Uintah Runtime System

University of Utah PSAAP II Center

Alan Humphrey, John Schmidt, Martin Berzins

Scientific Computing and Imaging Institute, University of Utah

SCI INSTITUTE

CARBON CAPTURE MULTIDISCIPLINARY SIMULATION CENTER

NNSA National Nuclear Security Administration

THE INSTITUTE FOR CLEAN AND SECURE ENERGY AT THE UNIVERSITY OF UTAH

THE UNIVERSITY OF UTAH

Outline:

I. Uintah Framework – Overview

II. Target Application – Utah PSAAP II Center

III. Uintah Runtime System

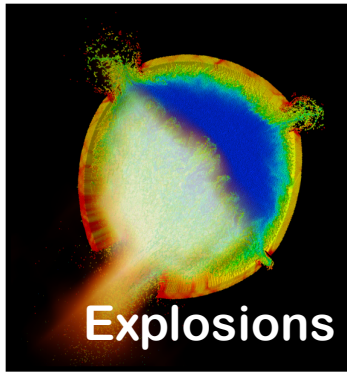
- *How parallelism and scaling are achieved*
- *Task graph abstraction*
- *Data model – DataWarehouse*
- *Data partitioning and distribution*
- *Programming model – overview and example*

IV. Performance portability - Kokkos

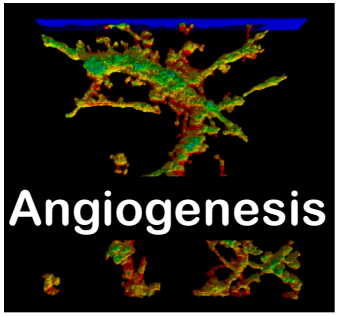
V. Summary and Questions

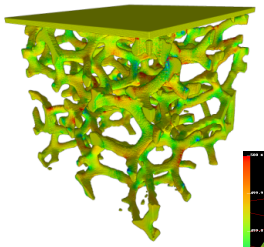
Uintah Overview

- Parallel, adaptive multi-physics framework
- Fluid-structure interaction problems
- Patch-based AMR – Structured Grid
- Particle system and mesh-based fluid solve

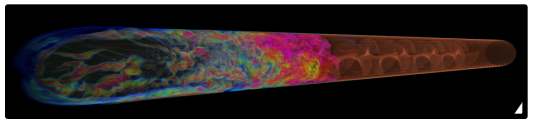


Plume Fires

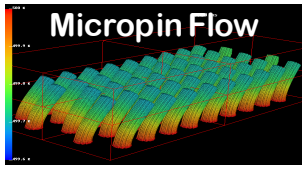


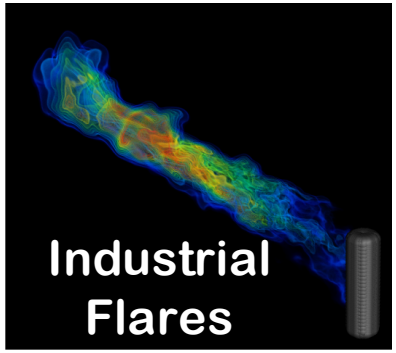


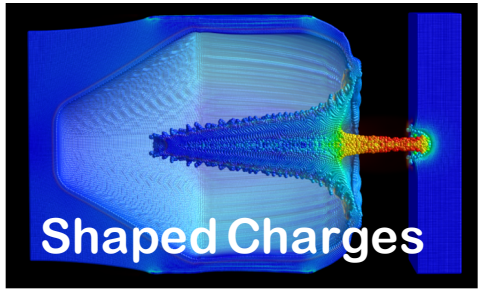
Foam Compaction

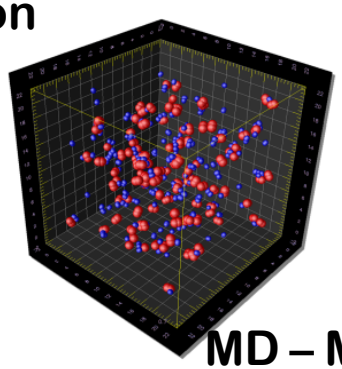


Chemical/Gas Mixing

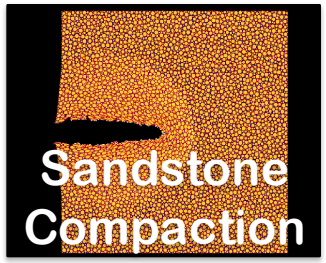








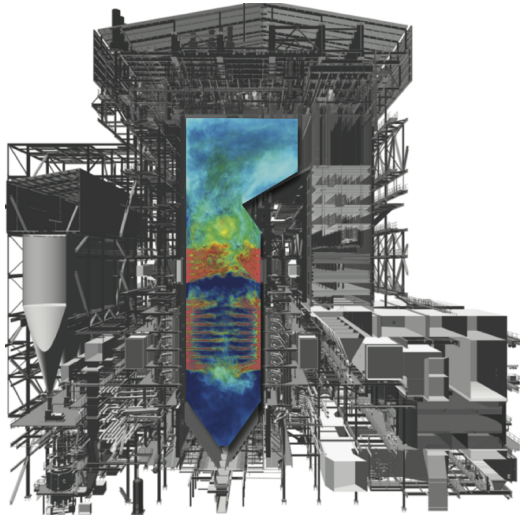
MD – Multiscale Materials Design

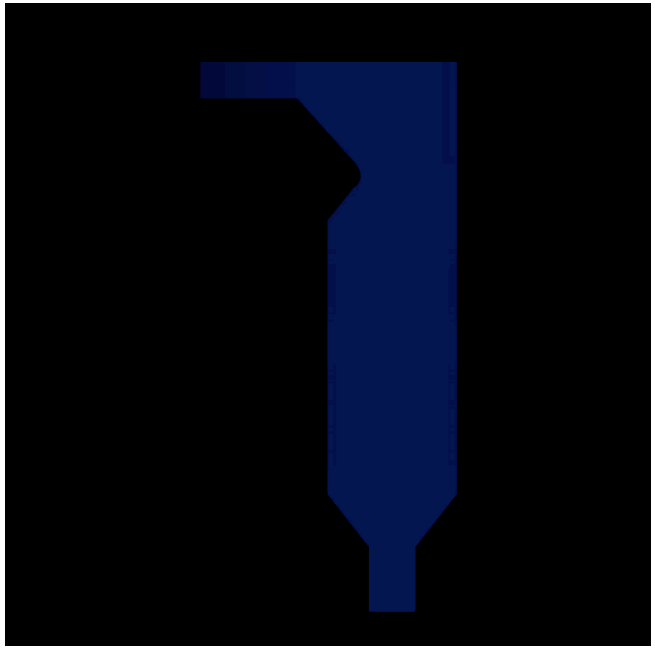


Exascale Target Problem

DOE NNSA PSAAP II Center – U. Utah

92 meters

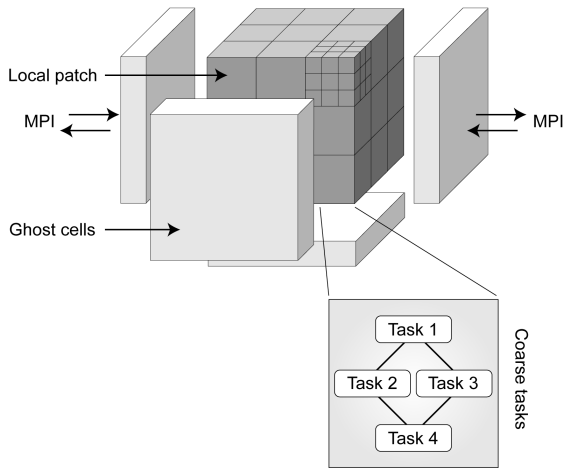




O_2 concentrations in coal boiler chamber

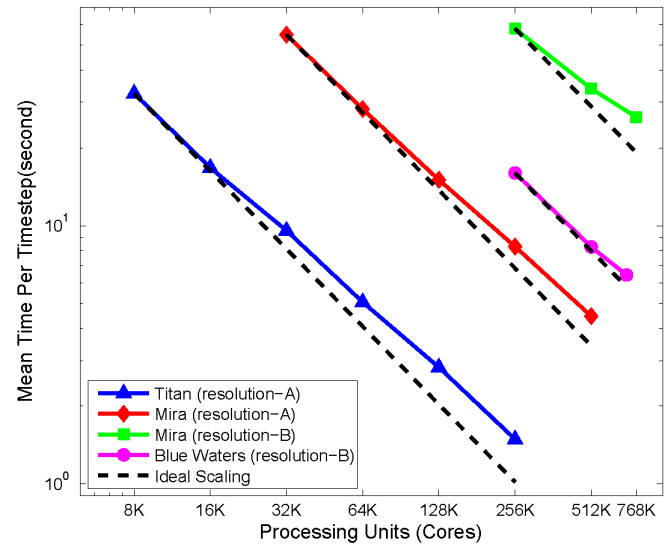
- Alstom Power 1000MWe "Twin Fireball" boiler
- Supply power for 1M people
- 1mm grid resolution = 9 x 10^{12} cells
- 1000 times larger than largest problems solved today

Uintah: Parallelism & Scaling



Asynchronous task-based paradigm

- **Domain decomposition**
- **Task-level parallelism**
- **Data parallelism within task**



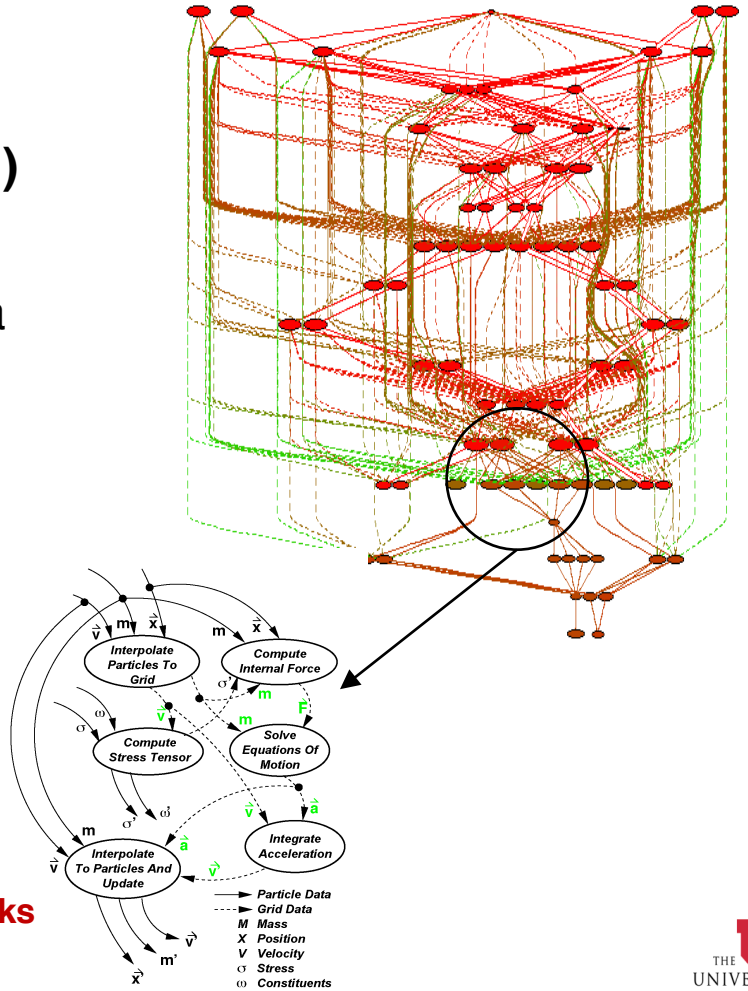
Strong Scaling:
Fluid-structure interaction problem using MPMICE algorithm w/ AMR

- **Clear separation of user code and runtime system**
- Task - serial code on generic "patch"
- Task specifies desired halo region
- Uintah manages MPI and load balancing
  - Also manages GPU data migration

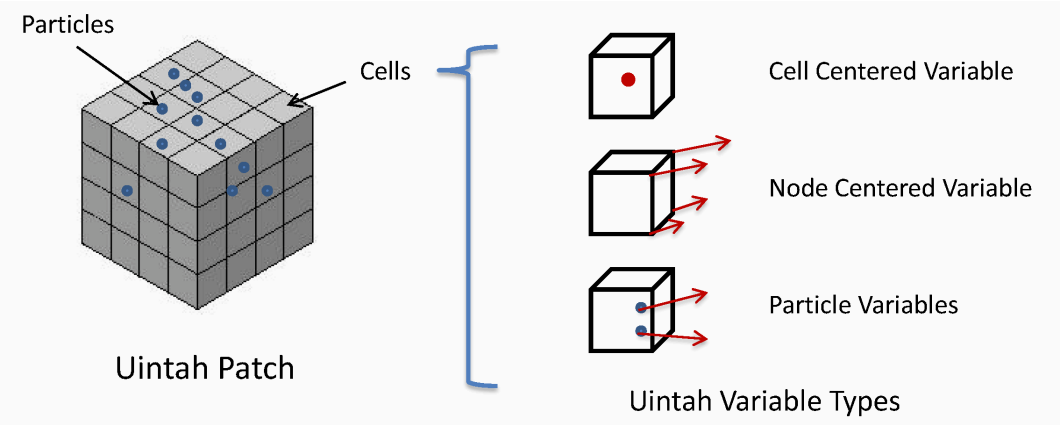
Existing components scale from 7k to 700K cores w/o changing any user code

Task-Graph Approach

- **Task Graph: Directed Acyclic Graph (DAG)**
- Explicit representation of comm & comp
- Allows apps to delegate decisions about data partitioning, parallelism, load balancing to framework
- Asynchronous, dynamic execution of tasks **<u>key idea, execute when ready</u>**
- Overlap communication & computation
- Design extended to GPUs without massive, code refactoring
  - **Infrastructure handles device API details**
  - **Provides convenient GPU APIs**
  - **User writes only GPU kernels for appropriate CPU tasks**



What is a Patch?



- Structured grid decomposed into patches
- **Unit work space** – block structured collection of cells with I,J,K indexing
- Patches uniquely assigned to cores

- Structured **GridVariable** (for Flows) are Cell, Face, Node Centered
- Unstructured Points (for Solids) are Particles or **ParticleVariables**
- ***Variables live on patches (cells)***

A Task?

- **Basic unit of work within Uintah – granularity user determined**
  - C++ method with computation (user written callback)
  - A specification of inputs/outputs with halo requirements
- **Interacts with local data repository – DataWarehouse**
  - DataWarehouse provides a simple get(), put() interface
- **Scheduled and executed via Task-Scheduler**

Uintah Grid Spec - Load Balancer

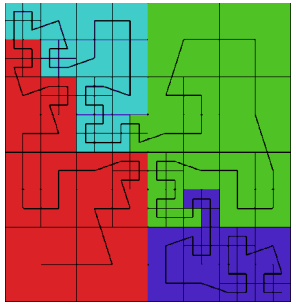
```
<Grid>
    <Level>
       <Box label = "1">
       <lower>      [0,0,0]        </lower>
       <upper>      [1.0,1.0,1.0]  </upper>
       <resolution> [64,64,64]     </resolution>
       <patches>    [2,2,2]        </patches>
       <extraCells> [1,1,1]        </extraCells>
     </Box>
   </Level>
 </Grid>
```

XML Problem Specification provided by user input file, e.g.

- **e.g., Grid spec, patches**
- **<patches> [x,y,z] </patches>**

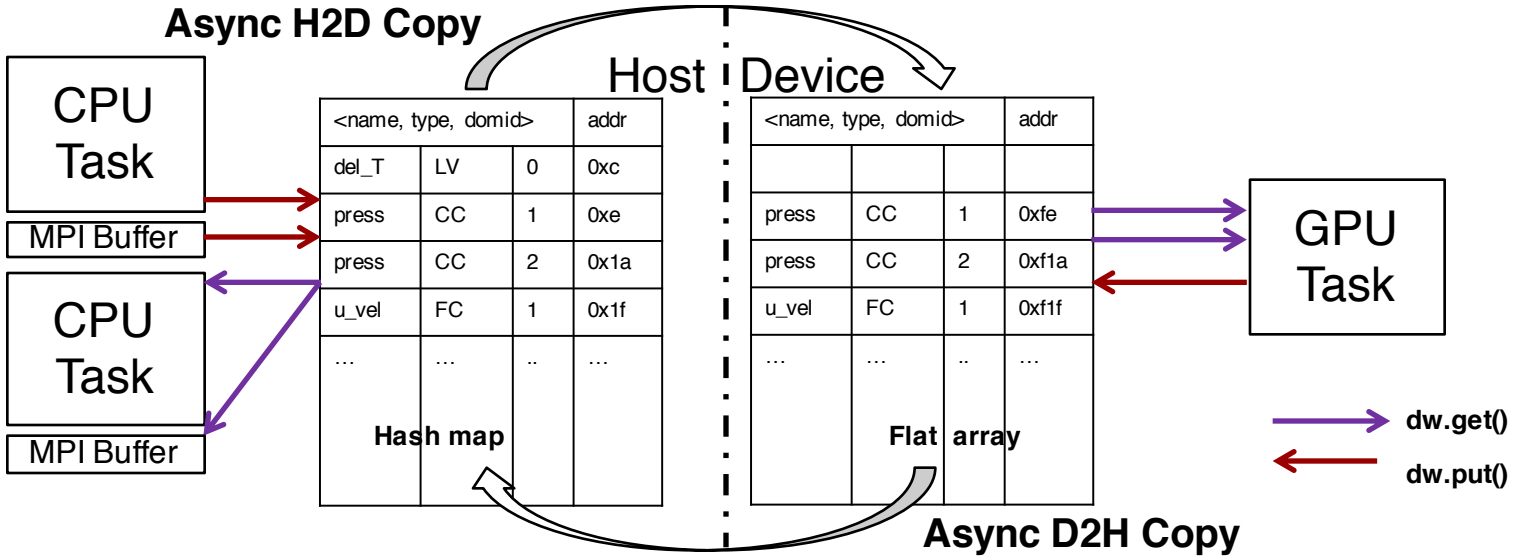
Uintah Load Balancer automatically assigns patches to nodes

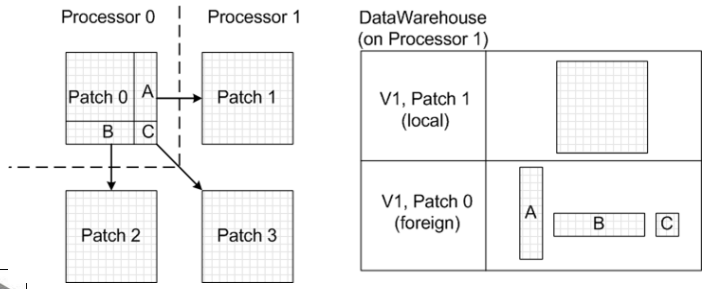
- **Uintah Load Balancing**
  - Cost Estimation Algorithms based on data assimilation
  - Use load balancing algorithms based on patches and a fast space filling curve algorithm [1]



1. J. Luitjens, M. Berzins, T.C. Henderson. "**Parallel Space Filling Curve Generation Through Sorting,**" In *Journal of Concurrency and Computation*, Vol. 19, No. 10, pp. 1387--1402. 2007.

Data Storage - DataWarehouse

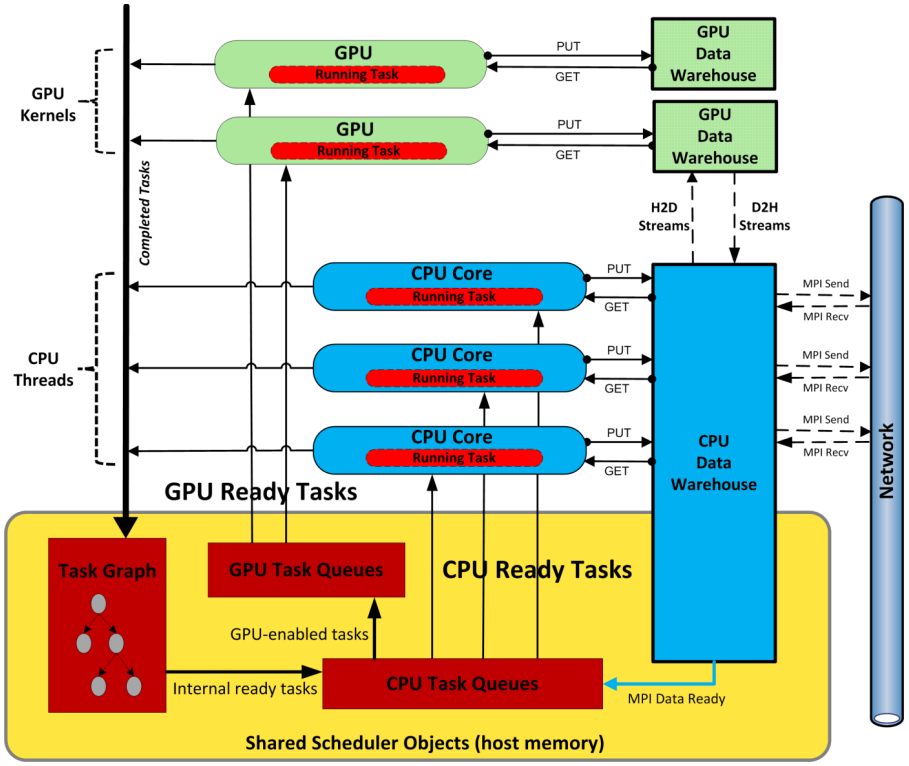




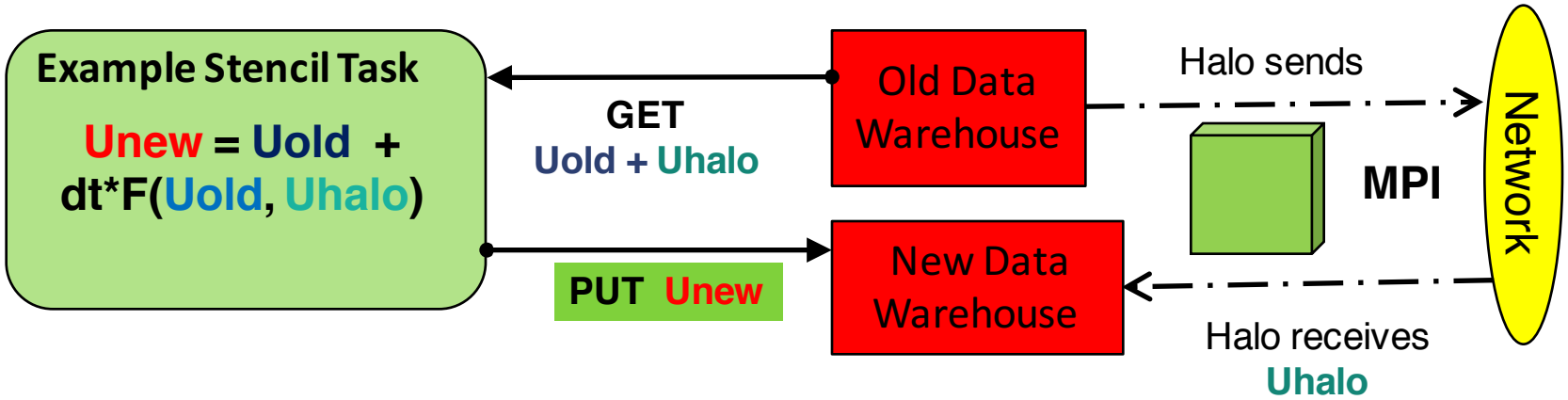
- Encapsulates model for global communication space
- Automatic, on-demand variable movement – MPI, H2D, D2H
  - Details hidden from developer
- Implemented interfaces for both CPU/GPU Tasks

Uintah Heterogeneous Runtime System

- Shared memory model on-node
  - 1 MPI rank per node / NUMA region
- MPI + Pthreads + CUDA
- Better load-balancing (node-level)
- Decentralized: All threads
  - *access CPU/GPU task queues*
  - *implicit work stealing*
  - *process their own MPI*
  - *interface with GPUs*
- Scalable, lock-free data structures
- Task code must be thread-safe

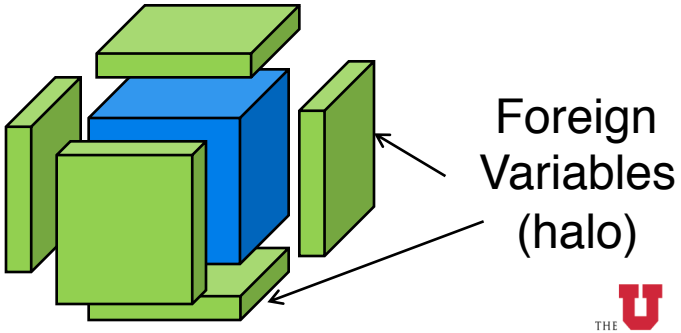


Uintah Programming Model for Stencil



Application specifies:

- **requires + halo and computes**
- **become dependencies in graph**
- **halo requirements - via MPI**
  - ***external dependencies***



Uintah Component Example
Scheduling a Task

inputs

outputs

```
void ICE::scheduleComputePressFC(        SchedulerP     & sched
                                , const PatchSet       * patches
                                , const MaterialSubset * press_matls
                                , const MaterialSet    * matls  )
{
  Task* task = new Task( "ICE::computePressFC", this, &ICE::computePressFC);

  Ghost::GhostType  gac = Ghost::AroundCells;
  task->requires( Task::OldDW,  lb->press_CCLabel  ,  press_matls,  gac, 1 );
  task->requires( Task::NewDW,  lb->sum_rho_CCLabel,  press_matl ,  gac, 1 );

  task->computes( lb->pressX_FCLabel, press_matls );
  task->computes( lb->pressY_FCLabel, press_matls );
  task->computes( lb->pressZ_FCLabel, press_matls );

  sched->addTask( task, patches, matls );  // infrastructure magic begins here
}
```

Halo Depth=1

Task Code

```
void ICE::computePressFC(  const ProcessorGroup * pg
                         , 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);
    Ghost::GhostType  gac = Ghost::AroundCells;

    // Get data from new Data Warehouse
    constCCVariable<double> press_CC;
    constCCVariable<double> sum_rho_CC;

    const int pressMatl = 0;
    old_dw->get( press_CC,  lb->press_CCLabel,    pressMatl,  patch,  gac, 1 );
    new_dw->get( sum_rho_CC,lb->sum_rho_CCLabel,  pressMatl,  patch,  gac, 1 );

    // Allocate memory and "put" it into new data warehouse
    SFCXVariable<double> pressX_FC;
    SFCYVariable<double> pressY_FC;
    SFCZVariable<double> pressZ_FC;

    new_dw->allocateAndPut( pressX_FC, lb->pressX_FCLabel,  pressMatl, patch );
    new_dw->allocateAndPut( pressY_FC, lb->pressY_FCLabel,  pressMatl, patch );
    new_dw->allocateAndPut( pressZ_FC, lb->pressZ_FCLabel,  pressMatl, patch );

    // do computation, stencil, etc
  } // end patch loop
```

Scheduling GPU Task

```
void
Ray::sched_rayTrace( const LevelP&   level,
                     SchedulerP&     sched,
                     Task::WhichDW   abskg_dw,
                     Task::WhichDW   sigma_dw,
                     Task::WhichDW   celltype_dw,
                     bool            modifies_divQ,
                     const int       radCalc_freq )
{
  Task *task;
  if ( Parallel::usingDevice() ) {       //  GPU Task implementation
    task = new Task( "Ray::rayTraceGPU", this, &Ray::rayTraceGPU,
                     modifies_divQ, abskg_dw, sigma_dw, celltype_dw,…  );
    task->usesDevice(true);
  } else {                              //  CPU Task implementation
    task = new Task( "Ray::rayTrace",    this, &Ray::rayTrace,
                     modifies_divQ, abskg_dw, sigma_dw, celltype_dw,… );
  }
```

Solid GPU capabilities BUT:

1. **Code Bifurcation – CPU and GPU kernels**
2. **User must write GPU kernel**

```
// calculate the thread indices
 int tidX = threadIdx.x + blockIdx.x * blockDim.x + patch.loEC.x;
 int tidY = threadIdx.y + blockIdx.y * blockDim.y + patch.loEC.y;
```

Performance Portability - Kokkos

Use Kokkos abstraction layer that maps loops onto machine (CPU GPU MIC) efficiently using cache aware memory models and vectorization / OpenMP

- Use **C++ template metaprogramming** for compile time data structures and functions and allow vectorization
- **Incremental** apps refactor to Kokkos parallel patterns/views

- **Replace patch cell iterator loops**

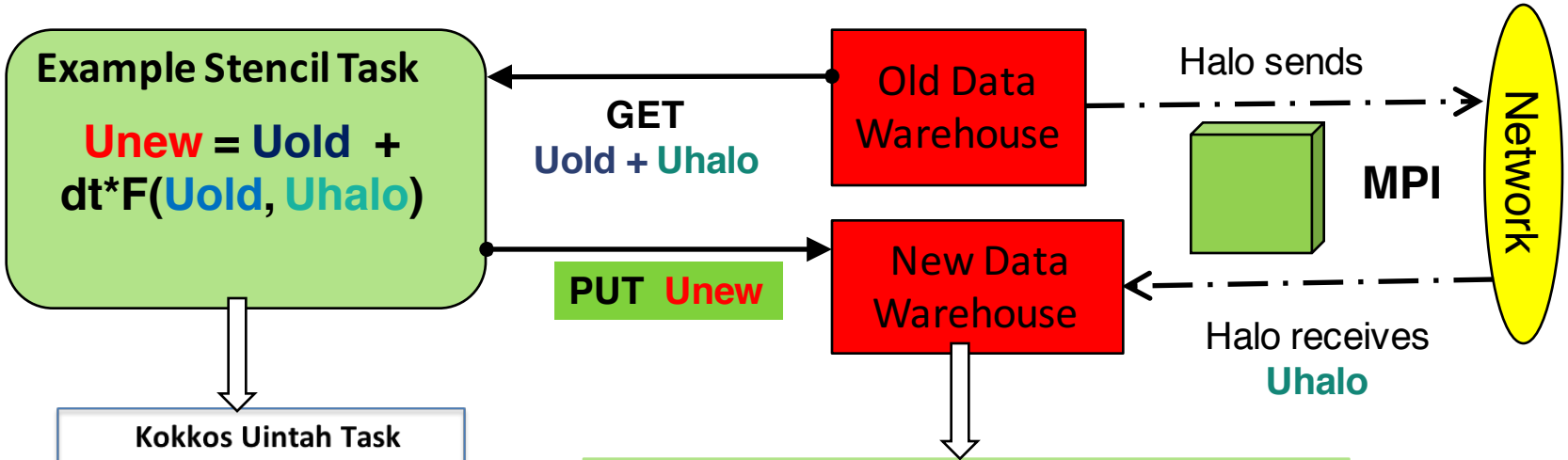
```
for (auto itr = patch.begin(); itr != patch.end(); ++itr) {
  IntVector iv = *itr;
  A[iv] = B[iv] + C[iv]; }
```

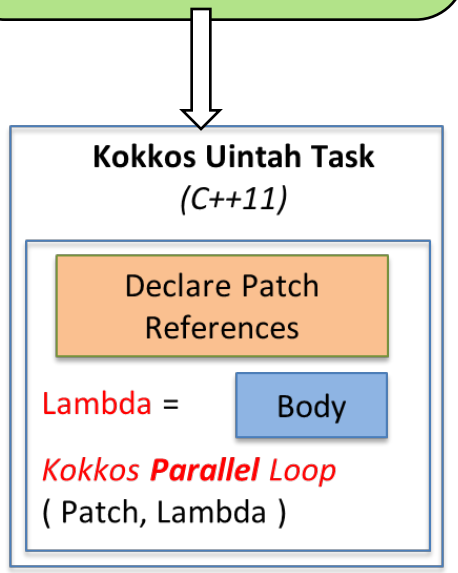
BECOMES


```
parallel_for(patch.range(), LAMBDA(int i, int j, int k) {
  A(i,j,k) = B(i,j,k) + C(i,j,k)});
```

- **Users prefer this to DSL**

Programming Model for Stencil- Kokkos





Kokkos Views Memory Structure
Cache, and vectorization friendly

Use **Kokkos** abstraction layer that maps loops onto machine specific data layouts and has appropriate memory abstractions

Acknowledgements

People, funding and computer time that have helped Uintah succeed

- Steve Parker, Justin Luitjens, Qingyu Meng, Dan Sunderland, Brad Peterson
- All those involved with Uintah (apps and infrastructure), past and present
- DOE ASC Strategic Academic Alliance Program 1998 -2010
- DOE NETL funding 2011-2014
- NSF Software funding and Peta-Apps 2007- 2015
- NSF XSEDE facilities and compute resources
- ALCC and Directors Discretionary time awards
- INCITE (4 awards 700M CPU hours in total)
- Argonne and Oak Ridge Facilities
- DOE NNSA PSAAP2 Center funding 2014-2019
- Argonne Aurora (180pf) early user program (2017-2019)

SCI INSTITUTE

THE UNIVERSITY OF UTAH

Questions?

Software Download

http://www.uintah.utah.edu