Vaango Developers Manual

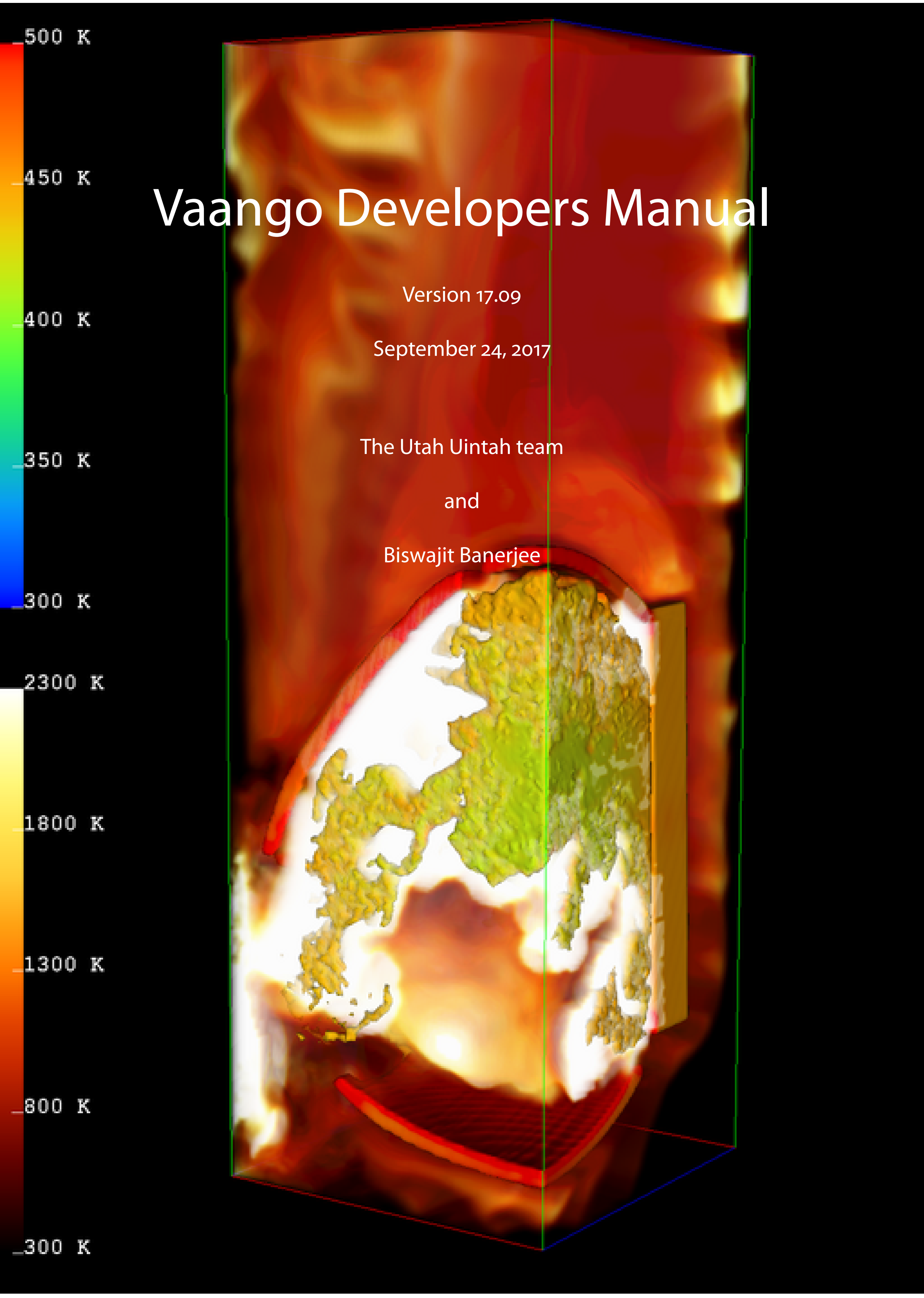
Version 17.09

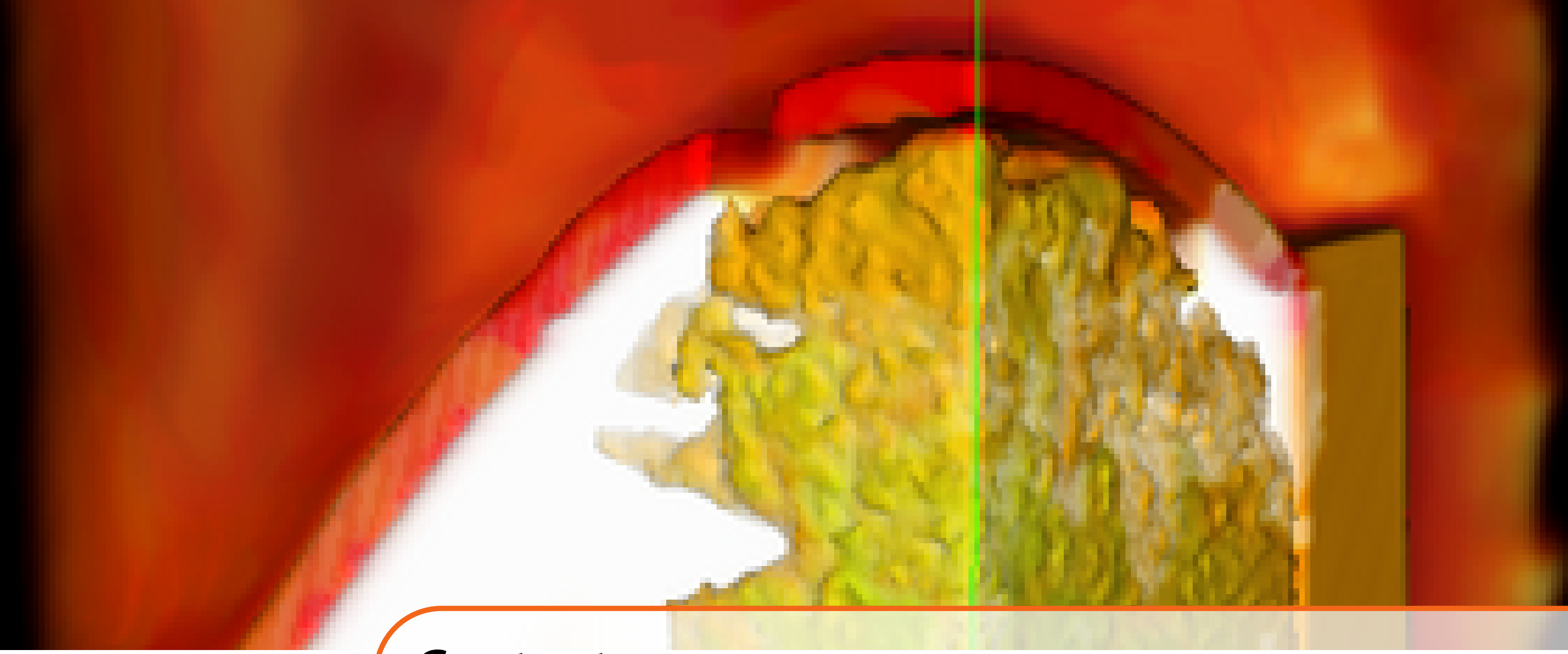
September 24, 2017

The Utah Uintah team

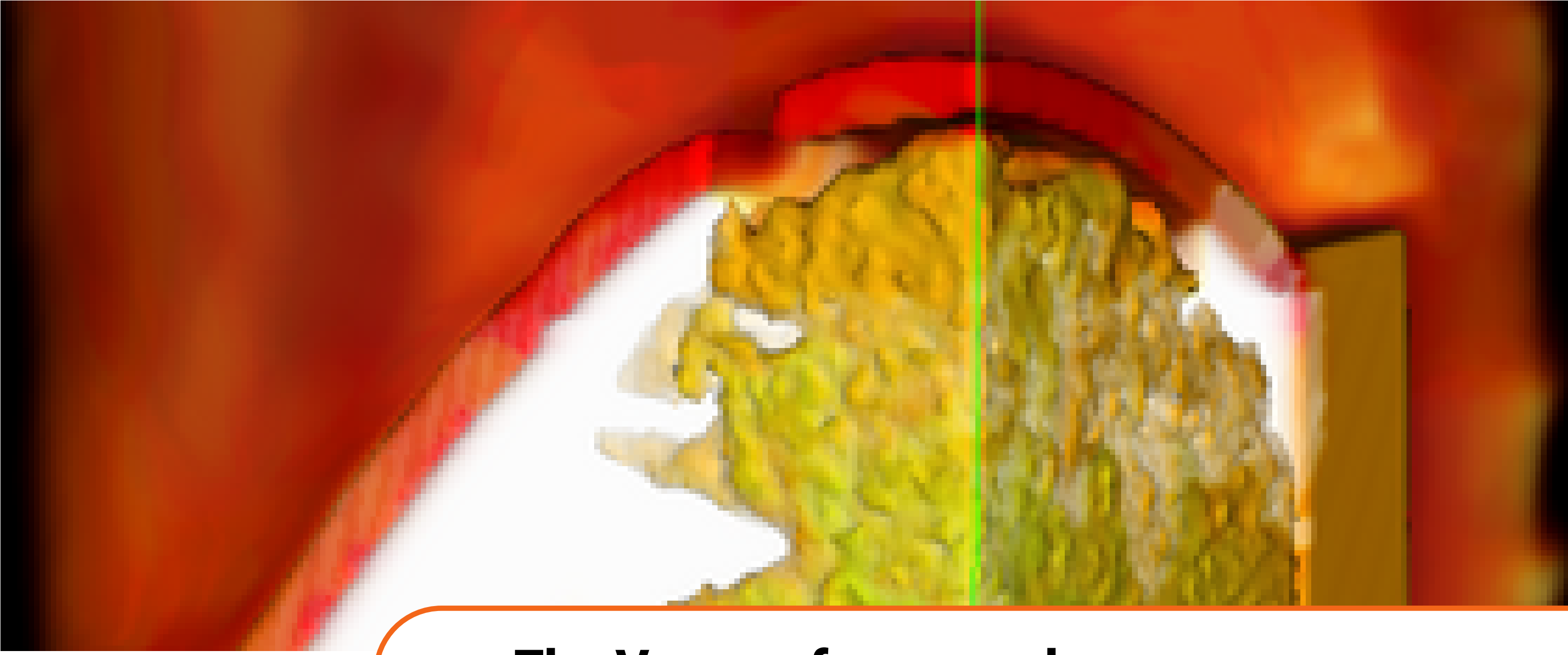
and

Biswajit Banerjee





Contents



1 — The Vaango framework

1.1 Historical information

Vaango is a fork of the Uintah Computational Framework (Uintah) created in 2012 to allow for the development of tools to solve solid mechanics problems in mechanical and civil engineering. The orginal Uintah code continues to be actively developed, but the focus of that code is multiphysics problems, particularly computational fluid dynamics (CFD) and chemical engineering. Around once a year, the underlying parallel computing infrastructure of Vaango is updated to keep up with developments in Uintah .

The Vaango framework, like 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. However, unlike the CFD problems that Uintah was designed for, the use of the grid is often only incidental to the solution of the governing PDEs of solid mechanics.

1.2 Overview

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.

Vaango 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) see figure 1.1. Vaango 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.

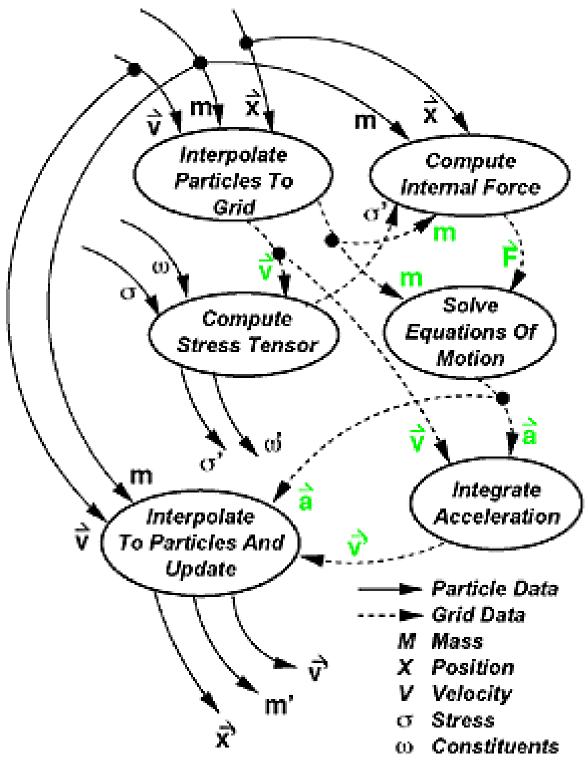


Figure 1.1: Example Task Graph

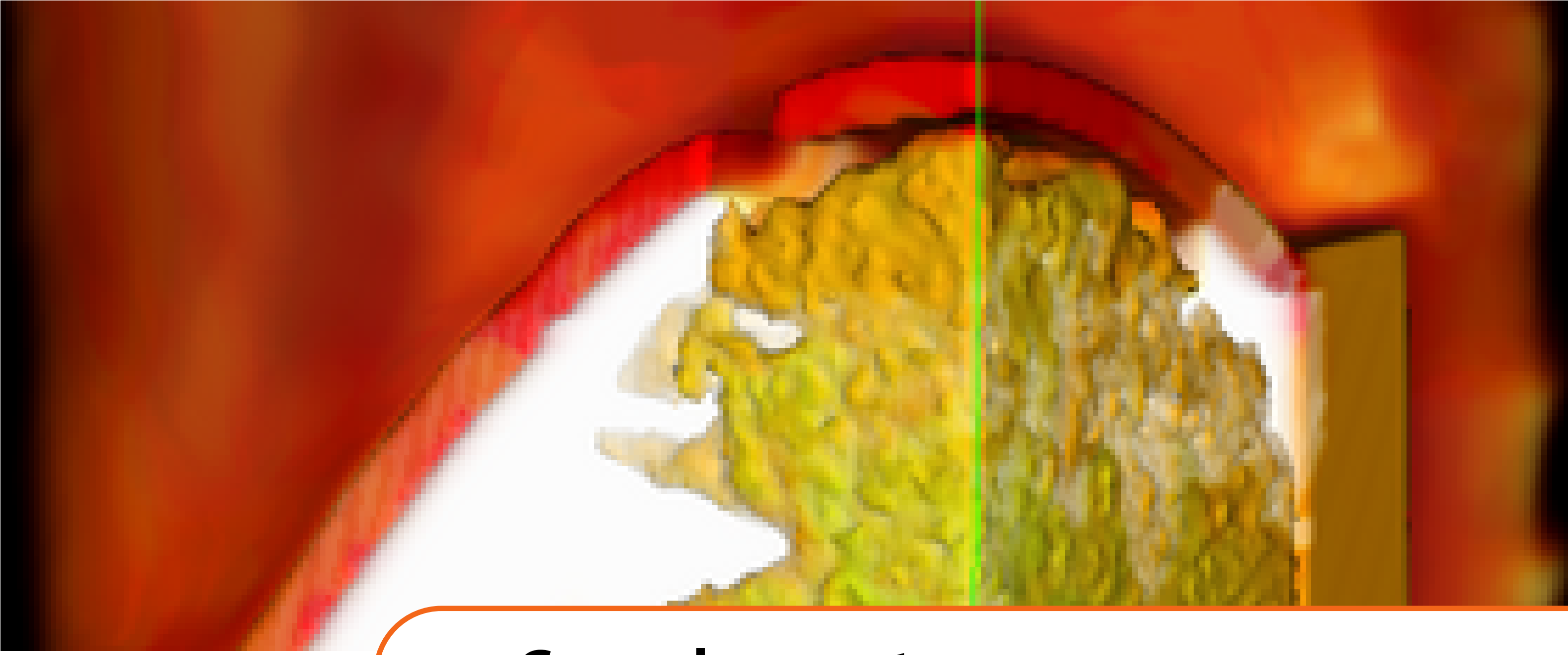
The primary advantage of a component-based approach is that it facilitates the separate development of simulation algorithms, models, and infrastructure. Components of the simulation can evolve independently. The component-based architecture allows pieces of the system to be implemented in a rudimentary form at first and then evolve as the technologies mature. Most importantly, Vaango 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.

1.3 An example

The sequence of steps performed in a simulation can be seen in the abbreviated example below. A complete version of this example can be found in the unit test for TabularPlasticity located at src/CCA/Components/MPM/ConstitutiveModel/UnitTests/testTabularPlasticity.cc .

```
try {
   // Read the input file
   ProblemSpecP ups = VaangoEnv::createInput();
   ups->getNode()->_private = (void *) ups_file.c_str();

   // Create the MPI/threading environment
   const ProcessorGroup* world = Uintah::Parallel::getRootProcessorGroup();

   // Create the simulation controller
   SimulationController* ctl = scinew AMRSimulationController(world, false,
      ups);

   // Create a regridder if needed
   RegridderCommon* reg = 0;

   // Create an implicit solver if needed
```

```
    SolverInterface* solve = SolverFactory::create(ups, world, "");

    // Create the component for the simulation (MPM/MPMICE/Peridynamics)
    UintahParallelComponent* comp = ComponentFactory::create(ups, world, false,
         "");

    // Create the simulation base object
    SimulationInterface* sim = dynamic_cast<SimulationInterface*>(comp);
    ctl->attachPort("sim", sim);
    comp->attachPort("solver", solve);
    comp->attachPort("regridder", reg);

    // Create a load balancer
    LoadBalancerCommon* lbc = LoadBalancerFactory::create(ups, world);
    lbc->attachPort("sim", sim);

    // Create a data archiver
    DataArchiver* dataarchiver = scinew DataArchiver(world, -1);
    Output* output = dataarchiver;
    ctl->attachPort("output", dataarchiver);
    dataarchiver->attachPort("load balancer", lbc);
    comp->attachPort("output", dataarchiver);
    dataarchiver->attachPort("sim", sim);

    // Create a task scheduler
    SchedulerCommon* sched = SchedulerFactory::create(ups, world, output);
    sched->attachPort("load balancer", lbc);
    ctl->attachPort("scheduler", sched);
    lbc->attachPort("scheduler", sched);
    comp->attachPort("scheduler", sched);
    sched->setStartAddr( start_addr );
    sched->addReference();

    // Run the simulation
    ctl->run();

    // Clean up after the simulation is complete
    delete ctl;
    sched->removeReference();
    delete sched;
    delete lbc;
    delete sim;
    delete solve;
    delete output;

  } catch (ProblemSetupException& e) {
    std::cout << e.message() << std::endl;
    thrownException = true;
  } catch (Exception& e) {
    std::cout << e.message() << std::endl;
    thrownException = true;
  } catch (...) {
    std::cout << "**ERROR** Unknown exception" << std::endl;
    thrownException = true;
  }
```



2 — General concepts

This chapter discusses the main concepts used in a Vaango simulation. These concepts are independent of the type of problem being considered and are core to the computational framework. Special treatment is needed for components such as MPM or Peridynamics .

2.1 Scheduler

The Scheduler in Vaango 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 or threading. Then, when the scheduler is executed, the tasks will execute in the pre-determined order.

2.1.1 needRecompile()

Each component has a needRecompile() function that is called once per timestep. If, for whatever reason, a Component determines that the list of tasks it had previously scheduled is no longer valid, then the Component must return 'true' when its needRecompile() function is called. This will cause the scheduler to rebuild the task graph (by asking each component to re-specify tasks). Note, rebuilding the taskgraph is a relatively expensive operation, so only should be done if necessary.

2.2 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 Vaango framework. The task component is assigned a name and in this particular example, it is called taskexample and a function 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 Vaango framework allows for the independent scheduling and computation of multi-material within a multi-physics calculation.

2.3 Simulation Component Class Description

Each Vaango 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 UintahParallelComponent 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

2.3.1 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.

2.3.2 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 .

2.3.3 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 .

2.3.4 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.

2.4 Data Storage Concepts

During the course of the simulation, data is computed and stored in a data structure called the DataWarehouse. The DataWarehouse is an abstraction for storage, retrieval, and access of VAANGO simulation data. The Data warehouse presents a localized view of the global data distribution.

2.4.1 Data Archiver

The Data Archiver is a component of the framework that allows for reading, saving, and accessing simulation data. It presents a global shared memory abstraction to the simulation data. The component developer does not have to worry about retrieving data that has been produced on a remote processor or sending data from a local processor to another processor for use. The framework takes care of these tasks implicitly for the component.

2.4.2 Two Data Warehouses

During each time step (assuming a single level problem), there are two data warehouses associated with the simulations. The `new_dw` and the `old_dw` (as they are commonly referred to in the code). The `old_dw` contains data that was generated in the previous timestep, and may not be modified. Data that is generated during the current timestep will be placed in the `new_dw` . At the end of a timestep, the current `old_dw` is deleted, and then replaced with the current `new_dw` . Then a new (empty) `new_dw` is created.

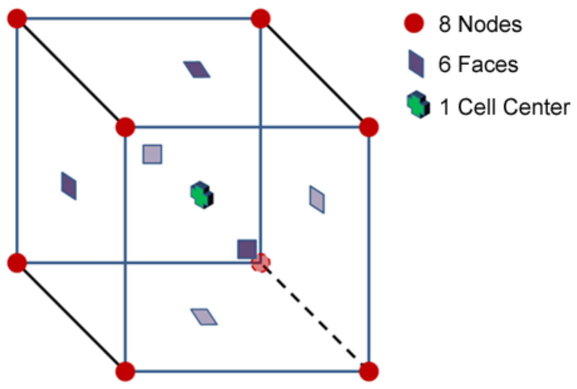


Figure 2.1: Variable locations with respect to a single cell.

At the end of a time step, some data is automatically transferred from the current time step's `new_dw` into the next time step's `old_dw`.

2.4.3 Storing and Retrieving Variables

In order to store data to the data warehouse, or to retrieve data, several things are required. First, the task (which procedure wishes access to the data) must have registered this information with the Scheduler during task creation time. Then, inside of the task, the following call is used to pull the data from the data warehouse:

```
SFCXVariable<double> uVelocity;
new_dw->get( uVelocity, d_lab->d_uVelocitySPBCLabel, matlIndex, patch, Ghost
    ::AroundFaces, Arches::ONEGHOSTCELL );
```

Similarly, to put data into the data warehouse, use this call:

```
PerPatch<CellInformationP> cellInfoP;
new_dw->put( cellInfoP, d_lab->d_cellInfoLabel, matlIndex, patch );
```

2.4.4 Variables

There are three (general) types of variables in VAANGO: Particle, Grid, and Reduction. Each type of variable is discussed below. Remember, in order to interact with the Data Warehouse, each variable must have a corresponding label.

Particle variables contain information about particles.

A grid variable is a representation of data across (usually) the entire computational domain. It can be used to represent temperature, volume, velocity, etc. It is implemented using a 3D array.

A reduction, in terms of distributed software (using, for example, MPI) is a point in which all (or some defined set of) processors all communicate with each other. It usually is an expensive operation (because it requires all processors to synchronize with each other and pass data). However, many times this operation cannot be avoided. VAANGO provides a "Reduction Variable" to facilitate this operation. Common reductions operations include finding the min or max of a single number on all processors, or creating the sum of a number from every processor, and returning the result to all processors.

Variables can be stored at several different locations. Variables are defined with respect to how they relate to a single cell in the computational domain. They may be located on the faces (FC) or nodes (NC) of the cell, or in the center (CC) of the cell (see Figure 2.1).

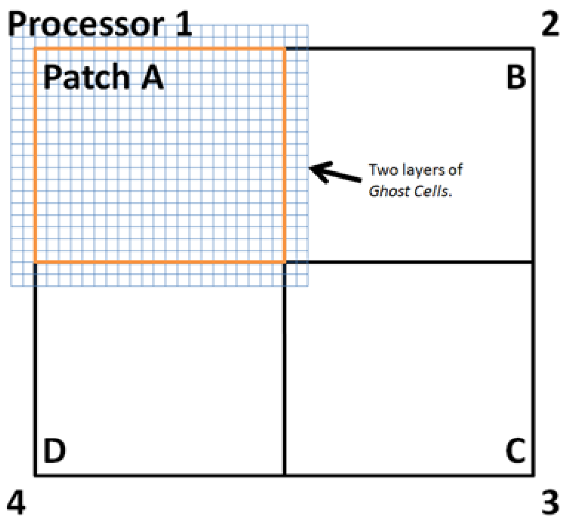


Figure 2.2: Schematic of ghost cell locations laid out with respect to patches.

2.5 Grid data

Data (Variables) used by the framework are stored on the Grid . The Grid is comprised of one or more (in the case of AMR) Levels. Each level contains one or more patches; and each patch contains a number of cells. The simulation data itself exists in (at) each cell (and is stored in a 'variable' which is implemented as a 3D array of the data in each cell across the patch).

2.5.1 Cells

Data in (at) each cell can be specified in several ways. Specifically the data can be Node centered (NC), Cell centered (CC), or Face centered (FC). For CC data, there is one value associated with each cell; for NC data, 4 values; and for FC data, 6 values.

2.5.2 Patches

For calculations to take place on a patch, information from bordering patches is required. This information is stored in boundary cells.

2.5.3 Boundary Cells

Boundary cells (Figure 2.2) represent portions of the computational domain outside the boundaries of the data assigned to a given processor. Specifically, they are almost always used as boundary cells to a patch that are necessary for computation, but are not "owned" by the patch that is currently being computed. As can be seen in Figure 2.2, patch A owns all the cells within the orange rectangle, but also has two layers of ghost cells. Portions of these cells are actually owned by patches B, C, and D. (The data found in these cells is automatically transfered (by the framework) from processors 2, 3, and 4 to processor 1 in order for the cell data to be available for use in computations on patch A.)

Many operations will require a stencil consisting of several cells in order to perform calculations at a given cell. However, at the border of a patch, there are no cells belonging to that patch that contain the required information - that information is "owned" by another processor. In this situation, data that belongs to another patch must be accessed. This is the purpose of ghost cells. The Data Warehouse takes care of moving the required ghost data from the "owner" processor to the neighbor processor so that the individual task can assume the required data is available.

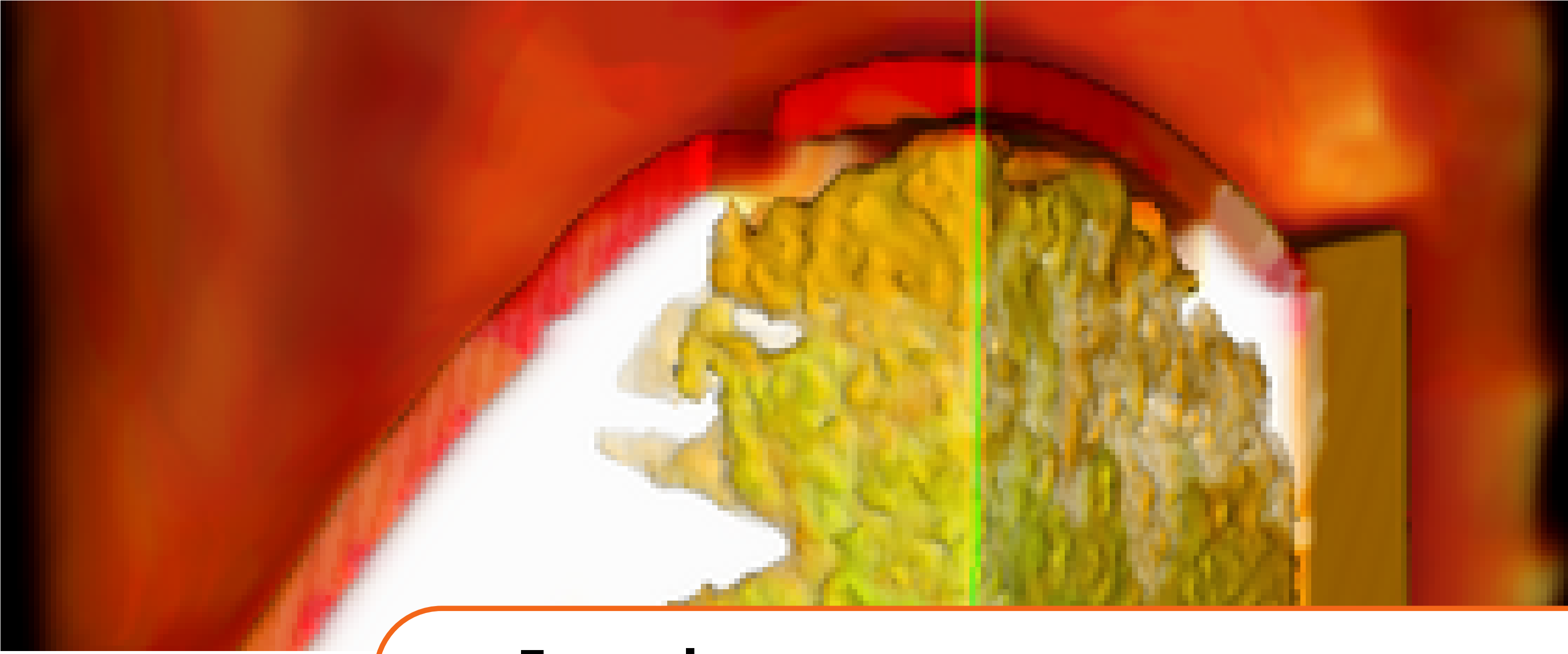
In summary, ghost cells exist between patches on the same level. They are cells that are owned by the neighboring patch but are required for computation due to the stencil width.

Extra cells exist at the edge of the computational domain and are used for boundary conditions. Extra cells exist on the boundaries of the level where no neighboring patches exist. This can either be the edge of a domain or on a coarse-fine interface.

2.5.4 Indexing grid cells

Vaango uses a straightforward indexing scheme. Each cell on a given level is uniquely identified by an x,y,z coordinate (stored in an `IntVector`). An IntVector is a vector of 3 integers that represent an X,Y,Z coordinate. See Core/Grid/Level.cc/h for more information. The following pseudo code shows how indices across levels are mapped to each other.

```
IntVector
Level::mapCellToCoarser(const IntVector& idx) const
{
  IntVector ratio = idx/d_refinementRatio;

  return ratio;
}

IntVector
Level::mapCellToFiner(const IntVector& idx) const
{
  IntVector r_ratio = grid->getLevel(d_index+1)->d_refinementRatio;
  IntVector fineCell = idx*r_ratio;

  return fineCell;
}
```



3 — Examples

This chapter will describe a set of example problems showing various stages of algorithm complexity and how the Vaango framework is used to solve the discretized form of the solutions. Emphasis will not be on the most efficient or fast algorithms, but intead will demonstrate straightforward implementations of well known algorithms within the Vaango Framework. Several examples will be given that show an increasing level of complexity.

All examples described are found in the directory src/CCA/Components/Examples .

3.1 Poisson1

Poisson1 solves Poisson's equation on a grid using Jacobi iteration. Since this is not a time dependent problem and Vaango is fundamentally designed for time dependent problems, each Jacobi iteration is considered to be a timestep. The timestep specified and computed is a fixed value obtained from the input file and has no bearing on the actual computation.

The following equation is discretized and solved using an iterative method. Each timestep is one iteration. At the end of the timestep, we the residual is computed showing the convergence of the solution and the next iteration is computed until.

The following shows a simplified form of the Poisson1 of the .h and .cc files found in the Examples directory. The argument list for some of the methods are eliminated for readibility purposes. Please refer to the actual source for a complete description of the arguments required for each method.

```
class Poisson1 : public UintahParallelComponent, public SimulationInterface {
public:
  Poisson1(const ProcessorGroup* myworld);
  virtual ~Poisson1();
  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:
  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);

   SimulationStateP sharedState_;
   double delt_;
   const VarLabel* phi_label;
   const VarLabel* residual_label;
   SimpleMaterial* mymat_;

   Poisson1(const Poisson1&);
   Poisson1& operator=(const Poisson1&);
 };
```

The private methods and data shown are the functions that are function pointers referred to in the task descriptions. The VarLabel data type stores the names of the various data that can be referenced uniquely by the data warehouse. The SimulationStateP data type is essentially a global variable that stores information about the materials that are needed by other internal Vaango framework components. SimpleMaterial is a data type that refers to the material properties.

Within each schedule function, i.e. sheduleInitialize, scheduleComputeStableTimestep, and scheduleTimeAdvance, a task is specified that has a function pointer associated with it. The function pointers point to the actual implementation of the specific task and have a different argument list than the associated schedule method.

The typical task implementation, i.e. timeAdvance() contains the following arguments: ProcessorGroup, PatchSubset, MaterialSubset, and two DataWarehouse objects. The purpose of the ProcessorGroup is to hold various MPI information such as the MPI_Communicator, the rank of the process and the number of processes that are actually being used.

3.1.1 Description of Scheduling Functions

The actual implementation with descriptions are presented following the code snippets.

```
 Poisson1::Poisson1(const ProcessorGroup* myworld)
 : UintahParallelComponent(myworld)
 {
   phi_label = VarLabel::create("phi",
   NCVariable<double>::getTypeDescription());
   residual_label = VarLabel::create("residual",
   sum_vartype::getTypeDescription());

 }

 Poisson1::~Poisson1()
 {
   VarLabel::destroy(phi_label);
   VarLabel::destroy(residual_label);
 }
```

Typical constructor and destructor for simple examples where the data label names (phi and residual) are created for data wharehouse storage and retrieval.

```
 void Poisson1::problemSetup(const ProblemSpecP& params, const ProblemSpecP&
     restart_prob_spec, GridP& /*grid*/, SimulationStateP& sharedState)
 {
   sharedState_ = sharedState;
   ProblemSpecP poisson = params->findBlock("Poisson");
```

```
  poisson->require("delt", delt_);

  mymat_ = scinew SimpleMaterial();

  sharedState->registerSimpleMaterial(mymat_);
}
```

The problemSetup is based in a xml description of the input file. The input file is parsed and the delt tag is set. The sharedState is assigned and is used to register a material and store it for later use by the Vaango internals.

```
void Poisson1::scheduleInitialize(const LevelP& level, SchedulerP& sched)
{
  Task* task = scinew Task("Poisson1::initialize",
  this, &Poisson1::initialize);

  task->computes(phi_label);
  task->computes(residual_label);
  sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
}
```

A task is defined which contains a name and a function pointer, i.e. initialize which is described later in Poisson1.cc The task defines two variables that are computed in the initialize function, phi and residual. The task is then added to the scheduler. This task is only computed once at the beginning of the simulation.

```
void Poisson1::scheduleComputeStableTimestep(const LevelP& level, SchedulerP&
    sched)
{
  Task* task = scinew Task("Poisson1::computeStableTimestep",
  this, &Poisson1::computeStableTimestep);

  task->requires(Task::NewDW, residual_label);
  task->computes(sharedState_->get_delt_label());
  sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
}
```

A task is defined for the computing the stable timestep and uses the function pointer, computeStable-Timestep defined later in Poisson1.cc. This requires data from the New DataWarehouse, and the next timestep size is computed and stored.

```
void
Poisson1::scheduleTimeAdvance( const LevelP& level, SchedulerP& sched)
{
  Task* task = scinew Task("Poisson1::timeAdvance", this, &Poisson1::
      timeAdvance);

  task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
  task->computes(phi_label);
  task->computes(residual_label);
  sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
}
```

The timeAdvance function is the main function that describes the computational algorithm. For simple examples, the entire algorithm is usually defined by one task with a small set of data dependencies. However, for more complicated algorithms, it is best to break the algorithm down into a set of tasks with each task describing its own set of data dependencies.

For this example, a single task is described and the timeAdvance function pointer is specified. Data from the previous timestep (OldDW) is required for the current timestep. For a simple seven (7) point stencil, only one level of ghost cells is required. The algorithm is set up for nodal values, the ghost cells are specified by the the Ghost::AroundNodes syntax. The task computes both the new data values for phi and a residual.

3.1.2 Description of Computational Functions

```
void Poisson1::computeStableTimestep(const ProcessorGroup* pg, const
    PatchSubset* /*patches*/, const MaterialSubset* /*matls*/, DataWarehouse*,
     DataWarehouse* new_dw)
{
  if(pg->myrank() == 0){
    sum_vartype residual;
    new_dw->get(residual, residual_label);
    cerr << "Residual=" << residual << '\n';
  }
  new_dw->put(delt_vartype(delt_), sharedState_->get_delt_label());
}
```

In this particular example, no timestep is actually computed, instead the original timestep specified in the input file is used and stored in the data warehouse (new_dw→put(delt_vartype(delt_), sharedState_→get_delt_label())). The residual computed in the main timeAdvance function is retrieved from the data warehouse and printed out to standard error for only the processor with a rank of 0.

```
void Poisson1::initialize(const ProcessorGroup*, const PatchSubset* patches,
    const MaterialSubset* matls, DataWarehouse* /*old_dw*/, DataWarehouse*
    new_dw)
{
  int matl = 0;
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
```

The node centered variable (NCVariable<double>phi) has space reserved in the DataWarehouse for the given patch and the given material (int matl = 0;). The phi variable is initialized to 0 for every grid node on the patch.

```
    NCVariable<double> phi;
    new_dw->allocateAndPut(phi, phi_label, matl, patch);
    phi.initialize(0.);
```

The boundary faces on the xminus face of the computational domain are specified and set to a value of 1. All other boundary values are set to a value of 0 as well as the internal nodes via the phi.initialize(0.) construct. Vaango provides helper functions for determining which nodes are on the boundaries. In addition, there are convenient looping constructs such as NodeIterator that alleviate the need to specify triply nested loops for visiting each node in the domain.

```
    if(patch->getBCType(Patch::xminus) != Patch::Neighbor){
      IntVector l,h;
      patch->getFaceNodes(Patch::xminus, 0, l, h);

      for(NodeIterator iter(l,h); !iter.done(); iter++){
        phi[*iter]=1;
      }
    }
    new_dw->put(sum_vartype(-1), residual_label);
  }
}
```

The initial residual value of -1 is stored at the beginning of the simulation.

The main computational algorithm is defined in the timeAdvance function. The overall algorithm is based on a simple Jacobi iteration step.

```
void Poisson1::timeAdvance(const ProcessorGroup*, const PatchSubset* patches,
     const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw
    )
{
  int matl = 0;
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    constNCVariable<double> phi;
```

Data from the previous timestep is retrieved from the data warehouse and copied to the current timestep's phi variable (newphi).

```
    old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
    NCVariable<double> newphi;

    new_dw->allocateAndPut(newphi, phi_label, matl, patch);
    newphi.copyPatch(phi, newphi.getLowIndex(), newphi.getHighIndex());
```

The indices for the patch are obtained and altered depending on whether or not the patch's internal boundaries are on the coincident with the grid domain. If the patch boundaries are the same as the grid domain, the boundary values are not overwritten since the lower and upper indices are modified to only specify internal nodal grid points.

```
    double residual=0;
    IntVector l = patch->getNodeLowIndex__New();
    IntVector h = patch->getNodeHighIndex__New();

    l += IntVector(patch->getBCType(Patch::xminus) == Patch::Neighbor?0:1,
    patch->getBCType(Patch::yminus) == Patch::Neighbor?0:1,
    patch->getBCType(Patch::zminus) == Patch::Neighbor?0:1);
    h -= IntVector(patch->getBCType(Patch::xplus)  == Patch::Neighbor?0:1,
    patch->getBCType(Patch::yplus)  == Patch::Neighbor?0:1,
    patch->getBCType(Patch::zplus)  == Patch::Neighbor?0:1);
```

The Jacobi iteration step is applied at each internal grid node. The residual is computed based on the old and new values and stored as a reduction variable (sum_vartype) in the data warehouse.

```
    //-----------------------------------
    //  Stencil
    for(NodeIterator iter(l, h);!iter.done(); iter++){
      IntVector n = *iter;

      newphi[n]=(1./6)*(
      phi[n+IntVector(1,0,0)] + phi[n+IntVector(-1,0,0)] +
      phi[n+IntVector(0,1,0)] + phi[n+IntVector(0,-1,0)] +
      phi[n+IntVector(0,0,1)] + phi[n+IntVector(0,0,-1)]);

      double diff = newphi[n] - phi[n];
      residual += diff * diff;
    }
    new_dw->put(sum_vartype(residual), residual_label);
  }
}
```

3.1.3 Input file

The input file that is used to run this example is given below and is given in SCIRun/src/Packages/Uintah/StandAlone/inputs/Examples/poisson1.ups . Relevant sections of the input file that can be modified are found in the <Time> section, and the <Grid> section, specifically, the number of patches and the grid resolution.

```
<Uintah_specification>
  <Meta>
    <title>Poisson1 test</title>
  </Meta>
  <SimulationComponent>
    <type> poisson1 </type>
  </SimulationComponent>
  <Time>
    <maxTime>       1.0       </maxTime>
    <initTime>      0.0       </initTime>
    <delt_min>      0.00001   </delt_min>
    <delt_max>      1         </delt_max>
    <max_Timesteps> 100        </max_Timesteps>
    <timestep_multiplier>  1  </timestep_multiplier>
  </Time>
  <DataArchiver>
    <filebase>poisson.uda</filebase>
    <outputTimestepInterval>1</outputTimestepInterval>
    <save label = "phi"/>
    <save label = "residual"/>
    <checkpoint cycle = "2" timestepInterval = "1"/>
  </DataArchiver>
  <Poisson>
    <delt>.01</delt>
    <maxresidual>.01</maxresidual>
  </Poisson>
  <Grid>
    <Level>
      <Box label = "1">
        <lower>      [0,0,0]          </lower>
        <upper>      [1.0,1.0,1.0]    </upper>
        <resolution>[50,50,50]       </resolution>
        <patches>    [2,1,1]          </patches>
      </Box>
    </Level>
  </Grid>
</Uintah_specification>
```

Running the Poisson1 Example

To run the poisson1.ups example,

```
  cd ~/SCIRun/dbg/Packages/Uintah/StandAlone/
```

create a symbolic link to the inputs directory:

```
  ln -s ~/SCIRun/src/Packages/Uintah/StandAlone/inputs
```

For a single processor run type the following:

```
  vaango inputs/Examples/poisson1.ups
```

For a two processor run, type the following:

```
  mpirun -np 2 vaango -mpi inputs/Examples/poisson1.ups
```

Changing the number of patches in the poisson1.ups and the resolution, enables you to run a more refined problem on more processors.

ADVICE: For non-AMR problems, it is advised to have at least the same number of patches as processors. You can always have more patches than processors, but you cannot have fewer patches than processors.

3.2 Poisson2

The next example also solves the Poisson's equation but instead of iterating in time, the subscheduler feature iterates within a given timestep, thus solving the problem in one timestep. The use of the subscheduler is important for implementing algorithms which solve non-linear problems which require iterating on a solution for each timestep.

The majority of the schedule and computational functions are similar to the Poisson1 example and are not repeated. Only the revised code is presented with explanations about the new features of Uintah.

```
void Poisson2::scheduleTimeAdvance( const LevelP& level, SchedulerP& sched)
{
  Task* task = scinew Task("timeAdvance", this, &Poisson2::timeAdvance, level
      , sched.get_rep());
  task->hasSubScheduler();
  task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
  task->computes(phi_label);
  LoadBalancer* lb = sched->getLoadBalancer();
  const PatchSet* perproc_patches = lb->getPerProcessorPatchSet(level);
  sched->addTask(task, perproc_patches, sharedState_->allMaterials());
}
```

Within this function, the task is specified with two additional arguments, the level and the scheduler sched.get_rep() . The task must also set the flag that a subscheduler will be used within the scheduling of the various tasks. Similar code to the Poisson1 example is used to specify what data is required and computed during the actual task execution. In addition, a loadbalancer component is required to query the patch distribution for each level of the grid. The task is then added to the top level scheduler with the requisite information, i.e. patches and materials.

The actual implementation of the timeAdvance function is also different from the Poisson1 example. The code is specified below with text explaining the use of the subscheduler. The new feature of the subscheduler shows the creation of a the iterate task within the subscheduler. This task will perform the actual Jacobi iteration for a given timestep.

```
void Poisson2::timeAdvance(const ProcessorGroup* pg, const PatchSubset*
    patches, const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse
    * new_dw, LevelP level, Scheduler* sched)
{
```

The subscheduler is instantiated and initialized.

```
  SchedulerP subsched = sched->createSubScheduler();
  subsched->initialize();
  GridP grid = level->getGrid();
```

An iterate task is created and added to the subscheduler. The typical computes and requires are specified for a 7 point stencil used in Jacobi iteration scheme with one layer of ghost cells. The new task is added to the subscheduler. A residual variable is only computed within the subscheduler and not passed back to the main scheduler. This is in contrast to the phi variable which was specified in scheduleTimeAdvance in the computes, as well as being specified in the computes for the subscheduler. Any variables that are only computed and used in an iterative step of an algorithm do not need to be added to the dependency specification for the top level task.

```
  // Create the tasks
  Task* task = scinew Task("iterate", this, &Poisson2::iterate);
  task->requires(Task::OldDW, phi_label, Ghost::AroundNodes, 1);
  task->computes(phi_label);
  task->computes(residual_label);
  subsched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
```

The subscheduler has its own data wharehouse that is separate from the top level's scheduler's data warehouse. This data warehouse must be initialized and any data from the top level's data warehouse must be passed to the subscheduler's version. This data resides in the data warehouse position NewDW .

```
    // Compile the scheduler
    subsched->advanceDataWarehouse(grid);
    subsched->compile();

    int count = 0;
    double residual;
    subsched->get_dw(1)->transferFrom(old_dw, phi_label, patches, matls);
```

Within each iteration, the following must occur for the subscheduler: the data warehouse's new data must be moved to the OldDW position, since any new values will be stored in NewDW and the old values cannot be overwritten. The OldDW is referred to in the subscheduler via the subsched->get_dw(0) and the NewDW is referred to in the subscheduler via subsched->get_dw(1) . Once the iteration is deemed to have met the tolerance, the data from the subscheduler is transferred to the scheduler's data warehouse.

```
    // Iterate
    do {
      subsched->advanceDataWarehouse(grid);
      subsched->get_dw(0)->setScrubbing(DataWarehouse::ScrubComplete);
      subsched->get_dw(1)->setScrubbing(DataWarehouse::ScrubNonPermanent);
      subsched->execute();

      sum_vartype residual_var;
      subsched->get_dw(1)->get(residual_var, residual_label);
      residual = residual_var;

      if(pg->myrank() == 0)
        cerr << "Iteration " << count++ << ", residual=" << residual << '\n';
    } while(residual > maxresidual_);

    new_dw->transferFrom(subsched->get_dw(1), phi_label, patches, matls);
  }
```

The iteration cycle is identical to Poisson1 's timeAdvance algorithm using Jacobi iteration with a 7 point stencil. Refer to the discussion about the algorithm implementation in the Poisson1 description.

```
  void Poisson2::iterate(const ProcessorGroup*, const PatchSubset* patches,
       const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw)
  {
    for(int p=0;p<patches->size();p++){
      const Patch* patch = patches->get(p);
      for(int m = 0;m<matls->size();m++){
        int matl = matls->get(m);
        constNCVariable<double> phi;
        old_dw->get(phi, phi_label, matl, patch, Ghost::AroundNodes, 1);
        NCVariable<double> newphi;
        new_dw->allocateAndPut(newphi, phi_label, matl, patch);
        newphi.copyPatch(phi, newphi.getLow(), newphi.getHigh());
        double residual=0;
        IntVector l = patch->getNodeLowIndex__New();
        IntVector h = patch->getNodeHighIndex__New();
        l += IntVector(patch->getBCType(Patch::xminus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::yminus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::zminus) == Patch::Neighbor?0:1);
        h -= IntVector(patch->getBCType(Patch::xplus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::yplus) == Patch::Neighbor?0:1,
        patch->getBCType(Patch::zplus) == Patch::Neighbor?0:1);
        for(NodeIterator iter(l, h);!iter.done(); iter++){
          newphi[*iter]=(1./6)*(
          phi[*iter+IntVector(1,0,0)]+phi[*iter+IntVector(-1,0,0)]+
          phi[*iter+IntVector(0,1,0)]+phi[*iter+IntVector(0,-1,0)]+
          phi[*iter+IntVector(0,0,1)]+phi[*iter+IntVector(0,0,-1)]);
          double diff = newphi[*iter]-phi[*iter];
          residual += diff*diff;
        }
        new_dw->put(sum_vartype(residual), residual_label);
      }
    }
  }
```

The input file src/StandAlone/inputs/Examples/poisson2.ups is very similar to the poisson1.ups file shown above. The only additional tag that is used is the <maxresidual > specifying the tolerance within the iteration performed in the subscheduler.

To run the poisson2 input file execute the following in the dbg build StandAlone directory:

```
vaango inputs/Examples/poisson2.ups
```

3.3 Burger

In this example, the inviscid Burger's equation is solved in three dimensions:

$$\frac{du}{dt} = -u\frac{du}{dx} \tag{3.1}$$

with the initial conditions:

$$u = \sin(\pi x) + \sin(2\pi y) + \sin(3\pi z) \tag{3.2}$$

using Euler's method to advance in time. The majority of the code is very similar to the Poisson1 example code with the differences shown below.

The initialization of the grid values for the unknown variable, u, is done at every grid node using the NodeIterator construct. The x,y,z values for a given grid node is determined using the function, patch->getNodePosition(n), where n is the nodal index in i,j,k space.

```
void Burger::initialize(const ProcessorGroup*, const PatchSubset* patches,
    const MaterialSubset* matls, DataWarehouse*, DataWarehouse* new_dw)
{
  int matl = 0;
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);

    NCVariable<double> u;
    new_dw->allocateAndPut(u, u_label, matl, patch);

    //Initialize
    // u = sin( pi*x ) + sin( pi*2*y ) + sin(pi*3z )
    IntVector l = patch->getNodeLowIndex__New();
    IntVector h = patch->getNodeHighIndex__New();

    for( NodeIterator iter=patch->getNodeIterator__New(); !iter.done(); iter
        ++ ){
      IntVector n = *iter;
      Point p = patch->nodePosition(n);
      u[n] = sin( p.x() * 3.14159265358 ) + sin( p.y() * 2*3.14159265358)  +
             sin( p.z() * 3*3.14159265358);
    }
  }
}
```

The timeAdvance function is quite similar to the Poisson1's timeAdvance routine. The relavant differences are only shown.

```
void Burger::timeAdvance(const ProcessorGroup*, const PatchSubset* patches,
    const MaterialSubset* matls, DataWarehouse* old_dw, DataWarehouse* new_dw)
{
  int matl = 0;
  //Loop for all patches on this processor
  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);
    . . . . .
```

The grid spacing and timestep values are stored.

```
    // dt, dx
    Vector dx = patch->getLevel()->dCell();
    delt_vartype dt;
    old_dw->get(dt, sharedState_->get_delt_label());
    . . . . .
```

Refer to the description in Poisson1 about the specification of the NodeIterator limits. The Euler algorithm is applied to solve the ordinary differential equation in time.

```
    //Iterate through all the nodes
    for(NodeIterator iter(l, h);!iter.done(); iter++){
      IntVector n = *iter;
      double dudx = (u[n+IntVector(1,0,0)] - u[n-IntVector(1,0,0)]) /(2.0 *
          dx.x());
      double dudy = (u[n+IntVector(0,1,0)] - u[n-IntVector(0,1,0)]) /(2.0 *
          dx.y());
      double dudz = (u[n+IntVector(0,0,1)] - u[n-IntVector(0,0,1)]) /(2.0 *
          dx.z());
      double du = - u[n] * dt * (dudx + dudy + dudz);
      new_u[n]= u[n] + du;
    }
```

Zero flux Neumann boundary conditions are applied to the node points on each of the grid faces.

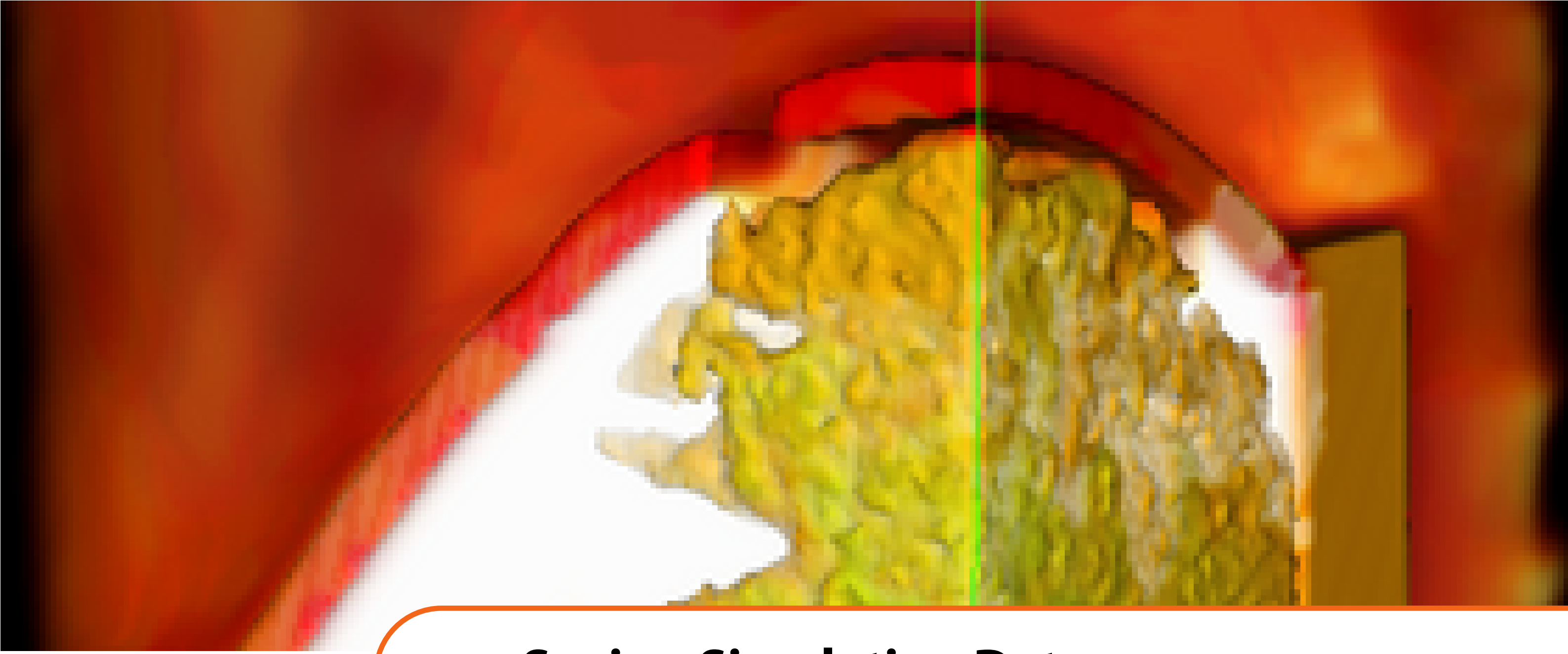
```
    //__________________________________
    // Boundary conditions: Neumann
    // Iterate over the faces encompassing the domain
    vector<Patch::FaceType>::const_iterator iter;
    vector<Patch::FaceType> bf;
    patch->getBoundaryFaces(bf);
    for (iter  = bf.begin(); iter != bf.end(); ++iter){
      Patch::FaceType face = *iter;

      IntVector axes = patch->faceAxes(face);
      int P_dir = axes[0]; // find the principal dir of that face

      IntVector offset(0,0,0);
      if (face == Patch::xminus || face == Patch::yminus || face == Patch::
          zminus){
        offset[P_dir] += 1;
      }
      if (face == Patch::xplus || face == Patch::yplus || face == Patch::
          zplus){
        offset[P_dir] -= 1;
      }

      Patch::FaceIteratorType FN = Patch::FaceNodes;
      for (CellIterator iter = patch->getFaceIterator__New(face,FN);!iter.
          done(); iter++){
```

```
          IntVector n = *iter;
          new_u[n] = new_u[n + offset];
        }
      }
    }
  }
```

The input file for the Burger (burger.ups) problem is very similar to the poisson1.ups with the addition, that the timestep increment used in the timeAdvance is quite small, 1.e-4 for stability reasons.

To run the Burger input file execute the following in the dbg build StandAlone directory:

```
  vaango inputs/Examples/burger.ups
```



4 — Saving Simulation Data

The UCF automatically saves data as specified in the .ups file. This works for all data that is stored in the Data Warehouse. However, there are times when a component will need to save its own data. (It is preferable that the Data Warehouse be updated to manage this Component data, but sometimes this is not expedient...) In these cases, here is an outline on how to save that data:

- Create a function that will save your data:

```
void
Component::saveMyData( const PatchSubset * patches )
{
  ...
}
```

- From the last Task in your algorithm, call the function:

```
saveMyData( patches );
```

- In the function, you need to do several things: Make sure that it is an output timestep. (Your component must have saved a pointer to the Data Archiver. Most components already do this... if not, you can do it in Component::problemSetup() .)

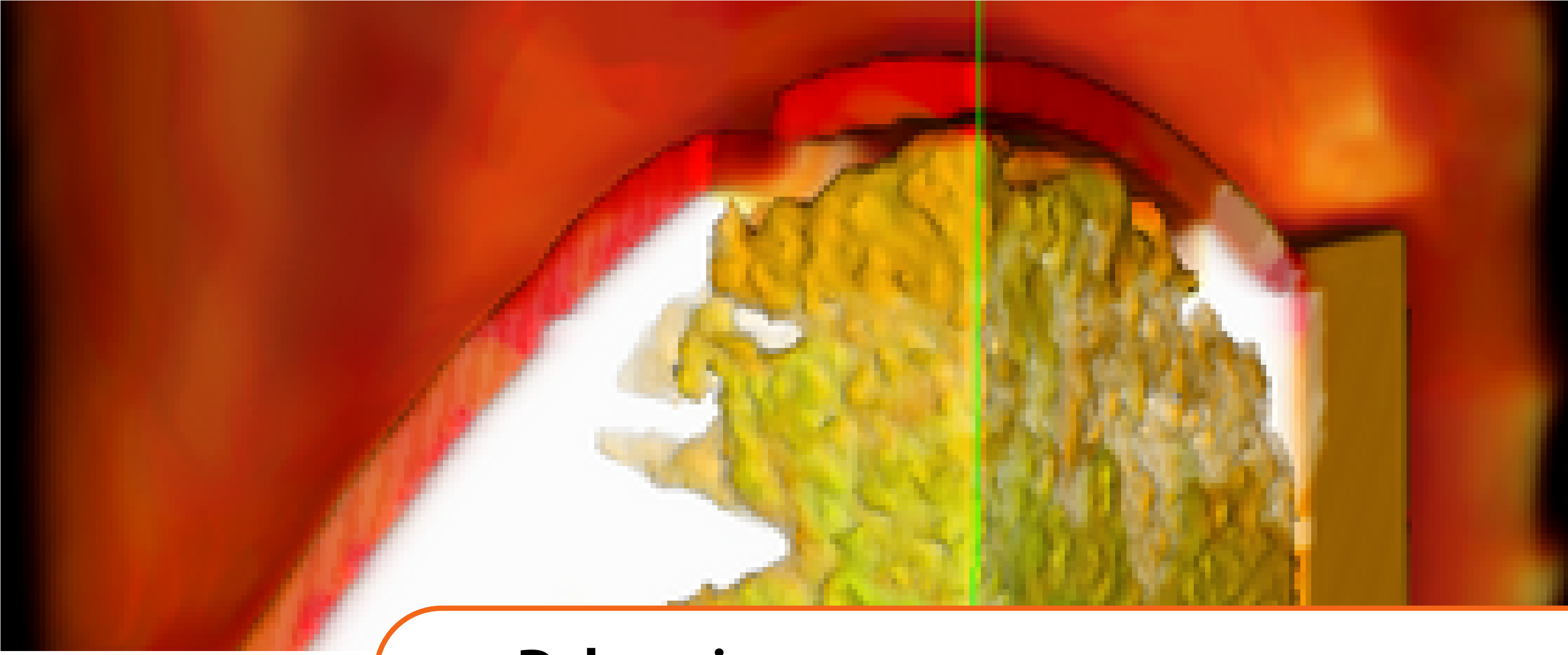
```
if( dataArchiver_->isOutputTimestep() ) {
  // Dump your data... (see below)
}
```

- You need to create a directory to put the data in:

```
const int      & timestep = d_sharedState->getCurrentTopLevelTimeStep();
ostringstream   data_dir;

// This gives you ".../simulation.uda.###/t#####/mydata/
data_dir << dataArchiver()->getOutputLocation() << "/t" << setw(5) <<
    setfill('0') << timestep
<< "/mydata";

int result = MKDIR( data_dir.c_str(), 0777 );  // Make sure you #
    include <Core/OS/Dir.h>

if( result != 0 ) {
  ostringstream error;
  error << "Component::saveMyData(): couldn't create directory '") +
      data_dir.str() + "' (" << result << ").";
    throw InternalError( error.str(), __FILE__, __LINE__);
  }
```

- Loop over the patches and save your data... (Note, depending on how much data you have, you need to determine whether you want to write the files as a binary file, or as an ascii file. Ascii files are easier for a human to look at for errors, but take a lot more time to read and space to write.)

```
  for( int pIndex = 0; pIndex < patches->size(); pIndex++ ){
    const Patch * patch = patches->get( pIndex );
    ostringstream file_name;
    // WARNING... not sure this is the correct naming convention...
       PLEASE VERIFY AND UPDATE THIS DOC.
    file_name << data_dir << "/p" << setw(5) << setfill('0') << pIndex;

    ofstream output( ceFileName, ios::out | ios::binary);
    if( !output ) {
      throw InternalError( string( "Component::saveMyData(): couldn't
          open file '") + file_name.str() + "'.",
      __FILE__, __LINE__);
    }

    // Output data
    for( loop over data ) {
      output << data...
    }

    output.close();
  } // end for pIndex
```



5 — Debugging

Debugging multi-processor software can be very difficult. Below are some hints on how to approach this.

5.1 Debugger

Use a debugger such as GDB to attach to the running program. Note, if you are an Emacs user, running GDB through Emacs' gdb-mode makes debugging even easier!

5.1.1 Serial Debugging

Whenever possible, debug vaango running in serial (ie. no MPI). This is much easier for many reasons (though may also not be possible in many situations). To debug serially, use:

```
gdb vaango <input_file>
```

5.1.2 Parallel Debugging

If it is necessary to use a parallel run of vaango to debug, a few things must take place.

1) The helpful macro `WAIT_FOR_DEBUGGER();` must be inserted into the code (vaango.cc) just before the real execution of the components begins ctl->run():

```
WAIT_FOR_DEBUGGER();
ctl->run();
```

Recompile the code (will only take a few seconds), then run through MPI as normal.

Caveats

1. Unless you know that the error occurs on a specific processor, you will need to attach a debugger to every process... so it behooves the developer to narrow the problem down to as few processors as possible.
2. If you did know the processor, then the `WAIT_FOR_DEBUGGER()` will need to be placed inside an 'if' statement to check for the correct MPI Rank.
3. Try to use only one node. If you need more 'processors', you (most likely) can double up on the same machine. If vaango must span multiple nodes, then you will need to log into each node separately to attach debuggers (see below).

5.1.3 Running Vaango

Run vaango :

```
mpirun -np 8 -m mpihosts vaango inputs/ARCHES/helium_1m.ups
```

Create a new terminal window for each processor you are using. (In the above case, 8 windows.)

You will notice that the output from vaango will include lines (one for each processor used) like this:

```
updraft2:22793 waiting for debugger
```

Attach to each process in the following manner (where PID is the process id number (22793 in the above case)):

```
cd Uintah/bin/StandAlone
gdb -p <PID>
```

Within GDB, you will then need to break each process out of the WAIT loop in the following manner:

```
up 2
set wait=false
cont
```

The above GDB commands must be run from each GDB session. Once all the WAITs are stopped, vaango will begin to run.