THE UNIVERSITY OF UTAH

Using Multiple DAGS to Ensure Portability and Scalability in Large Scale Computation Using Uintah

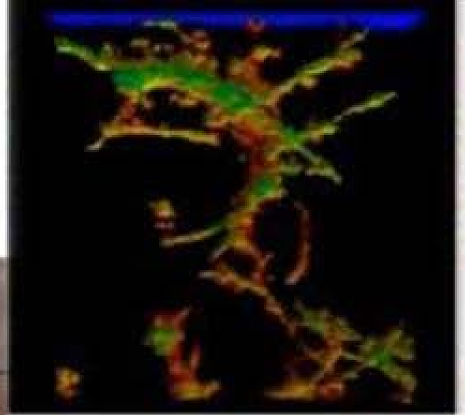
www.uintah.utah.edu

John Schmidt

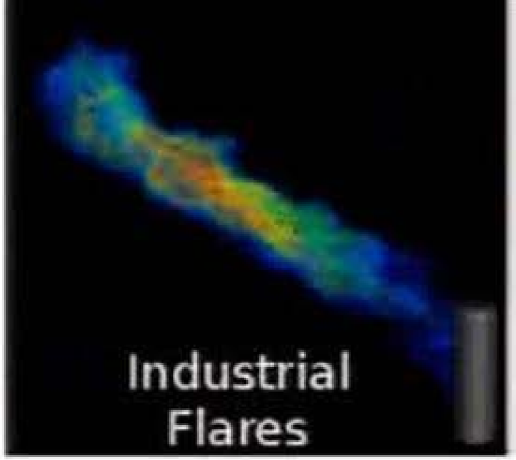
1. Outline of Uintah Component Model
2. Burgers Equation Example Code
3. Task Graph Generation
4. Scalability Examples
5. MiniAero Development & Insights

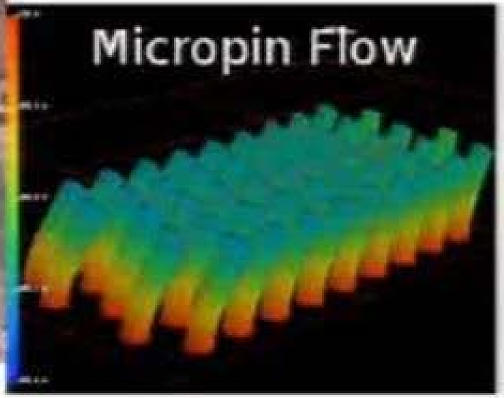
Message : Coding for an abstract graph interface provides portability

Example Uintah Applications

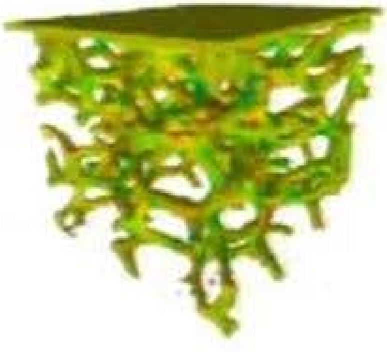


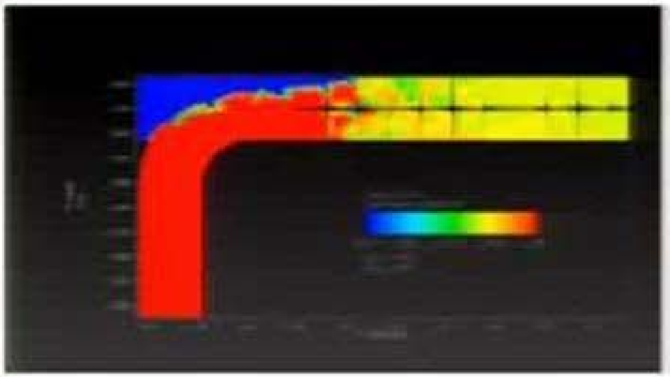
Angiogenesis



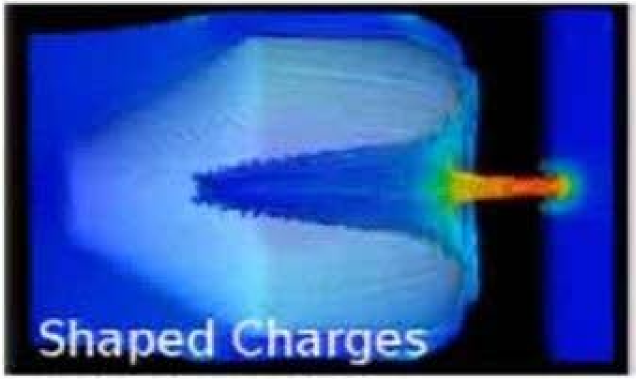


Foam
Compaction





Carbon capture and cleanup



Achieving Scalability & Portability

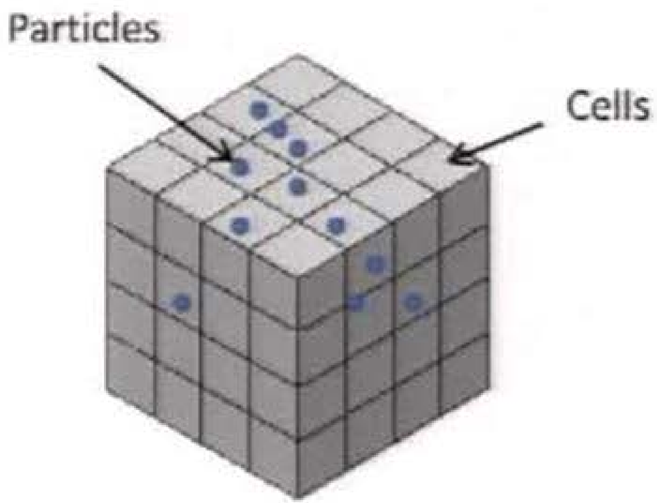
- Get the Abstractions Right
- Abstract Task Graph
  - Encapsulates computation and communication
- Data Storage – DataWarehouse
  - Encapsulates model for describing the global computational space
  - Data moves from node/processor via MPI under the covers as needed
    - hidden from the developer
- Programming for a Patch
  - Multiple patches combine to form the global Grid
  - Unit work space – block structured collection of cells with I,J,K indexing scheme
  - Single or multiple patches per Core/Processor

Abstract Taskgraph

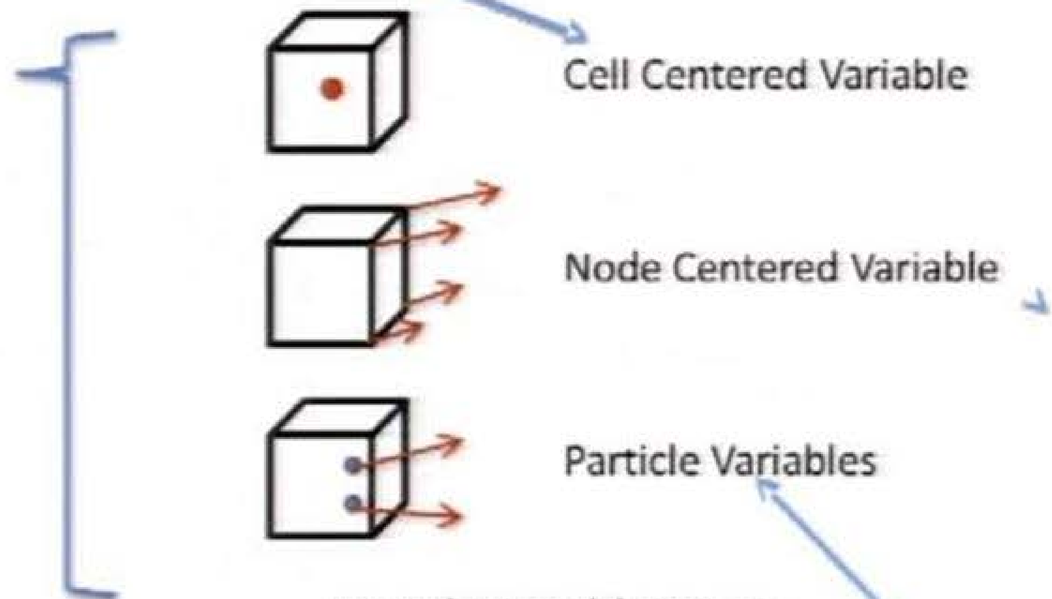
- Convenient abstraction that provides a mechanism for achieving parallelism
- Explicit representation of computation and communication
- Components delegate decisions about parallelism to a scheduler component using variable dependencies and computational workloads for global resource optimization (load balancing)
- Efficient fine-grained coupling of multi-physics components
- Flexible load balancing
- Separation of Application development from Parallelism.
  - Component developers don't have to be parallelism experts

Uintah Patch and Variables

ICE is a cell-centered finite volume method for Navier Stokes equations



Uintah Patch



Uintah Variable Types

- Structured Grid Variable (for Flows) are Cell Centered Nodes, Face Centered Nodes.
- Unstructured Points (for Solids) are Particles

ARCHES is a combustion code using several different radiation models and linear solvers

MPM is a novel method that uses particles and nodes

Exchange data with ICE, not just boundary condition

What is a Task?

- Two features:
  - A pointer to a function that performs the actual work
  - A specification of the inputs & outputs

```
Task* task = new 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() );
```

Component Basics

```
class Example : public UintahParallelComponent, public SimulationInterface {
  public:
    virtual void problemSetup(. . . .);

    virtual void scheduleInitialize(. . . );

    virtual void scheduleComputeStableTimestep(. . . );

    virtual void scheduleTimeAdvance(. . . );

  private:
    void intialize(. . .  );

    void computeStableTimestep(. . . );

    void timeAdvance( . . . );        <==========  Algorithmic Implementation

}
```

Burgers Example

```
<Grid>
  <Level>
    <Box label = "1">
     <lower>      [0,0,0]         </lower>
     <upper>      [1.0,1.0,1.0]  </upper>
     <resolution> [50,50,50]      </resolution>
     <patches>    [2,2,2]         </patches>
     <extraCells> [1,1,1]         </extraCells>
    </Box>
  </Level>
</Grid>
```

$$U_t + UU_x = 0$$

25 cubed patches
8 patches
One level of halos

```
void  Burger::scheduleTimeAdvance( const LevelP& level,
                                   SchedulerP& sched )
{
 .....
  task->requires( Task::OldDW, u_label, Ghost::AroundNodes, 1 );
  task->requires( Task::OldDW, sharedState_->get_delt_label() );

  task->computes( u_label );

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

Get old solution from old data warehouse
One level of halos
Compute new solution

Burgers Equation code $U_t + UU_x = 0$

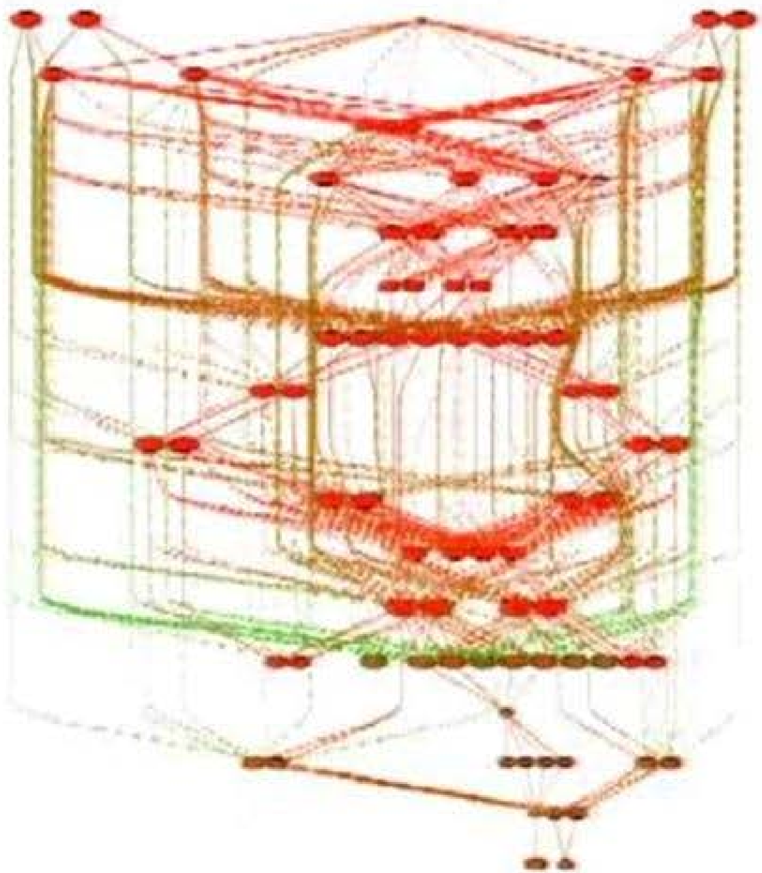
```
void Burger::timeAdvance( const ProcessorGroup*, const PatchSubset* patches,
                          const MaterialSubset* matls, DataWarehouse* old_dw,
                      DataWarehouse* new_dw) {
 for(int p=0;p<patches->size();p++){//Loop for all patches on this processor
   // Get data from  data warehouse including 1 layer of  "ghost" nodes from
   // surrounding patches
  old_dw->get( u, lb_->u, matl, patch, Ghost::AroundNodes, 1 );

  Vector dx = patch->getLevel()->dCell(); // dt, dx Time and space increments
  old_dw->get(dt, sharedState_->get_delt_label());

  new_dw->allocateAndPut(new_u, lb_->u, matl, patch );  // allocate memory for
                                                        // result: new_u

  // define iterator ranges  l and h, etc.
  // A lot of code has been pruned...
 for(NodeIterator iter(l, h);!iter.done(); iter++){ // iterate through all the nodes
   IntVector n = *iter;
   double dudx = (u[n+IntVector(1,0,0)] - u[n-IntVector(1,0,0)]) /(2.0 * dx.x());
   double du     = - u[n] * dt * (dudx);
   new_u[n]      = u[n] + du;
 }
```

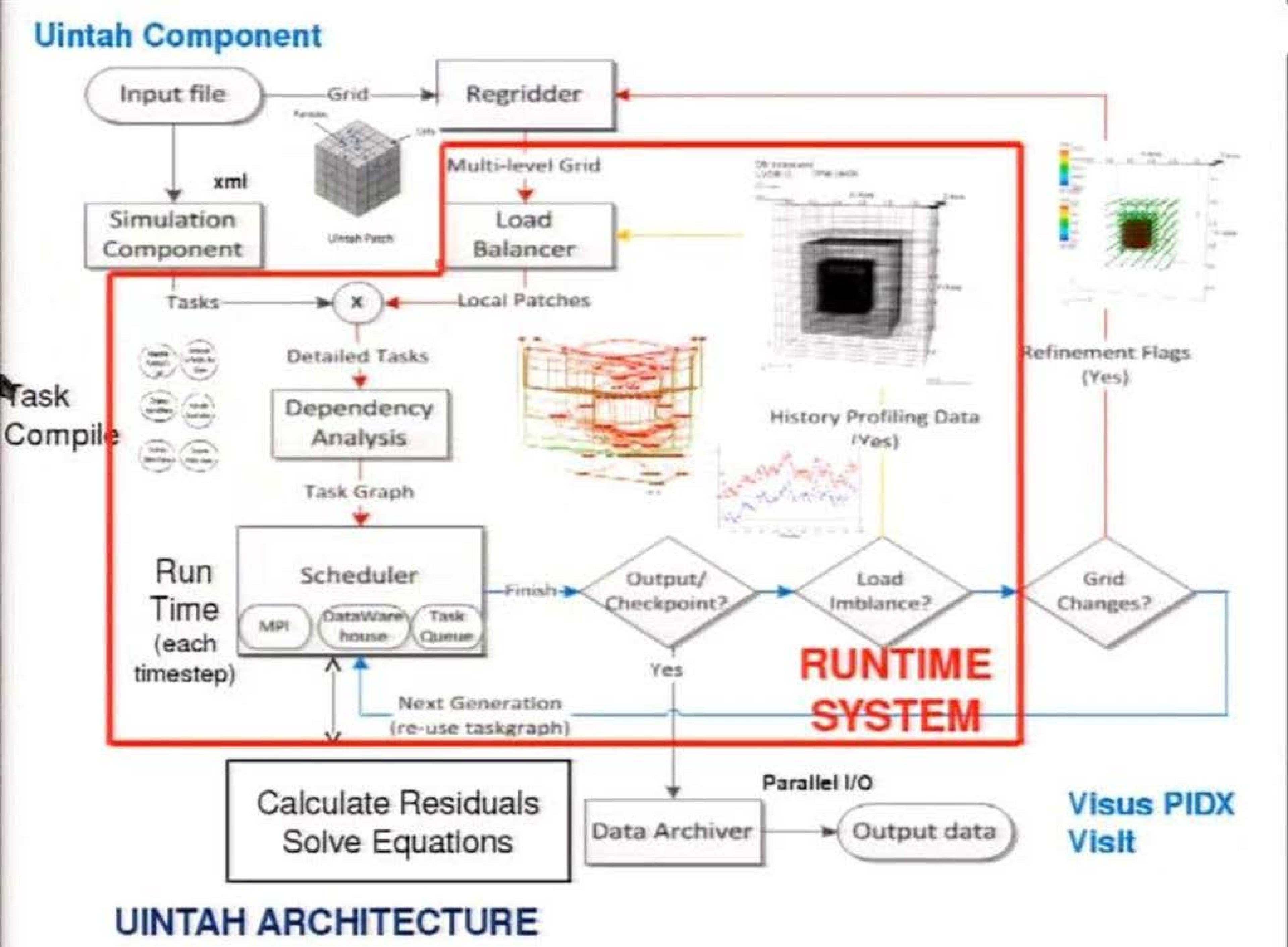
Uintah Distributed Task Graph

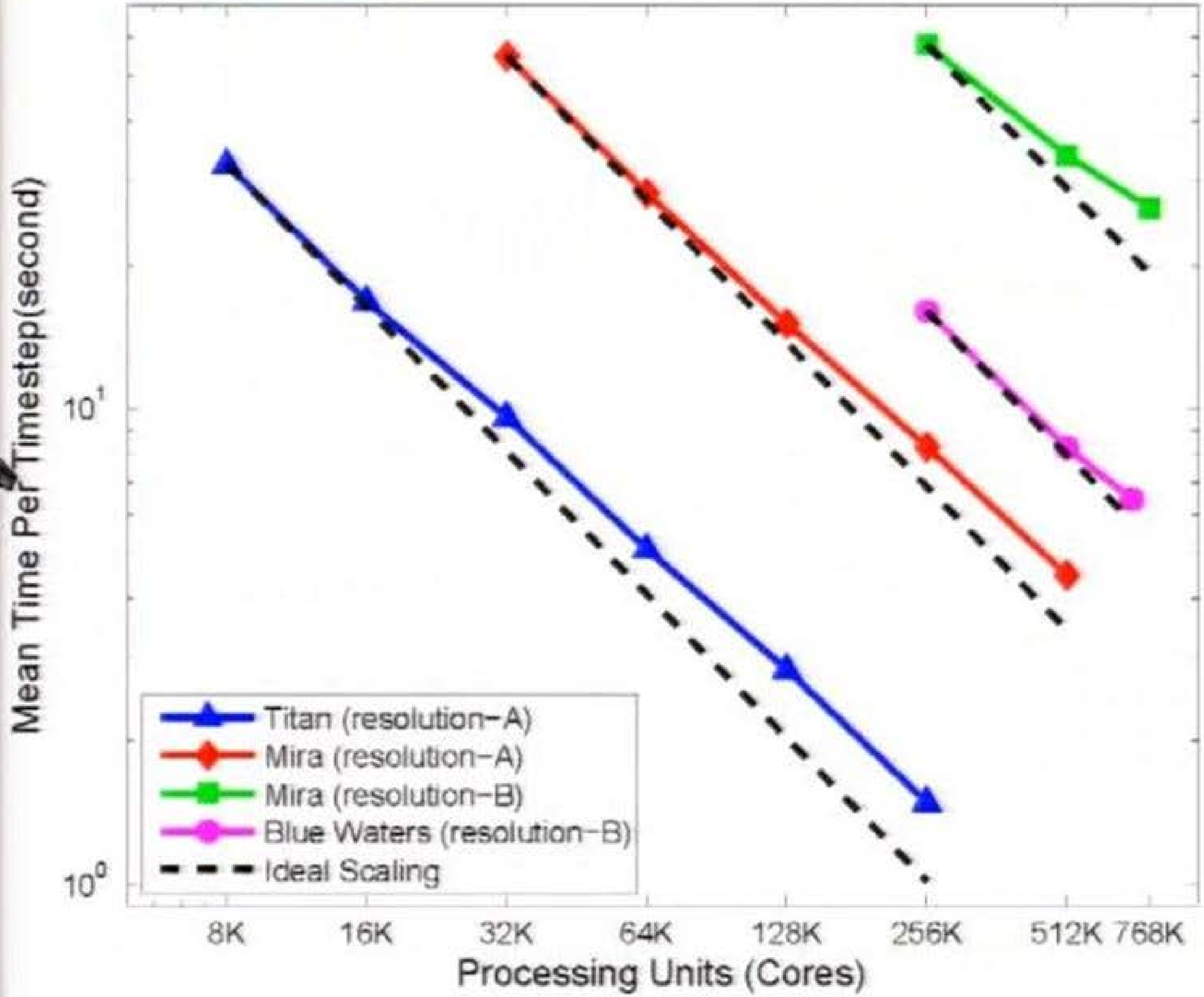


4 patches single level ICE task graph

- 2 million tasks per timestep globally on 98K cores
- Tasks on local and neighboring patches
- Same code callback by each patch
- Variables in data warehouse(DW)
  - Read - get() from OldDW and NewDW
  - Write- put() to NewDW
- Communication along edges

Although individual task graphs are quite "linear" per mesh patch they are offset when multiple patches execute per core

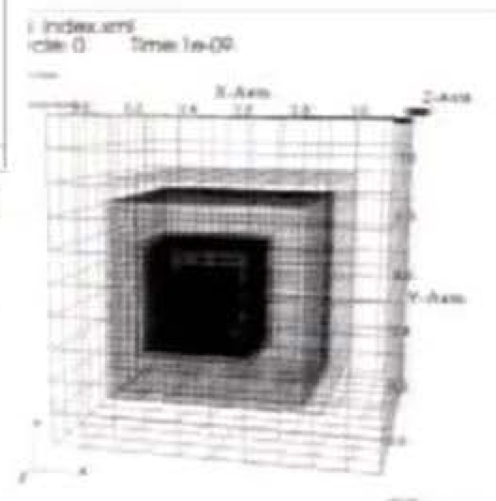

Uintah Component
Input file
Grid
Regridder
xml
Uintah Patch
Simulation Component
Multi-level Grid
Load Balancer
Tasks
Local Patches
Detailed Tasks
Task Compile
Dependency Analysis
Task Graph
History Profiling Data (Yes)
Refinement Flags (Yes)
Run Time (each timestep)
Scheduler
MPI
DataWarehouse
Task Queue
Finish
Output/ Checkpoint?
Load Imblance?
Grid Changes?
Yes
Next Generation (re-use taskgraph)
RUNTIME SYSTEM
Calculate Residuals Solve Equations
Data Archiver
Parallel I/O
Output data
Visus PIDX VisIt
UINTAH ARCHITECTURE



MPM AMR ICE Strong Scaling

Mira DOE BG/Q 768K cores
Blue Waters Cray XE6/XK7 700K+ cores

Resolution B
29 Billion particles
4 Billion mesh cells
1.2 Million mesh patches

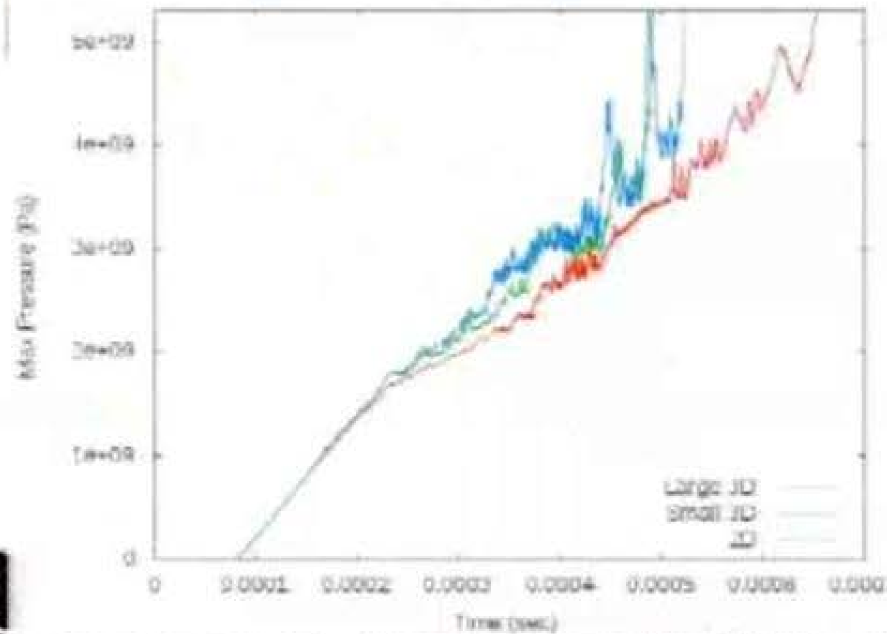


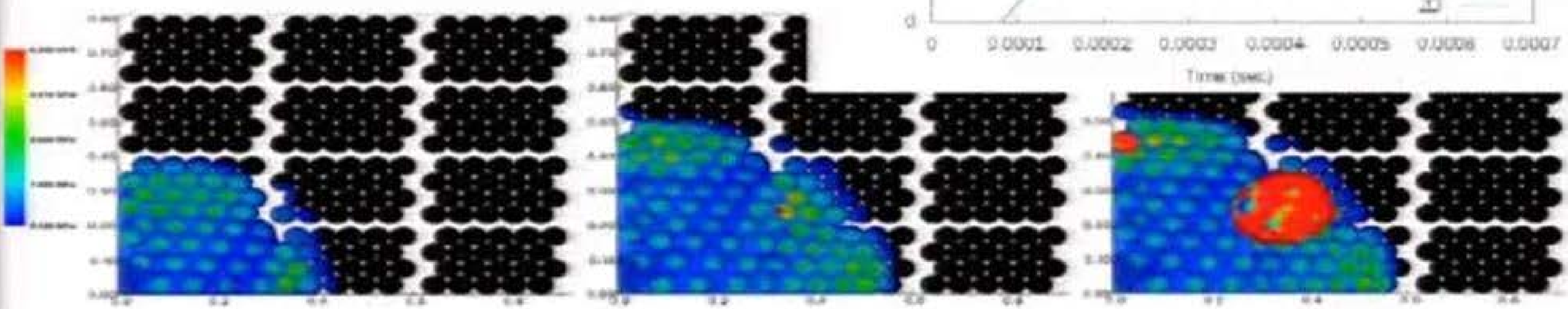
Complex fluid-structure interaction problem with adaptive mesh refinement, see SC13/14 paper NSF funding.

NSF funded modeling of Spanish Fork Accident 8/10/05

- Speeding truck with 8000 explosive boosters each with 2.5-5.5 lbs of explosive overturned and caught fire
- Experimental evidence for a transition from deflagration to detonation?

Deflagration wave moves at ~400m/s not all explosive consumed. Detonation wave moves 8500m/s all explosive consumed.



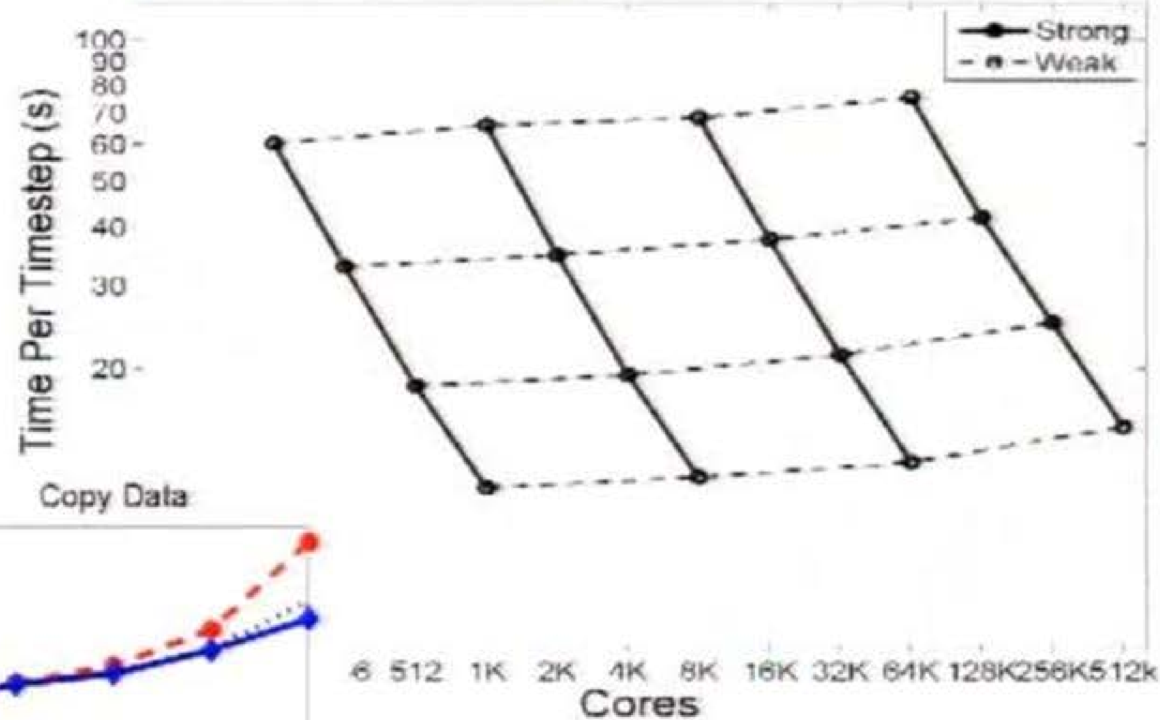


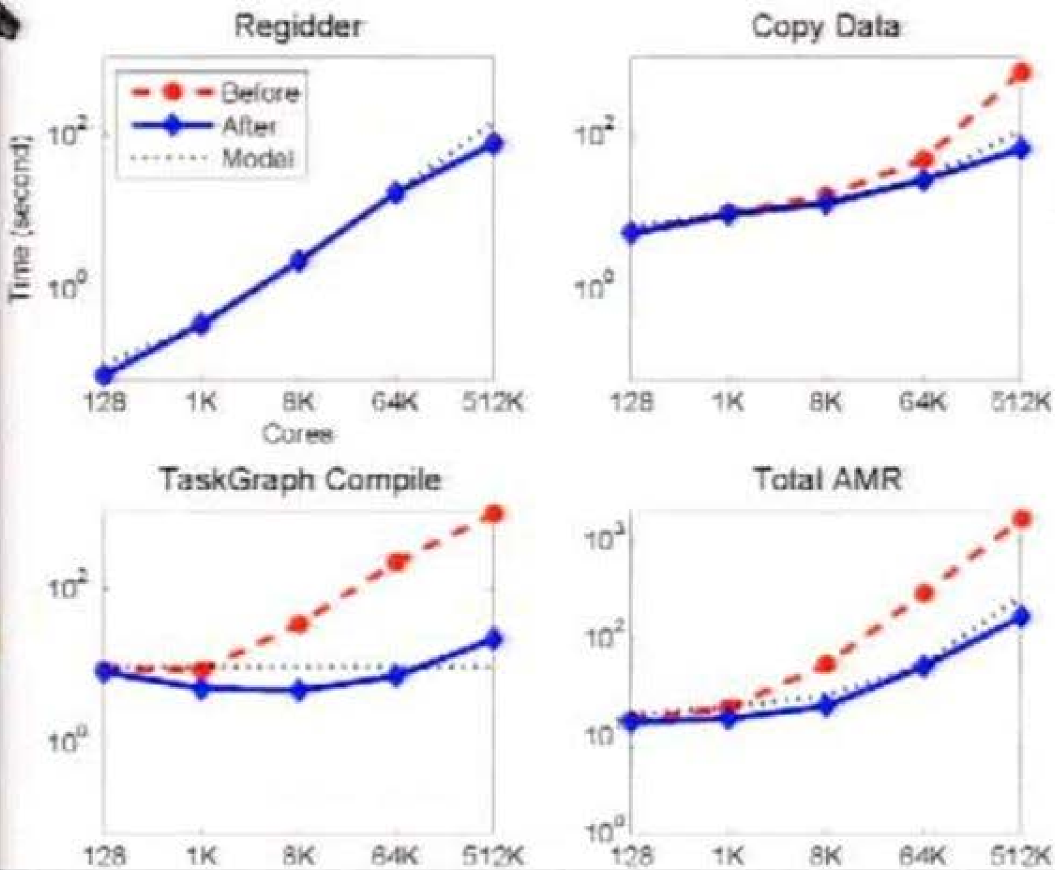
0.534 msec   0.634 msec   0.644 msec

Spanish Fork Accident

500K mesh patches
1.3 Billion mesh cells
7.8 Billion particles

Detonation MPMICE: Scaling on Mira BGQ





At every stage when we move
to the next generation of problems
Some of the algorithms and data
structures need to be replaced .

Scalability at one level is no certain
Indicator for problems or machines
An order of magnitude larger

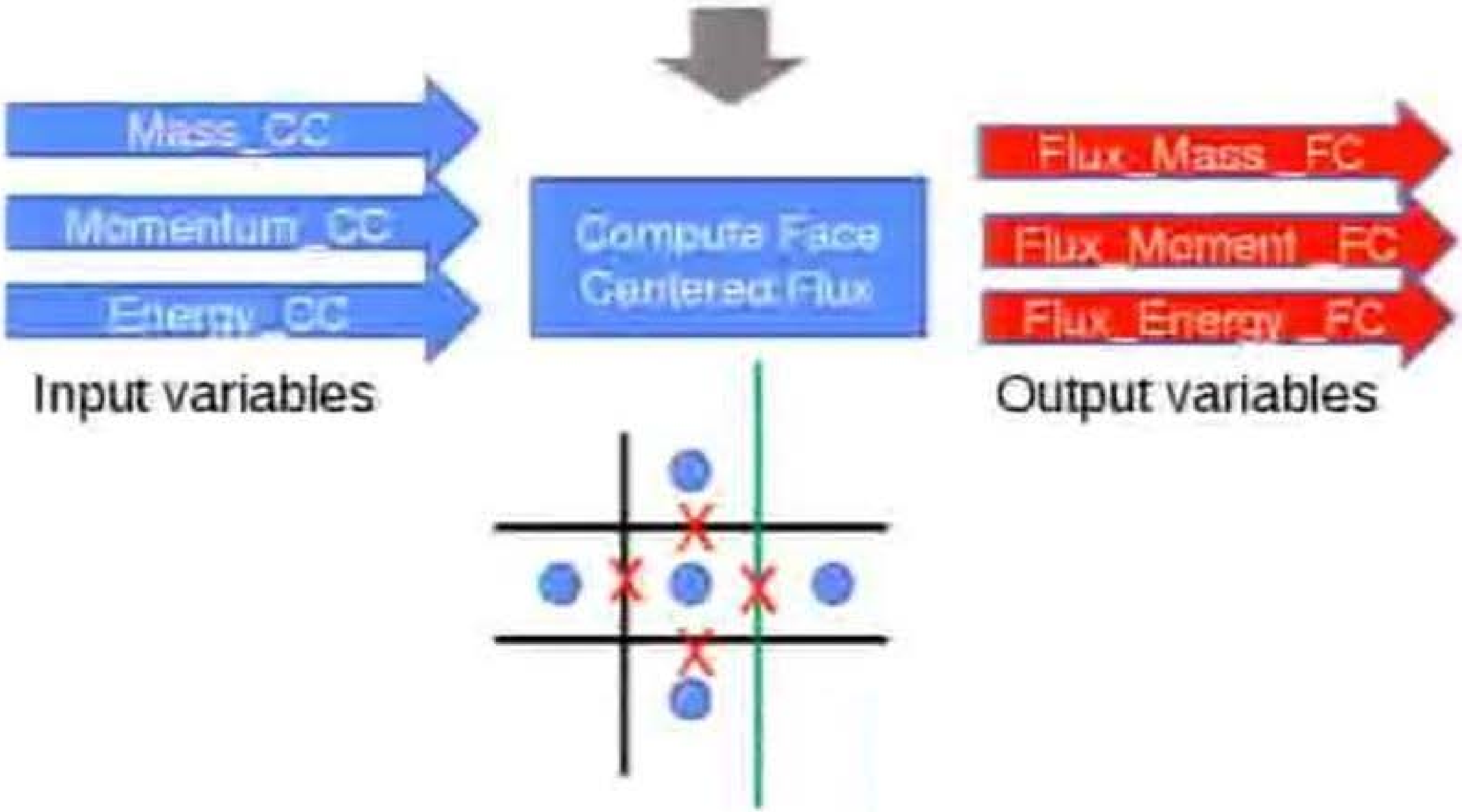
MiniAero Project

- **Project with Sandia National Lab**
  - Evaluate Uintah Framework – port app to framework
  - Team of 15+ developers from Sandia and Utah had a 3 day coding session to port the MiniAero mini-app (Finite Volume, RK4 -- CFD application).
- **Uintah Perspective**
  - How quickly can we educate new developers and implement a new component?
  - 90% of the port was completed during the visit
  - Within < 2 weeks from visit, basic features were functional on multi-cpu systems.
  - Within < 2 months, port was completed and scalable to 128K cores on Titan.

Task from the MiniAero Algorithm

Compute face-centered fluxes (mass, momentum, energy):

$$Fluxes_{fc} = f(m_{cc}, V\rho_{cc}, E_{cc})$$



Scheduling MiniAero Algorithm Task

"Hello World" task in Uintah

```
void MiniAero::schedFaceCenteredFlux(const LevelP& level,
                    SchedulerP&    sched,
                    const int RK_step)
{
  Task* task = scinew Task( "MiniAero::faceCenteredFlux", this,
              &MiniAero::faceCenteredFlux,RK_step);

  Ghost::GhostType  gac = Ghost::AroundCells;

  task->requires( Task::NewDW, flux_mass_CClabel, gac,1 );
  task->requires( Task::NewDW, flux_mom_CClabel, gac,1 );
  task->requires( Task::NewDW, flux_energy_CClabel, gac,1 );

  task->computes( flux_mass_FCXlabel);
  task->computes( flux_mom_FCXlabel);
  task->computes( flux_energy_FCXlabel);
  . . . .  /* Same for FCY and FCZ quantities) */

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

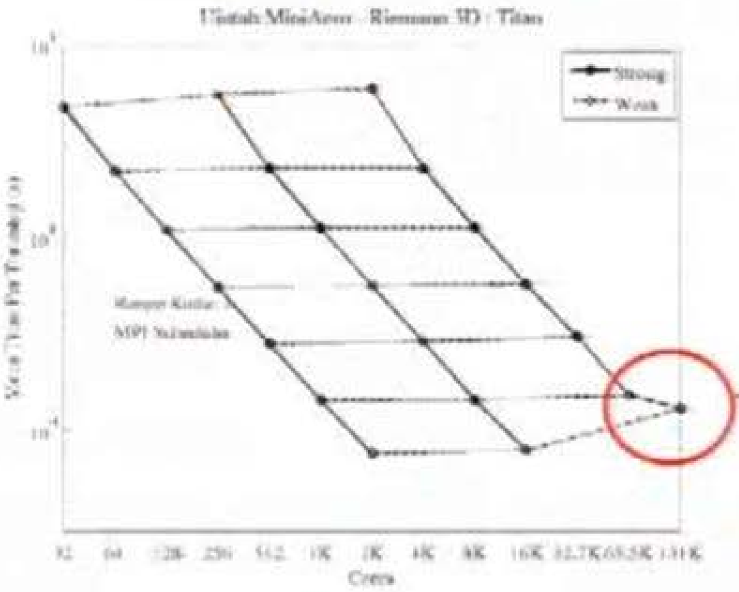
Task from the MiniAero Algorithm (3/3)

```
…continued
// Compute Face Centered Fluxes from Cell Centered
for (CellIterator iter = patch->getSFCXIterator(); !iter.done(); iter++){
 IntVector c = *iter;
 IntVector offset(-1,0,0);
 Flux_mass_FCX[c]=0.5*(flux_mass_CC[c][0] + flux_mass_CC[c+offset][0]);
 Flux_mom_FCX[c][0]=0.5*(flux_mom_CC[c](0,0)+flux_mom_CC[c+offset](0,0));
  Flux_mom_FCX[c][1] = 0.5*(flux_mom_CC[c](0,1) + flux_mom_CC[c+offset](0,1));
  Flux_mom_FCX[c][2] = 0.5*(flux_mom_CC[c](0,2) + flux_mom_CC[c+offset](0,2));
  Flux_energy_FCX[c] = 0.5*(flux_energy_CC[c][0]+flux_energy_CC[c+offset][0]);
}
for (CellIterator iter = patch->getSFCYIterator(); !iter.done(); iter++){
  IntVector c = *iter;
  IntVector offset(0,-1,0);
  Flux_mass_FCY[c] = 0.5*(flux_mass_CC[c][1] + flux_mass_CC[c + offset][1]);
  Flux_mom_FCY[c][0] = 0.5*(flux_mom_CC[c](1,0) + flux_mom_CC[c+offset](1,0));
  Flux_mom_FCY[c][1] = 0.5*(flux_mom_CC[c](1,1) + flux_mom_CC[c+offset](1,1));
  Flux_mom_FCY[c][2] = 0.5*(flux_mom_CC[c](1,2) + flux_mom_CC[c+offset](1,2));
  Flux_energy_FCY[c] = 0.5*(flux_energy_CC[c][1]+flux_energy_CC[c+offset][1]);
}
 . . . . . /* similarly for SFCZIterator directory
```

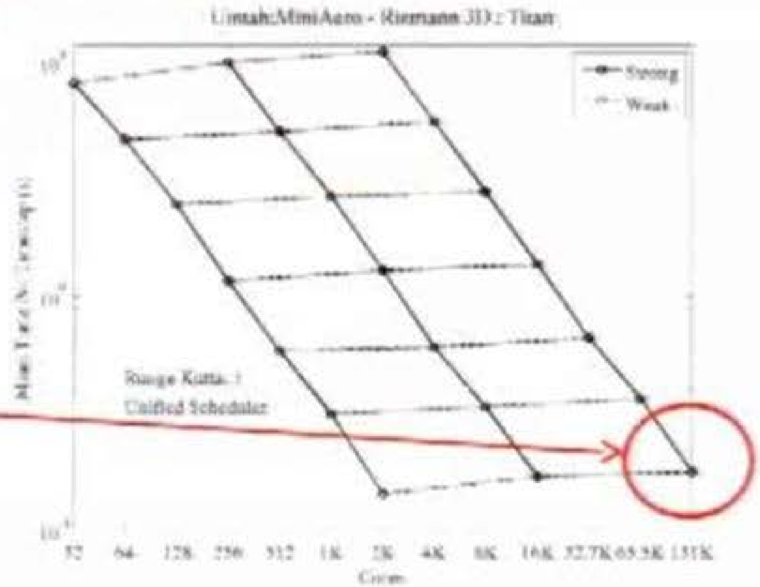
MiniAero Scaling

Resolution
16M, 128M, 1B mesh cells
32 > 128K mesh patches

Runtime – MPI ONLY



Runtime – MPI + Pthreads



Changing the scheduling and execution of tasks from an MPI only to a hybrid MPI + Pthreads improves the strong and weak scalability at large core counts (overlapping communication and computation, reducing MPI communication)

NO CHANGE to the application code, only happens in the runtime

Summary

- **DAG abstraction** powerful concept for portable and scalable applications.
- **Functional view of Tasks** on a patch with Inputs/Outputs map to different stages of an algorithm. Simplified development model.
- **Components can be developed quickly** without requiring in depth knowledge of MPI, threads, or other communication primitives. Simplifies developer time and expertise.
- **Scalable out of the box**. New components can leverage the advances in the runtime system.

Acknowledgments: This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number(s) DE-NA0002375, and by DOE ALCC award CMB109, "Large Scale Turbulent Clean Coal Combustion", for time on Titan. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. We would like to thank Todd Harman for generating the scaling plots for the MiniAero work.