SCI INSTITUTE
TECHNICAL REPORT

Uintah Application Development

John A. Schmidt

UUSCI-2008-005

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

November 6, 2008

Abstract:

The Uintah framework for solving partial differential equations on structured adaptive mesh refinement grids is described so that individuals wishing to understand and develop new components can do so. An overview of the installation procedure for Uintah, as well as the framework components including a high level view of the scheduler, and the tasks is presented. A simple poisson solver example is developed that outlines the basic elements that are required to implement a new computational component. The poisson solver is then extended to illustrate the novel features of the scheduling algorithm that are required for any iterative algorithm. This is then followed by a description of a component that solves the Burger equation. Finally, discussions of the boundary conditions both within the component and in the input file are described with additional information about the input file format for a generic Uintah component.

THE UNIVERSITY OF UTAH

Uintah Application Development

John A. Schmidt

November 6, 2008

Contents

Chapter 1

Uintah Software Components

Three software components need to be installed in order to develop and customize Uintah. These include Thirdpary, SCIRun, and Uintah. Thirdparty is composed of several libraries needed to build the visualization tool, SCIRun. SCIRun is a dataflow visualization tool which understands the data output format of Uintah. And finally, Uintah is a framework for performed structured Adaptive Mesh Refinement calculations for partial differential equations.

1.0.1 Downloading the Software

Uintah/SCIRun can be obtained via svn from the following website:

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

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

The Thirdparty library can similarly be obtained via:

```
svn co https://code.sci.utah.edu/svn/Thirdparty/3.1.0 Thirdparty
```

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

1.0.2 Installing Thirdparty, SCIRun, and Uintah

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 /Thirdpartythirdparty.src/

After reading the README.txt file type the follow as the root user:

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

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/3.1.0/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/3.1.0/Linux/gcc-4.3.1-2-32bit/
```

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

Chapter 2

Overview of the Uintah Framework

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.

2.1 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.

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 imple-

menting 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.

2.3 Tasks and Scheduler Description – Programmer Interface

2.3.1 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

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.

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`.

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`.

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.

2.3.2 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.

Chapter 3

Examples

This chapter will describe a set of example problems showing various stages of algorithm complexity and how the Uintah 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 Uintah Framework. Several examples will be given that show an increasing level of complexity that will serve as a guide to others interested in implmenting structured AMR algorithms for PDEs.

All examples described are found in the directory *SCIRun/src/Packages/Uintah/CCA/Components/Examples*

3.1 Poisson1

Poisson1 is solves Poisson's equation on a grid using Jacobi iteration. Since this is not a time dependent problem and Uintah 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_d
                               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 Uintah 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 Uintah 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, `computeStableTimestep` 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 conditions are applied to each face of the compuational domain using the construct given below. In this particular example based on the input file described a bit later, the value of 1.0 is applied to the xminus face. All other faces will have the Phi variable assigned the value of 0.0.

```
  for (Patch::FaceType face = Patch::startFace; face <= Patch::endFace;
         face=Patch::nextFace(face)) {

      if (patch->getBCType(face) == Patch::None) {
        int numChildren = patch->getBCDataArray(face)->getNumberChildren(matl);
        for (int child = 0; child < numChildren; child++) {
          Iterator nbound_ptr, nu;

          const BoundCondBase* bcb = patch->getArrayBCValues(face,matl,"Phi",nu
                                                             nbound_ptr,child);

          const BoundCond<double>* bc =
            dynamic_cast<const BoundCond<double>*>(bcb);
          double value = bc->getValue();
          for (nbound_ptr.reset(); !nbound_ptr.done();nbound_ptr++) {
            phi[*nbound_ptr]=value;
```

```
        }
      }
    }
  }


  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);
}
```

```
}
```

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 boundary conditions, 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>
  <BoundaryConditions>
    <Face side = "x-">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 1. </value>
      </BCType>
    </Face>
    <Face side = "x+">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 0. </value>
      </BCType>
    </Face>
    <Face side = "y-">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 0. </value>
      </BCType>
    </Face>
    <Face side = "y+">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 0. </value>
      </BCType>
    </Face>
    <Face side = "z-">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 0. </value>
      </BCType>
    </Face>
    <Face side = "z+">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 0. </value>
```

```
        </BCType>
      </Face>
    </BoundaryConditions>
    <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:

```
sus inputs/Examples/poisson1.ups
```

For a two processor run, type the following:

```
mpirun -np 2 sus -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 subcheduler 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 */SCIRun/src/Packages/Uintah/StandAlone/inputs/Ex-*

amples/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 */SCIRun/d-bg/Packages/Uintah/StandAlone* directory:

```
sus inputs/Examples/poisson2.ups
```

3.3 Burger

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

```
du/dt = -u du/dx
```

with the following initial conditions:

```
u = sin(pi x) + sin(2pi y) + sin(3pi z)
```

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)  +  s
    }
  }
}
```

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.
  double dudy = (u[n+IntVector(0,1,0)] - u[n-IntVector(0,1,0)]) /(2.0 * dx.
  double dudz = (u[n+IntVector(0,0,1)] - u[n-IntVector(0,0,1)]) /(2.0 * dx.
  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::zmin
    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(
        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 */SCIRun/dbg/-Packages/Uintah/StandAlone* directory:

```
sus inputs/Examples/burger.ups
```

Chapter 4

Specifying Boundary Conditions

The input file has a section under the tag BoundaryConditions with the following format:

```
<Face side = "x+">
    <BCType id = "0"   label = "Phi"     var = "Dirichlet">
          <value> 0. </value>
    </BCType>
</Face>
```

The **id** stands for the material number and can be either an integer starting from `0` or it can be set to `all`. The **label** denotes the string variable name that is actually specified in the algorithmic section of the component and will be further described below. The **var** is usually either Dirichlet or Neumann, but it is really up to the algorithmic component to describe how the boundary condition is applied. Finally, there is the **value** tag that can either be a floating point value or a vector value specified in the following manner, `[0.,0.,0.]`

Typically within the application of the boundary conditions, different routines are derived based on a Neumann or Dirichlet boundary condition. A common way of applying the boundary conditions and described below in the following code snippet is to loop over all the faces. Then check to see if a patch face is not on the interior (getBCType(face) == Patch::None), then retrieve the actual boundary condition information.

Within each face, boundary conditions may be assigned as the composition of various geometry regions. The most common configuration is the **side** shown above. However, one can assign boundary conditions that would approximate an inlet or outlet condition, that is a circle within a side given as:

```
    <Face side = "x+">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
            <value> 0. </value>
      </BCType>
  </Face>

<Face cirlce = "x+" origin = "0 0 0" radius = "1.0">
      <BCType id = "0"   label = "Phi"     var = "Dirichlet">
            <value> 0. </value>
      </BCType>
  </Face>
```

With situations such as this, the application code must loop over all of the geometry regions denoted by the following.

```
          int numChildren =
            patch->getBCDataArray(face)->getNumberChildren(matl);
          for (int child = 0; child < numChildren; child++) {



 for (Patch::FaceType face = Patch::startFace; face <= Patch::endFace;
           face=Patch::nextFace(face)) {

        if (patch->getBCType(face) == Patch::None) {
          int numChildren =
            patch->getBCDataArray(face)->getNumberChildren(matl);
          for (int child = 0; child < numChildren; child++) {
            Iterator nbound_ptr, nu;

            const BoundCondBase* bcb = patch->getArrayBCValues(face,matl,"Phi",
```

```
                                                          nu,nbound_ptr,
                                                          child);

          const BoundCond<double>* bc =
            dynamic_cast<const BoundCond<double>*>(bcb);

          double value = bc->getValue();
          string type = bc->getType();

          if (type == "Dirichlet")
             for (nbound_ptr.reset(); !nbound_ptr.done();nbound_ptr++)
                 phi[*nbound_ptr]=value;


       }

    }
}
```

Within each face, an **Iterator** is returned that contains the list of cells and nodes that must be visited with the boundary conditions. In the above example, the nodal boundary points are returned and are subsequently used to apply the Dirichlet boundary conditions. In the above example, the *nu* is used to denote the not-used cell iterator. However, if the algorithm were cell-centered, a snippet of code would look like this:

```
Iterator cells,nodes;

const BoundCondBase* bcb = patch->getArrayBCValues(face,matl,"Phi",cells
                                                    nodes,child);

for (cells.reset(); !cells.done();cells++)
       phi[*cells] = value;
```

Chapter 5

Input File Specification

The Uintah framework uses xml input files to specify the various parameters required in any simulation component. The application developer is free to use any tags to specify the data needed by the simulation. The essential tags that are required by Uintah include the following:

```
<Uintah_specification>

        <SimulationComponent>

<Time>

<DataArchiver>

<Grid>
```

The poisson1.ups file (found in /inputs/Examples/poisson1.ups) is examined closely to illustrate essential features of what each of the above tags mean and any subsequent tags that required to fill out the problem description.

Each input file must contain the tag **Uintah_specification** and within this tag each of the major tags are specified including **SimulationComponent**, **Time**, **DataArchiver**, and **Grid**.

```
<Uintah_specification>

    <SimulationComponent>
        <type> poisson1 </type>
    </SimulationComponent>
```

The **SimulationComponent** indicates which component is called from the many components that are a part of the Uintah framework.

```
    <Time>
        <maxTime>       1.0       </maxTime>
        <initTime>      0.0       </initTime>
        <delt_min>      0.00001   </delt_min>
        <delt_max>      1         </delt_max>
        <max_Timesteps> 10        </max_Timesteps>
        <timestep_multiplier>  1  </timestep_multiplier>
     </Time>
```

Within the **Time** tag, the maximum time (**maxTime**), the initial time (**initTime**), the initial timestep size (**delt_min**), the maximum timestep size (**delt_max**), the number of timesteps (**max_Timesteps**), and the timestep multiplier (**timestep_multiplier**) are specified. The **max_Timesteps** is an optional tag and can be used to aid for debugging and algorithm implementation and verification. The units of time are completely problem dependent, but consistency must be followed.

```
    <DataArchiver>
        <filebase>poisson.uda</filebase>
        <outputTimestepInterval>1</outputTimestepInterval>
        <save label = "phi"/>
        <save label = "residual"/>
        <checkpoint cycle = "2" timestepInterval = "1"/>
    </DataArchiver>
```

The **DataArchiver** describes the directory containing all of the simulation data generated and what data is saved during the computation. In

the above example, data is stored in the directory poisson.uda. By convention, the ".uda" designation is affixed to represent the Uintah Data Archive name. The tag **outputTimestepInterval** describes how frequently data is saved. In this case, data is saved for each timestep. The tag **save** describe the various data that is saved. In the above case, two pieces of data are saved, the Phi variable and the residual. Finally, there is the restarting a long running simulation by checkpointing the data with a certain frequency (**timestepInterval**) and how many checkpointed data sets are retained before being overwritten (**cycle**).

Each simulation component has its own section that is completely user defined. In the case of the Poisson example, there is a section of the input file that relates to input parameters to the poisson solver.

```
<Poisson>
   <delt>.01</delt>
   <maxresidual>.01</maxresidual>
</Poisson>
```

The tags **delt** and **maxresidual** are specified. Depending on the complexity of the component, this section of the input file can be either very simple such as above, or very detailed.

Finally, the **Grid** tag is used to describe both the boundary conditions and the description of the grid. The boundary conditions are specified for each face of the domain and can have any number of **BCType** tags depending on the complexity of the equations.

Within the **Grid** section, the domain boundaries (**lower** and **upper**) are specified as well as the resolution. The Uintah framework allows for easy parallelization and the tag **patches** describe how many patches are allocated along each dimension. In this example, the grid domain is decomposed into 2 patches along the x direction and 1 in each the y and z directions.

```
<Grid>
  <BoundaryConditions>
     <Face side = "x-">
       <BCType id = "0"   label = "Phi"     var = "Dirichlet">
                          <value> 1. </value>
       </BCType>
```

```
      </Face>

   . . . .

   </BoundaryConditions>

   <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>
```