LLNL-TR-608824

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

# LULESH Programming Model and Performance Ports Overview

I. Karlin

December 21, 2012

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

# LULESH Programming Model and Performance Ports Overview

Ian Karlin[1], Abhinav Bhatele[1], Brad Chamberlain[2], Jonathan Cohen[1], Zachary Devito[3], Maya Gokhale[1], Riyaz Haque[4], Rich Hornung[1], Jeff Keasler[1], Dan Laney[1], Edward Luke[5], Scott Lloyd[1], Jim McGraw[1], Rob Neely[1], David Richards[1], Martin Schulz[1], Charels H. Still[1], Felix Wang[6], and Daniel Wong[7]

[1]Lawrence Livermore National Laboratory
[2]Cray Inc.
[3]Stanford University
[4]University of California, Los Angeles
[5]Mississippi State University
[6]University of Illinois at Urbana-Champaign
[7]University of Southern California

LLNL-TR-608824

## Contents

Contents . . . 2
1 Introduction . . . 4
2 Serial Implementation . . . 5
3 Programming Model Ports . . . 5
3.1 MPI . . . 6
3.2 OpenMP . . . 6
3.3 C Version . . . 6
3.4 MPI plus OpenMP . . . 6
3.5 CUDA . . . 6
3.6 A++ . . . 6
3.7 Chapel . . . 7
3.8 Charm++ . . . 7
3.9 Liszt . . . 7
3.10 Loci . . . 7
3.11 In Progress/Planned . . . 8
4 Tuned LULESH Versions . . . 8
4.1 Performance Optimizations . . . 8
4.1.1 Loop Fusion and Array Contraction . . . 8
4.1.2 Global Allocation . . . 8
4.1.3 Increased Vectorization . . . 9
4.1.4 NUMA Aware Data Allocation . . . 9
4.2 Naming Convention Overview . . . 9
4.3 Best Versions . . . 9
4.3.1 Lawrence Livermore Linux Clusters . . . 9
4.3.2 BlueGene/Q . . . 10
4.3.3 Intel Xeon Phi . . . 10
5 Other LULESH Versions . . . 10
5.1 Fully Unstructured . . . 10
5.2 Vista Database . . . 10
5.3 One Dimensional Communication Pattern . . . 10
5.4 Clean Code . . . 10
5.5 Transactional Memory . . . 10
5.6 Critical Sections . . . 10
5.7 Fault Tolerant Checkpoint . . . 11
5.8 Planned Versions . . . 11
6 Performance Measurement and Correctness Checking . . . 11
6.1 Performance Measurement . . . 11
6.2 Correctness Checking . . . 11
6.3 Changes not to make to LULESH . . . 13
6.3.1 Mesh Representation . . . 13
6.3.2 Loop Structure . . . 13
6.3.3 Extra Compute . . . 13
7 Acknowledgments . . . 13

References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

## Abstract

*This document gives a description of various versions of the LULESH proxy application described in [1]. All the the codes described in this document are available at* `https://codesign.llnl.gov/lulesh.php`*. This report describes versions that fall into three categories: performance tuning exploration, programming model exploration and other versions. In this document we focus on giving a brief overview of what makes each version different from the others. The purpose is to allow others to reuse our work as a starting point for their exploration. Therefore, references to publications and presentations are provided to allow those who want more detailed information to be able to find it.*

## 1 Introduction

Hydrodynamics is widely used to model continuum material properties and material interactions in the presence of applied forces. It can consume up to one third the runtime of these applications. To provide a simpler, but still full-featured problem to test various tuning techniques and different programming models the Livermore Unstructured Lagrange Explicit Shock Hydro (LULESH) mini-app was created as one of five challenge problems in the DARPA UHPC program [1]. Hydrodynamics was chosen as one of the challenge problems because it consumes 27% of data center resources throughout DOD. Mini-apps like LULESH provide a smaller full featured program that allow for less time consuming exploration of various programming models and performance tuning techniques [10, 9]. The successful lessons learned from the mini-app exploration can then be applied to a full application.

In this document we describe versions of LULESH that are implemented in different programming models, performance tuned, and/or have additional functionality beyond the reference code. Each of the versions described here is hosted or linked to from `https://codesign.llnl.gov/lulesh.php` and is available for download. They are released under a BSD open source license. In this report we do not go into detail about any particular port or model, but instead provide a high level overview of what is available and the differences between each of these versions. References are provided where available to allow readers access to further details. We have tried to be as comprehensive as possible based on the work we know about, but encourage submission of any additional work both in code versions and publications.

The rest of the document is organized as follows. Section 2 contains an overview of the serial reference version of LULESH. Then Section 3 describes programming models other than C++ that LULESH is implemented in. Section 4 details various tuned versions of LULESH including a brief description of the optimizations that have been applied, and limits on applying them, to ensure applicability back to the real application. Next Section 5 discusses other work that adds functionality to LULESH. Finally, Section 6 describes how to measure the performance of a LULESH version and how to verify that a code is a correct implementation. Also, it details changes to LULESH that should not be made. These changes while technically correct, since they will not change the answer, are not possible to make in ALE3D the application LULESH is based on.

## 2 Serial Implementation

The serial reference implementation of the LULESH specification [1] is a mesh based physics code on an unstructured hexahedral mesh with two *centerings*. The *element centering* (at the center of each hexahedral) stores thermodynamic variables, such as energy and pressure. The *nodal centering* (where the corners of hexahedrals intersect) stores kinematics values, such as positions and velocities.

The actual 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 a time stepping algorithm followed by a time constraint calculation. The 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 intense 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 found 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 algorithm involves advancing the element quantities. To advance element quantities first kinematic values are calculated for the elements based on the new nodal positions and velocities. Next artificial viscosities are calculated in two phases with element based values computed first followed by the region based values. 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 algorithm at the next time step.

The last phase of LULESH is calculating the maximum timestep that can be taken to advance the simulation further. The time constraints are used to limit how far in time the simulation advances at the next time step. The time step is limited by two functions, which are each only applied to elements whose volume are changing. 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 timestep. The simulation can be run without any constraints if a small enough value is chosen as a fixed time step.

## 3 Programming Model Ports

LULESH has been implemented in a wide variety of programming models. In this section, we describe briefly each of the ports with an emphasis placed on implementation differences from the serial implementation and the other models. A brief classification of the less common models is presented, but details are not covered. For more information about most of these ports along with a look at the programmability, optimizability and performance portability of the LULESH implementations in these languages please see [10].

### 3.1 MPI

The MPI [12] implementation of LULESH adds communication of ghost fields in two places. After the force calculation the *fx*, *fy* and *fz* force values are communicated, so neighboring elements have the same values at the nodes. Then a second exchange of *delv_xi*, *delv_eta* and *delv_zeta* 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.

### 3.2 OpenMP

The OpenMP [4] 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. The base OpenMP version used OpenMP critical sections to protect this potential data race. Section 5 contains other versions of LULESH that handle these race conditions differently. However, the OpenMP version described here has been the best performing on all architectures tried so far.

### 3.3 C Version

A pure C version of the OpenMP version of the code was created to as a starting point for an OpenACC port. The only changes to get to this version are removing C++ constructs and other minor syntactic changes. The OpenACC port is in progress and will be released when completed.

### 3.4 MPI plus OpenMP

The hybrid MPI plus OpenMP LULESH implementation uses MPI processes for between node communication, and OpenMP to spawn threads within a process. OpenMP directives are put on the same loops as the pure OpenMP code to obtain strong scaling within an MPI process. The reduction of the hydro and courant constraints are made 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.

### 3.5 CUDA

Mapping LULESH to CUDA is similar to mapping the code to OpenMP. 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. Also, a reduction of the courant and hydro constraints on a node are needed. The same solution as used in the OpenMP codes are applied in CUDA. Some optimizations specific to the Fermi based GPU are contained in this version and the reader is referred to [10] for more details.

### 3.6 A++

A++ is an array language implemented in C++ as a class library. A++ provides arrays as objects (data), and simple array operations (functions that operate on array objects). More information on the language can be found in several online sources. The A++ version of LULESH was initially created to explore a wide range of options for manipulating data layouts via array notation. This version of the code is purely sequential, but the extensive use of array operations exposes significant amount of potential concurrency. The use of array operations reduced overall code length by about 35%. This version avoids the use of global array accesses; all data is passed by parameter to all functions. As a result, the code can be useful for more quickly understanding the algorithms and data dependences within LULESH. On the other hand, no effort has been expended to get the execution performance of the A++ LULESH to levels comparable to other versions.

### 3.7 Chapel

Chapel uses a block-imperative syntax with optional support for object-oriented programming, type inference, and other productivity-oriented features to support both task- and data-parallel styles of programming [3]. 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 are 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 Nodes and Elements domains. Over the course of the code's evolution, it was changed from using regular 3D domains to an irregular 1D domain representation, and from using dense representations of the material elements to sparse ones. Work on improving the benchmark source is under way. Also, the Chapel version of LULESH is one of the applications being used to assess and prioritize general improvements to the Chapel compiler.

### 3.8 Charm++

Charm++ is a parallel programming system that implements additions to C++ coupled with an adaptive runtime system based on message-driven migrateable objects [8]. The Charm++ port leaves the original C++ code that implements the physics computations of LULESH unchanged. The changes to LULESH involved expressing the domain level information in terms of *chares*, which each store a portion of a domain. The chares discretize the problem into sub-domains, with appropriate ghost regions allowing over decomposition of the domain. The Charm++ version of the port includes the same three communication patterns as the MPI implementation. Therefore, the only difference from the MPI, other than the over decomposition is that the execution flow of the program is asynchronous and data driven in Charm++.

### 3.9 Liszt

Liszt is a Scala-based domain-specific language for solving partial-differential equations on meshes [5]. 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. 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 both C++ code and CUDA can be generated to target both GPUs and CPUs.

### 3.10 Loci

Loci is a C++ framework that implements a declarative logic-relational programming model that is implicitly parallel and uses relational abstractions to describe distributed irregular data structures [11]. The Loci implementation uses built-in 3D vector data-types to simplify the algorithm description. Most of the work is offloaded to the compiler/runtime. The Loci preprocessor generates the loop over entities and the run-time 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. Finally, Loci schedules communication such that the reductions are applied consistently in parallel.

### 3.11 In Progress/Planned

An OpenCL port is currently being debugged. An initial implementation in OpenACC has been started, but is not yet complete. Once these are complete more information will be added about them.

## 4 Tuned LULESH Versions

In exploring how to improve the performance of LULESH on various computing platforms we have created tuned versions of LULESH. The code versions described here were created to learn various lessons or to try out different techniques. They are not meant to be exhaustive or systematic in the scope they cover. In this section, we first describe the optimizations that have been applied to LULESH and then describe the naming convention, or how to identify what optimizations are in each code version on the website. We then highlight the best known version on various machines for which, we have LULESH test data.

### 4.1 Performance Optimizations

In this section we list the various performance tuning optimizations we have applied to LULESH. More detail on how they were applied to LULESH and their impact on performance can be found in [9].

#### 4.1.1 Loop Fusion and Array Contraction

Loop fusion is an optimization that involves combining multiple loops over the same iteration space into a single loop. When loops are fused arrays that only store data produced in the first loop that is consumed in the second loop can be contracted to scalars [6]. In the OpenMP port fused versions of LULESH take the 45 loops that occur at each timestep and reduce them to 12 loops; however, other programming models may have slightly different loop counts. Fusion also reduces the number of OpenMP parallel regions to 12 from 30. When fusing these loops, the fact that LULESH is a proxy application meant to mimic ALE3D must be taken into account. The loops fused in the code on the website represent the most aggressive fusion possible in the real code with two exceptions. First, the courant and hydro calculations can be fused, but were not because of their relatively small runtime. Second, the boundary condition loops can be fused with the acceleration update and nodal position and velocity update calculation. However, this may require extra compute or data motion since the boundary condition loop is only over part of the domain.

We also reorganized the code where data validity checks occurred. We moved four conditional statements closer to the location where the data were initially calculated. By doing this we removed redundant checks and also reduced data motion. These changes are included in all loop fused versions.

#### 4.1.2 Global Allocation

By global allocation we mean all temporary variables are allocated outside of the main timestep loop. This optimization reduces the runtime of the serial sections of LULESH where mallocs and frees occur. Also, on systems that use lazy page allocation, runtimes are decreased due to not having to setup a virtual to physical page mapping every timestep. Another way to a generate similar, but not quite as good performance gains, on some systems is to use large pages or transparent huge pages. Also, large page sizes and tcmalloc from the google perf tools library [7] provided similar performance gains. However, the impacts of page fragmentation with these tools on more complex codes has not been studied.

#### 4.1.3 Increased Vectorization

We achieved increased vectorization in two ways: re-rolling loops and more aggressive techniques. By re-rolling loops more information was provided to some compilers. Therefore, they were able to vectorize loops that had been previously unrolled. This technique has been applied to six loops in *CalcElemFBHourglassForce* and one loop in *SumElemStressesToNodeForces*. The second method used to increase vectorization was significantly more invasive and is detailed in [9]. It results in up to 87% of all flops being vectorized, but is dependent on the compiler to vectorize the code.

#### 4.1.4 NUMA Aware Data Allocation

The Mcsup library [2] has been used to perform NUMA aware data allocations. It works by trapping all allocations of memory and making sure data is in the same NUMA domain as the task that accesses it. However, Mcsup is not released software; so, we do not provide any NUMA aware versions at this time. However, when the data in LULESH is allocated in a NUMA aware manner performance should scale linearly.

### 4.2 Naming Convention Overview

| Name | Meaning |
|---|---|
| LULESH | The file implements the LULESH algorithm. |
| MPI | This code is a MPI implementation. |
| OPENMP | This code is an OpenMP implementation. |
| ¡Model Name¿ | This code is implemented in the ¡Model Name¿ programming model. |
| FUSE | Loop fusion, array contraction optimizations and if checks described in Section 4.1.1 are performed. |
| ALLOC | Data allocation optimizations to remove all mallocs and frees from the timestep loop as described in Section 4.1.2 are performed. |
| VECLOOP | Vectorization by creating loops from previously unrolled code as described in Section 4.1.3 is performed. |
| VECAGG | Vectorization performing aggressive optimizations as described in Section 4.1.3 is performed. |

**Tab. 1. LULESH file naming convention and meaning.**

The performance versions on the website are named using the names and meanings in Table 1. All codes use a single file when possible for simplicity. Files are named such that all optimizations and programming models included in an implementation are concatenated together with an underscore separating each descriptor. For example, LULESH_MPI_OMP_ALLOC.cc is a C++ implementation containing hybrid OMP and MPI parallelism along with the data allocation optimizations.

### 4.3 Best Versions

In this section we highlight the best performing OpenMP codes for various platforms on which we have run extensive tests.

#### 4.3.1 Lawrence Livermore Linux Clusters

On all the Linux clusters we have tried at Lawrence Livermore the version LULESH_OMP_FUSE_ALLOC_VECAGG has performed the best when compiled with icc version 12.1.339 or newer. Platforms tried include an AMD Barcelona system, and Intel Nehalam, Westmere and Sandybridge systems. For icc the best compiler options of the ones we tried are -O3 and -mavx when applicable.

#### 4.3.2 BlueGene/Q

On the BlueGene/Q architecture the LULESH_OMP_FUSE_ALLOC code is the best performer. For this platform we used version 12.0 of the xlc compiler and using the -O5 -qsimd=noauto options produced the fasted code. At runtime the *OMP_WAIT_POLICY=ACTIVE* and *BG_SMP_FAST_WAKEUP=YES* flags were set.

#### 4.3.3 Intel Xeon Phi

For the Intel Xeon Phi the LULESH_OMP_FUSE_ALLOC_VECAGG has shown the best performance. We ran this code using *KMP_AFFINITY=compact* and one less core than is on the card. It was compiled using icc 13.0.0.79 with the -O3 -mmic options.

## 5 Other LULESH Versions

In this section we describe other LULESH versions that are available including how they differ and include features from the previous sections.

### 5.1 Fully Unstructured

The original LULESH implementations use a block structured grid with an unstructured access pattern. This version changes the mesh such that it is fully un-structured. The mesh is read in from a file and one example is included with this version.

### 5.2 Vista Database

The original LULESH implementations have hard coded equation of state (EOS) data. In this version instead of hard coded EOS, data lookups, are performed to a Vista database. The database is a simplified version of the one found in ALE3D and allows the proxy-app to capture another characteristic of the application it mimics.

### 5.3 One Dimensional Communication Pattern

The one dimensional communication pattern code changes how data are transferred in the hybrid MPI plus OpenMP version of LULESH. Instead of having twenty-six neighbors to communicate to each MPI process has only two. For massively multi-threaded hardware, such as the Intel MIC and GPU, this communication pattern can be beneficial.

### 5.4 Clean Code

This version of LULESH is rewritten to remove all global data structures and to pass data by reference. Also, the loop fusion, data allocation and loop rolled up vector optimizations are applied to it. The goal is to create a code that expresses as much information as possible to the compiler and increase readability. It outperforms the best version on both BlueGene/Q and Linux clusters when similar options are used.

### 5.5 Transactional Memory

The transactional memory version of LULESH builds off the OpenMP implementation. In it the extra data motion to store all eight values to be summed to the corners of a node and perform the summation in the hourglass and stress routines are removed. Instead the summations are performed as they are in the serial version with the twenty-four summations to nodes protected within a single transaction region. Currently only the BlueGene/Q hardware supports transactions and on this machine using transactional memory is significantly slower than using extra data motion to prevent races. This finding is true even in cases where there are no rollbacks.

### 5.6 Critical Sections

Like the transactional memory version this code builds from the OpenMP implementation. However, instead of using transactions to prevent data races it uses the `omp critical` directive. Tests show this is the slowest way to handle data races on Livermore machines.

### 5.7 Fault Tolerant Checkpoint

With the fault-tolerant version, variables and objects identified as persistent are transparently saved at checkpoint time [13]. Global variables in an application can be marked persistent by using a compiler attribute defined as PERM. The checkpoint method uses a persistent memory allocator that replaces the standard dynamic memory allocation functions with compatible versions that provide persistent memory to application programs. Memory allocated with this allocator will persist between program invocations after a call to a checkpoint function. This function essentially saves the state of the heap and registered global variables to a file which may reside in flash memory or other node local storage. Since the checkpoint data consists of an image file per node, it can be used by other inter-node or global checkpoint schemes such as Scalable Checkpoint and Restart (SCR). A second fault-tolerant version was implemented to demonstrate the use of SCR in conjunction with the persistent allocator. In this case, nodes query SCR for a new filename before writing the persistent memory image to the indicated SCR file. After closing a file, SCR manages its replication and distribution.

### 5.8 Planned Versions

A multi-region version of LULESH is planned. This version will mimic the control flow and access patterns used by a multi-material code, but still perform the same one material calculation.

## 6 Performance Measurement and Correctness Checking

In this section we describe how to measure the performance of LULESH. We also detail how to determine if an implementation is correct. Finally, we list a set of changes that should not be made to LULESH.

### 6.1 Performance Measurement

Performance measurements of LULESH should ignore problem setup time and only measure the time to timestep to solution because for large simulations setup will be a small part of the overall runtime. Also, the setup in LULESH is not representative of the file IO and setup phase of ALE3D. The performance of LULESH can be expressed in two meaningful ways both of which are output by the versions on the website. Wall clock time gives how long a version takes to perform the simulation. Grind time gives the time it takes to update a single zone for one iteration of the timestep loop.

When measuring the performance there are two options. One can use a fixed timestep and not use the courant and hydro constraints. Alternatively, a dynamic timestep can be used and the courant and hydro constraints timed.

### 6.2 Correctness Checking

| Size | Iteration Count | Final Origin Energy |
|---|---|---|
| $5^3$ | 306 | 1.670602e+05 |
| $10^3$ | 593 | 5.174622e+05 |
| $45^3$ | 1495 | 4.261463e+05 |
| $50^3$ | 1566 | 4.052502e+05 |
| $70^3$ | 1816 | 3.453062e+05 |
| $90^3$ | 2026 | 3.063249e+05 |

**Tab. 2. Common on node sizes used for correctness checking and timing studies of LULESH**

There is no method to establish the exact correctness of a LULESH port. However, the versions

```
printf("   Iteration count     = %i \n",    domain.cycle());
printf("   Final Origin Energy = %12.6e \n", domain.e(0));

Real_t    MaxAbsDiff = Real_t(0.0);
Real_t TotalAbsDiff = Real_t(0.0);
Real_t    MaxRelDiff = Real_t(0.0);

for (Index_t j=0; j<edgeElems; ++j) {
   for (Index_t k=j+1; k<edgeElems; ++k) {
      Real_t AbsDiff = FABS(domain.e(j*edgeElems+k)
                       - domain.e(k*edgeElems+j));
      TotalAbsDiff  += AbsDiff;
      if (MaxAbsDiff <AbsDiff) MaxAbsDiff = AbsDiff;
      Real_t RelDiff = AbsDiff / domain.e(k*edgeElems+j);
      if (MaxRelDiff <RelDiff)  MaxRelDiff = RelDiff;
   }
}

printf("   Testing Plane 0 of Energy Array:\n");
printf("        MaxAbsDiff   = %12.6e\n",   MaxAbsDiff   );
printf("        TotalAbsDiff = %12.6e\n",   TotalAbsDiff );
printf("        MaxRelDiff   = %12.6e\n\n", MaxRelDiff   );
```

**Fig. 1. C code to perform correctness checking of LULESH.**

on the website were tested using the C code in Figure 1 that calculates and prints the following three metrics. The iteration count of the implementation should match exactly. The energy at the origin when the simulation completes should be correct to at least six digits. The three measures of symmetry should all be $10^{-8}$ or smaller in double precision for $100^3$ mesh discretizations of smaller. While errors may still exist in your version of LULESH even if all these tests are passed we have found that they have caught innumerable major and minor bugs. As a reference we provide the final origin energy and number of iterations for sizes used to validate many versions in Table 2. The small sizes are helpful to quickly turnaround debugging runs and the larger ones for final validation.

### 6.3 Changes not to make to LULESH

When creating LULESH, certain simplifications were made in its structure and design to generate a single code with a serial version of less than 3000 lines and no external dependencies. Therefore, certain parts of LULESH can be changed to improve performance or ease implementation. However, making those changes will result in LULESH no longer being representative of ALE3D. In this section we list several changes, but this list may not be exhaustive.

#### 6.3.1 Mesh Representation

LULESH uses a block structured mesh accessed via an indirect reference pattern. Therefore, one can write a correct version of LULESH without any of the indirection arrays, increasing performance and code simplicity. However, ALE3D is a fully unstructured application so accessing LULESH's arrays in this manner would oversimplify LULESH.

#### 6.3.2 Loop Structure

In Section 4 we mentioned the loops we fused and did not fuse when tuning LULESH. There are two reasons why loops in LULESH should not be fused. The first is multi-material loops. Since LULESH solves the sedov problem on a single material it does not have loops over materials. However, in ALE3D there are loops over materials that make fusing neighboring loops impractical. The multi-region version of LULESH will contain the added loops to show where multi-material loops would go if we were simulating a multi-material problem in LULESH. The second reason for not fusing loops is code complexity and maintainability. While the stress computation in LULESH can be fused with the hourglass computation, when there are multiple options for both stress and hourglass routines as in ALE3D, fusing them together creates code maintenance issues.

#### 6.3.3 Extra Compute

In the stress calculation of LULESH there is extra computational work performed. While not necessary for solving the sedov problem this compute was put in to model other computation in ALE3D that would not otherwise be represented when solving the sedov problem in LULESH. It also serves as a test of the compiler's ability to optimize code, which is important when using LULESH as a benchmark. The compute is found in the first call to the shape derivative calculation.

## 7 Acknowledgments

The versions of LULESH documented here were developed by many people. Jeff Keasler and Rich Horuung are responsible for the original serial and MPI versions. Jeff also created the Vista View, OpenMP, Hybrid MPI plus OpenMP, fully unstructured and one dimensional communication codes. Jon Cohen developed the CUDA version of the code. The A++ and clean code versions were created by Jim McGraw. The Chapel port was performed by the Chapel team. The Loci port was developed at Mississippi St. Felix Wang wrote the Charm++ port and Riyaz Haque developed the Liszt port. Maya Gokhale, Scott Lloyd and Daniel Wong wrote the fault tolerant version. The performance tuning work was done by Ian Karlin and Jim McGraw. Jeff Keasler and Ian Karlin created the pure C version of the code. Ian Karlin is responsible for the transactional memory and critical section codes.

Funding for this work came from the following sources. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0001. This work partially supported by the DOE Office of Advanced Scientific Computing Research. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

## References

[1] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[2] A.H. Baker, T. Gamblin, M. Schulz, and U.M. Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 275 –286, may 2011.

[3] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[4] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46 –55, jan-mar 1998.

[5] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat 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*, SC '11. ACM, 2011.

[6] 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*, pages 281–295, New Haven, CT, Aug. 2004.

[7] Sanjay Ghemawat. Tcmalloc: Thread-caching malloc, 2012. http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[8] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[9] I. Karlin, J. Mcgraw, E. Gallarado, J. Keasler, E. A. Leon, and B. Still. Memory and parallelism tuning exploration using the lulesh proxy application, November 2012.

[10] 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 Still. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[11] 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*, 15(03):477–502, 2005.

[12] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[13] Daniel Wong and Maya Gokhale. A memory-mapped approach to checkpointing. Technical Report LLNL-CONF-499091-DRAFT, 2011.