Hydrodynamics Challenge Problem
Lawrence Livermore National Laboratory

LLNL-TR-490254

Contents

Chapter 1

Hydrodynamics Challenge Problem

The hydrodynamics challenge problem represents a classical HPC physics problem, namely high deformation event modeling via Lagrangian shock hydrodynamics. This challenge problem solves the Sedov blast wave problem for one material in three dimensions. The problem has an analytic solution, and can be scaled to arbitrarily large problem sizes. The reference code is drawn from a production LLNL hydrodynamics code.

1.1 Introduction

1.1.1 Motivation for Hydrocodes for Defense Applications

Computer simulations of a wide variety of science and engineering problems require modeling hydrodynamics, which describes the motion of materials relative to each other when subject to forces. Herein, we refer to computer codes that solve the equations of hydrodynamics as *hydrocodes*.

Many important DoD simulation problems involve complex multi-material systems that undergo large deformations. Examples include armor defense, penetration mechanics, blast effects, structural integrity, and conventional munitions such as shaped charges and explosively formed projectiles. Indeed, the original motivation to develop hydrocodes was to solve problems with defense applications.

1.1.1.1 Hydrocodes in DoD HPC Workloads

The FY2010 Requirements Analysis Report issued by the DoD High Performance Computing Modernization Program (HPCMP) Office shows that a major portion of DoD HPC activities involves hydrocodes [5]. The report surveyed 496 projects across the Services and various Agencies, representing 4,050 HPCMP users at more than 125 locations, including government, contractors, and academia. Each project was grouped into one of ten categories. The Computational Fluid Dynamics (CFD) category accounted for the most projects (37% of the total) and the most users (27% of the total). The Computational Structural Mechanics (CSM) category was fourth with about 10% of total users. According to the report, hydrocodes are among the most used of all applications with several ranked in the top ten in terms of number of users. In addition, *of all* non-real time applications, four DOE hydrocodes (CTH, ALE3D, Sierra, and Alegra) are ranked in the top ten in terms of CPU hours.

1.1.1.2 Hydrocodes on UHPC Architectures

Numerical algorithms used in hydrocodes present computational issues not found in other challenge problems. A typical hydrocode approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. A node on the mesh is a point where mesh lines intersect. Finite difference equations that approximate differential

operators in the equations couple variables on the mesh (e.g, at nodes and elements) via stencil operations. Other computations, involving material properties and equation of state, are interleaved with the stencil operations. The operations must be performed in a specific order for numerical accuracy and computational robustness.

Due to these computational characteristics, plus the importance of hydrocodes to DoD HPC efforts, a hydrodynamics challenge problem is an important inclusion in a code suite used to evaluate UHPC architectures.

UHPC "petaflop in a rack" architectures that can efficiently solve the hydro challenge problem will create a new capability for simulation-coupled sensor processing. An increase in processing power in the field could enable simulation-in-the-loop for real-time algorithms to evaluate tactical threats and deliver appropriate responses for threat elimination. An LLNL/DARPA *imaging* example [8] of this capability includes simulation to improve the accuracy of radar propagation in buildings, as explored in the DARPA VisiBuilding program.

1.1.2 Hydrodynamics Methods

There are two alternative specifications of the governing equations for hydrodynamics. While mathematically equivalent, the formulations naturally lead to different numerical solution algorithms. The formulations can be found in any book on continuum mechanics; e.g., see [2]. In the Eulerian frame of reference, physical variables such as density, pressure, and velocity, are defined as functions at fixed spatial positions over time. Thus, the Eulerian form of the equations can be thought of as describing the spatial distribution of the flow variables at each instant in time. The Lagrangian formulation exploits the fact that values of some physical variables refer to identifiable pieces of matter at certain positions in space, a view similar to particle mechanics. In the Lagrangian formulation, the flow variables are defined as functions of time and particular material elements and describe the dynamical history of those elements. Both formulations are used in hydrocodes and have advantages and disadvantages for various applications.

Eulerian hydrocodes, such as CTH, typically employ an orthogonal mesh for accuracy of the numerical approximation. Materials flow through the mesh, which is fixed in time and space, as a simulation progresses. Eulerian codes are particularly useful for problems that exhibit strong shearing and vortical motion such as that found in turbulent flows. However, moving material boundaries and interactions among materials are less natural to express with Eulerian methods than with Lagrangian methods. For example, each material in a multi-material mesh element may be represented as a fraction of element volume. Material interfaces are not directly represented and thus tend to diffuse in a non-physical manner when not aligned with the mesh. Without additional numerical machinery to construct material interfaces, Eulerian methods require very fine mesh resolution for good spatial accuracy. The left-hand image in Fig. 1.1 illustrates how a fixed mesh, Eulerian code may not resolve a material interface accurately.

In Lagrangian hydrocodes, the initial mesh configuration partitions the problem domain into material elements and element boundaries are constructed to align with material interfaces. As a simulation evolves, the mesh follows the motion of these elements through space and time. Lagrangian methods handle moving boundaries and multiple materials naturally and can provide a highly accurate solution for many problems without requiring an excessively fine mesh. The right-hand image in Fig. 1.1 illustrates this. However, when the flow involves sufficiently complex structure (e.g., strong shearing or vorticity), Lagrangian methods can perform poorly as mesh elements distort and possibly tangle.

ALE (Arbitrary Lagrangian Eulerian) codes (such as ALE3D [1]) have been developed to seek a

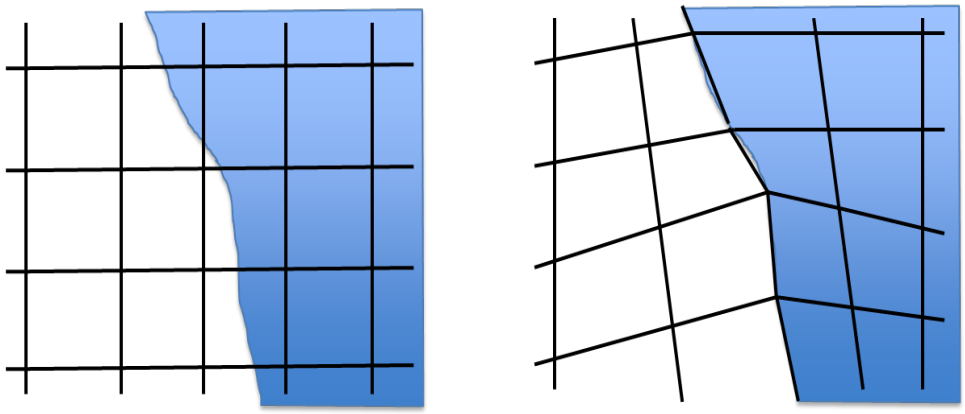

Figure 1.1. The accuracy of a simulation involving a boundary between two materials (here shown in white and blue) usually depends on how well the mesh can represent the boundary. The left image shows a fixed Eulerian mesh and a material interface that does not align with mesh element boundaries. Since the interface is not represented explicitly, its position must be inferred from material volume fractions in the elements. The right image shows how Lagrangian mesh nodes follow the motion of the (same) material interface directly, thus representing it much more accurately.

compromise between the Eulerian and Lagrangian formulations. ALE methods can accurately solve problems involving moving boundaries, multiple materials, and strong shearing and vortical flow regions. The general strategy is to evolve the problem using the Lagrangian algorithm until the mesh reaches a level of distortion such that continuing in this fashion is problematic. At this point, the mesh is *relaxed* to a more numerically-desirable configuration. Then, the simulation variables are mapped to the new mesh and the simulation continues.

1.2 Lagrangian Hydrodynamics: A Closer Look

While integration schemes and numerical algorithms employed in hydrocodes vary, most codes posses similar computational characteristics and data access patterns. In the interest of algorithm simplicity and smaller code size, while capturing essential features of production hydrocodes, we have chosen a Lagrangian hydrodynamics methodology for our challenge problem.

1.2.1 Governing Equations

The inviscid compressible hydrodynamics equations represent the conservation of mass, momentum, and energy [2]. In the Lagrangian description, the differential equations are:

$$\frac{D\rho}{Dt} = -\rho \overrightarrow{\nabla} \cdot \overrightarrow{\mathbf{U}} \tag{1.1}$$

$$\rho \frac{D\overrightarrow{\mathbf{U}}}{Dt} = \overrightarrow{\nabla} \cdot \boldsymbol{\Sigma} \tag{1.2}$$

$$\frac{De}{Dt} = \frac{1}{\rho} Tr(\boldsymbol{\epsilon}_{tot} \cdot \boldsymbol{\Sigma}) = -p \frac{DV_{spec}}{Dt} + V_{spec} Tr(\boldsymbol{\epsilon} \cdot \mathbf{S}) \tag{1.3}$$

The variables on the left hand-side of the equations are density ρ, the velocity vector $\overrightarrow{\mathbf{U}}$, and the internal energy e. The total strain rate tensor is:

$$(\boldsymbol{\epsilon}_{tot})_{ij} = \frac{1}{2}\left(\frac{\partial U_i}{\partial X_j} + \frac{\partial U_j}{\partial X_i}\right) \tag{1.4}$$

where $\overrightarrow{\mathbf{X}} = (X_1, X_2, X_3) = (x, y, z)$ is the spatial position vector and $\overrightarrow{\mathbf{U}} = (U_1, U_2, U_3) = (U_x, U_y, U_z)$. The total stress tensor is

$$\boldsymbol{\Sigma}_{ij} = \mathbf{S}_{ij} - p\delta_{ij} \tag{1.5}$$

where p is the isotropic pressure:

$$p = -\frac{1}{3}Tr(\boldsymbol{\Sigma}) \tag{1.6}$$

and δ_{ij} is the *Dirac delta* tensor. The tensor $\mathbf{S}$ contains the stress deviator terms $\mathbf{S}_{ij} = \boldsymbol{\Sigma}_{ij} + p\delta_{ij}$. The variable V_{spec} is the specific volume: $V_{spec} = \frac{1}{\rho}$. The tensor $\boldsymbol{\epsilon}$ is the deviatoric strain tensor:

$$\boldsymbol{\epsilon} = (\boldsymbol{\epsilon}_{tot}) - \frac{1}{3}\overrightarrow{\nabla} \cdot \overrightarrow{\mathbf{U}} \tag{1.7}$$

Here, the Lagrange time derivative (or material derivative) is a total derivative moving with the flow field:

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \overrightarrow{\mathbf{U}} \cdot \overrightarrow{\nabla} \tag{1.8}$$

The pressure is usually determined by an equation of state that gives it as a function of density and internal energy: $p = EOS(\rho, e)$. The stress deviator terms are usually determined by some (material) constitutive relations.

In the interest of simplicity, our challenge problem code solves the Euler equations describing a single material and assume an inviscid approximation of the stress tensor; i.e., no shearing stresses $\mathbf{S}_{ij} = 0$. The resulting equations are:

$$\frac{D\rho}{Dt} = -\rho\overrightarrow{\nabla} \cdot \overrightarrow{\mathbf{U}} \tag{1.9}$$

$$\rho\frac{D\overrightarrow{\mathbf{U}}}{Dt} = -\overrightarrow{\nabla} \cdot p \tag{1.10}$$

$$\frac{De}{Dt} = -\frac{p}{\rho}\overrightarrow{\nabla} \cdot \overrightarrow{\mathbf{U}} = -p\frac{DV_{spec}}{Dt} \tag{1.11}$$

1.2.2 Mesh Quantities

To simulate on a computer a physical problem described by the partial differential equations in Section 1.2.1, it is common to define a discrete *mesh* which covers the physical region of interest. The mesh partitions the region into a collection of disjoint *elements*. In our case, the mesh defines each element to be a potentially distorted hexahedron in three-dimensional space.

The equations are solved using a *staggered mesh* approximation [9]. That is, thermodynamic variables such as ρ, e, and p are approximated as piece-wise constant functions within each element. This is known as *single-point quadrature*; the value of the function in an element is represented at the

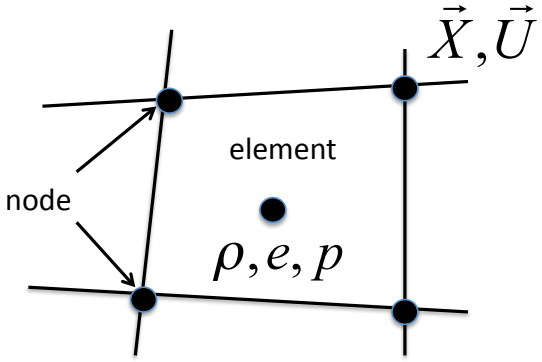

Figure 1.2. Variables on a staggered mesh. Thermodynamic variables are represented at element centers. Kinematic variables are represented at nodes. The figure shows a two-dimensional mesh for simplicity; a three-dimensional mesh representation is the obvious extension.

point at the element center. Kinematic variables such as $\vec{\mathbf{X}}$ and $\vec{\mathbf{U}}$ are defined at the element nodes. The single-point quadrature mesh elements used in the challenge problem implementation, while less accurate than alternatives, have a long history of demonstrated robustness for modeling realistic problems involving plastic flow and shock discontinuities. The spatial relationships among these variables are illustrated in Fig. 1.2. Spatial gradients are computed using finite element approximations. The reference code (see Section 1.3) uses specific computational operations to perform the finite element approximations that are employed in production hydrocodes.

1.2.3 Numerical Time Integration

After setting the initial values of the solution variables on the mesh and defining appropriate boundary conditions, the solution evolves by integrating the equations in time. As is common in hydrodynamics simulations, the challenge problem implementation uses an explicit time stepping algorithm to advance the solution through a sequence of discrete time increments. That is, the solution at time t^n is advanced to time $t^{n+1} = t^n + \Delta t^n$, where n is the step number and $\Delta t^n = t^{n+1} - t^n$ is the time increment.

An accurate and robust Lagrangian time integration algorithm requires that several issues be addressed: time increment selection, artificial viscosity, and an hourglass filter. Production Lagrangian hydrocodes treat these concerns in various ways all of which adds to algorithm and code complexity. The inclusion of these mechanisms in our challenge problem code allows us to maintain essential features of production hydrocodes.

The Courant-Friedrichs-Lewy (CFL) condition determines the maximum size of each time increment based on the shortest distance across any mesh element and the sound speed of the material in the element [4]. The stability condition insures that the simulation does not propagate information faster in the numerical approximation than is dictated by the governing equations. Since the same time increment is used to advance the solution over the entire mesh, the determination of the maximum allowable increment usually requires a collective communication operation.

To model the entropy-conserving properties of the governing equations properly, the discrete equations must be augmented with a dissipation mechanism. In reality, physical viscosity has a dissipation length scale of a few molecular mean free paths which cannot be represented at the length scale of

typical mesh elements. Nevertheless, the artificial viscosity mechanismis employed in hydrocodes usually were developed using the exemplar of real physical viscosity. The artificial viscosity employed in our reference code is based on the model developed by LLNL scientist Christensen [3], which is a major advance in computational hydrodynamics algorithms.

A consequence of single-point quadrature is that the stress is contant within each element which produces spurious energy modes which are not resisted by the element stress. These zero energy modes are called "hourglass" modes due to their shapes. They can produce physically erroneous response which may be unstable. To remedy this, the reference implementation uses the Flanagan-Belytschko kinematic hourglass filter [6]. This approach, used commonly in Lagrange finite element hydrocodes, defines a stiffness force within each element that damps the unwanted modes.

Before summarizing the Lagrange time integration step, we note that, due to the use of an artificial viscosity, the reference implementation actually solves modified forms of the momentum and energy equations described in Section 1.2.1 in our code. For the Euler equations, these equations become:

$$\rho \frac{D\overrightarrow{\mathbf{U}}}{Dt} = -\overrightarrow{\nabla} \cdot (p+q) \tag{1.12}$$

$$\frac{De}{Dt} = -(p+q)\frac{DV_{spec}}{Dt} \tag{1.13}$$

Here, q is the artificial viscosity which acts like a pressure term.

Finally, note that the Lagrangian integration approach implies that the mass in each element is constant in time. Thus, the equation for mass conservation can be rewritten as

$$\rho V = \rho_0 \tag{1.14}$$

where V is a relative volume (i.e., element volume divided by initial element volume $\mathbf{V}_0$) and ρ_0 is a reference density (i.e., element mass (constant) divided by $\mathbf{V}_0$). The reference code uses this alternative form of the mass conservation equation; in particular, V is evolved in time rather than ρ. Eq. 1.14 shows how to determine ρ in an element from V when needed.

1.2.4 Summary of the Lagrange Time Step

Each discrete Lagrange time step advances the solution variables $(V^n, p^n, e^n, \overrightarrow{\mathbf{X}}^n, \overrightarrow{\mathbf{U}}^n)$ at the current time t^n to their values $(V^{n+1}, p^{n+1}, e^{n+1}, \overrightarrow{\mathbf{X}}^{n+1}, \overrightarrow{\mathbf{U}}^{n+1})$ at the new time t^{n+1}, where $\Delta t^n = t^{n+1} - t^n$ is the time increment. In this process, the reference code first advances variables at mesh nodes, then updates element variables based on the new node variable values. The steps are summarized as follows:

1. Calculate the time increment Δt^n.
2. Construct a force at each mesh node. This involves integrating p^n and q^n over a control volume at each node, and calculating the hourglass filter contribution from each element to each of its surrounding nodes. The result is a force vector $\overrightarrow{\mathbf{F}}$ at each node on the mesh.
3. Compute the acceleration at each mesh node. Computing the acceleration uses the force at the node and Newton's Second Law of Motion: $\overrightarrow{\mathbf{F}} = m_0 \overrightarrow{\mathbf{A}}$. Here, m_0 is the mass at a node; for our problem, this mass is constant in time since the mass in each element surrounding a node remains constant. In this step, it may also be necessary to impose boundary conditions on the acceleration.

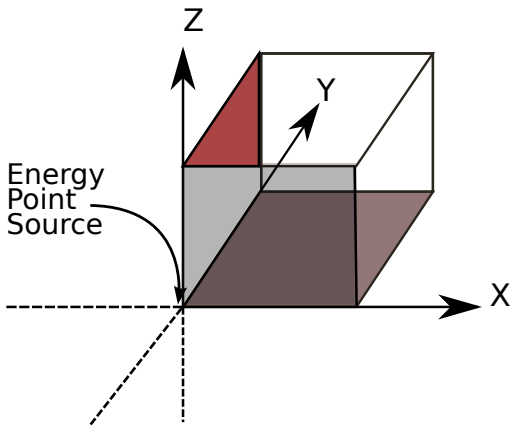

Figure 1.3. Our challenge problem code solves the Sedov problem on a three- dimensional parallelepiped domain, one corner of which lies at the origin. The box in the figure is the computational region which includes a finite portion of one octant of three-dimensional space. On the colored faces of the box, we impose symmetry boundary conditions. The other three faces are free boundaries. The origin represents a point source of energy imposed as an initial condition.

4. Compute the new velocity at each mesh node. The new velocity $\overrightarrow{\mathbf{U}}^{n+1}$ is computed by discrete integration of the acceleration: $\frac{D\overrightarrow{\mathbf{U}}}{Dt} = \overrightarrow{\mathbf{A}}$.
5. Compute the new position at each mesh node. The new position $\overrightarrow{\mathbf{X}}^{n+1}$ is computed by discrete integration of the velocity: $\frac{D\overrightarrow{\mathbf{X}}}{Dt} = \overrightarrow{\mathbf{U}}$.
6. Update various element variables based on node variables advanced to t^{n+1}. Such element variables include V^{n+1} and accessory variables used to calculate the artificial viscosity, determine timestep constraints, etc.
7. Calculate new artificial viscosity q^{n+1} in each element.
8. Apply material model properties based on material, volume, etc. in each element. This includes equation of state evaluation, which provides the new pressure p^{n+1}, and internal energy e^{n+1}.
9. Compute time constraints which will be applied when Δt^{n+1} is calculated for next step.

1.3 Reference Code Overview

1.3.1 Sedov Blast Wave Problem

Our challenge problem reference code simulates the Sedov blast wave problem [7] in three spatial dimensions. The solution is generated by solving the Euler equations described in Section 1.2.3. The problem solution is spherically-symmetric; that is, the solution is the same along each ray extending from the origin. We solve the problem in a parallelepiped representing a finite portion of one octant of three-dimensional space. The problem geometry is illustrated in Figure 1.3. The xy, xz, and yz coordinate planes are *symmetry* planes for this problem geometry. On these planes we impose symmetry boundary conditions; that is, the normal component of $\overrightarrow{\mathbf{U}}$ is always zero on each of these boundaries. On the remaining boundaries, we use *free* boundary conditions; that is, the solution at the boundary is allowed to evolve freely.

The initial conditions include a point source of energy deposited at the origin. All other variables

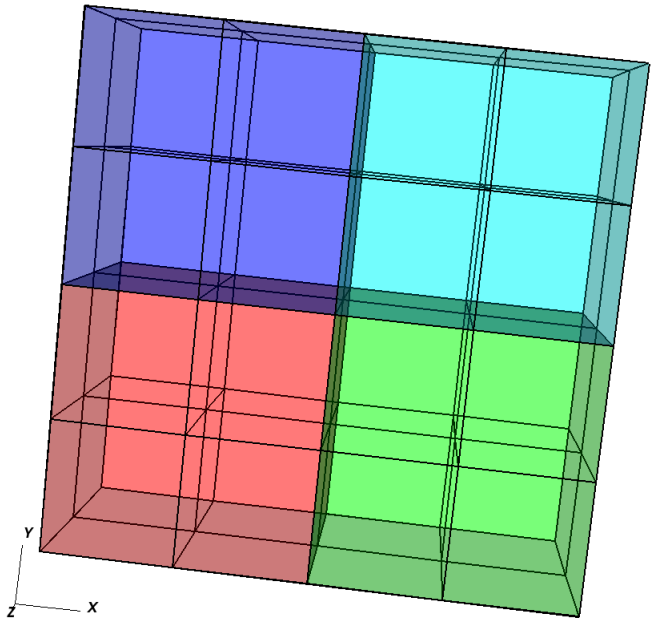

Figure 1.4. A simple Cartesian mesh on a box domain in three-dimensional space. The mesh is divided into four domains, each with eight elements. Each domain is colored differently.

apart from $\vec{\mathbf{X}}$ and V are initially set to zero. The initial values of $\vec{\mathbf{X}}$ and V are set based on the initial mesh configuration. In the box, we prescribe an uniform Cartesian mesh for the initial mesh. Recall from Section 1.2.2 that the mesh partitions the problem into a collection of hexahedral elements whose corners are the nodes of the mesh defined by the values of $\vec{\mathbf{X}}$.

Note that adjacent elements share nodes and thus the values of variables represented at those nodes. In the code implementation, the mesh is partitioned into logically-rectangular collections of elements called *domains*. Each domain serves as a *data locality context*: in the reference implementation, data for variables represented on the portion of the mesh contained in the domain are stored within a single data structure. Each domain structure holds data for all its elements and nodes surrounding those elements. Note that nodes on domain boundaries are shared by neighboring domains. If domains are mapped to processors with disjoint memory subsystems, it is often convenient to replicate the boundary data. Figure 1.4 shows a simple Cartesian mesh partitioned into 4 domains.

Thus, we have a three level hierarchy for computation, the *problem* level which is the union of all domains (i.e., the whole mesh), the domain level, and the element level. The problem may contain one or more domains, and a domain may contain one or more elements. This allows for flexible scalability from a single element problem to a very large problem with an arbitrary number of elements, where the computation is defined over the same physical volume of space. In our serial implementation the problem contains one domain. The serial implementation LULESH v1.0 is available from the CHASM web site.

1.3.2 Symbols Used in Code Description

Tables 1.1, 1.2, and 1.3 define symbols used in the description of operations in our reference code implementation. In particular, the tables relate the variables defined in our description of the governing

Variable	Description	Accessor Method(s)
$\overrightarrow{\mathbf{X}} = (x, y, z)$	position vector	`x()`, `y()`, `z()`
$\overrightarrow{\mathbf{U}} = (U_x, U_y, U_z)$	velocity vector	`xd()`, `yd()`, `zd()`
$\overrightarrow{\mathbf{A}} = (A_x, A_y, A_z)$	acceleration vector	`xdd()`, `ydd()`, `zdd()`
$\overrightarrow{\mathbf{F}} = (F_x, F_y, F_z)$	force vector	`fx()`, `fy()`, `fz()`
m_0	nodal mass	`nodalMass()`

Table 1.1. Node Variables

Variable	Description	Accessor Method
p	pressure	`p()`
e	internal energy	`e()`
q	artificial viscosity	`q()`
V	relative volume	`v()`
$\dot{V}/V$	relative volume change per volume	`vdov()`
$\Delta V = \mathbf{V}^{n+1} - \mathbf{V}^n$	relative volume change	`delv()`
$\mathbf{V}_0$	reference (initial) volume	`volo()`
l_{char}	characteristic length	`arealg()`
$\boldsymbol{\epsilon} = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz})$	diagonal terms of deviatoric strain	`dxx()`, `dyy()`, `dzz()`
q_{lin}	artificial viscosity linear term	`ql()`
q_{quad}	artificial viscosity quadratic term	`qq()`

Table 1.2. Element Variables

equations (Section 1.2.3) to routines used in the code to access them. The first column in each table lists the variable symbols, the second column describes those variable, and the third column indicates the mesh class method used to access the variables on a domain.

Many operations in the code involve loops over the nodes or elements of the domain. In production hydro codes, the collection of variables referenced in a *single* element calculation must be passed through many levels of function calls. Thus as good software engineering practice, it is common to *gather* those variables into a set of local data structures. We follow this practice in the reference code; computational blocks use the local data structures rather than the domain-level data. When the operations are complete, we *scatter* the data from the local arrays back to the arrays on the domain. For example, in a loop over elements, we gather variable data at the nodes around each element into local arrays of length 8 (the number of nodes surrounding a hexahedral element). We designate the gather operation as:

Variable	Description	Accessor Method
t^n	current simulation time	`time()`
Δt^n	current time increment	`deltatime()`
$\Delta t_{Courant}$	Courant time constraint	`dtcourant()`
Δt_{hydro}	hydro time constraint	`dthydro()`
u_{cut}	node velocity cut-off value	`ucut()`

Table 1.3. Scalar Variables

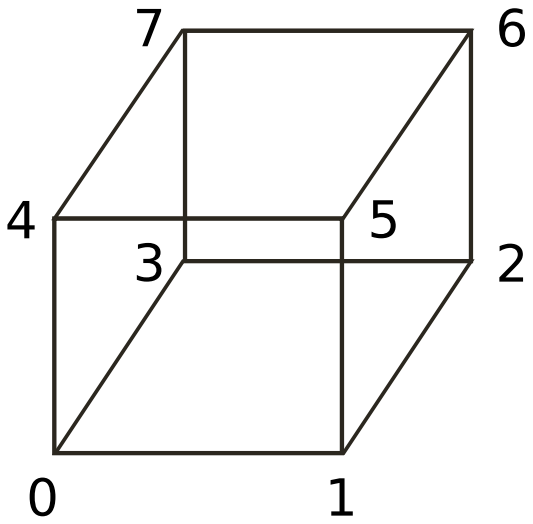

Figure 1.5. Node Numbering

$$\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\to\tilde{n})} : f \mapsto \tilde{f}$$

where f is a nodal variable on the mesh and $\tilde{f}$ is the array holding the values of f for the nodes around element e. Thus, the *tilde* over an item indicates a node variable whose values have been gathered into a local array for computation associated with a mesh element. The numbering scheme for the nodes around the element used to access items in the local array is illustrated in Fig. 1.5.

The inverse mapping operation that scatters the node values in the local array to the mesh arrays is denoted as:

$$\boldsymbol{\mathcal{S}}^{(e)}_{(\tilde{n}\to n(e))} : \tilde{f} \mapsto f$$

Similarly, while processing the local array of node data for an element, we sometimes gather a subset of the local array entries in a smaller array of that holds the node values for a particular *face* of the element. Such a local array has length 4 (the number of nodes surrounding an element face). We designate the operation of gathering the local node array entries for face k of an element as:

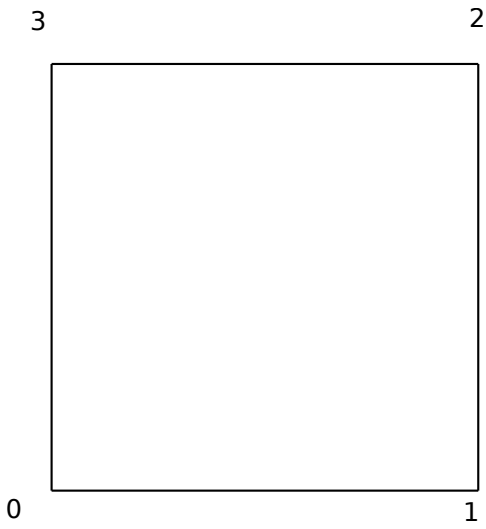

Figure 1.6. Face Node Numbering

$$\boldsymbol{\mathcal{G}}_{(\tilde{n}\to\hat{n})}^{(\tilde{n}_k)} : \tilde{f} \mapsto \hat{f}$$

Thus, the *hat* over an item indicates a node variable whose values have been gathered into a local array for computation associated with a face of a mesh element. The inverse mapping operation which scatters the node values for face k back to the larger local array for element node data is denoted as:

$$\boldsymbol{\mathcal{S}}_{(\hat{n}\to\tilde{n})}^{(\tilde{n}_k)} : \hat{f} \mapsto \tilde{f}$$

The numbering scheme for the nodes around an element face used to access items in the local node array corresponding to an element face is illustrated in Fig. 1.6.

1.3.3 Main Program

The main routine in our code performs of the following steps:

1. **Initialize the problem decomposition, mesh, global data structures**. This step contains the following sub-steps:
   (a) Create one or more domains. The number and spatial configuration of the domains depends on how we wish to decompose the problem. Each domain will contain all the state variables, such as pressure and node coordinates, for a portion of the mesh. We start by allocating the memory for those variables. Next, we create a lattice of node coordinates for each domain as a simple array. Each element on the domain is defined by the eight nodes that surround it.
   (b) Create a material index set. The goal of our challenge problem is to emulate the computational characteristics of a real hydrodynamics code as closely as possible. Such codes model problems containing multiple materials, where elements containing a given material are often identified by an index set. Although our challenge problem code uses only one material, we create an index set for each domain containing all elements of the domain and use this to emulate the algorithmic machinery found in a real code.

(c) Define the initial problem state. After identifying the basic entities in the mesh, we initialize all the state variables associated with the mesh. At this point, we also initialize a variety of scalar parameters dealing with material models and time step control.

(d) Create boundary conditions. Finally we mark certain entities lying on the mesh boundary so we can apply acceleration boundary conditions to model the mesh symmetries as described above in Section 1.2.1 for our challenge problem.

2. **Advance the solution variables to the final simulation time.**
The time integration process is implemented in the code as a *while-loop* near the end of the *main* routine. Each iteration of the loop advances the solution variables from the current simulation time t^n to a new simulation time t^{n+1} over a discrete time increment $\Delta t^n = t^{n+1} - t^n$. Here n is the timestep count. The integration loop is complete when t^{n+1} reaches the predetermined simulation stopping time. The body of the *while-loop* contains the following two sub-steps:

(a) Calculate Δt^n; see Section 1.4.

(b) Advances the solution variables from t^n to t^{n+1} using a "leap frog" time integration scheme for the Lagrange update; see Section 1.5.

1.4 Time Increment Calculation

The routine `CalcTimeIncrement()` computes the time increment Δt^n for the current timestep loop iteration. We aim for a "target" value of $t^{final} - t^n$ for Δt^n. However, the actual time increment is allowed to grow by a certain prescribed amount from the value used in the previous step and is subject to the constraints $\Delta t_{Courant}$ and Δt_{hydro} described in Section 1.5.3.

1.5 Lagrange Leapfrog Algorithm

The routine `LagrangeLeapFrog()` advances the solution from t^n to t^{n+1} over the time increment Δt^n. The process of advancing the solution is comprised of two major parts. The first described in Section 1.5.1 advances nodal variables on the mesh. The second described in Section 1.5.2 advances element variables on the mesh.

In the following description of our challenge problem reference implementation, we indicate the discrete integration time (via a superscript) for only those variables which comprise our primary solution state variables as discussed in Section 1.2.4. We emphasize that this does not imply that it is necessary to store more than one time level of data for such variables. Rather, we feel the time notation helps the reader relate the following code description to the integration algorithm. Other variables are associated with an integration time due to their implicit dependence on the solution state.

1.5.1 Advance Node Quantities

The routine `LagrangeNodal()` advances the nodal mesh variables, primarily velocity $\overrightarrow{\mathbf{U}}$ and position $\overrightarrow{\mathbf{X}}$. The main steps in this process are:

1. Calculate nodal forces (Section 1.5.1.1).
2. Calculate nodal accelerations (Section 1.5.1.2).
3. Apply acceleration boundary conditions as needed (Section 1.5.1.3).
4. Integrate nodal accelerations to obtain updated nodal velocities $\overrightarrow{\mathbf{U}}^{n+1}$ (Section 1.5.1.4).
5. Integrate nodal velocities to obtain updated nodal positions $\overrightarrow{\mathbf{X}}^{n+1}$ (Section 1.5.1.5).

1.5.1.1 Calculate Node Forces

The routine `CalcForceForNodes()` calculates a three-dimensional force vector $\overrightarrow{\mathbf{F}}$ at each mesh node based on the values of mesh variables at time t^n. First, the components of $\overrightarrow{\mathbf{F}}$ are set to zero at each node. Then, a volume force contribution is calculated within each mesh element. The force in each element is used to distribute a force contribution to each of its surrounding nodes. The total nodal force is accumulated as all elements in the mesh are traversed.

The routine `CalcVolumeForceForElems()` calculates the volume force contribution for each mesh element. The main steps in this process are:

1. Initialize stress terms for each element. Recall from Section 1.2.1 that our assumption of an inviscid isotropic stress tensor implies that the three principal stress components are equal, and the shear stresses are zero. Thus, we initialize the diagonal terms of the stress tensor $\boldsymbol{\Sigma}$ to $-(p+q)$ in each element. This is done in the routine `InitStressTermsForElems()`.
2. Integrate the volumetric stress contributions for each element. This is done in the routine `IntegrateStressForElems()` which does the following for each element in a loop over elements:
   (a) Gather node coordinates for the element into local arrays:
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\to\tilde{n})} : \overrightarrow{\mathbf{X}} \mapsto \overrightarrow{\tilde{\mathbf{X}}}$
   (b) Calculate normal vectors at element nodes (as an interpolation of element face normals) (routine `CalcElemNodeNormals()`).
   - Set node normals of element to zero, $\overrightarrow{\tilde{\mathbf{N}}} = 0$.
   - Enumerate the six faces of an element $\boldsymbol{\mathcal{G}}^{(\tilde{n}_k)}_{(\tilde{n}\to\hat{n})} : \overrightarrow{\tilde{\mathbf{X}}} \mapsto \overrightarrow{\hat{\mathbf{X}}}, k = 0 \ldots 5$.
     For each face, calculate a normal vector, scale the magnitude by one quarter, and sum the scaled vector into each of the four nodes of the element corresponding to the face.

   (c) Sum force contribution in element to local vector for each node around element (routine `SumElemStressesToNodeForces()`).

$$\tilde{F}_x = \tilde{\boldsymbol{\Sigma}}_{xx}\overrightarrow{\tilde{\mathbf{N}}}_x$$
$$\tilde{F}_y = \tilde{\boldsymbol{\Sigma}}_{yy}\overrightarrow{\tilde{\mathbf{N}}}_y$$
$$\tilde{F}_z = \tilde{\boldsymbol{\Sigma}}_{zz}\overrightarrow{\tilde{\mathbf{N}}}_z$$

   (d) Add local force contribution in each element node to proper node force vectors on mesh:
   $\boldsymbol{\mathcal{S}}^{(e)}_{(\tilde{n}\to n(e))} : \overrightarrow{\tilde{\mathbf{F}}} \mapsto \overrightarrow{\mathbf{F}}$
3. Perform a diagnostic check: check each element volume to make sure it is positive; if a volume less than or equal to zero is found, the code exits.
4. Calculate the hourglass control contribution for each element. This is done in the routine `CalcHourglassControlForElems()` which does the following for each element in a loop over elements:
   (a) Gather node coordinates for element into local arrays.
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\to\tilde{n})} : \overrightarrow{\mathbf{X}} \mapsto \overrightarrow{\tilde{\mathbf{X}}}$
   (b) Calculate element volume derivatives (routine `CalcElemVolumeDerivative()`).
   Starting with a formula for the volume of a hexahedron, take the derivative of that volume formula with respect to the coordinates at one of the nodes. By symmetry, the formula for one node can be applied to each of the other seven nodes.

(c) Perform a diagnostic check for element volume less than or equal to zero. If such a volume is found, the code exits.

Lastly, the routine `CalcFBHourglassForceForElems()` is called which calculates the Flanagan-Belytschko hourglass control force for each element; the paper [6] describes the mathematics behind the algorithm. The hourglass filter contributions for each element are independently added directly to (domain level) $\overrightarrow{\mathbf{F}}$, which completes the nodal force calculation.

1.5.1.2 Calculate Node Accelerations

The routine `CalcAccelerationForNodes()` calculates a three-dimensional acceleration vector $\overrightarrow{\mathbf{A}}$ at each mesh node from $\overrightarrow{\mathbf{F}}$. The acceleration is computed using Newton's Second Law of Motion, $\overrightarrow{\mathbf{F}} = m_0 \overrightarrow{\mathbf{A}}$, where m_0 is the mass at the node; i.e.,

$$\overrightarrow{\mathbf{A}} = \overrightarrow{\mathbf{F}} / m_0$$

Note that since the mass in each element is constant in time for our calculations, the mass at each node is also constant in time. The nodal mass values are set during the problem set up.

1.5.1.3 Apply Boundary Conditions

The routine `ApplyAccelerationBoundaryConditions()` applies *symmetry* boundary conditions at nodes on the boundaries of the mesh where these were specified during problem set up. A symmetry boundary condition sets the normal component of $\overrightarrow{\mathbf{A}}$ at the boundary to zero. This implies that the normal component of the velocity vector $\overrightarrow{\mathbf{U}}$ will remain constant in time.

Recall that the benchmark Sedov problem is spherically-symmetric and that we simulate it in a cubic domain containing a single octant of the sphere. To maintain spherical symmetry of the domain, we apply symmetry boundary conditions along the faces of the cubic domain that contact the planes separating the octants of the sphere. This forces the normal components of $\overrightarrow{\mathbf{U}}$ to be zero along these boundary faces for all time since they were initialized to zero.

1.5.1.4 Advance Node Velocities

The routine `CalcVelocityForNodes()` integrates the acceleration at each node to advance the velocity at the node to t^{n+1}, i.e.,

$$\overrightarrow{\mathbf{U}}^{n+1} = \overrightarrow{\mathbf{U}}^{n} + \overrightarrow{\mathbf{A}} \Delta t^n$$

Note that this routine also applies a *cut-off* to each velocity vector value. Specifically, if a value is below some prescribed value, that term is set to zero. The reason for this cutoff is to prevent spurious mesh motion which may arise due to floating point roundoff error when the velocity is near zero.

1.5.1.5 Advance Node Positions

The routine `CalcPositionForNodes()` performs the last step in the nodal advance portion of the algorithm by integrating the velocity at each node to advance the position of the node to t^{n+1}, i.e.,

$$\overrightarrow{\mathbf{X}}^{n+1} = \overrightarrow{\mathbf{X}}^{n} + \overrightarrow{\mathbf{U}}^{n+1} \Delta t^n$$

1.5.2 Advance Element Quantities

The routine `LagrangeElements()` advances the element mesh quantities, primarily pressure p, internal energy e, and relative volume V. The artificial viscosity q in each element is also calculated here. The main steps in this process are:

1. Calculate element quantities based on nodal kinematic quantities (Section 1.5.2.1).
2. Calculate element artificial viscosity terms (Section 1.5.2.2).
3. Apply material properties in each element needed to calulate updated pressure p^{n+1} and internal energy e^{n+1} (Section 1.5.2.3).
4. Compute updated element volume $\mathbf{V}^{n+1}$ (Section 1.5.2.4).

1.5.2.1 Calculate Kinematic Element Quantities

The routine `CalcLagrangeElements()` calculates various element quantities that are based on the new kinematic node quantities $\overrightarrow{\mathbf{U}}^{n+1}$ and $\overrightarrow{\mathbf{X}}^{n+1}$ computed as described in Section 1.5.1.4 and Section 1.5.1.5, respectively.

First, `CalcLagrangeElements()` calls the routine `CalcKinematicsForElems()` which calculates terms in the total strain rate tensor $\boldsymbol{\epsilon}_{tot}$ that are used to compute the terms in the deviatoric strain rate tensor $\boldsymbol{\epsilon}$. These calculations are done for each element in a loop over elements as follows:

1. Gather node coordinates for element into local arrays.
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\rightarrow\tilde{n})} : \overrightarrow{\mathbf{X}} \mapsto \overrightarrow{\tilde{\mathbf{X}}}$
2. Calculate volume from element coordinates via call to `CalcElemVolume()`. Then compute $\mathbf{V}^{n+1} = volume/\mathbf{V}_0$ and relative volume change $\Delta V = \mathbf{V}^{n+1} - \mathbf{V}^n$.
3. Calculate the characteristic length l_{char} for the element. The length is computed in the routine `CalcElemCharacteristicLength()` which divides the $volume$ of the element by the area of its largest face.
4. Gather node velocities for element into local arrays.
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\rightarrow\tilde{n})} : \overrightarrow{\mathbf{U}} \mapsto \overrightarrow{\tilde{\mathbf{U}}}$
5. Modify nodal positions in local array so that they are at the time halfway between t^n and t^{n+1}:
   $\overrightarrow{\tilde{\mathbf{X}}} -= \frac{1}{2}\Delta t \overrightarrow{\tilde{\mathbf{U}}}$
6. Call the routine `CalcElemShapeFunctionDerivatives()` which calculates shape function derivatives for the element which are used to compute the velocity gradient for the element.
7. Calculate element velocity gradient which defines the terms of $\boldsymbol{\epsilon}_{tot}$. The diagonal entries of $\boldsymbol{\epsilon}_{tot}$ are then used to initialize the diagonal entries of the strain rate tensor $\boldsymbol{\epsilon}$.

Lastly, the routine `CalcLagrangeElements()` calculates $\dot{V}/V = (\epsilon_{xx} + \epsilon_{yy} + \epsilon_{zz})/3$ and subtracts $\dot{V}/V$ from each strain component to define the final deviatoric strain rate tensor.

1.5.2.2 Calculate Artificial Viscosity

The routine `CalcQForElems()` calculates the artificial viscosity term q for each element. The paper [3] describes the mathematical aspects of the algorithm. The calculation is partitioned into two parts.

The routine `CalcMonotonicQGradientsForElems()` performs the first part. For each element in a loop over elements, this routine does the following:

1. Gather nodal coordinates for element into local arrays.
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\rightarrow\tilde{n})} : \overrightarrow{\mathbf{X}} \mapsto \overrightarrow{\tilde{\mathbf{X}}}$

2. Gather nodal velocities for element into local arrays.
   $\boldsymbol{\mathcal{G}}^{(e)}_{(n(e)\rightarrow\tilde{n})} : \overrightarrow{\mathbf{U}} \mapsto \tilde{\overrightarrow{\mathbf{U}}}$
3. Compute various discrete spatial gradients of nodal coordinates and velocity gradients with respect to a reference coordinate system. The mapping $(x, y, z) \mapsto (\xi, \eta, \zeta)$ maps the element to a unit cube. Mapping the element to the unit cube simplifies the process of defining a single value for q in the element from the gradient information. The resulting gradients for the coordinates and velocity are obtained from the mesh object via the methods `delx_xi()`, `delx_eta()`, `delx_zeta()`, and `delv_xi()`, `delv_eta()`, `delv_zeta()`, respectively.

The routine `CalcMonotonicQForElems()` performs the second part of the q calculation. This routine calls `CalcMonotonicQRegionForElems()` which uses the spatial gradient information computed earlier to compute linear and quadratic terms for q, q_{lin} and q_{quad}, respectively. The actual element values of q are calculated during the application of material properties in each element. This is described next in Section 1.5.2.3.

1.5.2.3 Apply Material Properties

The routine `ApplyMaterialPropertiesForElems()` updates the pressure and internal energy variables to their values at the new time, p^{n+1} and e^{n+1}. The routine first initializes a temporary array with the values of $\mathbf{V}^{n+1}$ for each element that was computed earlier (Section 1.5.2.1). Then, upper and lower cut-off values are applied to each array value to keep the relative volumes within a prescribed range (not too close to zero and not too large). Next, the routine `EvalEOSForElems()` is called and the array of modified relative element volumes is passed to it.

The routine `EvalEOSForElems()` calculates updated values for pressure p^{n+1} and internal energy e^{n+1}. The computation involves several loops over elements to pack various mesh element arrays (e.g., p, e, q, etc.) into local temporary arrays. Several other quantities are computed and stored in element length temporary arrays also. The temporary arrays are needed because the routine `CalcEnergyForElems()` calculates p^{n+1} and e^{n+1} in each element in an iterative process that requires knowledge of those variables at time t^n while it computes the new time values.

The routine `CalcEnergyForElems()` calls `CalcPressureForElems()` repeatedly: The function `CalcPressureForElems()` is the Equation of State model for a "gamma law" gas:

$$P = (\gamma - 1)\frac{\rho}{\rho_0}e$$

and the value `c1s` passed to the routine is defined to be $\gamma - 1$. The Equation of State calculation is a core part of any hydrocode. In a production code, one of any number of Equation of State functions may be called to generate a pressure that is needed to close the system of equations and generate a unique solution.

When `EvalEOSForElems()` returns, the mesh element arrays for p, e, and q are set to their updated values. The algorithms in `CalcEnergyForElems()` and `EvalEOSForElems()` are described in [3].

Lastly, the routine `CalcSoundSpeedForElems()` calculates the sound speed c_{sound} in each element using p^{n+1} and e^{n+1}:

$$c_{sound} = \frac{p^{n+1}e^{n+1} + (\mathbf{V}^{n+1})^2 p^{n+1}(\gamma - 1)(\frac{1}{\mathbf{V}^{n+1}-1} + 1)}{\rho_0}$$

The maximum value of c_{sound} is used to calculate constraints on Δt^{n+1} which will be used for the next time advance step.

1.5.2.4 Update Element Volumes

The routine `UpdateVolumesForElems()` updates the relative volume to $\mathbf{V}^{n+1}$. This routine basically resets the current volume $\mathbf{V}^n$ in each element to the new volume $\mathbf{V}^{n+1}$ computed in Section 1.5.2.3 so the simulation can continue to the next time increment.

Note that this routine applies a *cut-off* to the relative volume V in each element. Specifically, if V is sufficiently close to one (a prescribed tolerance), then V is set to one. The reason for this cutoff is to prevent spurious deviations of volume from their initial values which may arise due to floating point roundoff error.

1.5.3 Calculate Time Constraints

After all solution variables are advanced to t^{n+1}, the constraints $\Delta t_{Courant}$ and Δt_{hydro} for the next time increment Δt^{n+1} are calculated in the routine `CalcTimeConstraintsForElems()`. Each constraint is computed in each element and then the final constraint value is the minimum over all element values. The constraints are applied during the computation of Δt for the next time step.

The routine `CalcCourantConstraintForElems()` calculates the Courant timestep constraint $\Delta t_{Courant}$. This constraint is calculated only in elements whose volumes are changing; that is, $\dot{V}/V \neq 0$. If all element volumes remain the same, there is no Courant constraint applied during the subsequent Δt calculation. Essentially, $\Delta t_{Courant}$ for an element is the ratio: characteristic length for the element (calculated in Section 1.5.2.1) divided by the sound speed c_{sound} in the element. However, when the element is undergoing *compression*, that is $\dot{V}/V < 0$, additional terms are added to the denominator to reduce $\Delta t_{Courant}$ further.

The routine `CalcHydroConstraintForElems()` calculates the hydro timestep constraint Δt_{hydro}. Similar to $\Delta t_{Courant}$, Δt_{hydro} is calculated only in elements whose volumes are changing. When an element is undergoing volume change, Δt_{hydro} for the element is some maximum allowable element volume change (prescribed) divided by $\dot{V}/V$ in the element.

1.6 Benchmarks and Metrics

The Lagrange algorithm can be applied to a wide range of hydrodynamics simulation problems. To simplify the challenge problem code, we limit its application to a well-know test problem that can be used to verify correctness of the implementation and analyze execution performance. Specifically, the problem setup phase is hard-coded for the Sedov blast wave problem [7]. An example calculation of this problem is shown in Figure 1.7. The Sedov problem has a known analytic solution and can be scaled to arbitrarily large problem sizes with mesh resolution being the primary scaling factor.

The serial reference code LULESH v1.0, compiled with g++ 4.1.2 using -O3 option, has been timed running on one core of a 2.3 GHz. Opteron 8356 processor. Using a mesh with a single domain of 45x45x45 elements (22 floating point values per element, 16 integers per element) and 46x46x46 nodes (13 floating point values per node), the wall clock time was 8 minutes and 58 seconds.

This standard configuration runs in 1495 iterations of the main loop. We expect that a UHPC system should be able to run a calculation containing more than one billion mesh elements and potentially more than that, depending on available memory.

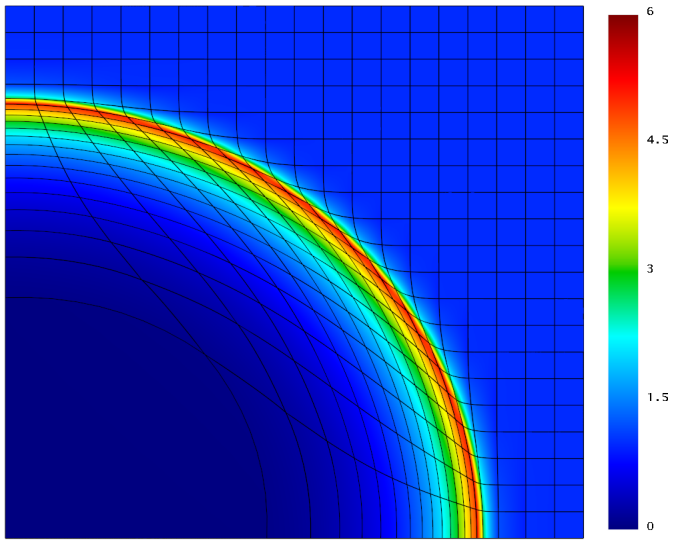

Figure 1.7. The Sedov blast wave problem models an expanding shock front originating from a point blast.

1.6.1 Benchmarks

Several of the key kernels found in the Lagrange time step have been extracted for ease of detailed performance analysis on various architectures. The kernels consist of the hexahedron volume calculation, the force calculation, and calculation of the linear and quadratic artificial viscocity coefficients. Each kernel is described below. The code for these kernels are provided in the CHASM code repository.

1.6.1.1 Kernel 1 – Hexahedron volume calculation

This kernel tests the ability of the compiler to optimize some very common code constructs. The kernel calculation involves data access through index arrays. It exhibits an access pattern that gathers data from node centered fields, and stores data on element centered fields. Index arrays are a common feature found in unstructured meshes, as are operation between fields defined over different mesh centerings. This computaion contains a large amount of available parallelism, since all arrays accesses are to independent memory locations. This kernel contains initialization code and may be compiled and run.

1.6.1.2 Kernel 2 – Force calculation

This kernel generates nodal forces from element stresses. It is possible to rewrite this code in a more performance efficient form, but the style used here is what is found in actual finite element codes. The code is split into routines that make it easier for engineers to think about a tiny part of a problem with regular structure (i.e. a single face or element within a mesh), which then gets assembled into a more complicated structure at a larger scale (i.e. faces are assembled into elements, and elements are assembled into domains). The code is written in this style to reduce both the complexity of both the code and the physics problem being solved. The parallelism may be limited due to the way the code

is (necessarily) structured. The indexed array usage may also complicate automatic parallelization, although the introduction of compiler directives to expose parallelism may be an acceptable solution. This kernel contains initialization code and may be compiled and run.

1.6.1.3 Kernel 3 – Calculate linear and quadratic artificial viscosity coefficients

This kernel contains conditional logic, and is somewhat representative of conditional data dependencies that exist in real codes. An alternative routine CalcEnergyForElements is even more representative, but it is not as compact as the Kernel 3 code fragment. This kernel does not contain initialization code.

1.6.2 Metrics

We propose various metrics to evaluate performance. The metrics include:

- Grind Time: A common metric used to evaluate performance of finite element codes is "grind time," which is the average time required to update the solution variables in an element through one time increment, typically in microseconds. Current production codes typically execute the Sedov problem on x86-64/Linux systems with a grind between 2 and 3 microseconds on one processor. The grind time for the reference code in the configuration described above was 3.95 microseconds.
- Memory Bandwidth: Our implementation utilizes an unstructured mesh representation which is often employed in production hydro codes for flexibility in defining complex geometries. The use of mesh connectivity arrays, such as those that define nodes associated with each mesh element, results in indirection that can stress system memory bandwidth.
- Scalability and Parallelism: Our implementation is designed to treat each mesh element as the smallest unit of work. This will allow a very large amount of fine-grained parallelism to be exercised by UHPC systems. By dividing the mesh into domains, coarse grain parallelism may be employed.
- Programmability: Our challenge problem reference implementation is derived from a production hydrocode containing several hundred thousand lines of source code. However, our reference implementation consists of only a few thousand lines. This will allow exploration of implementation alternatives suited to novel UHPC architectures, potentially involving significant code rewriting. Also, since we have strived to retain essential features of the original code in the reference implementation, these implementation modifications will be relevant to production hydrocodes.

1.7 Architecture Characterization

We have chosen to implement a 3D hydrodynamics challenge problem in order to simplify scalability, expose more parallelism, and present more realistic communication patterns among data elements. Our problem has been carefully constructed to characterize a variety of system-specific parameters:

1.7.1 Primitive data types

There are three data types used in the LULESH code:

- **Real_t** Can be defined as float, double, long double, or any other numeric type suitable for real number arithmetic. Furthermore, since LULESH has been written in C++, LULESH can be extended to support user defined types for fixed-point or interval arithmetic if those features are not natively supported in the hardware, compiler, or runtime system.
- **Integer_t** Different integer word sizes perform with different efficiency due to clock-cycle and memory bandwidth limitations. Currently, the only direct integer quantity used in LULESH is

a 12 bit flag variable.

- **Index_t** This is the type used for array indexing and loop-index variables. In theory, an architecture may support different word sizes for this type of variable, again for both clock-cycle/register efficiency and memory bandwidth purposes.

1.7.2 Memory Locality

We have isolated all the LULESH data structures in the mesh object at the top of the LULESH source code file. This reduces the code changes needed to map LULESH to the memory subsystem of a specific architecture to just a few lines of code per data structure decision.

- **Interleave** Three dimensional coordinate data can be represented as three separate arrays for the x, y, and z coordinate components. It can also be represented as a single array in which each array entry consists of three coordinate components. The decision of how to bundle coordinate data efficiently depends on the implementation of both the core's micro-architecture and the memory subsystem. For example, a single array could reduce register pressure in an inner loop, while using multiple independent arrays may expose more opportunities for vectorization.
- **Latency** Many memory subsystems consist of different cache levels, each having a different latency. We have introduced a *domain* as a locality context that can be resized to fit differing memory hierarchies. Rather than running one large domain that won't fit into cache or local store, the mesh can instead be described as multiple smaller domains each of whose local data fits into the local store. This flexibility should allow TA-1 teams to experiment with varying granularity of task in terms of the number of mesh elements being processed by a task.
- **Contention** For multi-core architectures with a cache hierarchy, re-organizing the arrays may reduce or eliminate cache contention. By isolating the allocation of arrays in the LULESH mesh object, it is easy to experiment with alternative layouts.

1.7.3 Network Locality

Domain level locality contexts not only allow experimentation with memory locality optimizations, they also allow evaluation of network communication and locality within the network. By carefully mapping domains to the network architecture, we can isolate the effects of bandwidth limitations and contention in the network. We note that although the challenge problem has been implemented as a simple cubic mesh, no assumption can be made about topology. In a "real" mesh, there will be reduced and enhanced connectivity points. That is, nodes on the mesh may be shared by fewer or more elements than in the logically-rectangular configuration. While it is true that every element will always be defined in terms of eight nodes, there are no guarantees about the number of elements sharing a given node. TA-1 teams should avoid using optimizations that depend on the dividing the problem into simple nested loops over a fixed rectangular block topology since this is not representative for many real-world calculations.

- **Bandwidth** In a parallel implementation of the hydrodynamics challenge problem, there are two major interacting factors, mesh discretization and domain partitioning. For example, a 1D slab domain partitioning imposes a (spatially) linear nearest-neighbor communication pattern. If this domain partitioning is chosen as a "base configuration" for best case network communication, the mesh must be discretized accordingly.
- **Contention** In a standard 3D domain stencil computation over domains, each domain must support bi-directional communication with other domains with which it shares a face. Furthermore, for our challenge problem and many "real-world" problems, we will require bi-directional communication between domains that touch at corners. Very few network architectures directly

support this corner coupling, so corner coupling introduces the possibility of message contention (and associated latency skew which accentuates or causes a load imbalance).

Finally, the LULESH challenge problem can be run in two modes. It can be run with a fixed time increment size so that only "nearest-neighbor" data exchanges are required between domains, or it can be run in a Courant/hydro limited manner that requires an additional global reduction across all the domains. In fixed time step mode, assuming domains are mapped across compute nodes of the UHPC machine, the amount of communication between compute nodes should be constant regardless of scale. It will depend only on the surface area of the domain boundaries and how many domains are mapped to a compute node. It should be noted that if a fixed time step approach is used, the time increment must be specified at the beginning of the simulation and must be reasonable (e.g., sufficiently small) to allow the simulation to execute to completion without numerical difficulties.

The Courant/hydro option, which allows the time increment to change for each time step, is required for most real-world problems typically. When the Courant/hydro option is used, the global reduction often introduces a pronounced latency at large scale when the domains are mapped to separate compute nodes communicating over an interconnection network. Additionally, the run time of the Courant/hydro variant will go up exponentially, because more time steps are required to run to completion as the resolution (i.e., mesh spacing) is refined. By default, the code runs in a mode that requires a global reduction for each iteration of the simulation loop.

1.7.4 Parallelism

There should be many opportunities for vectorization, threading, and node-level parallelism in the LULESH code.

- **Vectorization** Since we chose to implement a 3D hydrocode, there is a sizable core computation applied to a single volumetric element. This should expose good vectorization opportunities at the leaf function level. There are also vectorization opportunities available in several of the LULESH code inner-loops.
- **Many-core threading** The code includes many subroutines that are computationally intensive yet contain few conditional code paths to be executed, making them easy to implement in GPU-like kernels. In some cases, it is possible to trade off the use of conditional code paths with unconditional code, some components of which may be superfluous. Specifically, in the artificial viscosity calculation, there is no need to compute artificial viscosity in an expanding element (mesh.vdov(i) > 0). Thus it is valid to omit evaluation of CalcMonotonicQGradients-ForElems() and CalcMonotonicQRegionForElems() for any expanding element, but checking for the condition and omitting the calculations will introduce conditional logic. Many-core TA-1 architectures may give better performance by omitting the conditional check and always performing the artificial viscosity calculations on all elements.
- **Multi-core threading (pthread/OpenMP)** Several subroutines contain data dependent control paths that perform reasonably well with this kind of technology. Using the example of artificial viscosity, multi-core node architectures might have better performance by including the conditional path.

Factors that may reduce available parallelism include the use of partial-sum operations, and the indexed array indirection in several parts of the code. There are many partial sums in the code due to the fact that many operations in real hydrodynamic codes are written in terms of element local coordinate systems. It is possible that a rewrite of the code could eliminate many of these partial sums. However, TA-1 teams should use caution in re-writing the partial sum calculations in a way

that depends on a simple rectangular topology, as those optimizations would not be applicable in a different mesh with more complex mesh connectivity.

LULESH is presently a serial code. However, it has been structured to facilitate distributed memory parallelism across nodes of a cluster. Comments in the code have been inserted to indicate where inter-domain parallelism should occur.

- In `TimeIncrement`, there needs to be a parallel reduction when running in fixed timestep mode at the start of the top level `if` statement.
- At the end of `CalcForceForNodes`, there should be a point to point communication among nodes to exchange fx, fy, and fz (nodal centering) arrays on boundary nodes and then sum exchanged boundary values into local values of `fx`, `fy`, and `fz`.
- In `CalcQForElems`, communication is necessary to populate halo elements for `delv_xi`, `delv_eta`, and `delv_zeta` (element centered) arrays.
- In the main driver loop, communication is required to exchange x,xd,y,yd,z,and zd (node centered) arrays on boundaries. The value received from the "owner domain" will overwrite the local value. If a node contains multiple domains, then only one of them will be the node owner domain (there will be a semi-arbitrary selection of who is owner).

All "network-based" parallelism will occur at those code locations in our parallel implementation, and these are the locations where most any implementation will find it is necessary to send data between nodes/cabinets/racks. The "monoQ" communication might be something that can be worked around with an algorithm change, but the other communication is necessary. We note in closing that a domain implies a locality context and does not necessarily imply MPI communication between domains.

1.7.5 Compiler

LULESH includes many constructs commonly found in actual hydrodynamics software, and thus offers an opportunity to evaluate and tune compilers for these common constructs. Specifically, the LULESH implementation makes it possible to test inlining, constant propagation, vectorization, and optimizations related to indexed array usage.

We have tried to organize the LULESH code so that a variety of compiler and run time optimizations can be tested; e.g., the use of C++ STL libraries or pointer based arrays. We have tried to make it simple to trade off these underlying implementation details with just a few lines of code changes in the LULESH mesh object.

Additionally, the code will exercise compiler optimizations of specific C language keywords such as static, inline, and restrict. The first LULESH code release consists of a single file. TA-1 teams may choose to divide the code into multiple files for ease of separate compilation and as an opportunity to test inter-file optimization capabilities of their compilers. Finally, different compilers often have proprietary directives to enable the generation of proprietary opcodes for their target processors. Such constructs are not provided in our released code, but TA-1 teams may want to experiment with such architecture-specific directives.

Bibliography

[1] Ale3d web site. https://wci.llnl.gov/codes/ale3d/.

[2] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 2000.

[3] R. B. Christensen. Godunov methods on a staggered mesh: An improved artificial viscosity. *Lawrence Livermore National Laboratory Report, UCRL-JC-105-269*, 1991. Available online at https://e-reports-ext.llnl.gov/pdf/219547.pdf.

[4] R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal*, pages 215–234, March 1967. English translation of the Germal original published in Mathematische Annalen 100(1):32–74, 1928.

[5] DoD High Performance Computing Modernization Program Office. Department of Defense High Performance Computing Modernization Program FY 2010 Requirements Analysis Report. 2010.

[6] D. P. Flanagan and T. Belytschko. A uniform strain hexahedron and quadrilateral with orthogonal hourglass control. *International Journal for Numerical Methods in Engineering*, pages 679–706, March 1981.

[7] L. I. Sedov. *Similarity and Dimensional Methods in Mechanics*. Academic Press, 1959.

[8] Dan White. A code to model electromagnetic phenomena. *Lawrence Livermore National Laboratory S&T Report*, November 2007.

[9] M. L. Wilkins. *Methods in Computational Physics*. Academic Press, 1964.