure, without modifying the source:

```
const map = [1..n] dmapped Block([1..n]);
// ...rest of code unchanged...
```

It should be noted the same map construct is also used to block computations and change data distributions among nodes. Other optimizations, such as data structure transformations, are applied similarly without touching the source.

For CHARM++, we discuss the discretization of the domain into multiple smaller sub-domains on a processor, leading to a blocked computation. The creation of a chare or arrays of chares is done simply by calling its class constructor using the language constructs provided through CHARM++. For example in LULESH, if the class name of the chare array is `SubDomain`, defined in the interface file as `array [3D] SubDomain`, the handle provided by the runtime system to the programmer is `CProxy_SubDomain`. Calling the constructor is done by using the method `ckNew` on one of these handles. After any constructor arguments, chare arrays also take size arguments. Any chare can use this method to create other chares. For LULESH, the sub-domains were created during the initialization phase of the main chare. The sub-domain chare array did not take any other constructor arguments so only the dimension sizes were specified:

```
subdomains = CProxy\_SubDomain::ckNew(chareDimX,
    chareDimY, chareDimZ);
```

The actual discretization of the domain was performed at runtime by way of command line arguments, and it was found that about eight chares per processor produced the best performance in most cases.

CHARM++ also simplifies the process of checkpointing for fault tolerance with minimal coding overhead. Since CHARM++ programs are already equipped to migrate chares across processing units for load balancing, the mechanism to save program state is essentially already in place. Furthermore, the CHARM++ runtime system provides routines to perform the checkpoint operation of these chares to disk or memory.

By being domain-specific, Liszt enables and/or simplifies optimizations at compilation. A higher level syntax provides more information than C++ and the similarity of expressions means optimizations such as loop fusion can be performed using similar changes to those made in C++. For instance, in Liszt, the following two loops:

```
val dt = timestep()
for (v <- vertices(mesh)) {
  velocity(v) = acceleration(v) * dt
}
for (v <- vertices(mesh)) {
  position(v) = velocity(v) * dt
}
```

can be fused together as a single loop:

```
val dt = timestep()
for (v <- vertices(mesh)) {
  position(v) = acceleration(v) * dt * dt
}
```

eliminating the (temporary) field `velocity`. While manually performing this optimization in Liszt is somewhat less effort than in C++, the advantage comes from enabling a compiler to have more information. Therefore a compiler is more likely to be able to perform this optimization automatically with no programmer intervention.

The Loci framework is implemented using a preprocessor that translates rule descriptions into C++ kernels along with a runtime system that manages kernel coordination in the form of runtime generated execution schedules or plans. Generally, programs are optimized for an architecture by the preprocessor and runtime system. Since the assembly of kernels is performed at runtime, loop fusion of the type that is described for the Liszt DSL is not possible without some sort of just-in-time compiler support, however, Loci does identify computation chains that represents loops that could be fused. Instead of fusing the loops, the kernel computations are blocked into segments that can fit into the L1 cache and then chained giving nearly the same benefit as loop fusion.

In addition to these optimizations at the vector instruction level, Loci manages the lifetime of intermediate values of the computation in order to reduce aggregate memory requirements of the program while improving cache reuse at higher levels. Loci programs can typically see about a 10-20% performance improvement when intermediate variable lifetimes are optimized. In addition, the work replication optimization where select values are recomputed on processors saving interprocessor communication can be enabled. Under this optimization, the replication of work can be determined based on the relative cost of the computation versus communication costs and is generally an effective approach for high latency network fabrics.

VIII. Future Work and Lessons Learned

Implementing LULESH in some of the new models highlighted deficiencies in their ability to express scientific applications. This led the developers of the respective languages to make enhancements as LULESH was being written. We also identified other issues in some of these models which can be considered future work. For example, Chapel used LULESH as a motivating example to implement fully unstructured grid support in the language. While LULESH is a block structured mesh using an unstructured access pattern, it is meant as a proxy for fully unstructured codes, so this language feature allows more complex applications to be expressed in Chapel.

For the Liszt language, LULESH was the first computationally intensive code ported to the model and the port showed that Liszt handles memory-bound loops well. However, the need to improve the code generation for compute-heavy loops was shown. Other lessons learned from the port include several ideas for new Liszt abstractions and fine-grained control over data and workload distribution that will allow algorithms such as anisotropic diffusion to be coded in Liszt.

In summary, this paper presents a unique study exploring the implementation of a single application in eight different parallel programming models ranging from on-node and off-node to hybrid models and models for accelerators. In the spirit of co-designing hardware, system software (which includes programming languages/models, runtimes and tools), and applications, we implemented LULESH, a shock hydrodynamics application in these eight traditional and emerging programming approaches. In this paper, we described our experiences and details of the LULESH implementations. We also outlined various optimizations and showed how new programming models reduce required programmer effort relative to C++.

This study was not intended to find one champion language that can be used for all kinds of applications and architectures in the future; but to explore new approaches that can help achieve extreme-scale performance and high programmer productivity at the same time. Models such as Chapel, Loci, and Liszt show promise in terms of high programmer productivity. The source lines of code (SLOC) needed to produce a parallel implementation in these models was less than the serial C++ code. However, some of these models require more developmental effort to match the performance of the MPI implementation. When comparing models for the GPU, the Liszt port is almost one third the size of the CUDA implementation and did not require any changes from the CPU version to the GPU version.

In terms of performance, Loci along with Charm++ were the two performance competitive programming models among the emerging approaches. When evaluating this result, one has to remember that being older emerging language approaches, these technologies have had more time to mature their compiler and runtime technologies. These results should be viewed as an example of what can be done with other models given the time and resources to mature.

This study shows that the newer approaches contain many higher level constructs and compiler-driven advantages for tuning codes to multiple architectures. While each model facilitates a certain set of optimizations, no model in its current form handles all optimizations we looked at. Because of the power of these high-level tunability advantages, we hope that it would take less time and effort to port between significantly different architectures with the newer languages and still achieve reasonably good performance. In addition, since tuning is handled at a high level, the risk of programmer error should be smaller. Therefore, programmer productivity in these emerging approaches will be higher.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-586774). This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. This work was partially supported by the DOE Office of Advanced Scientific Computing Research. Accordingly, the U.S. government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. government purposes.

Neither the U.S. government nor Lawrence Livermore National Security, LLC (LLNS), nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, or process disclosed, or represents that its use would not infringe privately owned rights. The views and opinions of authors expressed herein do not necessarily state or reflect those of the U.S. government or LLNS, and shall not be used for advertising or product endorsement purposes.

REFERENCES

[1] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

[2] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46 –55, jan-mar 1998.

[3] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000. [Online]. Available: http://doi.acm.org/10.1145/352029.352035

[4] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Procesor Parallelism*. O'Reily Media, July 2007.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[6] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," vol. 180, p. 012037, 2009.

[7] J. M. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947 – 1980, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819198000933

[8] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Apra, "Advances, applications and performance of the global arrays shared memory programming toolkit," vol. 20, no. 2, pp. 203–231, Summer 2006.

[9] D. Loveman, "High performance fortran," *Parallel Distributed Technology: Systems Applications, IEEE*, vol. 1, no. 1, pp. 25 –42, feb. 1993.

[10] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[11] C. Moler, "MATLAB-a mathematical visualization laboratory," in *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, 29 1998-march 3 1988, pp. 480 –481.

[12] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: a domain specific language for building portable mesh-based PDE solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011.

[13] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A structured english query language," in *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, ser. SIGFIDET '74. New York, NY, USA: ACM, 1974, pp. 249–264. [Online]. Available: http://doi.acm.org/10.1145/800296.811515

[14] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[15] "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)," http://computation.llnl.gov/casc/ShockHydro.

[16] "NVIDIA CUDA C Programming Guide Version 4.2," Tech. Rep., April 2012.

[17] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[18] E. A. Luke and T. George, "Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis," *Journal of Functional Programming, Special Issue on Functional Approaches to High-Performance Parallel Programming*, vol. 15, no. 03, pp. 477–502, 2005.

[19] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: co-array fortran and unified parallel C," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005.

[20] A. Podobas, M. Brorsson, and K.-F. Faxén, "A comparison of some recent task-based parallel programming models," in *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers, (MULTIPROG '2010)*, 2010.

[21] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, pp. 4–8, Mar. 2011. [Online]. Available: http://doi.acm.org/10.1145/1964218.1964220

[22] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A comparison of context-oriented programming languages," in *International Workshop on Context-Oriented Programming*, ser. COP '09. New York, NY, USA: ACM, 2009, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/1562112.1562118

[23] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, "Programming model for a heterogeneous x86 platform," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 431–440. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542525

[24] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, D. Iten, and V. Litvinov, "Authoring User-defined Domain Maps in Chapel." Cray, Inc., May 2011.

[25] O. S. Lawlor and L. V. Kalé, "Supporting dynamic parallel object arrays," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 371–393, 2003.

[26] J. Ullman, *Principles of Database and Knowledgebase Systems*. Computer Science Press, 1988, pp. 53–66.

[27] Y. Zhang and E. A. Luke, "Concurrent composition using Loci," *IEEE/AIP Computing in Science and Engineering*, vol. 11, no. 3, pp. 27–35, May/June 2009.

[28] K. Soni, N. Cain, and E. A. Luke, "Work replication: A communication optimization in Loci," in *Proceedings of the ISCA 21nd International Conference on Parallel and Distributed Computing and Communication Systems*, New Orleans, LA, September 2008, pp. 50–55.

[29] I. Karlin, J. Mcgraw, E. Gallarado, J. Keasler, E. A. Leon, and B. Still, "Memory and parallelism tuning exploration using the LULESH proxy application," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC 2012)*, November 2012.

[30] G. Gao, R. Olson, V. Sarkar, and R. Thekkath, "Collective loop fusion for array contraction." in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, Aug. 2004, pp. 281–295.

[31] J. Keasler, "Performance Portable C++," *Dr. Dobbs Journal*, pp. 40–47, June 2008.

[32] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, 1998, pp. 1–27.

[33] B. L. Chamberlain, S.-E. Choi, S. J. Deitz, and A. Navarro, "User-Defined Parallel Zippered Iterators in Chapel," in *Proceedings of Fifth Conference on Partitioned Global Address Space Programming Models*, October 2011, pp. 1–11.

[34] D. Wheeler. (2012) SLOCCount. [Online]. Available: http://www.dwheeler.com/sloccount