# Uintah

Jim Guilkey, Todd Harman, Justin Luitjens, John Schmidt,
Jeremy Thornock, J. Davison de St. Germain

April 14, 2009

## Contents

**Abstract**

# 1 Overview of Uintah

The Uintah Computational Framework, i.e. Uintah consists of a set of software components and libraries that facilitate the solution of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids using hundreds to thousands of processors.

One of the challenges in designing a parallel, component-based multi-physics application is determining how to efficiently decompose the problem domain. Components, by definition, make local decisions. Yet parallel efficiency is only obtained through a globally optimal domain decomposition and scheduling of computational tasks. Typical techniques include allocating disjoint sets of processing resources to each component, or defining a single domain decomposition that is a compromise between the ideal load balance of multiple components. However, neither of these techniques will achieve maximum efficiency for complex multi-physics problems.

Uintah uses a non-traditional approach to achieving parallelism, employing an abstract taskgraph representation to describe computation and communication. The taskgraph is an explicit representation of the computation and communication that occur in the coarse of a single iteration of the simulation (typically a timestep or nonlinear solver iteration). Uintah components delegate decisions about parallelism to a scheduler component, using variable dependencies to describe communication patterns and characterizing computational workloads to facilitate a global resource optimization. The taskgraph representation has a number of advantages, including efficient fine-grained coupling of multi-physics components, flexible load balancing mechanisms and a separation of application concerns from parallelism concerns. However, it creates a challenge for scalability which we overcome by creating an implicit definition of this graph and representing it in a distributed fashion.

The primary advantage of a component-based approach is that it facilitates the separate development of simulation algorithms, models, and infrastructure. Components of the simulation can evolve independently. The component-based architecture allows pieces of the system to be implemented in a rudimentary form at first and then evolve as the technologies mature. Most importantly, Uintah allows the aspects of parallelism (schedulers, load-balancers, parallel input/output, and so forth) to evolve independently of the simulation components. Furthermore, components enable replacement of computation pieces without complex decision logic in the code itself.

# 2 Scheduler

The Scheduler in Uintah is responsible for determining the order of tasks and ensuring that the correct inter-processor data is made available when necessary. Each software component passes a set of tasks to the scheduler. Each task is responsible for computing some subset of variables, and may require previously computed variables, possibly from different processors. The scheduler will then compile this task information into a task graph, and the task graph will contain a sorted order of tasks, along with any information necessary to perform inter-process communication via MPI. Then, when the scheduler is executed, the tasks will execute in the pre-determined order.

# 3 Tasks

A task contains two essential components: a pointer to a function that performs the actual computations, and the data inputs and outputs, i.e. the data dependencies required by the function. When a task requests a previously computed variable from the data warehouse, the number of ghost cells are also specified. The Unitah framework uses the ghost cell information to excecute inter-process communication to retrieve the necessary ghost cell data.

An example of a task description is presented showing the essential features that are commonly used by the application developer when implementing an algorithm within the Uintah framework. The task component is assigned a name and in this particular example, it is called `taskexample` and a funtion pointer, `&Example::taskexample`. Following the instantiation of the task itself, the dependency information is assigned to the tasks. In the following example, the task requires data from the previous timestep (`Task::OldDW`) associated with the name variable1_label and requires one ghost node (`Ghost::AroundNodes,1`) level of information which will be retrieved from another processor via MPI. In addition, the task will compute two new pieces of data each associated with different variables, i.e. `variable1_label`, and `variable2_label`. Finally, the task is added to the scheduler component with specifications about what patches and materials are associated with the actual computation.

```
Task* task = scinew Task("Example::taskexample",this,
```

```
                          &Example::taskexample);
task->requires(Task::OldDW, variable1_label, Ghost::AroundNodes, 1);
task->computes(variable1_label);
task->computes(variable2_label);
sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
```

For more complex problems involving multiple materials and multi-physics calculations, a subset of the materials may only be used in the calculation of particular tasks. The Uintah framework allows for the independent scheduling and computation of multi-material within a multi-physics calculation.

# 4 Tasks and Scheduler Description – Programmer Interface

## 4.1 Simulation Component Class Description

Each Uintah component can be described as a C++ class that is derived from two other classes: **UintahParallelComponent** and a **SimulationInterface**. The new derived class must provide the following virtual methods: `problemSetup`, `scheduleInitialize`, `scheduleComputeStableTimestep`, and `scheduleTimeAdvance`. Here is an example of the typical *.h file that needs to be created for a new component.

```
class Example : public UintahParallelComponent, public SimulationInterface {
  public:

    virtual void problemSetup(const ProblemSpecP& params,
                              const ProblemSpecP& restart_prob_spec,
                              GridP& grid, SimulationStateP&);

    virtual void scheduleInitialize(const LevelP& level,SchedulerP& sched);

    virtual void scheduleComputeStableTimestep(const LevelP& level,
                                               SchedulerP&);
```

```
    virtual void scheduleTimeAdvance(const LevelP& level, SchedulerP&);

  private:
    Example(const ProcessorGroup* myworld);
    virtual ~Example();


    void initialize(const ProcessorGroup*,
                    const PatchSubset* patches,
                    const MaterialSubset* matls,
                    DataWarehouse* old_dw,
                    DataWarehouse* new_dw);


    void computeStableTimestep(const ProcessorGroup*,
                               const PatchSubset* patches,
                               const MaterialSubset* matls,
                               DataWarehouse* old_dw,
                               DataWarehouse* new_dw);

    void timeAdvance(const ProcessorGroup*,
                     const PatchSubset* patches,
                     const MaterialSubset* matls,
                     DataWarehouse* old_dw,
                     DataWarehouse* new_dw);

}
```

Each new component inherits from the classes **UintahParrallelComponent** and **SimulationInterface**. The component overrides default implementations of various methods. The above methods are the essential functions that a new component must implement. Additional methods to do AMR will be described as more complex examples are presented.

The roles of each of the scheduling methods are described below. Each scheduling method, i.e. `scheduleInitialize`, `scheduleComputeStableTimestep`,

and `scheduleTimeAdvance` describe

### 4.1.1 ProblemSetup

The purpose of this method is to read a problem specification which requires a minimum of information about the grid used, time information, i.e. time step size, length of time for simulation, etc, and where and what data is actually saved. Depending on the problem that is solved, the input file can be rather complex, and this method would evolve to establish any and all parameters needed to initially setup the problem.

### 4.1.2 ScheduleInitialize

The purpose of this method is to initialize the grid data with values read in from the problemSetup and to define what variables are actually computed in the TimeAdvance stage of the simulation. A task is defined which references a function pointer called `initialize`.

### 4.1.3 ScheduleComputeStableTimestep

The purpose of this method is to compute the next timestep in the simulation. A task is defined which references a function pointer called `computeStableTimestep`.

### 4.1.4 ScheduleTimeAdvance

The purpose of this method is to schedule the actual algorithmic implementation. For simple algorithms, there is only one task defined with a minimal set of data dependencies specified. However, for more complicated algorithms, the best way to schedule the algorithm is to break it down into individual tasks. Each task of the algorithm will have its own data dependencies and function pointers that reference individual computational methods.

## 4.2 Data Storage Concepts

During the course of the simulation, data is computed and stored in a data structure called the DataWarehouse. Data that is from a previous timestep is stored in the Old DataWarehouse, called `OldDW`, and data that is computed in current timestep is stored in the New DataWarehouse, called `NewDW`. At

the end of the timestep, current timestep data is moved to the old data warehouse for the next timestep in the simulation.

# 5 Downloading the Software

The Uintah problem solving environment provides a framework for solving PDEs on structured grids

Uintah can be obtained either from a tarball (http://www.uintah.utah.edu) or by using svn to download the latest source from the following:

```
svn co https://code.sci.utah.edu/svn/SCIRun/trunk Uintah
```

The above command checks out the Uintah source tree and installs it into a directory called Uintah in the users home directory.

The Thirdparty library can similarly be obtained via:

```
svn co https://code.sci.utah.edu/svn/Thirdparty/1.25.4 Thirdparty
```

The Thirdparty library source code is downloaded into a directory called Thirdparty.

## 5.1 Installing Thirdparty, SCIRun, and Uintah

### 5.1.1 Thirdparty Install

Please read the README.txt found in */Thirdparty*.

Thirdparty should be installed in */usr/local/Thirdparty*. As root, create this directory:

```
\# mkdir /usr/local/Thirdparty
```

Chang to the Thirdparty directory you checked out, i.e. cd /Thirdparty
After reading the README.txt file type the follow as the root user:
**32bit OS:**

```
\# ./install.sh /usr/local/Thirdparty/ 32
```

**64bit OS:**

```
\# ./install.sh /usr/local/Thirdparty/ 64
```

### 5.1.2 Configuring Uintah

cd to */SCIRun* and create the following directories: dbg and opt
cd to dbg and type the following to configure for a debug build:

```
./src/configure --enable-debug --enable-sci-malloc
--enable-package=Uintah
--with-thirdparty=/usr/local/Thirdparty/1.25.5/Linux/gcc-4.3.1-2-32bit/
```

Then build the software by typing `make` at the command line. Once the debug build has finished which can take roughly an hour on a single processor Pentium IV computer, cd to the opt/ and type the following to configure for an optimized build:

```
./src/configure '--enable-optimze=-march=pentium4 -msse -msse2
-mfpmath=sse -03' --disable-sci-malloc --enable-assertion-level=0
--enable-package=Uintah
--with-thirdparty=/usr/local/Thirdparty/1.25.4/Linux/gcc-4.3.1-2-32bit/
```

Then build the software by typing `make` at the command line

# 6 Computational Framework

## 6.1 Mechanics of Running sus

- Explain how to run on multiple processors
- what do the different command line options mean
- how to restart an uda

## 6.2 Time Related Variables

Uintah components are timestepping codes. As such, one of the first entries in each input file describes the timestepping parameters. An input file segment is given below that encompasses all of the possible parameters. Most are self-explanatory, and not all are required, (e.g `<max_Timestep>`, `<max_delt_increase>`, `<end_on_max_time_exactly>` and `<delt_init>` are all optional). `<timestep_multiplier>` serves as a CFL number, that is, a number, usually less than 1.0, that is used to moderate the timestep automatically calculated by the individual components.

```
<Time>
    <maxTime>            1.0         </maxTime>
    <initTime>           0.0         </initTime>
    <delt_min>           0.0         </delt_min>
    <delt_max>           1.0         </delt_max>
    <delt_init>          1.0e-9      </delt_init>
    <max_delt_increase>  2.0         </max_delt_increase>
    <timestep_multiplier>1.0         </timestep_multiplier>
    <max_Timestep>       100         </max_Timestep>
    <end_on_max_time_exactly>true    </end_on_max_time_exactly>
</Time>
```

## 6.3 Data Archiver

- variable labels
- checkpointing
- different options for specifying the output frequency

## 6.4 Geometry objects

- different objects available and how to specify them
- what is res
- operators, union, difference
- example of combining several geom_objects

## 6.5 Boundary conditions

- describe how to have a jet in the floor of the domain.

## 6.6 Grid specification

Explain how a grid is specified and what these tags mean

```
<Level>
    <Box label="1">
       <lower>        [0,0,0]          </lower>
       <upper>        [5,5,5]          </upper>
       <extraCells>   [1,1,1]          </extraCells>
       <patches>      [1,1,1]          </patches>
    </Box>
    <spacing>         [0.5,0.5,0.5]    </spacing>
</Level>
```

## 6.7 Adapative Mesh Refinement

- need to discuss the input options for the different regridders.
- How is a cell flagged as needing to be refined

## 6.8 load Balancer

- to be filled in

## 6.9 uda

- discuss the directory structure of an uda - discuss how to modify an input parameter before you restart an uda.

## 6.10 Visualization tools

- VisIT
- manta?

## 6.11 Tools

- puda, lineextract, timeextract, compare_uda

### 6.11.1 plotting tools

- plotStats
- plotRegridder
- plotCPU_usage
- plotComponents

## 6.12 Code

-explain the basic directory structure of src

```
|-- CCA
|-- Components
|   |-- Angio
|   |-- Arches
|   |-- DataArchiver
|   |-- Examples
|   |-- ICE
|   |-- LoadBalancers
|   |-- MPM
|   |-- MPMArches
|   |-- MPMICE
|   |-- Models
|   |-- OnTheFlyAnalysis
|   |-- Parent
|   |-- PatchCombiner
|   |-- ProblemSpecification
|   |-- Regridder
|   |-- Schedulers
|   |-- SimulationController
|   |-- Solvers
|   |-- SpatialOps
|   `-- SwitchingCriteria
```

```
‘-- Ports
|-- Core
|-- R_Tester
|-- StandAlone
|-- Teem
|-- VisIt
|-- build_scripts
|-- include
|-- on-the-fly-libs
|-- orderAccuracy
|-- osx
|-- scripts
|-- tau
|-- testprograms
‘-- tools
```

# 7 Arches

## 7.1 Introduction

The Arches component solves a low-mach formulation with a pressure projection of the Navier-Stokes equations for turbulent, variable density, reacting flows. Turbulent flow scales are modeled using a Large Eddy Simulation (LES) approach. Chemistry is handled using chemistry parameterization and closure models for subgrid scale mixing. Modes of heat transfer include radiation. blah blah blah

## 7.2 Theory - Algorithm Description

The essential governing equations for the Arches component, written in finite volume form, include the mass balance, momentum balance, mixture fraction balance, and energy balance equations. Using a bold-face symbol to represent a vector quantity, the equations are:

1. The mass balance,

$$\int_V \frac{\partial \rho}{\partial t} dV + \oint_S \rho \mathbf{u} \cdot d\mathbf{S} = 0 \, , \tag{1}$$

   where $\rho$ is density and $\mathbf{u}$ is the velocity vector.

2. The momentum balance,

$$\int_V \frac{\partial \rho \mathbf{u}}{\partial t} dV + \oint_S \rho \mathbf{u}\mathbf{u} \cdot d\mathbf{S} = \oint_S \tau \cdot d\mathbf{S} - \int_V \nabla p dV + \int_V \rho \mathbf{g} dV \, , \tag{2}$$

   where $\tau$ is the deviatoric stress tensor defined as $\tau_{ij} = 2\mu S_{ij} - \frac{2}{3}\mu \frac{\partial u_k}{\partial x_k}\delta_{ij}$, the second isotropic term in $\tau_{ij}$ is absorbed into the pressure projection for the current low-Mach scheme, and $S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)$. Also in Equation 2, $\mathbf{g}$ is the gravitational body force and $p$ is pressure.

3. The mixture fraction balance,

$$\int_V \frac{\partial \rho f}{\partial t} dV + \oint_S \rho \mathbf{u} f \cdot d\mathbf{S} = \oint_S D \nabla f \cdot d\mathbf{S} \, , \tag{3}$$

   where $f$ is the mixture fraction and a Fick's law form of the diffusion term assuming equal diffusivities results in a single diffusion coefficient, $D$.

4. The thermal energy balance,

$$\int_V \frac{\partial \rho h}{\partial t} dV + \oint_S \rho \mathbf{u} h \cdot d\mathbf{S} = \oint_S k \nabla h \cdot d\mathbf{S} - \oint_S q \cdot d\mathbf{S} \ , \qquad (4)$$

where $h$ is the sum of the chemical plus sensible enthalpy, $q$ is the radiative flux, a Fourier's law form of the conduction term is used with a diffusion coefficient, $k$, and the pressure term is neglected.

These equations are solved in an LES context, meaning filters are applied to the equations. Here, we use Favre filtering, defined as

$$\overline{\phi} = \frac{\overline{\rho \phi}}{\overline{\rho}},$$

to isolate the density in the filtered equations. The filtering operations result in the classic turbulence closure problem and thus models are required. The model options are for Arches are discussed below.

The set of filtered equations are discretized in space and time and solved on a staggered, finite volume mesh. The staggering scheme consists of four offset grids. One grid stores the scalar quantities and the remaining three grids store each component of the velocity vector. The velocity components are situated so that the center of their control volume is located on the face centers of the scalar grid in their respective direction.

The equations are solved in an explicit manner.

## 7.3 Uintah Specification

### 7.3.1 Basic Inputs

### 7.3.2 Turbulence

### 7.3.3 Properties

### 7.3.4 BoundaryConditions

### 7.3.5 Physical Constants

### 7.3.6 Solvers

## 7.4 Examples

## 7.5 References

# 8 ICE

## 8.1 Introduction

finite volume first order tempral discretizatioon all speed with the option of explicit or semi-implicit

## 8.2 Theory - Algorithm Description

# 9 Introduction

The work presented here describes a numerical approach for "full physics" simulations of dynamic fluid structure interactions involving large deformations and material transformations (e.g., phase change). "Full physics" refers to problems involving strong interactions between the fluid field and solid field temperatures and velocities, with a full Navier Stokes representation of fluid materials and the transient, nonlinear response of solid materials. These interactions may include chemical or physical transformation between the solid and fluid fields.

Approaches to fluid structure interaction (FSI) problems are typically divided into two classes. "Separated" approaches treat individual materials as occupying distinct regions of space, with interactions occurring only at material interfaces. The details of those interactions vary between implementations, and are often a function of the degree, or "strength" of the coupling between the fluid and solid fields. Because of the separated nature of the materials, only one set of state variables is needed at any point in space, since only one material is allowed to exist at that point. "Averaged" model approaches allow **all** materials to exist at any point in space with some probability. Variables describing the material state vary continuously throughout the computational domain, thus, the state of every material is defined at every point in space. Distinct material interfaces are not defined, rather the interaction between materials is computed in an average sense, and, as such, interactions among materials may take place anywhere.

While both the separated and averaged model approaches have their respective merits, the averaged model, when carried out on an Eulerian grid, allows arbitrary distortion of materials and material interfaces. However, these distortions can be catastrophic for the solid material, as the deformation history of the solid must be transported through the Eulerian grid. This

transport can lead to non-physical stresses and the interface between materials is also subject to diffusion. The latter problem can be mitigated via surface tracking and the use of a single valued velocity field [**?**, **?**], but this does not eliminate the problems of stress transport.

The approach described here uses the averaged model approach and addresses the issue of stress transport by integrating the state of the solid field in the "material" frame of reference through use of the Material Point Method (MPM) [**?**, **?**]. MPM is a particle method for solid mechanics that allows the solid field to undergo arbitrary distortion. Because the fluid state is integrated in the Eulerian frame, it can also undergo arbitrary distortion. MPM uses a computational "scratchpad" grid to advance the solution to the equations of motion, and by choosing to use the same grid used in the Eulerian frame of reference, interactions among the materials are facilitated on this common computational framework. By choosing to use an infinitely fast rate of momentum transfer between the materials, the single velocity field limit is obtained, and the interface between materials is limited to, at most, a few cells. Thus, in the differential limit, the separated model can be recovered. This means that with sufficient grid resolution, the accuracy of the separated model and the robustness of the averaged model can be enjoyed simultaneously.

The theoretical and algorithmic basis for the fluid structure interaction simulations presented here is based on a body of work of several investigators at Los Alamos National Laboratory, primarily Bryan Kashiwa, Rick Rauenzahn and Matt Lewis. Several reports by these researchers are publicly available and are cited herein. It is largely through our personal interactions that we have been able to bring these ideas to bear on the simulations described herein.

An exposition of the governing equations is given in the next section, followed by an algorithmic description of the solution of those equations. This description is first done separately for the materials in the Eulerian and Lagrangian frames of reference, before details associated with the integrated approach are given. Because this manuscript is focused on explosions of energetic devices, some of the models used to close the governing equations are described briefly. Finally, results from three calculations are presented. The first two of these are intended to serve as validation of the general approach and the models used, while the third is an unvalidated demonstration calculation. The reader is encouraged to browse Section **??** at this point to better appreciate the direction that the subsequent development is headed.

# 10 Governing Equations

The governing multi-material model equations are stated and described, but not developed, here. Their development can be found in [**?**]. Here, our intent is to identify the quantities of interest, of which there are 8, as well as those equations (or closure models) which govern their behavior. Consider a collection of $N$ materials, and let the subscript r signify one of the materials, such that $\mathrm{r} = 1, 2, 3, \ldots, N$. In an arbitary volume of space $V(\mathbf{x}, t)$, the averaged thermodynamic state of a material is given by the vector $[M_\mathrm{r}, \mathbf{u}_\mathrm{r}, e_\mathrm{r}, T_\mathrm{r}, v_\mathrm{r}, \theta_\mathrm{r}, \boldsymbol{\sigma}_\mathrm{r}, p]$, the elements of which are the r-material mass, velocity, internal energy, temperature, specific volume, volume fraction, stress, and the equilibration pressure. The r-material averaged density is $\rho_\mathrm{r} = M_\mathrm{r}/V$. The rate of change of the state in a volume moving with the velocity of r-material is:

$$\frac{1}{V}\frac{D_\mathrm{r} M_\mathrm{r}}{Dt} = \textstyle\sum_{s=1}^{N} \Gamma_\mathrm{rs} \tag{5}$$

$$\frac{1}{V}\frac{D_\mathrm{r}(M_\mathrm{r}\mathbf{u}_\mathrm{r})}{Dt} = \theta_\mathrm{r}\boldsymbol{\nabla}\cdot\boldsymbol{\sigma} + \boldsymbol{\nabla}\cdot\theta_\mathrm{r}(\boldsymbol{\sigma}_\mathrm{r} - \boldsymbol{\sigma}) + \rho_\mathrm{r}\mathbf{g} + \textstyle\sum_{s=1}^{N}\mathbf{f}_\mathrm{rs} + \sum_{s=1}^{N}\mathbf{u}_\mathrm{rs}^{+}\Gamma_\mathrm{rs} \tag{6}$$

$$\frac{1}{V}\frac{D_\mathrm{r}(M_\mathrm{r} e_\mathrm{r})}{Dt} = -\rho_\mathrm{r} p\frac{D_\mathrm{r} v_\mathrm{r}}{Dt} + \theta_\mathrm{r}\boldsymbol{\tau}_\mathrm{r} : \nabla\mathbf{u}_\mathrm{r} - \boldsymbol{\nabla}\cdot\mathbf{j}_\mathrm{r} + \textstyle\sum_{s=1}^{N} q_\mathrm{rs} + \sum_{s=1}^{N} h_\mathrm{rs}^{+}\Gamma_\mathrm{rs} \tag{7}$$

Equations (5-7) are the averaged model equations for mass, momentum, and internal energy of r-material, in which $\boldsymbol{\sigma}$ is the mean mixture stress, taken here to be isotropic, so that $\boldsymbol{\sigma} = -p\mathbf{I}$ in terms of the hydrodynamic pressure $p$. The effects of turbulence have been explicitly omitted from these equations, and the subsequent solution, for the sake of simplicity. However, including the effects of turbulence is not precluded by either the model or the solution method used here.

In Eq. (6) the term $\sum_{s=1}^{N}\mathbf{f}_\mathrm{rs}$ signifies a model for the momentum exchange among materials. This term results from the deviation of the r-field stress from the mean stress, averaged, and is typically modeled as a function of the relative velocity between materials at a point. (For a two material problem this term might look like $\mathbf{f}_{12} = K_{12}\theta_1\theta_2(\mathbf{u}_1 - \mathbf{u}_2)$ where the coefficient $K_{12}$ determines the rate at which momentum is transferred between materials). Likewise, in Eq. (7), $\sum_{\mathrm{s}=1}^{N} q_\mathrm{rs}$ represents an exchange of heat energy among materials. For a two material problem $q_{12} = H_{12}\theta_1\theta_2(T_2 - T_1)$ where $T_\mathrm{r}$ is the r-material temperature and the coefficient $H_\mathrm{rs}$ is analogous to a convective

heat transfer rate coefficient. The heat flux is $\mathbf{j}_\mathrm{r} = -\rho_\mathrm{r} b_\mathrm{r} \boldsymbol{\nabla} T_\mathrm{r}$ where the thermal diffusion coefficient $b_\mathrm{r}$ includes both molecular and turbulent effects (when the turbulence is included).

In Eqs. (5-7) the term $\Gamma_\mathrm{rs}$ is the rate of mass conversion from s-material into r-material, for example, the burning of a solid reactant into gaseous products. The rate at which mass conversion occurs is governed by a reaction model, two examples of which are given in Section **??**. In Eqs. (6) and (7), the velocity $\mathbf{u}^+_\mathrm{rs}$ and the enthalpy $h^+_\mathrm{rs}$ are those of the s-material that is converted into r-material. These are simply the mean values associated with the donor material.

The temperature $T_\mathrm{r}$, specific volume $v_\mathrm{r}$, volume fraction $\theta_\mathrm{r}$, and hydrodynamic pressure $p$ are related to the r-material mass density, $\rho_\mathrm{r}$, and specific internal energy, $e_\mathrm{r}$, by way of equations of state. The four relations for the four quantites $(T_\mathrm{r}, v_\mathrm{r}, \theta_\mathrm{r}, p)$ are:

$$
\begin{aligned}
e_\mathrm{r} &= e_\mathrm{r}(v_\mathrm{r}, T_\mathrm{r}) && (8)\\
v_\mathrm{r} &= v_\mathrm{r}(p, T_\mathrm{r}) && (9)\\
\theta_\mathrm{r} &= \rho_\mathrm{r} v_\mathrm{r} && (10)\\
0 &= 1 - \textstyle\sum_{s=1}^{N} \rho_\mathrm{s} v_\mathrm{s} && (11)
\end{aligned}
$$

Equations (8) and (9 )are, respectively, the caloric and thermal equations of state. Equation (10) defines the volume fraction, $\theta$, as the volume of r-material per total material volume, and with that definition, Equation (11), referred to as the multi-material equation of state, follows. It defines the unique value of the hydrodynamic pressure $p$ that allows arbitrary masses of the multiple materials to identically fill the volume $V$. This pressure is called the "equilibration" pressure [**?**].

A closure relation is still needed for the material stress $\boldsymbol{\sigma}_\mathrm{r}$. For a fluid $\boldsymbol{\sigma}_\mathrm{r} = -p\mathbf{I} + \boldsymbol{\tau}_\mathrm{r}$ where the deviatoric stress is well known for Newtonian fluids. For a solid, the material stress is the Cauchy stress. The Cauchy stress is computed using a solid constitutive model and may depend on the the rate of deformation, the current state of deformation ($\mathbf{E}$), the temperature, and possibly a number of history variables. Such a relationship may be expressed as:

$$
\boldsymbol{\sigma}_\mathrm{r} \equiv \boldsymbol{\sigma}_\mathrm{r}(\boldsymbol{\nabla}\mathbf{u}_\mathrm{r}, \mathbf{E}_\mathrm{r}, T_\mathrm{r}, \ \ldots) \qquad (12)
$$

The approach described here imposes no restrictions on the types of constitutive relations that can be considered. More specific discussion of some of the models used in this work is found in Sec. **??**

Equations (5-12) form a set of eight equations for the eight-element state vector
$[M_{\rm r}, \mathbf{u}_{\rm r}, e_{\rm r}, T_{\rm r}, v_{\rm r}, \theta_{\rm r}, \boldsymbol{\sigma}_{\rm r}, p]$, for any arbitrary volume of space $V$ moving with the r-material velocity. The approach described here uses the reference frame most suitable for a particular material type. As such, there is no guarantee that arbitrary volumes will remain coincident for materials described in different reference frames. This problem is addressed by treating the specific volume as a dynamic variable of the material state which is integrated forward in time from initial conditions. In so doing, at any time, the total volume associated with all of the materials is given by:

$$V_t = \Sigma_{r=1}^{N} M_{\rm r} v_{\rm r} \tag{13}$$

so the volume fraction is $\theta_{\rm r} = M_{\rm r} v_{\rm r} / V_t$ (which sums to one by definition). An evolution equation for the r-material specific volume, derived from the time variation of Eqs. (8-11), has been developed in [**?**]. It is stated here as:

$$\begin{aligned} \frac{1}{V}\frac{D_{\rm r}(M_{\rm r} v_{\rm r})}{Dt} = f_{\rm r}^{\theta} \boldsymbol{\nabla}\cdot\mathbf{u} \quad &+ \quad \left[v_{\rm r}\Gamma_{\rm r} - f_{\rm r}^{\theta}\Sigma_{s=1}^{N} v_{\rm s}\Gamma_{\rm s}\right] \\ &+ \quad \left[\theta_{\rm r}\beta_{\rm r}\frac{D_{\rm r}T_{\rm r}}{Dt} - f_{\rm r}^{\theta}\Sigma_{s=1}^{N}\theta_{\rm s}\beta_{\rm s}\frac{D_{\rm s}T_{\rm s}}{Dt}\right] . \end{aligned} \tag{14}$$

where $f_{\rm r}^{\theta} = \frac{\theta_{\rm r}\kappa_{\rm r}}{\sum_{\rm s=1}^{N}\theta_{\rm s}\kappa_{\rm s}}$, and $\kappa_{\rm r}$ is the r-material bulk compressibility.

The evaluation of the multi-material equation of state (Eq. (11)) is still required in order to determine an equilibrium pressure that results in a common value for the pressure, as well as specific volumes that fill the total volume identically.

# 11 Numerical Implementation

A description of the means by which numerical solutions to the equations in Section 10 are found is presented next. This begins with separate, brief, overviews of the methodologies used for the Eulerian and Lagrangian reference frames. The algorithmic details necesssary for integrating them to achieve a tightly coupled fluid-structure interaction capability is provided in Sec. **??**.

## 11.1 Eulerian Multi-Material Method

The Eulerian method implemented here is a cell-centered, finite volume, multi-material version of the ICE (for Implicit, Continuous fluid, Eulerian) method [**?**] developed by Kashiwa and others at Los Alamos National Laboratory [**?**]. "Cell-centered" means that all elements of the state are colocated at the grid cell-center (in contrast to a staggered grid, in which velocity components may be centered at the faces of grid cells, for example). This colocation is particularly important in regions where a material mass is vanishing. By using the same control volume for mass and momentum it can be assured that as the material mass goes to zero, the mass and momentum also go to zero at the same rate, leaving a well defined velocity. The technique is fully compressible, allowing wide generality in the types of problems that can be addressed.

Our use of the cell-centered ICE method employs time splitting: first, a Lagrangian step updates the state due to the physics of the conservation laws (i.e., right hand side of Eqs. 5-7); this is followed by an Eulerian step, in which the change due to advection is evaluated. For solution in the Eulerian frame, the method is well developed and described in [**?**].

In the mixed frame approach used here, a modification to the multi-material equation of state is needed. Equation (11) is unambiguous when all materials are fluids or in cases of a flow consisting of dispersed solid grains in a carrier fluid. However in fluid-structure problems the stress state of a submerged structure may be strongly directional, and the isotropic part of the stress has nothing to do with the hydrodynamic (equilibration) pressure $p$. The equilibrium that typically exists between a fluid and a solid is at the interface between the two materials: there the normal part of the traction equals the pressure exerted by the fluid on the solid over the interface. Because the orientation of the interface is not explicitly known at any point (it is effectively lost in the averaging) such an equilibrium cannot be computed
.

The difficulty, and the modification that resolves it, can be understood by considering a solid material in tension coexisting with a gas. For solid materials, the equation of state is the bulk part of the constitutive response (that is, the isotropic part of the Cauchy stress versus specific volume and temperature). If one attempts to equate the isotropic part of the stress with the fluid pressure, there exist regions in pressure-volume space for which Eq. (11) has no physical solutions (because the gas pressure is only positive).

This can be seen schematically in Fig. **??**, which sketches equations of state for a gas and a solid, at an arbitrary temperature.

Recall that the isothermal compressiblity is the negative slope of the specific volume versus pressure. Embedded structures considered here are solids and, at low pressure, possess a much smaller compressibility than the gasses in which they are submerged. Nevertheless the variation of condensed phase specific volume can be important at very high pressures, where the compressibilities of the gas and condensed phase materials can become comparable (as in a detonation wave, for example). Because the speed of shock waves in materials is determined by their equations of state, obtaining accurate high pressure behavior is an important goal of our FSI studies.

To compensate for the lack of directional information for the embedded surfaces, we evaluate the solid phase equations of state in two parts. Above a specified postive threshold pressure (typically 1 atmosphere), the full equation of state is respected; below that threshold pressure, the solid phase pressure follows a polynomial chosen to be $C^1$ continuous at the threshold value and which approaches zero as the specific volume becomes large. The effect is to decouple the solid phase specific volume from the stress when the isotropic part of the stress falls below a threshold value. In regions of coexistence at states below the threshold pressure, $p$ tends to behave according to the fluid equation of state (due to the greater compressibility) while in regions of pure condensed phase material $p$ tends rapidly toward zero and the full material stress dominates the dynamics as it should.

## 11.2 Uintah Specification

### 11.2.1 Basic Inputs

The ICE component is selected by specifying: Each Uintah component is invoked using a single executable called *sus* , which chooses the type of simulation to execute based on the *SimulationComponent* tag in the input file. In the case of ICE simulations, this looks like:

```
<SimulationComponent type="ice" />
```

near the top of the inputfile. The system of units **must**be to be consistent (mks, cgs) and the majority of input files will be in meter-kilogram-sec system.

### 11.2.2 Physical Constants

```
<PhysicalConstants>
   <gravity>            [0,0,0]   </gravity>
   <reference_pressure> 101325.0  </reference_pressure>
</PhysicalConstants>
```

### 11.2.3 Material Properties

For each ICE material the thermodynamic and transport properties must be specified, in addition to the initial conditions of the fluid inside of each geom_object. Below is the an example of how to specify an invisid ideal gas over square region with dimensions $6mX6mX6m$. The initial conditions of the gas in that region are $T = 300, \rho = 1.179, v_x = 1, v_y = 2, v_z = 3$ (Note, the pressure XML tag is not used as an initial condition and is simply there to make the user aware of what the pressure would be at that thermodynamic state.)

```
<MaterialProperties>
   <ICE>
     <material>
       <EOS type = "ideal_gas">                 </EOS>
       <dynamic_viscosity>   0.0                </dynamic_viscosity>
       <thermal_conductivity>0.0                </thermal_conductivity>
       <specific_heat>      716.0               </specific_heat>
       <gamma>               1.4                </gamma>
       <geom_object>
           <box label="wholeDomain">
               <min>       [ 0.0, 0.0, 0.0 ]  </min>
               <max>       [ 6.0, 6.0, 6.0 ]  </max>
           </box>
           <res>                 [2,2,2]      </res>
           <velocity>      [1.,2.,3.]         </velocity>
           <density>       1.1792946927374306 </density>
           <pressure>      101325.0           </pressure>
           <temperature>   300.0              </temperature>
       </geom_object>
     </material>
  </ICE>
</MaterialProperties>
```

### 11.2.4 Equation of State

Below is a list of the various equations of state, along with the user defined constants, that are available. The reader should consult the literature for the theoretical development and applicability of the equations of state to the problem being solved. The most commonly used EOS is the ideal gas law

$$p = (\gamma - 1)c_v \rho T \tag{15}$$

and is specified in the input file with:

```
<EOS type="ideal_gas"/>
```

The Thomsen Hartka EOS for cold liquid water (1-100 atm pressure range) is specified with [**?**, **?**]

```
<EOS type="Thomsen_Hartka_water">
  <a>  2.0e-7     </a>    <!-- (K/Pa)     -->
  <b>  2.6        </b>    <!-- (J/kg K^2) -->
  <co> 4205.7     </co>   <!-- (J/Kg K)   -->
  <ko> 5.0e-10    </ko>   <!-- (1/Pa)     -->
  <To> 277.0      </To>   <!-- (K)        -->
  <L>  8.0e-6     </L>    <!-- (1/K^2)    -->
  <vo> 1.00008e-3 </vo>   <!-- (m^3/kg)   -->
</EOS>
```

The input specification for the "JWLC", "JWL++" and "Murnahan" equations of state from [**?**] are:

```
<EOS type = "JWLC">
  <A>   2.9867e11   </A>
  <B>   4.11706e9   </B>
  <C>   7.206147e8  </C>
  <R1>    4.95      </R1>
  <R2>    1.15      </R2>
  <om>    0.35      </om>
  <rho0>  1160.0    </rho0>
</EOS>

<EOS type = "JWL">
  <A>     1.6689e12 </A>
  <B>     5.969e10  </B>
  <R1>      5.9     </R1>
  <R2>      2.1     </R2>
  <om>      0.45    </om>
  <rho0>  1835.0    </rho0>
</EOS>
```

```
<EOS type = "Murnahan">
  <n>     7.4       </n>
  <K>     39.0e-11  </K>
  <rho0>  1160.0    </rho0>
  <P0>    101325.0  </P0>
</EOS>
```

The "hard sphere" or "Abel" equation of state for dense gases is

$$p(v - b) = RT \tag{16}$$

where b corresponds to the volume occupied by the molecules themselves [**?**]. Input parameters are specified using:

```
<EOS type="hard_sphere_gas">
   <b> 1.4e-3 </b>
</EOS>
```

Non-idea gas equation of state used in HMX combustion simulations the Twu-Sim-Tassone(TST) EOS is

$$p = \frac{(\gamma - 1)c_v T}{v - b} - \frac{a}{(v + 3.0b)(v - 0.5b)} \tag{17}$$

Input parameters are specified using:

```
<EOS type="TST">
  <a>       -260.1385968     </a>
  <b>       7.955153678e-4   </b>
  <u>       -0.5             </u>
  <w>       3.0              </w>
  <Gamma>   1.63             </Gamma>
</EOS>
```

The input parameters for the Tillotson equation of state [**?**] for soils :

```
<EOS type = "Tillotson">
  <a>       .5      </a>
  <b>       1.3     </b>
  <A>       4.5e9   </A>
  <B>       3.0e9   </B>
  <E0>      6.e6    </E0>
  <Es>      3.2e6   </Es>
  <Esp>     18.0e6  </Esp>
  <alpha>   5.0     </alpha>
  <beta>    5.0     </beta>
  <rho0>    1700.0  </rho0>
</EOS>
```

### 11.2.5 Exchange Properties

The heat and momentum exchange coefficients $K_{rs}$ and $H_{rs}$, which determine the rate at which momentum and heat are transferred between materials, are specified in the following format.

```
0->1,   0->2,  0->3
        1->2,  1->3
               2->3
```

For a two material problem the coefficients would be:

```
<exchange_properties>
   <exchange_coefficients>
      <momentum>  [0, 1e15, 1e15 ]     </momentum>
      <heat>      [0, 1e10, 1e10 ]     </heat>
   </exchange_coefficients>
</exchange_properties>
```

### 11.2.6 BoundaryConditions

Boundary conditions must be specified on each face of the computational domain $(x^-, x^+, y^-, y^+, z^-, z^+)$ for the variables $P, \mathbf{u}, \mathbf{T}, \rho, \mathbf{v}$ for each material. The three main types of numerical boundary conditions that can be applied are "Neumann", "Dirichlet" and "Symmetric". A Neumann boundary condition is used to set the gradient or $\frac{\partial q}{\partial n}|_{surface} = value$ at the boundary. The value of the primative variable in the boundary cell is given by,

$$q[\text{boundary cell}] = q[\text{interior cell}] - value * dn; \tag{18}$$

if we use a first order upwind discretization of the gradient. Dirichlet boundary conditions set the value of primative variable in the boundary cell using

$$q[\text{boundary cell}] = value; \tag{19}$$

```
<Grid>
  <BoundaryConditions>
    <Face side = "x-">
      <BCType id = "0"   label = "Pressure"     var = "Neumann">
                            <value> 0. </value>
      </BCType>
      <BCType id = "all" label = "Velocity"     var = "Neumann">
                            <value> [0.,0.,0.] </value>
      </BCType>
```

```
      <BCType id = "all" label = "Temperature"  var = "Neumann">
                           <value> 0.0 </value>
      </BCType>
      <BCType id = "all" label = "Density"      var = "Neumann">
                           <value> 0.0 </value>
      </BCType>
      <BCType id = "all" label = "SpecificVol"  var = "computeFromDensity">
                           <value> 0.0  </value>
      </BCType>
    </Face>
      .
      [other faces]
      .
  </BoundaryConditions>
</Grid>
```

There is also the field `id = "all"` . In principal, one could set different boundary condition types for different materials. In practice, this is rarely used, so the usage illustrated here should be used. Note that pressure field id is always 0.Symmetric boundary conditions are set using:

```
<Face side = "y-">
    <BCType id = "all" label = "Symmetric" var = "symmetry"> </BCType>
</Face>
```

In addition to "Dirichlet", "Neumann", and "Symmetric" type boundary conditions ICE has several custom or experimental boundary conditions the user can access. The "Sine" boundary condition was designed to impose a pulsating pressure wave in the boundary cells by applying

$$p = p_{reference} + Asin(\omega t) \tag{20}$$

The input file parameters that control the frequency and magnitude of the wave are:

```
<SINE_BC>
  <omega>    1000 </omega>
  <A>        800 </A>
</SINE_BC>
```

and to specify them add

```
<BCType id = "0"   label = "Pressure"     var = "Sine">
                      <value> 0.0 </value>
```

```
</BCType>
<BCType id = "0"   label = "Temperature"  var = "Sine">
                      <value> 0.0 </value>
</BCType>
```

to the input file. For non-reflective boundary conditions the user should specify the "LODI" or locally one-dimensional invisid type [**?**]

```
<LODI>
  <press_infinity> 1.0132500000010138e+05  </press_infinity>
  <sigma>          0.27                    </sigma>
  <ice_material_index> 0                   </ice_material_index>
</LODI>
```

and

```
<Face side = "x+">
  <BCType id = "0"   label = "Pressure"     var = "LODI">
                        <value> 0. </value>
  </BCType>
  <BCType id = "0"   label = "Velocity"     var = "LODI">
                        <value> [0.,0.,0.] </value>
  </BCType>
  <BCType id = "0"   label = "Temperature"  var = "LODI">
                        <value> 0.0 </value>
  </BCType>
  <BCType id = "0"   label = "Density"      var = "LODI">
                        <value> 0.0 </value>
  </BCType>
  <BCType id = '0' label = "SpecificVol" var = "computeFromDensity">
                        <value> 0.0 </value>
  </BCType>
</Face>
```

This boundary condition is designed to suppress all the unwanted effects of an artifical boundary. **This BC is computationally expensive and only partially works and should be used with caution**. In flow fields where there are no passing through the outlet of the domain it reduces the reflected pressure waves significantly.

### 11.2.7 Semi-Implicit Pressure Solve

The equation for the change in the pressure field $\Delta P$ during a given timestep is given by

$$\frac{dP}{dt} = \frac{\sum_{m=1}^{N} \frac{\dot{m}}{V\rho_m^o} - \sum_{m=1}^{N} \nabla \cdot \widehat{\theta_m \vec{U_m}}^{*f}}{\sum_{m=1}^{N} \frac{\theta_m}{\rho_m^o c_m^2}} \tag{21}$$

which can be written in matrix form $Ax = b$ and solved with a linear solver. Details on the notation, discretization of Eq. 21 and the formation of $A$ and $b$ can be found in

```
src/CCA/Components/ICE/Docs/implicitPressSolve.pdf
```

The linear system $Ax = b$ can be solved using the default Uintah:conjugate gradient solver (cg) (slow) or one of the many that are available through the scalable linear solvers and preconditioner package hypre [**?**]. Experience has shown that the most efficient hypre preconditioner and solver are the pfmg and cg respectively. Below are typical values for both the Uintah:cg and hypre:cg solver

```
<ImplicitSolver>
   <max_outer_iterations>         20    </max_outer_iterations>
   <outer_iteration_tolerance>    1e-8  </outer_iteration_tolerance>
   <iters_before_timestep_restart> 5    </iters_before_timestep_restart>
   <Parameters variable="implicitPressure">

    <tolerance>     1.e-10  </tolerance>

    <!-- CGSolver options -->
    <norm>     LInfinity  </norm>
    <criteria> Absolute   </criteria>

    <!-- Hypre options -->
    <solver>         cg      </solver>
    <preconditioner> pfmg    </preconditioner>
    <maxiterations>  7500    </maxiterations>
    <npre>           1       </npre>
    <npost>          1       </npost>
    <skip>           0       </skip>
    <jump>           0       </jump>
   </Parameters>
</ImplicitSolver>
```

If the user is interested in altering the tolerance to which the equations are solved they should look at

`<tolerance> and <outer\_iteration\_tolerance>`

| XML tag | Description |
|---|---|
| max_outer_iterations | maximum number of iterations in the outer loop of the pressure solve. |
| outer_iteration_tolerance | tolerance XXXXDX |
| iters_before_timestep_restart | number of outer iterations before a timestep is restarted |
| tolerance | XXXX |

Table below shows some of the

### 11.2.8 XML tag description

| XML tag | Type | Dimensions | Description |
|---|---|---|---|
| cfl | double | | Courant Number. |
| gravity | Vector | $[L/t^2]$ | gravitational acceleration, $\vec{g}$. |
| global material properties | | | |
| dynamic_viscosity | double | $[M/Lt]$ | viscosity, $\mu$. |
| thermal_conditucivity | double | $[ML/t^3T]$ | thermal conductivity, $k$ |
| specific_heat | double | $[L^2/t^2T]$ | $c_p$ |
| gamma | double | | ratio of specific heats, $\gamma$. |
| geometry object related | | | |
| res | vector | | resolution used for defining geometry objects. |
| velocity | vector | $[L/t]$ | initial velocity, $\vec{u}$. |
| density | double | $[M/L^3]$ | initial density, $\rho$. |
| temperature | double | $[T]$ | initial temperature, $T$. |
| pressure | double | | Not used. |
| AMR Parameters | | | |
| orderOfInterpolation | integer | | Order of interpolation at the coarse/fine interfaces. |
| do_Refluxing | boolean | | on/off switch for correcting the flux of mass, momentum, and energy at the course/fine interfaces. |

## 11.3 Examples

# Shock Tube

## Problem Description

The shock tube problem is a standard 1D compressible flow problem that has been used by many as a validation test case [**?**, **?**, **?**]. At time $t = 0$ the computational domain is divided into two separate regions of space by a diaphram, with each region at a different density and pressure. The separated regions are at rest with a uniform temperature $= 300K$. The initial pressure ratio is $\frac{P_R}{P_L} = 10$ and density ratio is $\frac{\rho_R}{\rho_L} = 0.1$ The diaphram is instantly removed and a traveling shockwave, discontinutity and expansion fan form. The expansion fan moves towards the left while the shockwave and contact discontinutity move to the right. This problem tests the algorithm's ability to capture steep gradients and solve Eulers equations.

## Simulation Specifics

**Component used:** ICE

**Input file name:** rieman_sm.ups

**Command used to run input file:** sus inputs/UintahRelease/ICE/shockTube.ups

**Postprocessing command:**
inputs/UintahRelease/ICE/plot_shockTube_1L shockTube.uda y
This will generate a postscript file shockTube.ps

**Simulation Domain:** 1 x .001 x .001 m

**Cell Spacing:**
1 x 1 x 1 mm (Level 0)

**Example Runtimes:**
1 minute (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.005 sec.

# Results

Figure 1 shows a comparison of the exact versus simulated results at time $t = 5msec$.

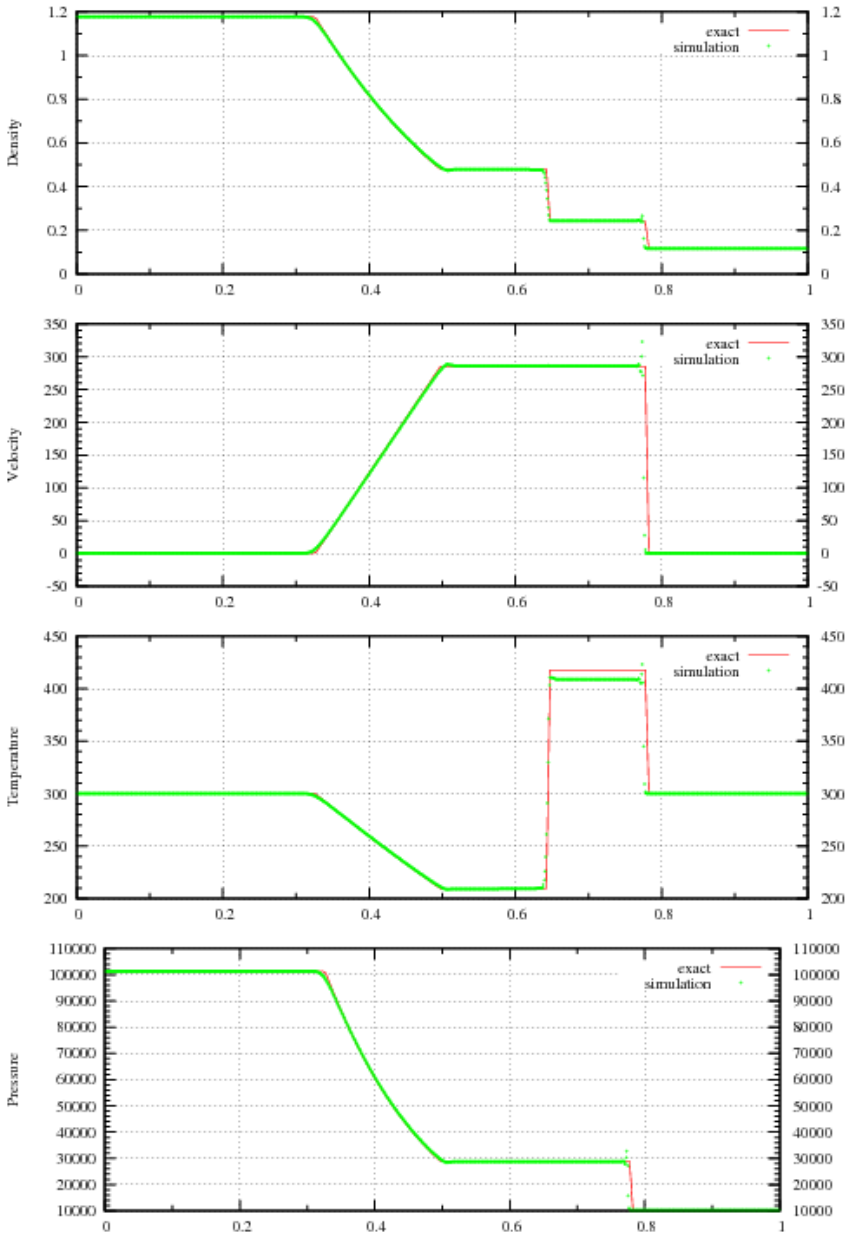


Figure 1: Shock tube results at time $t = 5msec$

# Shock Tube with Adaptive Mesh Refinement

## Simulation Specifics

**Component used:** ICE

**Input file name:** shocktube_AMR.ups

**Command used to run input file:** sus
inputs/UintahRelease/ICE/shocktube_AMR.ups

**Postprocessing command:**
inputs/UintahRelease/ICE/plot_shockTube_AMR shockTube_AMR.uda y
This will generate a postscript file shockTube_AMR.ps

**Simulation Domain:** 1 x .001 x .001 m

**Cell Spacing:**
10 x 1 x 1 mm (Level 0)
2.5 x 1 x1 mm (Level 1)
0.625 x1 x1 mm (Level 2)

**Example Runtimes:**
2ish minutes (1 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.005 sec.

# Results

Figure 2 shows a comparison of the exact versus simulated results at time $t = 5msec.$

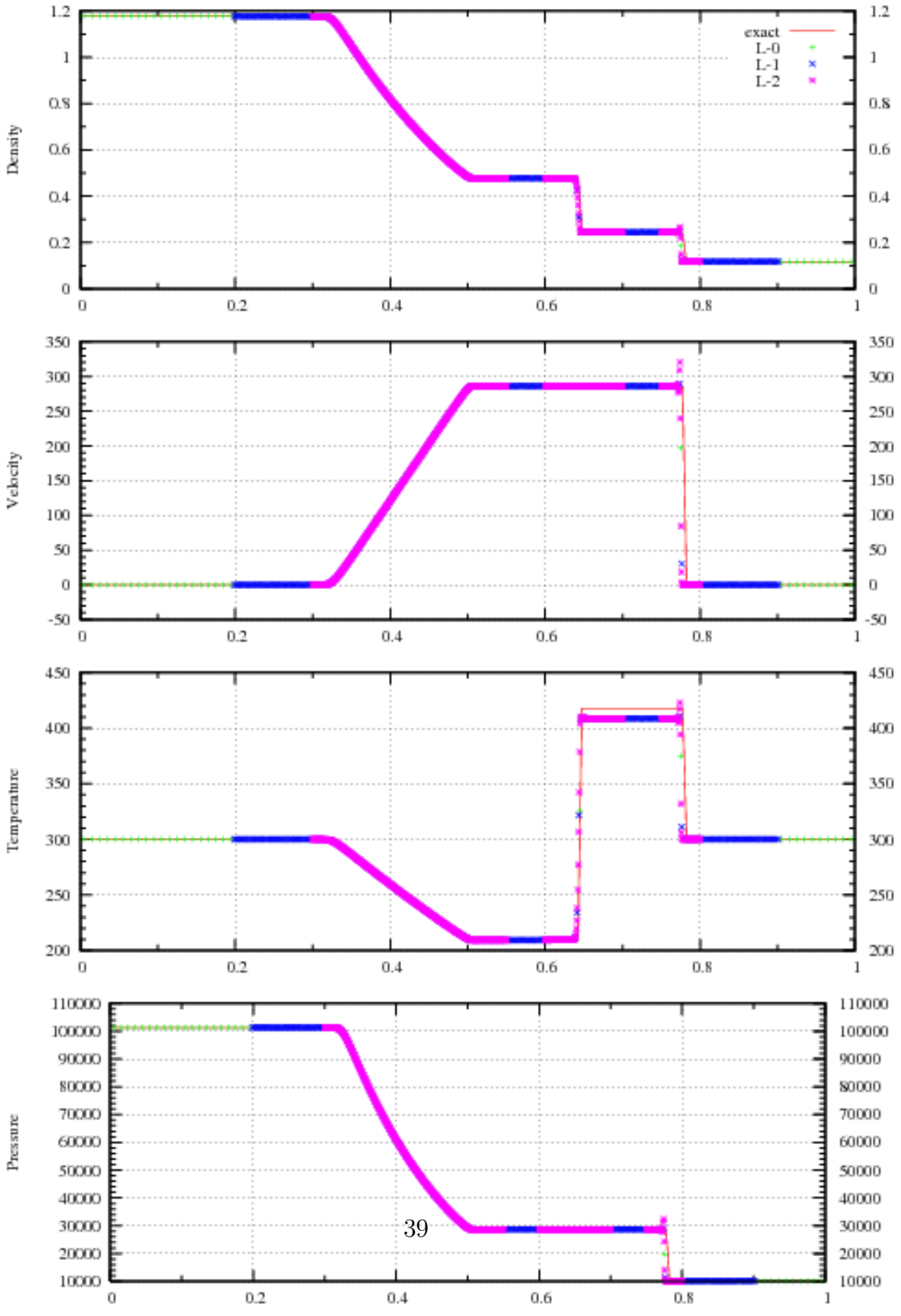


Figure 2: Shock tube results at time $t = 5msec$

# 2D Riemann Problem with Adaptive Mesh Refinement

## Problem Description

In two-dimensional Riemann problems there 15 different solutions that combine rarefaction waves, shock waves and a slip line or contact discontinuities [**?**, **?**]. Here we simulate 4 slip lines that form a symmetrical single vortex turning counter clockwise. At time $t = 0$ the computational domain is divided into four quadrants by the lines $x = 1/2, y = 1/2$ The initial condition for $V = (p, \rho, u, v)$ in the four quadrants are $V_{ll} = (1, 1, -0.75, 0.5), V_{lr} = (1, 3, -0.75, -0.5), V_{ul} = (1, 2, 0.75, 0.5), V_{ur} = (1, 1, 0.75, -0.5)$ where, $p$ is pressure, $\rho$ is the density of the polytropic gas, $u$ and $v$ are the $x$ and $y$ component of velocity.

## Simulation Specifics

**Component used:** ICE

**Input file name:** riemann2D_AMR.ups

**Command used to run input file:**
mpirun -np 5 sus inputs/UintahRelease/ICE/riemann2D_AMR.ups

**VisIT session file:** inputs/UintahRelease/ICE/riemann2D.session

**Simulation Domain:** 0.96 x 0.96m x 0.1 m

**Cell Spacing:**
40 x 40 x 1 mm (Level 0)
10 x 10 x 1 mm (Level 1)
2.5 x 2.5 x 1 mm (Level 2)

**Example Runtimes:**
5ish minutes (5 processors, 2.66 GHz Xeon)

**Physical time simulated:** 0.3 sec.

# Results

Figure 3 shows a flood and line contour plot(s) of the density of the gas at 0.03sec.

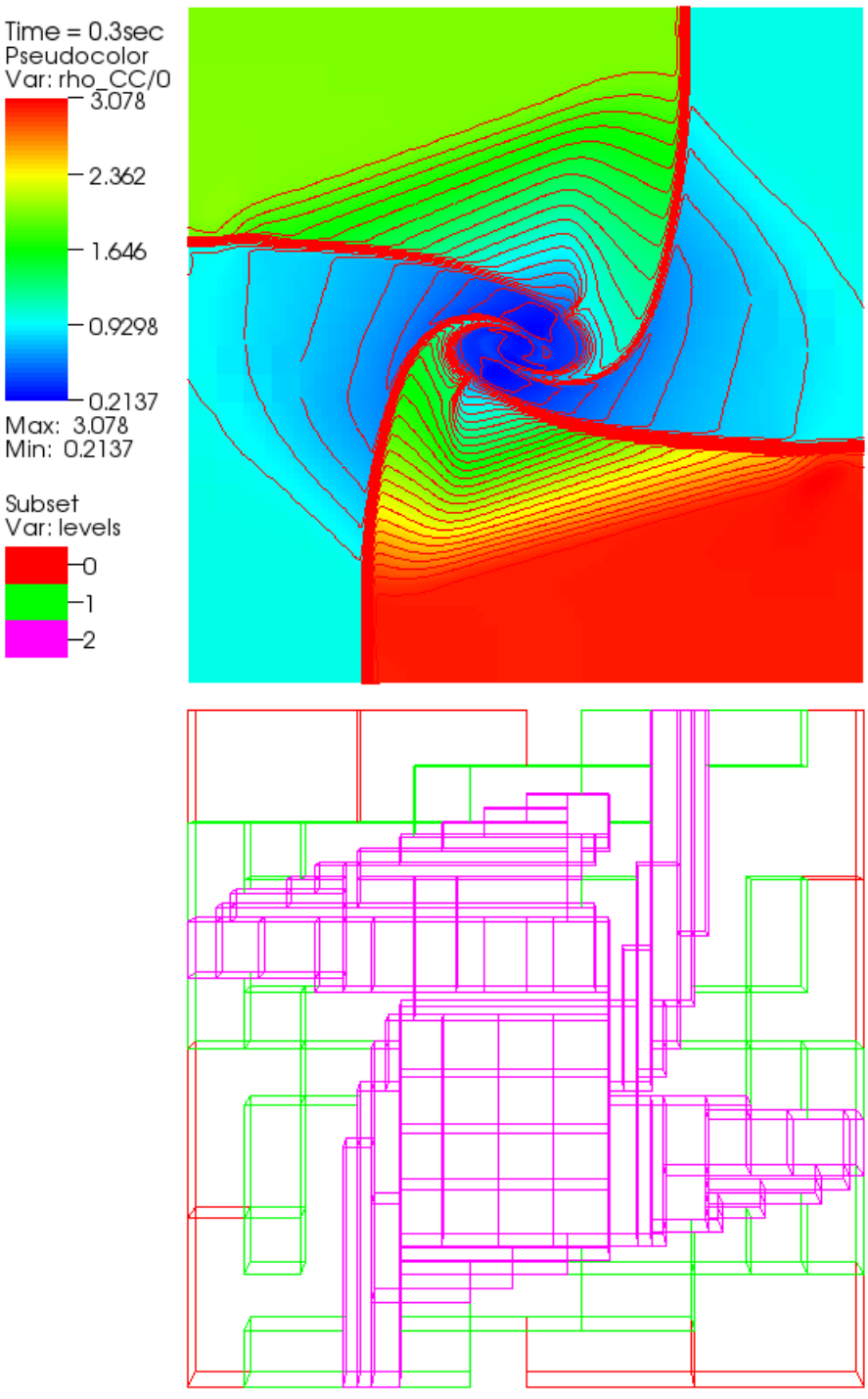


Figure 3: Contour plot of density for the 2D Riemann problem at time $t = 0.3sec$. Bottom plot shows the outline of the patches on the 3 levels.

# Explosion 2D

## Problem Description

For the multidimensional blast wave orexplosion test is a standard compressible flow problem that has been used by many as a validation test case. At time $t = 0$ there is a circular region of gas at the center of the domain at a relatively high pressure and density. The expansion of high pressure gas forms a circular shock wave and contact surface that expands into surrounding atmosphere. At the same time a circular rarefaction travels towards the origin. As the shock wave and contact surface move outwards they become weaker and at some point the contact reverses direction and travels inward. The rarefraction reflects from the center and forms an overexpanded region, creating a shock that travels inward [**?**]. At time $t = 0$ the computational domain is divided into two region, circular high pressure region with a radius $R = 0.4$ and the surrounding box 2x2x0.1. The initial condition inside of the circular region were $(p = 1, \rho = 1, u = 0, v = 0)$ and outside $(p = 0.1, \rho = 0.125, u = 0, v = 0)$. The fluid was an ideal, inviscid, polytropic gas.

## Simulation Specifics

**Component used:** ICE

**Input file name:** Explosion.ups

**Command used to run input file:**
mpirun -np 4 sus inputs/UintahRelease/ICE/Explosion.ups

**Visualization net file:** inputs/UintahRelease/ICE/Explosion.session

**Postprocessing command:**
inputs/ICE/Scripts/plot_explosion_AMR Explosion_AMR.uda y
This will generate a postscript file explosion_AMR.ps

**Simulation Domain:** 2 x 2 x .1

**Cell Spacing:**

62.5 x 62.5 x 10 (Level 0)
15.625 x 15.625 x 10 (Level 1)
3.9 x 3.9 x 10 (Level 2)

**Example Runtimes:**

20 minutes (4 processor, 2.66 GHz Xeon)

**Physical time simulated:** 0.25 (non-dimensional).

# Results

Figures **??** shows surface plots of the pressure and density at $t = 0.25$.

Since this test is symetrical we can use results from the equivalent 1 dimensional problem to compare against

## 11.4 References

# 12 MPM

## 12.1 Introduction

The material point method (MPM) was described by Sulsky et al. [**?**, **?**] as an extension to the FLIP (Fluid-Implicit Particle) method of Brackbill [**?**], which itself is an extension of the particle-in-cell (PIC) method of Harlow [**?**]. Interestingly, the name "material point method" first appeared in the literature two years later in a description of an axisymmetric form of the method [**?**]. In both FLIP and MPM, the basic idea is the same: objects are discretized into particles, or material points, each of which contains all state data for the small region of material that it represents. This includes the position, mass, volume, velocity, stress and state of deformation of that material. MPM differs from other so called "mesh-free" particle methods in that, while each object is primarily represented by a collection of particles, a computational mesh is also an important part of the calculation. Particles do not interact with each other directly, rather the particle information is accumulated to the grid, where the equations of motion are integrated forward in time. This time advanced solution is then used to update the particle state.

The method usually uses a regular structured grid as a computational mesh. While this grid, in principle, deforms as the material that it is representing deforms, at the end of each timestep, it is reset to its original undeformed position, in effect providing a new computational grid for each timestep. The use of a regular structured grid for each time step has a number of computational advantages. Computation of spatial gradients is simplified. Mesh entanglement, which can plague fully Lagrangian techniques, such as the Finite Element Method (FEM), is avoided. MPM has also been successful in solving problems involving contact between colliding objects, having an advantage over FEM in that the use of the regular grid eliminates the need for doing costly searches for contact surfaces[**?**].

In addition to the advantages that MPM brings, as with any numerical technique, it has its own set of shortcomings. It is computationally more expensive than a comparable FEM code. Accuracy for MPM is typically lower than FEM, and errors associated with particles moving around the computational grid can introduce non-physical oscillations into the solution. Finally, numerical difficulties can still arise in simulations involving large deformation that will prematurely terminate the simulation. The severity of all of these issues (except for the expense) has been significantly reduced with

the introduction of the Generalized Interpolation Material Point Method, or GIMP[**?**]. The basic concepts associated with GIMP will be described below. Throughout this document, MPM (which ends up being a special case of GIMP) will frequently be referred to interchangably with GIMP.

In addition, MPM can be incorporated with a multi-material CFD algorithm as the structural component in a fluid-structure interaction formulation. This capability was first demonstrated in the CFDLIB codes from Los Alamos by Bryan Kashiwa and co-workers[**?**]. There, as in the Uintah-MPMICE component, MPM serves as the Lagrangian description of the solid material in a multimaterial CFD code. Certain elements of the solution procedure are based in the Eulerian CFD algorithm, including intermaterial heat and momentum transfer as well as satisfaction of a multimaterial equation of state. The use of a Lagrangian method such as MPM to advance the solution of the solid material eliminates the diffusion typically associated with Eulerian methods. The Uintah-MPM component will be described in later chapter of this manual.

Subsequent sections of this chapter will first give a relatively brief description of the MPM and GIMP algorithms. This will, of course, be focused mainly on describing the capabilities of the Uintah-MPM component. This is followed by a section that attempts to relate the information in Section 12.2 to the implementation in Uintah. Following that is a description of the information that goes into an input file. Finally, a number of examples are provided, along with representative results.

## 12.2 Algorithm Description

Time and space prohibit an exhaustive description of the theoretical underpinnings of the Material Point Method. Here we will concentrate on the discrete equations that result from applying a weak form analysis to the governing equations. The interested reader should consult [**?**, **?**] for the development of these discrete equations in MPM, and [**?**] for the development of the equations for the GIMP method. These end up being very similar, the differences in how the two developments affect implementation will be described in Section 12.3.

In solving a structural mechanics problem with MPM, one begins by discretizing the object of interest into a suitable number of particles, or "material points". (**Aside:** What constitutes a suitable number is something of an open question, but it is typically advisable to use at least two particles in

each computational cell in each direction, i.e. 4 particles per cell (PPC) in 2-D, 8 PPC in 3-D. In choosing the resolution of the computational grid, similar considerations apply as for any computational method (trade-off between time to solution and accuracy, use of resolution studies to ensure convergence in results, etc.).) Each of these particles will carry, minimally, the following variables:

1. position - $\mathbf{x}_p$
2. mass - $m_p$
3. volume - $v_p$
4. velocity - $\mathbf{v}_p$
5. stress - $\boldsymbol{\sigma}_p$
6. deformation gradient - $\mathbf{F}_p$

The description that follows is a recipe for advancing each of these variables from the current (discrete) time $n$ to the subsequent time $n+1$. Note that particle mass, $m_p$, typically remains constant throughout a simulation unless solid phase reaction models are utilized, a feature that is not present in Uintah-MPM. (Such models are available in MPMICE, see Section 14.) It is also important to point out that the algorithm for advancing the timestep is based on the so-called Update Stress Last (USL) algorithm. The superiority of this approach over the Update Stress First (USF) approach was clearly demonstrated by Wallstedt and Guilkey [**?**]. USF was the formulation used in Uintah until mid-2008.

The discrete momentum equation that results from the weak form is given as:

$$\mathbf{ma} \quad = \quad \mathbf{F}^{\text{ext}} - \mathbf{F}^{\text{int}} \tag{22}$$

where $\mathbf{m}$ is the mass matrix, $\mathbf{a}$ is the acceleration vector, $\mathbf{F}^{\text{ext}}$ is the external force vector (sum of the body forces and tractions), and $\mathbf{F}^{\text{int}}$ is the internal force vector resulting from the divergence of the material stresses. The construction of each of these quantities, which are based at the nodes of the computational grid, will be described below.

The solution begins by accumulating the particle state on the nodes of the computational grid, to form the mass matrix $\mathbf{m}$ and to find the nodal

external forces $\mathbf{F}^{\text{ext}}$, and velocities, $\mathbf{v}$. In practice, a lumped mass matrix is used to avoid the need to invert a system of equations to solve Eq. 22 for acceleration. These quantities are calculated at individual nodes by the following equations, where the $\sum_p$ represents a summation over all particles:

$$m_i = \sum_p S_{ip} m_p, \qquad \mathbf{v}_i = \frac{\sum_p S_{ip} m_p \mathbf{v}_p}{m_i}, \qquad \mathbf{F}_i^{\text{ext}} = \sum_p S_{ip} \mathbf{F}_p^{\text{ext}} \tag{23}$$

and $i$ refers to individual nodes of the grid. $m_p$ is the particle mass, $\mathbf{v}_p$ is the particle velocity, and $\mathbf{F}_p^{\text{ext}}$ is the external force on the particle. The external forces that start on the particles typically the result of tractions, the application of which will be discussed in Section 12.5.8. $S_{ip}$ is the shape function of the $ith$ node evaluated at $\mathbf{x}_p$. The functional form of the shape functions differs between MPM and GIMP. This difference is discussed in Section 12.3.

Following the operations in Eq. 23, $\mathbf{F}^{\text{int}}$ is still required in order to solve for acceleration at the nodes. This is computed at the nodes as a volume integral of the divergence of the stress on the particles, specifically:

$$\mathbf{F}_i^{\text{int}} = \sum_p \mathbf{G}_{ip} \boldsymbol{\sigma}_p v_p, \tag{24}$$

where $\mathbf{G}_{ip}$ is the gradient of the shape function of the $ith$ node evaluated at $\mathbf{x}_p$, and $\boldsymbol{\sigma}_p$ and $v_p$ are the time $n$ values of particle stress and volume respectively.

Equation 22 can then be solved for $\mathbf{a}$.

$$\mathbf{a}_i = \frac{\mathbf{F}_i^{\text{ext}} - \mathbf{F}_i^{\text{int}}}{m_i} \tag{25}$$

An explicit forward Euler method is used for the time integration:

$$\mathbf{v}_i^L = \mathbf{v}_i + \mathbf{a}_i \Delta t \tag{26}$$

The time advanced grid velocity, $\mathbf{v}^L$ is used to compute a velocity gradient at each particle according to:

$$\nabla \mathbf{v}_p = \sum_i \mathbf{G}_{ip} \mathbf{v}_i^L \tag{27}$$

This velocity gradient is used to update the particle's deformation gradient, volume and stress. First, an incremental deformation gradient is computed:

$$\mathbf{F}_{n_p}^{n+1} = (\mathbf{I} + \nabla \mathbf{v}_p \Delta t) \tag{28}$$

Particle volume and deformation gradient are updated by:

$$v_p^{n+1} = \mathrm{Det}(\mathbf{F}_{\mathrm{n_p}}^{\mathrm{n+1}})\mathrm{v}_{\mathrm{p}}^{\mathrm{n}}, \qquad \mathbf{F}_{\mathrm{p}}^{\mathrm{n+1}} = \mathbf{F}_{\mathrm{n_p}}^{\mathrm{n+1}}\mathbf{F}_{\mathrm{p}}^{\mathrm{n}} \tag{29}$$

Finally, the velocity gradient, and/or the deformation gradient are provided to a constitutive model, which outputs a time advanced stress at the particles. Specifics of this operation will be further discussed in Section 12.5.5

At this point in the timestep, the particle position and velocity are explicitly updated by:

$$\mathbf{v}_p(t + \Delta t) \;=\; \mathbf{v}_p(t) + \sum_i S_{ip}\mathbf{a}_i \Delta t \tag{30}$$

$$\mathbf{x}_p(t + \Delta t) \;=\; \mathbf{x}_p(t) + \sum_i S_{ip}\mathbf{v}_i^L \Delta t \tag{31}$$

This completes one timestep, in that the update of all six of the variables enumerated above (with the exception of mass, which is assumed to remain constant) has been accomplished. Conceptually, one can imagine that, since an acceleration and velocity were computed at the grid, and an interval of time has passed, the grid nodes also experienced a displacement. This displacement also moved the particles in an isoparametric fashion. In practice, particle motion is accomplished by Equation 31, and the grid never deforms. So, while the MPM literature will often refer to resetting the grid to its original configuration, in fact, this isn't necessary as the grid nodes never leave that configuration. Regardless, at this point, one is ready to advance to the next timestep.

The algorithm described above is the core of the Uintah-MPM implementation. However, it neglects a number of important considerations. The first is kinematic boundary conditions on the grid for velocity and acceleration. The manner in which these are handled will be described in Section 12.4. Next, is the use of advanced contact algorithms. By default, MPM enforces no-slip, no-interpenetration contact. This feature is extremely useful, but it

also means that two bodies initially in "contact" (meaning that they both contain particles whose data are accumulated to common nodes) behave as if they are a single body. To enable multi-field simulations with frictional contact, or to impose displacement based boundary conditions, e.g. a rigid piston, additional steps must be taken. These steps implement contact formulations such as that described by Bardenhagen, et al.[**?**]. The *use* of the contact algorithms is described in Section 12.5.6, but the reader will be referred to the relevant literature for their development. Lastly, heat conduction is also available in the explicit MPM code, although it may be neglected via a run time option in the input file. Explicit MPM is typically used for high rate simulations in which heat conduction is negligible.

## 12.3 Shape functions for MPM and GIMP

In both MPM and GIMP, the basic idea is the same: objects are discretized into particles, or material points, each of which contains all state data for the small region of material that it represents. In MPM, these particles are spatially Dirac delta functions, meaning that the material that each represents is assumed to exist at a single point in space, namely the position of the particle. Interactions between the particles and the grid take place using weighting functions, also known as shape functions or interpolation functions. These are typically, but not necessarily, linear, bilinear or trilinear in one, two and three dimensions, respectively.

More recently, Bardenhagen and Kober [**?**] generalized the development that gives rise to MPM, and suggested that MPM may be thought of as a subset of their "Generalized Interpolation Material Point" (GIMP) method. In the family of GIMP methods one chooses a characteristic function $\chi_p$ to represent the particles and a shape function $S_i$ as a basis of support on the computational nodes. An effective shape function $\bar{S}_{ip}$ is found by the convolution of the $\chi_p$ and $S_i$ which is written as:

$$\bar{S}_{ip}(\mathbf{x}_p) = \frac{1}{V_p} \int_{\Omega_p \cap \Omega} \chi_p(\mathbf{x} - \mathbf{x}_p) S_i(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{32}$$

While the user has significant latitude in choosing these two functions, in

practice, the choice of $S_i$ is usually given (in one-dimension) as,

$$S_i(x) = \begin{cases} 1 + (x - x_i)/h & -h < x - x_i \leq 0 \\ 1 - (x - x_i)/h & 0 < x - x_i \leq h \\ 0 & \text{otherwise,} \end{cases} \tag{33}$$

where $x_i$ is the vertex location, and $h$ is the cell width, assumed to be constant in this investigation, although this is not a general restriction on the method. Multi-dimensional versions are constructed by forming tensor products of the one-dimensional version in the orthogonal directions.

When the choice of characteristic function is the Dirac delta,

$$\chi_p(\mathbf{x}) = \delta(\mathbf{x} - \mathbf{x}_p)V_p, \tag{34}$$

where $\mathbf{x}_p$ is the particle position, and $V_p$ is the particle volume, then traditional MPM is recovered. In that case, the effective shape function is still that given by Equation 33. Its gradient is given by:

$$G_i(x) = \begin{cases} 1/h & -h < x - x_i \leq 0 \\ -1/h & 0 < x - x_i \leq h \\ 0 & \text{otherwise,} \end{cases} \tag{35}$$

Plots of Equations 33 and 35 are shown below. The discontinuity in the gradient gives rise to poor accuracy and stability properties.

Typically, when an analyst indicates that they are "using GIMP" this implies use of the linear grid basis function given in Eq. 33 and a "top-hat" characteristic function, given by (in one-dimension),

$$\chi_p(x) = H(x - (x_p - l_p)) - H(x - (x_p + l_p)), \tag{36}$$

where $H(x)$ is the Heaviside function ($H(x) = 0$ if $x < 0$ and $H(x) = 1$ if $x \geq 0$) and $l_p$ is the half-length of the particle. When the convolution indicated in Eq. 32 is carried out using the expressions in Eqns. 33 and 36,

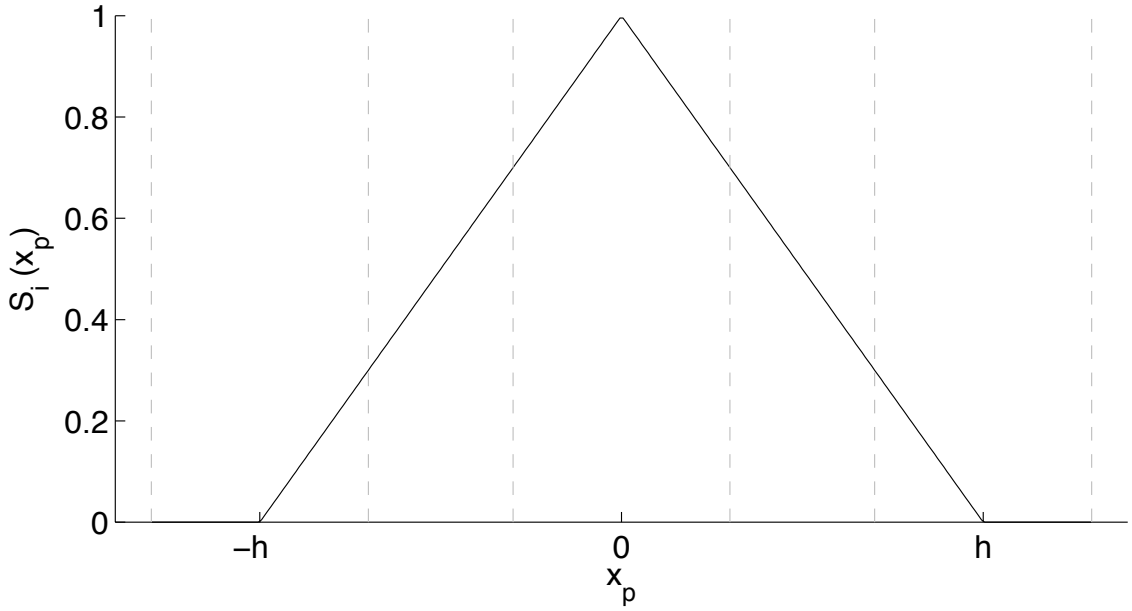


Figure 4: Effective shape function when using traditional MPM.

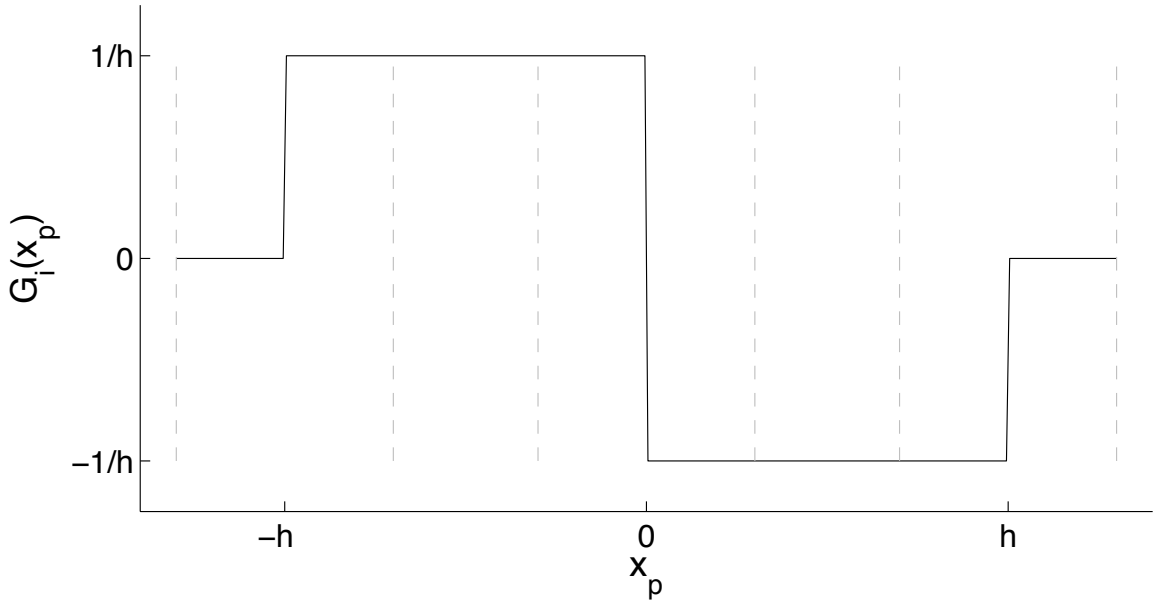


Figure 5: Gradient of the effective shape function when using traditional MPM.

a closed form for the effective shape function can be written as:

$$S_i\left(x_p\right) = \begin{cases} \frac{\left(h+l_p+\left(x_p-x_i\right)\right)^2}{4hl_p} & -h-l_p < x_p - x_i \leq -h+l_p \\ 1+\frac{\left(x_p-x_i\right)}{h} & -h+l_p < x_p - x_i \leq -l_p \\ 1-\frac{\left(x_p-x_i\right)^2+l_p^2}{2hl_p} & -l_p < x_p - x_i \leq l_p \\ 1-\frac{\left(x_p-x_i\right)}{h} & l_p < x_p - x_i \leq h-l_p \\ \frac{\left(h+l_p-\left(x_p-x_i\right)\right)^2}{4hl_p} & h-l_p < x_p - x_i \leq h+l_p \\ 0 & \text{otherwise,} \end{cases} \tag{37}$$

The gradient of which is:

$$G_i(x_p) = \begin{cases} \frac{h+l_p+\left(x_p-x_i\right)}{2hl_p} & -h-l_p < x_p - x_i \leq -h+l_p \\ \frac{1}{h} & -h+l_p < x_p - x_i \leq -l_p \\ -\frac{\left(x_p-x_i\right)}{hl_p} & -l_p < x_p - x_i \leq l_p \\ -\frac{1}{h} & l_p < x_p - x_i \leq h-l_p \\ -\frac{h+l_p-\left(x_p-x_i\right)}{2hl_p} & h-l_p < x_p - x_i \leq h+l_p \\ 0 & \text{otherwise,} \end{cases} \tag{38}$$

Plots of Equations 37 and 38 are shown below. The continuous nature of the gradients are largely responsible for the improved robustness and accuracy of GIMP over MPM.

There is one further consideration in defining the effective shape function, and that is whether or not the size (length in 1-D) of the particle is kept fixed (denoted as "UGIMP" here) or is allowed to evolve due to material deformations ("Finite GIMP" or "Contiguous GIMP" in (1) and "cpGIMP" here). In one-dimensional simulations, evolution of the particle (half-)length is straightforward,

$$l_p^n = F_p^n l_p^0, \tag{39}$$

where $F_p^n$ is the deformation gradient at time $n$. In multi-dimensional simulations, a similar approach can be used, assuming an initially rectangular or cuboid particle, to find the current particle shape. The difficulty arises in evaluating Eq. 32 for these general shapes. One approach, apparently effective, has been to create a cuboid that circumscribes the deformed particle shape [**?**]. Alternatively, one can assume that the particle size remains constant (insofar as it applies to the effective shape function evaluations only). This is the approach currently implemented in Uintah.

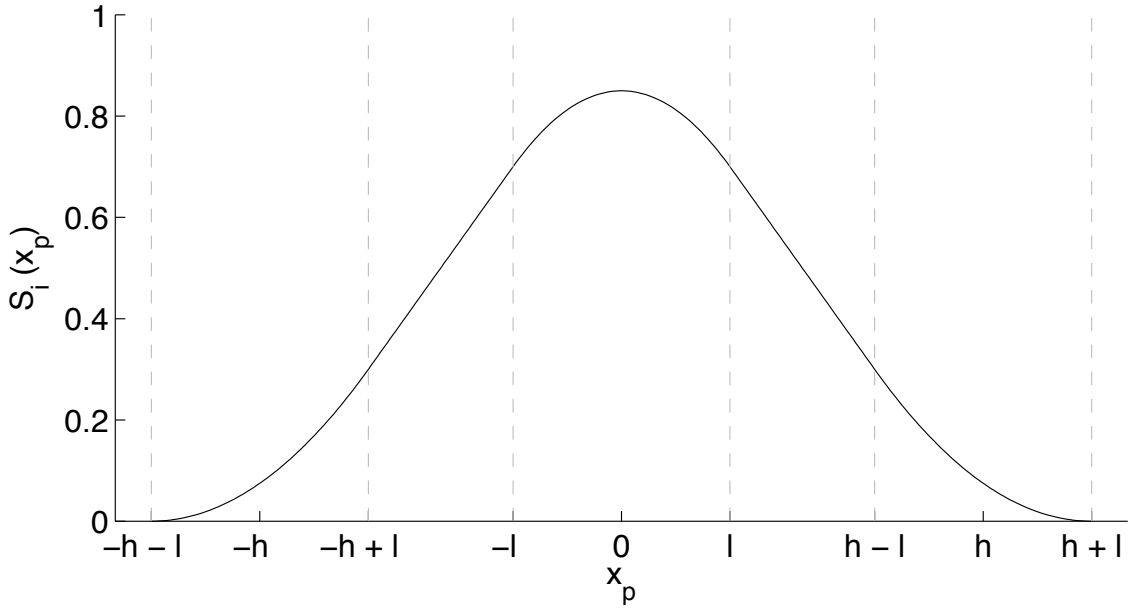


Figure 6: Effective shape function when using GIMP.

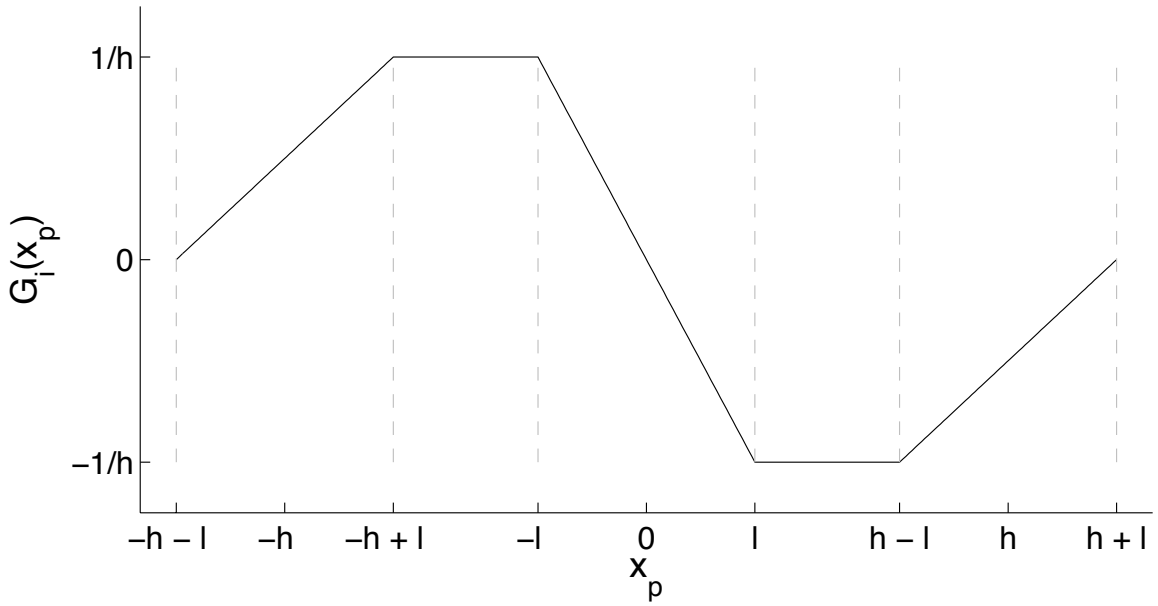


Figure 7: Gradient of the effective shape function when using GIMP.

## 12.4 Uintah Implementation

Users of Uintah-MPM needn't necessarily bother themselves with the implementation in code of the algorithm described above. This section is intended to serve as a reference for users who find themselves needing to modify the source code, or those who are simply interested. Anyone just wishing to run MPM simulations may skip ahead to Sections 12.5 and 12.6. The goal of this section is to provide a mapping from the the algorithm described above to the software that carries it out. This won't be exhaustive, but will be a good starting point for the motivated reader.

The source code for the Uintah-MPM implementation can be found in

```
src/CCA/Components/MPM
```

Within that directory are a number of files and subdirectories, these will be discussed as needed. For the moment, consider the various files that end in "MPM.cc":

```
AMRMPM.cc  FractureMPM.cc  ImpMPM.cc  RigidMPM.cc  SerialMPM.cc  ShellMPM.cc
```

`AMRMPM.cc` is the nascent beginnings of an AMR implementation of MPM. It is far from complete and should be ignored. `FractureMPM.cc` is an implementation of the work of Guo and Nairn [**?**], and while it is viable, it is undocumented and unsupported. `ShellMPM.cc` is a treatment of MPM particles as shell and membrane elements, developed by Biswajit Bannerjee. It is also viable, but also undocumented and unsupported. `ImpMPM.cc` is an implicit time integration form of MPM based on the work of Guilkey and Weiss [**?**]. It is also viable, and future releases of Uintah will include documentation of its capabilities and uses. For now, interested readers should contact Jim Guilkey directly for more information. `RigidMPM.cc` contains a very reduced level of functionality, and is used solely in conjunction with the MPMArches component.

This leaves `SerialMPM.cc`. This contains, despite its name, the parallel implementation of the algorithm described above in Section 12.2. For now, we will skip over the initialization procedures such as:

```
SerialMPM::problemSetup
SerialMPM::scheduleInitialize
SerialMPM::actuallyInitialize
```

and focus mainly on the timestepping algorithm described above. Reference will be made back to these functions as needed in Section 12.5.

Each of the Uintah components contains a function called `scheduleTimeAdvance` . The algorithms implemented in these components are broken into a number of steps. The implementation of these steps in Uintah take place in "tasks". Each task is responsible for performing the calculations needed to accomplish that step in the algorithm. Thus, each task requires some data upon which to operate, and it also creates some data, either as a final result, or as input to a subsequent task. Before individual tasks are executed, each is first "scheduled". The scheduling of tasks describes the dataflow and data dependencies for a given algorithm. By describing the data dependencies, both temporally and spatially, each task can be executed in the proper order, and communication tasks can automatically be generated by the Uintah infrastructure to achieve parallelism. Thus, scheduleTimeAdvance calls a series of functions, each of which schedules the individual tasks. Let's begin by looking at the `scheduleTimeAdvance` for `SerialMPM,` pasted below.

```
void
SerialMPM::scheduleTimeAdvance(const LevelP & level,
                               SchedulerP   & sched)
{
  MALLOC_TRACE_TAG_SCOPE("SerialMPM::scheduleTimeAdvance()");
  if (!flags->doMPMOnLevel(level->getIndex(), level->getGrid()->numLevels()))
    return;

  const PatchSet* patches = level->eachPatch();
  const MaterialSet* matls = d_sharedState->allMPMMaterials();

  scheduleApplyExternalLoads(             sched, patches, matls);
  scheduleInterpolateParticlesToGrid(     sched, patches, matls);
  scheduleExMomInterpolated(              sched, patches, matls);
  scheduleComputeContactArea(             sched, patches, matls);
  scheduleComputeInternalForce(           sched, patches, matls);

  scheduleComputeAndIntegrateAcceleration(sched, patches, matls);
  scheduleExMomIntegrated(                sched, patches, matls);
  scheduleSetGridBoundaryConditions(      sched, patches, matls);
  scheduleSetPrescribedMotion(            sched, patches, matls);
  scheduleComputeStressTensor(            sched, patches, matls);
  if(flags->d_doExplicitHeatConduction){
    scheduleComputeHeatExchange(          sched, patches, matls);
    scheduleComputeInternalHeatRate(      sched, patches, matls);
    scheduleComputeNodalHeatFlux(         sched, patches, matls);
```

```
    scheduleSolveHeatEquations(           sched, patches, matls);
    scheduleIntegrateTemperatureRate(     sched, patches, matls);
  }
  scheduleAddNewParticles(                sched, patches, matls);
  scheduleConvertLocalizedParticles(      sched, patches, matls);
  scheduleInterpolateToParticlesAndUpdate(sched, patches, matls);

  if(flags->d_canAddMPMMaterial){
    //  This checks to see if the model on THIS patch says that it's
    //  time to add a new material
    scheduleCheckNeedAddMPMMaterial(         sched, patches, matls);

    //  This one checks to see if the model on ANY patch says that it's
    //  time to add a new material
    scheduleSetNeedAddMaterialFlag(         sched, level,   matls);
  }

  sched->scheduleParticleRelocation(level, lb->pXLabel_preReloc,
                                    d_sharedState->d_particleState_preReloc,
                                    lb->pXLabel,
                                    d_sharedState->d_particleState,
                                    lb->pParticleIDLabel, matls);
  if(d_analysisModule){
    d_analysisModule->scheduleDoAnalysis( sched, level);
  }
}
```

The preceding includes scheduling for a number of rarely used features. For now, let's condense the preceding to the essential tasks:

```
void
SerialMPM::scheduleTimeAdvance(const LevelP & level,
                               SchedulerP   & sched)
{
  if (!flags->doMPMOnLevel(level->getIndex(), level->getGrid()->numLevels()))
    return;

  const PatchSet* patches = level->eachPatch();
  const MaterialSet* matls = d_sharedState->allMPMMaterials();

  scheduleApplyExternalLoads(             sched, patches, matls);
  scheduleInterpolateParticlesToGrid(     sched, patches, matls);
  scheduleExMomInterpolated(              sched, patches, matls);
  scheduleComputeInternalForce(           sched, patches, matls);
```

```
  scheduleComputeAndIntegrateAcceleration(sched, patches, matls);
  scheduleExMomIntegrated(                sched, patches, matls);
  scheduleSetGridBoundaryConditions(      sched, patches, matls);
  scheduleComputeStressTensor(            sched, patches, matls);
  scheduleInterpolateToParticlesAndUpdate(sched, patches, matls);

  sched->scheduleParticleRelocation(level, lb->pXLabel_preReloc,
                                    d_sharedState->d_particleState_preReloc,
                                    lb->pXLabel,
                                    d_sharedState->d_particleState,
                                    lb->pParticleIDLabel, matls);
}
```

As described above, each of the "schedule" functions describes dataflow, and it also calls the function that actually executes the task. The naming convention is illustrated by an example, `scheduleComputeAndIntegrateAcceleration` calls `computeAndIntegrateAcceleration`. Let's examine this particular task, which executes Equations 25 and 26, more carefully. First, the scheduling of the task:

```
void SerialMPM::scheduleComputeAndIntegrateAcceleration(SchedulerP& sched,
                                                        const PatchSet* patches,
                                                        const MaterialSet* matls)
{
  if (!flags->doMPMOnLevel(getLevel(patches)->getIndex(),
                           getLevel(patches)->getGrid()->numLevels()))
    return;

  printSchedule(patches,cout_doing,"MPM::scheduleComputeAndIntegrateAcceleration\t\t\t\t");

  Task* t = scinew Task("MPM::computeAndIntegrateAcceleration",
                        this, &SerialMPM::computeAndIntegrateAcceleration);

  t->requires(Task::OldDW, d_sharedState->get_delt_label() );

  t->requires(Task::NewDW, lb->gMassLabel,          Ghost::None);
  t->requires(Task::NewDW, lb->gInternalForceLabel, Ghost::None);
  t->requires(Task::NewDW, lb->gExternalForceLabel, Ghost::None);
  t->requires(Task::NewDW, lb->gVelocityLabel,      Ghost::None);

  t->computes(lb->gVelocityStarLabel);
  t->computes(lb->gAccelerationLabel);

  sched->addTask(t, patches, matls);
}
```

The `if` statement basically directs the schedule to only do this task on the finest level (MPM can be used in AMR simulations, but only at the finest level.) The `printSchedule` command is in place for debugging purposes, this type of print statement can be turned on by setting an environmental variable. The real business of this task begins with the declaration of the Task. In the task declaration, the function associated with that task is identified. Subsequent to that is a description of the data dependencies. Namely, this task `requires` the mass, internal and external forces as well as velocity on the grid. No ghost data are required as this task is a node by node calculation. It also requires the timestep size. Note also that most of the required data are needed from the `NewDW` where `DW` refers to DataWarehouse. This simply means that these data were calculated by an earlier task in the current timestep. The timestep size for this step was computed in the previous timestep, and thus is required from the `OldDW`. Finally, this task `computes` the acceleration and time advanced velocity at each node.

The code to execute this task is as follows:

```
void SerialMPM::computeAndIntegrateAcceleration(const ProcessorGroup*,
                                                const PatchSubset* patches,
                                                const MaterialSubset*,
                                                DataWarehouse* old_dw,
                                                DataWarehouse* new_dw)
{
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    printTask(patches, patch,cout_doing,"Doing computeAndIntegrateAcceleration\t\t\t\t");

    Ghost::GhostType  gnone = Ghost::None;
    Vector gravity = d_sharedState->getGravity();
    for(int m = 0; m < d_sharedState->getNumMPMMatls(); m++){
      MPMMaterial* mpm_matl = d_sharedState->getMPMMaterial( m );
      int dwi = mpm_matl->getDWIndex();

      // Get required variables for this patch
      constNCVariable<Vector> internalforce, externalforce, velocity;
      constNCVariable<double> mass;

      delt_vartype delT;
      old_dw->get(delT, d_sharedState->get_delt_label(), getLevel(patches) );

      new_dw->get(internalforce,lb->gInternalForceLabel, dwi, patch, gnone, 0);
      new_dw->get(externalforce,lb->gExternalForceLabel, dwi, patch, gnone, 0);
      new_dw->get(mass,         lb->gMassLabel,          dwi, patch, gnone, 0);
```

```
      new_dw->get(velocity,     lb->gVelocityLabel,      dwi, patch, gnone, 0);

      // Create variables for the results
      NCVariable<Vector> velocity_star,acceleration;
      new_dw->allocateAndPut(velocity_star, lb->gVelocityStarLabel, dwi, patch);
      new_dw->allocateAndPut(acceleration,  lb->gAccelerationLabel, dwi, patch);

      acceleration.initialize(Vector(0.,0.,0.));
      double damp_coef = flags->d_artificialDampCoeff;

      for(NodeIterator iter=patch->getExtraNodeIterator__New();
                       !iter.done();iter++){
        IntVector c = *iter;
        Vector acc(0.,0.,0.);
        if (mass[c] > flags->d_min_mass_for_acceleration){
          acc  = (internalforce[c] + externalforce[c])/mass[c];
          acc -= damp_coef*velocity[c];
        }
        acceleration[c] = acc +  gravity;
        velocity_star[c] = velocity[c] + acceleration[c] * delT;
      }
    }    // matls
  }
}
```

This task contains three nested for loops. First, is a loop over all of the "patches" that the processor executing this task is responsible for. Next is a loop over all materials (imagine a simulation involving the interaction between, say, tungsten and copper). Within this loop, the required data are retrieved from the `new_dw` (New DataWarehouse) and space for the data to be created is allocated. The final loop is over all of the nodes on the current patch, and the calculations described by Equations 25 and 26 are carried out. (This also includes a linear damping term not described above.)

Let's consider each task in turn. The remaining tasks will be described in much less detail, but the preceding dissection of a fairly simple task, along with a description of what the remaining tasks are intended to accomplish, should allow interested individuals to follow the remainder of the Uintah-MPM implementation.

1. `scheduleApplyExternalLoads` This task is mainly responsible for applying traction boundary conditions described in the input file. This is done by assigning external force vectors to the particles. If the user

wishes to apply a load that is not possible to acheive via the input file options, it is straightforward to modify the code here to do "one-off" tests.

2. `scheduleInterpolateParticlesToGrid` The name of this task was poorly chosen, but has persisted. This task carries out the operations given in Equation 23. It also sets boundary conditions on some of the variables, such as the grid temperature, and grid velocity (in the case of symmetry BCs).

3. `scheduleExMomInterpolated` This task actually exists in one of the contact models which can be found in the `Contact` directory. Each of those models has two main tasks. This is the the first of those. It is responsible for modifying the grid velocity computed by interpolateParticlesToGrid according to the rules for the particular contact model chosen in the input file. These models are briefly described in Section 12.5.6.

4. `scheduleComputeInternalForce` This task computes the volume integral of the divergence of stress. Specifically, it carries out the operation given in Equation 24. It also computes some diagnostic data, if requested in the input file, such as the reaction forces (tractions) on the boundaries of the computational domain.

5. `scheduleComputeAndIntegrateAcceleration` As described previously, this task carries out the operations described in Equations 25 and 26.

6. `scheduleExMomIntegrated` This is the second of the contact tasks (see above). It is responsible for modifiying the time advanced grid velocity computed in `computeAndIntegrateAcceleration.`

7. `scheduleSetGridBoundaryConditions` This task sets boundary conditions on the time advanced grid velocity. It also sets an acceleration boundary condition as well. However, rather than just setting the acceleration to a given value, it is computed by solving Equation 26 for acceleration, and then recomputing the acceleration (on all nodes) as:

$$\mathbf{a}_i = \frac{\mathbf{v}_i^L - \mathbf{v}_i}{\Delta t} \tag{40}$$

Doing this operation on all nodes has several advantages. For most interior nodes, the value for acceleration will be unchanged, but for nodes on the where the velocity has been altered by enforcing boundary conditions, and for nodes at which the contact models have altered the velocity, the acceleration will be modified to reflect that alteration.

8. `scheduleComputeStressTensor` The task, `computeStressTensor,` exists in each of the models in the `ConstitutiveModel` directory. Each model is responsible for carrying out the operations given in Equation 29, and of course, as the name implies, it also computes the material stress. This task has one additional important function, and that is computing the timestep size for the subsequent step. The CFL condition dictates that the timestep size be limited according to:

$$\Delta t < \frac{\Delta x}{c + |u|} \tag{41}$$

where $\Delta x$ is the cell spacing, $c$ is the wavespeed in the material, and $|u|$ is the magnitude of the local velocity. Because the wavespeed may depend on the state of stress that a material is in, this task provides a convenient time at which to make this calculation. A timestep size is computed for all particles, and the minimum for the particle set on a given patch is put into a "reduction variable". The Uintah infrastructure then does a global reduction to select the smallest timestep from across the domain.

9. `scheduleInterpolateToParticlesAndUpdate` This task carries out the operations in Equations 30 and 31, namely updating the particle state based on the grid solution.

10. `scheduleParticleRelocation` This task is not actually located in the MPM code, but in the Uintah infrastructure. The idea is that as particles move, some will cross patch boundaries, and their data will need to be sent to other processors. This task is responsible for identifying particles that have left the patch that they were on, finding where they went, and sending their data to the correct processor.

## 12.5 Uintah Specification

Uintah input files are constructed in XML format. Each begins with:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
```

while the remainder of the file is enclosed within the following tags:

```
<Uintah_specification>
</Uintah_specification>
```

The following subsections describe the remaining inputs needed to construct an input file for an MPM simulation. The order of the various sections of the input file is not important. **The MPM, ICE and MPMICE components are dimensionless calculations. It is the responsibility of the analyst to provide the following inputs using a consistent set of units.**

### 12.5.1 Common Inputs

Each Uintah component is invoked using a single executable called *sus* , which chooses the type of simulation to execute based on the *SimulationComponent* tag in the input file. For the case of MPM simulations, this looks like:

```
<SimulationComponent type="mpm" />
```

There are a number of fields that are required for any component. The first is that describing the timestepping parameters, these are largely common to all components, and are described in Section 6.2. The only one that bears commenting on at this point is:

```
<timestep_multiplier>    0.5     </timestep_multiplier>
```

This is effectively the CFL number for MPM simulations, that is the number multiplied by the timestep size that is automatically calculated by the MPM code. Experience indicates that one should generally keep this value below 0.5, and should expect to use smaller values for high-rate, large-deformation simulations.

The next field common to the input files for all components is:

```
<DataArchiver>
</DataArchiver>
```

This is described in Section 6.3. To see a list of variables available for saving in MPM simulations, execute the following command from the `StandAlone` directory:

```
inputs/labelNames mpm
```

Note that for visualizing particle data, one must save `p.x,` and at least one other variable by which to color the particles.

The other principle common field is that which describes the computational grid. For MPM, this is typically broken up into two parts, the `<Level>` section specifies the physical extents and spatial resolution of the grid. For more information, consult Section 6.6. The other part specifies kinematic boundary conditions on the grid boundaries. These are discussed below in Section 12.5.7.

### 12.5.2 Physical Constants

The only physical constant required (or optional for that matter) for MPM simulations is gravity, this is specified as:

```
<PhysicalConstants>
   <gravity>            [0,0,0]   </gravity>
</PhysicalConstants>
```

### 12.5.3 MPM Flags

There are many options available when running MPM simulations. These are generally specified in the `<MPM>` section of the input file. Below is a list of these options taken from `inputs/UPS_SPEC/mpm_spec.xml` This file also gives possible values for many of these. A description of their functionality is forthcoming, in the meantime, consult the code and input files.

```
    <MPM>
<!-- These are commonly used options -->
      <artificial_damping_coeff           spec="OPTIONAL DOUBLE 'positive'"/>
      <artificial_viscosity               spec="OPTIONAL BOOLEAN" />
      <artificial_viscosity_coeff1        spec="OPTIONAL DOUBLE" />
      <artificial_viscosity_coeff2        spec="OPTIONAL DOUBLE" />
      <axisymmetric                       spec="OPTIONAL BOOLEAN" />
      <boundary_traction_faces            spec="OPTIONAL STRING" />
      <DoExplicitHeatConduction           spec="OPTIONAL BOOLEAN" />
      <erosion                            spec="OPTIONAL NO_DATA"
            attribute1="algorithm REQUIRED STRING 'none, KeepStress, ZeroStress, RemoveMass'" />
      <interpolator                       spec="OPTIONAL STRING 'linear, gimp, 3rdorderBS, 4thor
      <minimum_particle_mass              spec="OPTIONAL DOUBLE 'positive'"/>
      <minimum_mass_for_acc               spec="OPTIONAL DOUBLE 'positive'"/>
```

```
      <maximum_particle_velocity          spec="OPTIONAL DOUBLE 'positive'"/>
      <testForNegTemps_mpm                spec="OPTIONAL BOOLEAN" />
      <time_integrator                    spec="OPTIONAL STRING 'explicit, fracture, implicit'"
      <use_load_curves                    spec="OPTIONAL BOOLEAN" />
      <UsePrescribedDeformation           spec="OPTIONAL BOOLEAN" />
      <withColor                          spec="OPTIONAL BOOLEAN" />

<!-- These are not commonly used options -->
      <accumulate_strain_energy           spec="OPTIONAL BOOLEAN" />
      <CanAddMPMMaterial                  spec="OPTIONAL BOOLEAN" />
      <create_new_particles               spec="OPTIONAL BOOLEAN" />
      <do_contact_friction_heating        spec="OPTIONAL BOOLEAN" />
      <do_grid_reset                      spec="OPTIONAL BOOLEAN" />
      <DoThermalExpansion                 spec="OPTIONAL BOOLEAN" />
      <ForceBC_force_increment_factor     spec="OPTIONAL DOUBLE" />
      <manual_new_material                spec="OPTIONAL BOOLEAN" />
      <interpolateParticleTempToGridEveryStep spec="OPTIONAL BOOLEAN" />
      <temperature_solve                  spec="OPTIONAL BOOLEAN" />

      <!-- FIXME:  THE FOLLOW APPLY ONLY TO THE IMPLICIT MPM CODE -->
      <dynamic                            spec="OPTIONAL BOOLEAN" />
      <solver                             spec="OPTIONAL STRING 'petsc, simple'" />
      <convergence_criteria_disp          spec="OPTIONAL DOUBLE 'positive'"/>
      <convergence_criteria_energy        spec="OPTIONAL DOUBLE 'positive'"/>
      <num_iters_to_decrease_delT         spec="OPTIONAL INTEGER" />
      <num_iters_to_increase_delT         spec="OPTIONAL INTEGER" />
      <iters_before_timestep_restart      spec="OPTIONAL INTEGER" />
      <DoTransientImplicitHeatConduction  spec="OPTIONAL BOOLEAN" />
      <delT_decrease_factor               spec="OPTIONAL DOUBLE" />
      <delT_increase_factor               spec="OPTIONAL DOUBLE" />
      <DoImplicitHeatConduction           spec="OPTIONAL BOOLEAN" />
      <DoMechanics                        spec="OPTIONAL BOOLEAN" />

      <!-- FIXME:  THE FOLLOW APPLY ONLY TO THE Fracture MPM CODE -->
      <dadx                               spec="OPTIONAL DOUBLE" />
      <smooth_crack_front                 spec="OPTIONAL BOOLEAN" />
      <calculate_fracture_parameters      spec="OPTIONAL BOOLEAN" />
      <do_crack_propagation               spec="OPTIONAL BOOLEAN" />
      <use_volume_integral                spec="OPTIONAL BOOLEAN" />
    </MPM>
```

### 12.5.4 Material Properties

The `Material Properties` section of the input file actually contains not only those, but also the geometry and initial condition data as well. Below is

a simple example, copied from `inputs/MPM/disks.ups`. The `name` field is optional. The first field is the material `<density>`. The `<constitutive_model>` field refers to the model used to generate a stress tensor on each material point. The use of these models is described in detail in Section 12.5.5. Next are the thermal transport properties, `<thermal_conductivity>` and `<specific_heat>`. Note that these are required even if heat conduction is not being computed. These are the required material properties. There are additional optional parameters that are used in other auxiliary calculations, for a list of these see the `inputs/UPS_SPEC/mpm_spec.xml` .

Next is the specification of the geometry, and, along with it, the initial state of the material contained in that geometry. For more information on how initial geometry can be specified, see Section **??**. Within the `<geom_object>` is the `<res>` field. This indicates how many particles per cell are to be used in each of the coordinate directions. Following that are initial values for velocity and temperature. Finally, the `<color>` designation has a number of uses, for example when one wishes to identify initially distinct regions of the same material. In Section 12.5.9 is a description of how this field is used to identify particles for on the fly data extraction.

An arbitray number of `<material>` fields can be specified. As the calculation proceeds, each of these materials has their own field variables, and, as such, each material behaves independently of the others. Interactions between materials occur as a result of "contact" models. Their use is described in detail in Section 12.5.6.

```
<MaterialProperties>
   <MPM>
       <material name="disks">
          <density>1000.0</density>
          <constitutive_model type="comp_mooney_rivlin">
             <he_constant_1>100000.0</he_constant_1>
             <he_constant_2>20000.0</he_constant_2>
             <he_PR>.49</he_PR>
          </constitutive_model>
          <thermal_conductivity>1.0</thermal_conductivity>
          <specific_heat>5</specific_heat>
          <geom_object>
              <cylinder label = "gp1">
                 <bottom>[.25,.25,.05]</bottom>
                 <top>[.25,.25,.1]</top>
                 <radius> .2 </radius>
              </cylinder>
```

```
            <res>[2,2,2]</res>
            <velocity>[2.0,2.0,0]</velocity>
            <temperature>300</temperature>
            <color>             0               </color>
          </geom_object>
        </material>

        <contact>
          <type>null</type>
          <materials>[0]</materials>
        </contact>
    </MPM>
  </MaterialProperties>
```

### 12.5.5 Constitutive Models

Until the Examples have been completed, the reader will need to refer to the source code and/or the many examples of the use of the constitutive models in the inputs/MPM directory.

### 12.5.6 Contact

When multiple materials are specified in the input file, each material interacts with its own field variables. In other words, each material has its own mass, velocity, acceleration, etc. Without any mechanism for their interaction, each material would behave as if it were the only one in the domain. Contact models provide the mechanism by which to specify rules for inter material interactions. There are a number of contact models from which to choose, the use of each is described next. See the input file segment in Section 12.5.4 for an example of their proper placement in the input file, namely, after all of the MPM materials have been described.

The simplest contact model is the `null` model, which indicates that no inter material interactions are to take place. This is typically only used in single material simulations. Its usage looks like:

```
<contact>
  <type>null</type>
</contact>
```

The next simplest model is the `single_velocity` model. The basic MPM formulation provides "free" no-slip, no-interpenetration contact, assuming

that all particle data communicates with a single field on the grid. For a single material simulation with multiple objects, that is the case. If one wishes to achieve that behavior in Uintah-MPM when multiple materials are present, the `single_velocity` contact model should be used. It is specified as:

```
<contact>
  <type>single_velocity</type>
  <materials>[0,1]</materials>
</contact>
```

Note that for this, and all of the contact models, the `<materials>` tag is optional. If it is omitted, the assumption is that all materials will interact via the same contact model. (This will be further discussed below.)

The ultimate in contact models is the `friction` contact model. For a full description, the reader is directed to the paper by Bardenhagen et al.[**?**]. Briefly, the model both overcomes some deficiences in the single velocity field contact (either the "free" contact or the model described above, which behave identically), and it enables some additional features. With single velocity field contact, initially adjacent objects are treated as if they are effectively stuck together. The friction contact model overcomes this by detecting if materials are approaching or departing at a given node. If they are approaching, contact is "enforced" and if they are departing, another check is made to determine if the objects are in compression or tension. If they are in compression, then they are still rebounding from each other, and so contact is enforced. If tension is detected, they are allowed to move apart independently. Frictional sliding is allowed, based on the value specified for `<mu>` and the normal force between the objects. An example of the use of this model is given here:

```
<contact>
   <type>friction</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

A slightly simplified version of the friction model is the `<approach>` model. It is the same as the frictional model above, except that it doesn't make the additional check on the traction between two bodies at each node. At times, it is necessary to neglect this, but some loss of energy will result. Specification is of the model is also nearly identical:

```
<contact>
   <type>approach</type>
   <materials>[0,1,2]</materials>
   <mu> 0.5 </mu>
</contact>
```

Finally, the contact infrastructure is also used to provide a moving displacement boundary condition. Imagine a billet being smashed by a rigid platen, for example. Usage of this model, known as `<specified>` contact, looks like:

```
<contact>
  <type>specified</type>
  <filename>TXC.txt</filename>
  <materials>[0,1,2]</materials>
  <master_material>[0]</master_material>
  <direction>[1,1,1]</direction>
  <stop_time>1.0 </stop_time>
  <velocity_after_stop>[0, 0, 0]</velocity_after_stop>
</contact>
```

For reasons of backwards compatibility, the `<type>specified</type>` is interchangable with `<type>rigid</type>`. By default, when either model is chosen, material 0 is the "rigid" material, although this can be over ridden by the use of the `<master_material>` field. If no `<filename>` field is specified, then the particles of the rigid material proceed with the velocity that they were given as their initial condition, either until the reach a computational boundary, or until the simulation time has reached `<stop_time>`, after which, their velocity becomes that given in the `<velocity_after_stop>` field. The `<direction>` field indicates in which cartesian directions contact should be specified. Values of 1 indicate that contact should be specified, 0 indicates that the subject materials should be allowed to slide in that direction. If a `<filename>` field *is* specified, then the user can create a text file which contains four entries per line. These are:

```
time1 velocity_x1 velocity_y1 velocity_z1
time2 velocity_x2 velocity_y2 velocity_z2
      .
      .
      .
```

The velocity of the rigid material particles will be set to these values, based on linear interpolation between times, until `<stop_time>` is reached.

Finally, it is possible to specify more than one contact model. Suppose one has a simulation with three materials, one rigid, and the other two deformable. The user may want to have the rigid material interact in a rigid manner with the other two materials, while the two deformable materials interact with each other in a single velocity field manner. Specification for this, assuming the rigid material is 0 would look like:

```
<contact>
    <type>single_velocity</type>
    <materials>[1,2]</materials>
</contact>

<contact>
    <type>specified</type>
    <filename>prof.txt</filename>
    <stop_time>1.0</stop_time>
    <direction>[0, 0, 1]</direction>
</contact>
```

An example of this usage can be found in `inputs/MPM/twoblock-single-rigid.ups`.

### 12.5.7 BoundaryConditions

Boundary conditions must be specified on each face of the computational domain $(x^-, x^+, y^-, y^+, z^-, z^+)$ each material. An example of their specification is as follows, where the entire `<Grid>` field is included for context:

```
<Grid>
   <BoundaryConditions>
     <Face side = "x-">
         <BCType id = "all" var = "Dirichlet" label = "Velocity">
               <value> [0.0,0.0,0.0] </value>
         </BCType>
     </Face>
     <Face side = "x+">
        <BCType id = "all" var = "Neumann" label = "Velocity">
             <value> [0.0,0.0,0.0] </value>
        </BCType>
     </Face>
     <Face side = "y-">
       <BCType id = "all" var = "Dirichlet" label = "Velocity">
             <value> [0.0,0.0,0.0] </value>
       </BCType>
     </Face>
```

```
      <Face side = "y+">
        <BCType id = "all" var = "Neumann" label = "Velocity">
               <value> [0.0,0.0,0.0] </value>
        </BCType>
      </Face>
      <Face side = "z-">
        <BCType id = "all" var = "symmetry" label = "Symmetric"> </BCType>
      </Face>
      <Face side = "z+">
        <BCType id = "all" var = "symmetry" label = "Symmetric"> </BCType>
      </Face>
    </BoundaryConditions>
    <Level>
```

... See Section 6.6 ...

```
    </Level>
 </Grid>
```

The three main types of numerical boundary conditions (BCs) that can be applied are "Neumann", "Dirichlet", and "Symmetric", and the use of each is illustrated above. In the case of MPM simulations, Neumann BCs are used when one wishes to allow particles to advect freely out of the computational domain. Dirichlet BCs are used to specify a velocity, zero or otherwise (indicated by the `<value>` tag), on one of the computational boundaries. Symmetric BCs are used to indicate a plane of symmetry. This has a variety of uses. The most obvious is simply when a simulation of interest has symmetry that one can take advantage of to reduce the cost of a calculation. Similarly, since Uintah is a three-dimensional code, if one wishes to achieve plane-strain conditions, this can be done by carrying out a simulation that is one cell thick with Symmetric BCs applied to each face of the plane, as in the example above. Finally, Symmetric BCs also provide a free slip boundary.

There is also the field `id = "all"` . In principal, one could set different boundary condition types for different materials. In practice, this is rarely used, so the usage illustrated here should be used.

### 12.5.8 Physical Boundary Conditions

It is often more convenient to apply a specified load at the MPM particles. The load may be a function of time. Such a load versus time curve is called a **load curve**. In Uintah, the load curve infrastructure is available for general

use (and not only for particles). However, it has been implemented only for a special case of pressure loading. Namely, a surface is specified through the use of the <geom_object> description, and a pressure vs. time curve is described by specifying their values at discrete points in time, between which linear interpolation is used to find values at any time. At $t = 0$, those particles in the vicinity of the the surface are tagged with a load curve ID, and those particles are assigned external forces such that the desired pressure is achieved.

We invoke the load curve in the <MPM> section (See Section 12.5.3)of the input file using <use_load_curves> true </use_load_curves>. The default value is <use_load_curves> false </use_load_curves>.

In Uintah, a load curve infrastructure is implemented in the file .../MPM/PhysicalBC/LoadCurve.h. This file is essentially a templated structure that has the following private data

```
// Load curve information
std::vector<double> d_time;
std::vector<T> d_load;
int d_id;
```

The variable d_id is the load curve ID, d_time is the time, and d_load is the load. Note that the load can have any form - scalar, vector, matrix, etc.

In our current implementation, the actual specification of the load curve information is in the <PhysicalBC> section of the input file. The implementation is limited in that it applies only to pressure boundary conditions for some special geometries (the implementation is in .../MPM/PhysicalBC/PressureBC.cc). However, the load curve template can be used in other, more general, contexts.

A sample input file specification of a pressure load curve is shown below. In this case, a pressure is applied to the inside and outside of a cylinder. The pressure is ramped up from 0 to 1 GPa on the inside and from 0 to 0.1 MPa on the outside over a time of 10 microsecs.

```
<PhysicalBC>
  <MPM>
    <pressure>
      <geom_object>
        <cylinder label = "inner cylinder">
          <bottom>           [0.0,0.0,0.0]   </bottom>
          <top>              [0.0,0.0,.02]   </top>
```

```
          <radius>           0.5             </radius>
        </cylinder>
      </geom_object>
      <load_curve>
        <id>1</id>
        <time_point>
          <time> 0 </time>
          <load> 0 </load>
        </time_point>
        <time_point>
          <time> 1.0e-5 </time>
          <load> 1.0e9 </load>
        </time_point>
      </load_curve>
    </pressure>
    <pressure>
      <geom_object>
        <cylinder label = "outer cylinder">
          <bottom>           [0.0,0.0,0.0]   </bottom>
          <top>              [0.0,0.0,.02]   </top>
          <radius>           1.0             </radius>
        </cylinder>
      </geom_object>
      <load_curve>
        <id>2</id>
        <time_point>
          <time> 0 </time>
          <load> 0 </load>
        </time_point>
        <time_point>
          <time> 1.0e-5 </time>
          <load> 101325.0 </load>
        </time_point>
      </load_curve>
    </pressure>
  </MPM>
</PhysicalBC>
```

The complete input file can be found in `inputs/MPM/thickCylinderMPM.ups`. An additional example which is used to achieve triaxial loading can be found at `inputs/MPM/TXC.ups`. There, the material geometry is a block, and so the regions described are flat surfaces upon which the pressure is applied.

### 12.5.9 On the Fly DataAnalysis

In the event that one wishes to monitor the data for a small region of a simulation at a rate that is more frequent than the what the DataArchiver can reasonably provide (for reasons of data storage and effect on run time), Uintah provides a `<DataAnalysis>` feature. As it applies to MPM, it allows one to specify a group of particles, by assigning those particles a particular value of the `<color>` parameter. In addition, a list of variables and a frequency of output is provided. Then, at run time, a sub-directory (`particleExtract/L-0` ) is created inside the uda which contains a series of files, named according to their particle IDs, one for each tagged particle. Each of these files contains the time and position for that particle, along with whatever other data is specified. **To use this feature, one must include the `<withColor> true </withColor>` tag in the `<MPM>` section of the input file.** (See Section 12.5.3.)

The following input file snippet is taken from `inputs/MPM/disks.ups`

```
<DataAnalysis>
   <Module name="particleExtract">

    <material>disks</material>
    <samplingFrequency> 1e10 </samplingFrequency>
    <timeStart>          0   </timeStart>
    <timeStop>          100  </timeStop>
    <colorThreshold>
      0
    </colorThreshold>

    <Variables>
      <analyze label="p.velocity"/>
      <analyze label="p.stress"/>
    </Variables>

  </Module>
</DataAnalysis>
```

For all particles that are assigned a color greater than the `<colorThreshold>`, the variables `p.velocity` and `p.stress` are saved every every 1/`<samplingFrequency>` time units, starting at `<timeStart>` until `<timeStop>`.

It is also possible to save grid based data with this module, see Section 8 for more information.

## 12.6 Examples

# Colliding Disks

## Problem Description

This is an implementation of an example calculation from [**?**] in which two elastic disks collide and rebound. See Section 7.3 of that manuscript for a description of the problem.

## Simulation Specifics

**Component used:** MPM

**Input file name:** disks_sulsky.ups

**Command used to run input file:** sus disks_sulsky.ups

**Simulation Domain:** 1.0 x 1.0 x 0.05 m

**Cell Spacing:**
.05 x .05 x .05 m (Level 0)

**Example Runtimes:**
1 minute (1 processor, 3.0 GHz Xeon)

**Physical time simulated:** 3.0 seconds

**Associate scirun network:** disks.srn

# Results

Figure 9 shows a snapshot of the simulation, as the disks are beginning to collide.

Additional data is available within the uda in the form of "dat" files. In this case, both the kinetic and strain energies are avaiable and can be plotted to create a graph similar to that in Fig. 5a of [**?**]. e.g. using gnuplot:

```
cd disks.uda.000
gnuplot
gnuplot> plot "StrainEnergy.dat", "KineticEnergy.dat"
gnuplot> quit
```

# Taylor Impact Test

## Problem Description

This is a simulation of an Taylor impact experiment calculation from [**?**] in a copper cylinder at 718 K that is fired at a rigid anvil at 188 m/s. The copper cylinder has a length of 30 mm and a diameter of 6 mm. The cylinder rebounds from the anvil after 100 $\mu$s.

## Simulation Specifics

**Component used:** MPM

**Input file name:** taylorImpact.ups

**Command used to run input file:** sus taylorImpact.ups

**Simulation Domain:** 8 mm x 33 mm x 8 mm

**Cell Spacing:**
1/3 mm x 1/3 mm x 1/3 mm (Level 0)

**Example Runtimes:**
1 hour 20 min. (1 processor, AMD Opteron 2.2 GHz)

**Physical time simulated:** 100 $\mu$seconds

**Associate scirun network:** taylorImpact.srn

## Results

Figure **??** shows a snapshot of the simulation after the cylinder begins to rebound.

Additional data are available within the uda in the form of "dat" files.

# Sphere Rolling Down an Inclined Plane

## Problem Description

Here, a sphere of soft plastic, initially at rest, rolls under the influence of gravity down a plane of a harder plastic. Gravity is oriented such that the plane is effectively angled at 45 degrees to the horizontal. This simulation demonstrates the effectiveness of the contact algorithm, described in [**?**]. Frictional contact, using a friction coefficient of $\mu = 0.495$ causes the ball to start rolling as it impacts the plane, after being dropped from barely above it.

## Simulation Specifics

**Component used:** MPM

**Input file name:** inclinedPlaneSphere.ups

**Command used to run input file:** sus inclinedPlaneSphere.ups

**Simulation Domain:** 12.0 x 2.0 x 4.8 m

**Cell Spacing:**
.2 x .2 x .2 m (Level 0)

**Example Runtimes:**
9 minutes (1 processor, 3.0 GHz Xeon)

**Physical time simulated:** 2.2 seconds

**Associate scirun network:** inclinedPlaneSphere.srn

## Results

Figure 8 shows a snapshot of the simulation, as the sphere is about halfway down the plane. Particles are colored according to velocity magnitude, note that the particles at the top of the sphere are moving most rapidly, and those near the surface of the plane are basically stationary, as expected.

# Crushing a Foam Microstructure

## Problem Description

This calculation demonstrates two important strength of MPM. The first is the ability to quickly generate a computational representation of complex geometries. The second is the ability of the method to handle large deformations, including self contact.

In particular, in this calculation a small sample of foam, the geometry for which was collected using microCT, is represented via material points. The sample is crushed to 87.5% compaction through the use of a rigid plate, which acts as a constant velocity boundary condition on the top of the sample. This calculation is a small example of those described in [**?**]. The geometry of the foam is created by image procesing the CT data, and based on the intensity of each voxel in the image data, the space represented by that voxel either recieves a particle with the material properties of the foam's constituent material, or is left as void space. This particle representation avoids the time consuming steps required to build a suitable unstructured mesh for this very complicated geometry.

## Simulation Specifics

**Component used:** MPM

Figure 8: Sphere rolling down an "inclined" plane. The gravity vector is oriented at a 45 degree angle relative to the plane. Particles are colored by velocity magnitude.

**Input file name:** foam.ups

**Instruction to run input file:** First, copy foam.ups and foam.pts.gz to the same directory as sus. Adjust the number of patches in the ups file based on the number of processors available to you for this run. First, uncompress the pts file:

```
gunzip foam.pts.gz
```

Then the command:

```
tools/pfs/pfs foam.pts
```

will divide the foam.pts file, which contains the geometric description of the foam, into number of patches smaller files, named foam.pts.0, foam.pts.1, etc. This is done so that for large simulations, each processor is only reading that data which it needs, and prevents the thrashing of the file system that would occur if each processor needed to read the entire pts file. This command only needs to be done once, or anytime the patch distibution is changed. Note that this step must be done even if only one processor is available.

To run this simulation:

```
mpirun -np NP sus foam.ups
```

where NP is the number of processors being used.

**Simulation Domain:** 0.2 X 0.2 X 0.2125 mm

**Number of Computational Cells:**
102 X 102 X 85 (Level 0)

**Example Runtimes:**
3 hours (32 processors, 2.4 GHz Xeon)
11 hours (2 processors, 3.0 GHz Xeon)

**Physical time simulated:** 3.75 seconds

**Associate scirun network:** foam.srn

# Results

Figure **??** shows a snapshot of the simulation, as the foam is at 50% compaction.

In this simulation, the reaction forces at 5 of the 6 computational boundaries are also recorded and can be viewed using a simple plotting package such as gnuplot. At each timestep, the internal force at each of the boundaries is accumulated and stored in "dat" files within the uda, e.g. BndyForce_zminus.dat. Because the reaction force is a vector, it is enclosed in square brackets which may be removed by use of a script in the inputs directory:

```
cd foam.uda.000
../inputs/ICE/Scripts/removeBraces BndyForce\_zminus.dat
gnuplot
gnuplot> plot "BndyForce\_zminus.dat" using 1:4
gnuplot> quit
```

These reaction forces are similar to what would be measured on a mechanical testing device, and help to understand the material behavior.

## 12.7 References

Figure 9: Compaction of a foam microstructure.

# 13 MPMArches

## 13.1 Introduction

## 13.2 Theory - Algorithm Description

## 13.3 Uintah Specification

### 13.3.1 Arches

### 13.3.2 Basic Inputs

### 13.3.3 Turbulence

### 13.3.4 Properties

### 13.3.5 BoundaryConditions

### 13.3.6 Physical Constants

### 13.3.7 Solvers

### 13.3.8 MPM

### 13.3.9 Basic Inputs

### 13.3.10 Physical Constants

### 13.3.11 Material Properties

### 13.3.12 Constitutive Models

### 13.3.13 Contact

### 13.3.14 BoundaryConditions

### 13.3.15 Physical Boundary Conditions

## 13.4 Examples

## 13.5 References

# 14 MPMICE

## 14.1 Introduction

## 14.2 Theory - Algorithm Description

## 14.3 Uintah Specification

### 14.3.1 ICE

### 14.3.2 Basic Inputs

### 14.3.3 Physical Constants

### 14.3.4 Material Properties

### 14.3.5 Equation of State

### 14.3.6 Exchange Properties

### 14.3.7 BoundaryConditions

### 14.3.8 Solvers

## 14.4 MPM

### 14.4.1 Basic Inputs

### 14.4.2 Physical Constants

### 14.4.3 Material Properties

### 14.4.4 Constitutive Models

### 14.4.5 Contact

### 14.4.6 BoundaryConditions

### 14.4.7 Physical Boundary Conditions

## 14.5 Examples

## 14.6 References