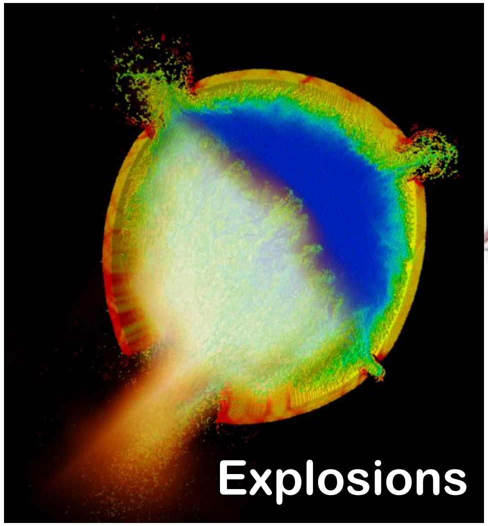
Uintah Framework

Justin Luitjens, Qingyu Meng, John Schmidt, Martin Berzins, Todd Harman, Chuch Wight, Steven Parker, et al

Uintah Parallel Computing Framework

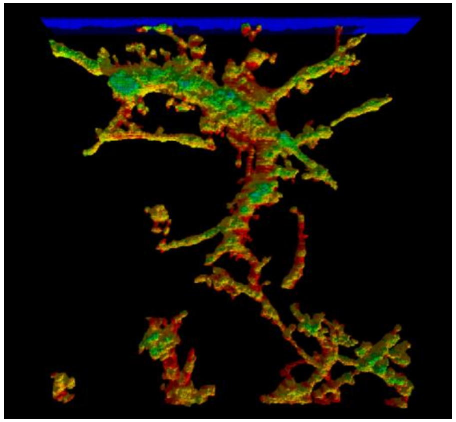
- Uintah - **far-sighted design by Steve Parker** :
  - **Component based design**
    - **Separated development**
    - **Swap components in and out**
    - **Code reuse**
  - **Automated parallelism**
    - **Engineer only writes "serial" code for a hexahedral patch**
    - **Complete separation of user code and parallelism**
    - **Asynchronous communication, message coalescing**
    - **Hybrid MPI/Threading**
  - **AMR Support**
    - **Automated load balancing & regridding**
  - **Multiple Simulation Components**
    - **ICE, MPM, Arches, MPMICE, et al.**
  - **Simulation of a broad class of fluid-structure interaction problems**

Uintah Applications

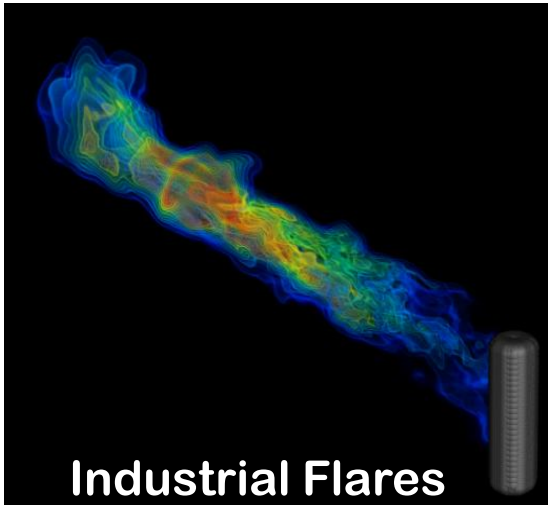


Explosions

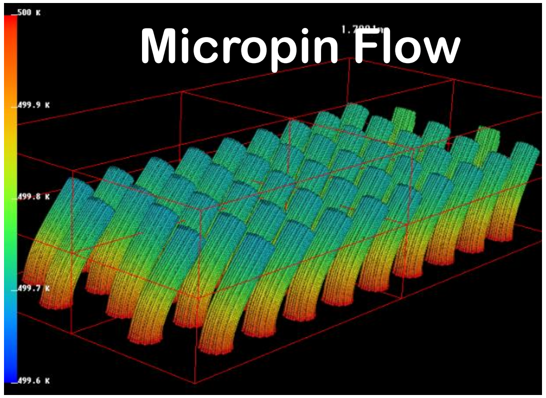
Plume Fires



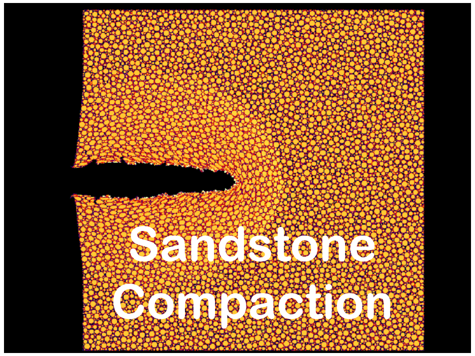
Angiogenesis



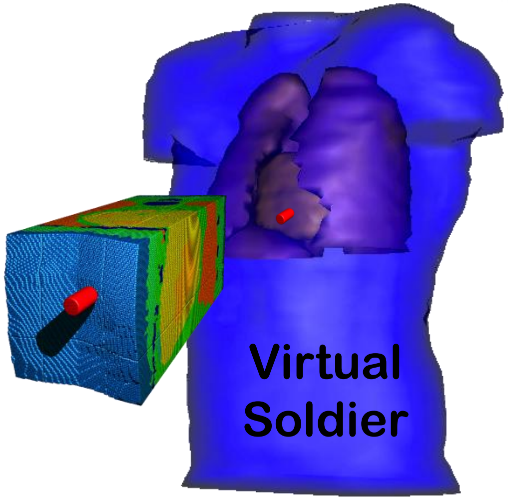
Industrial Flares



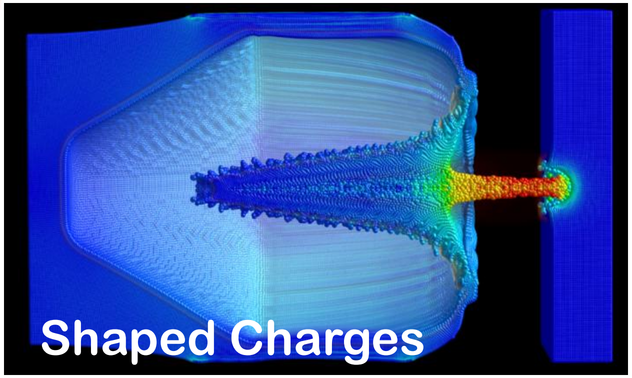
Micropin Flow



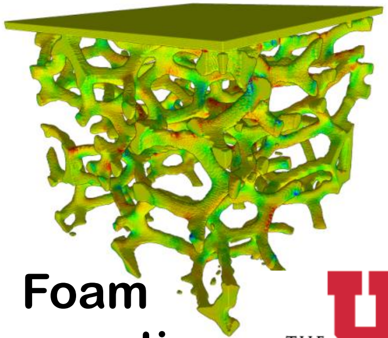
Sandstone Compaction



Virtual Soldier



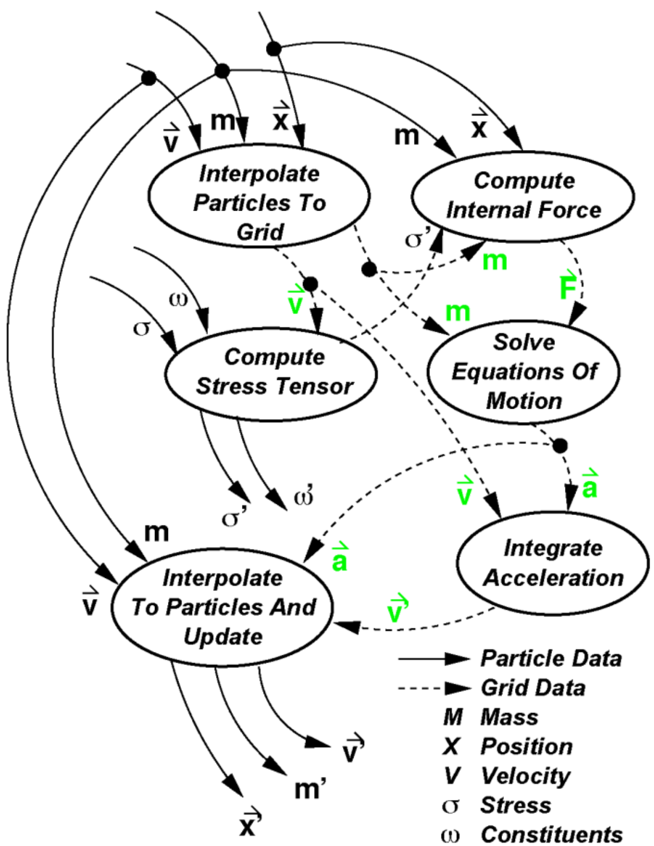
Shaped Charges

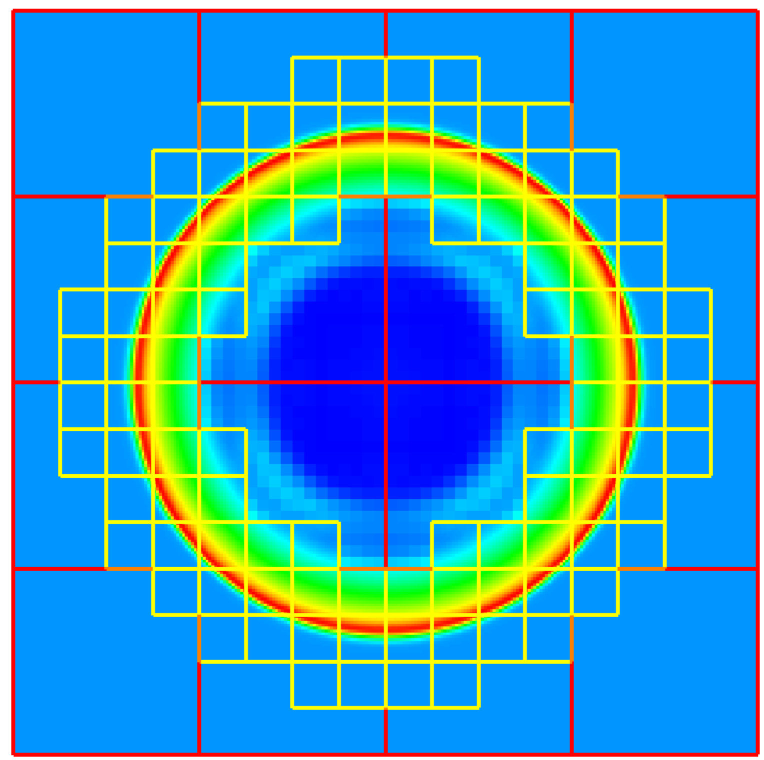


Foam Compaction

THE UNIVERSITY OF UTAH

How Does Uintah Work?



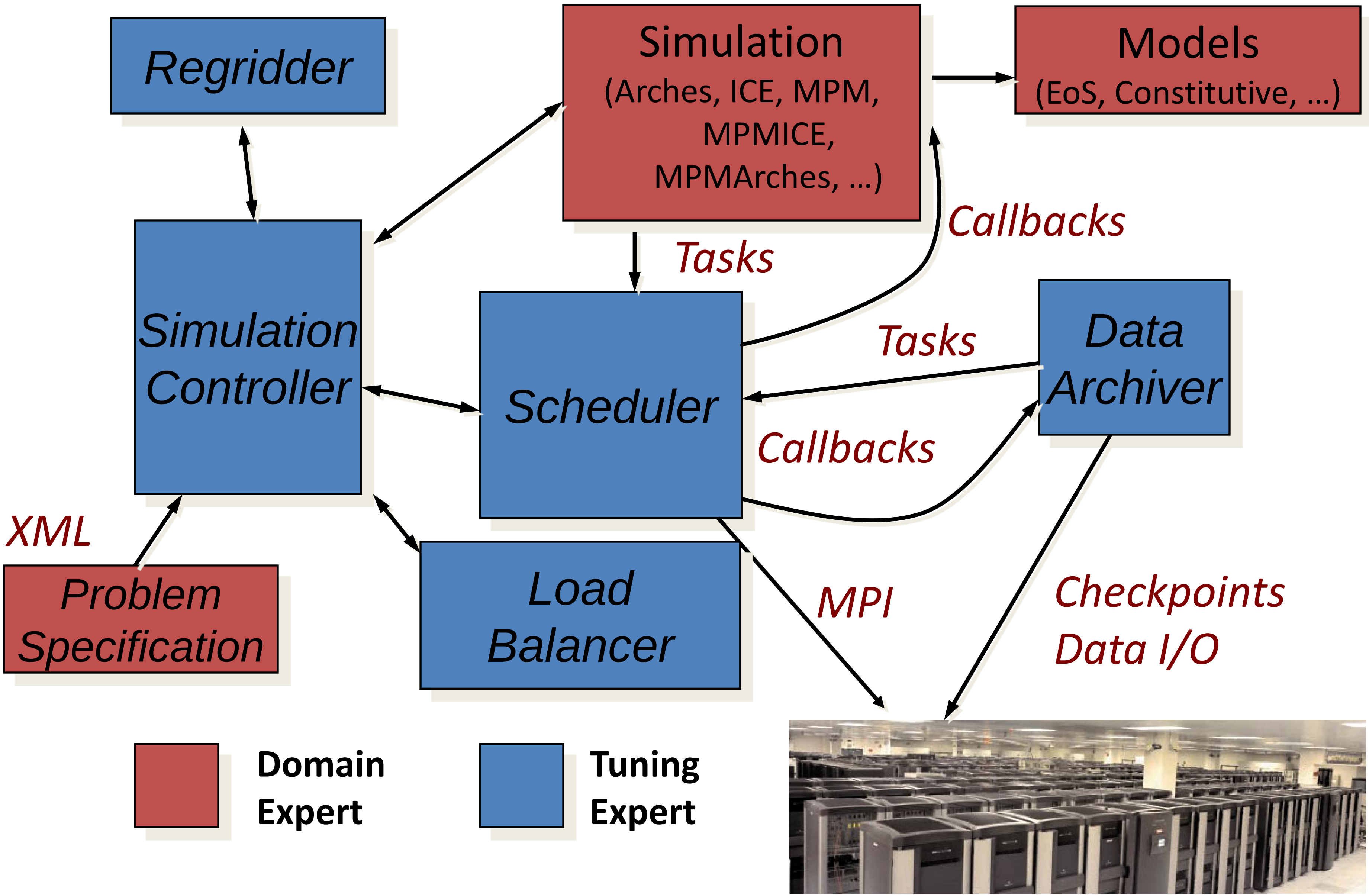


Task-Graph Specification

- Computes & Requries

Patch-Based Domain Decomposition

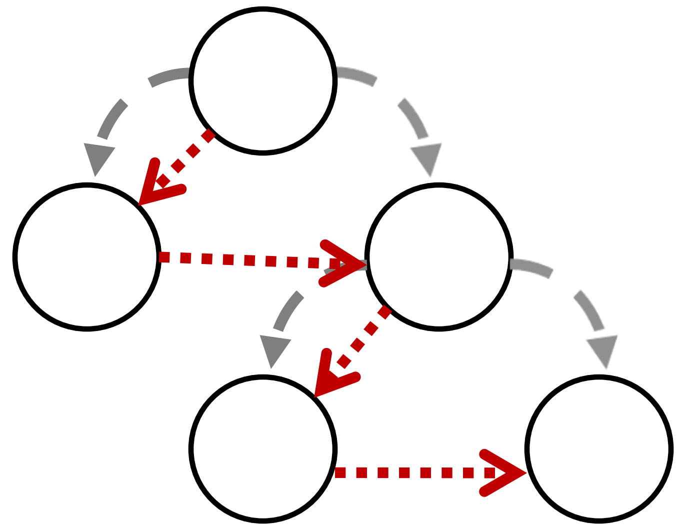
How Does Uintah Work?

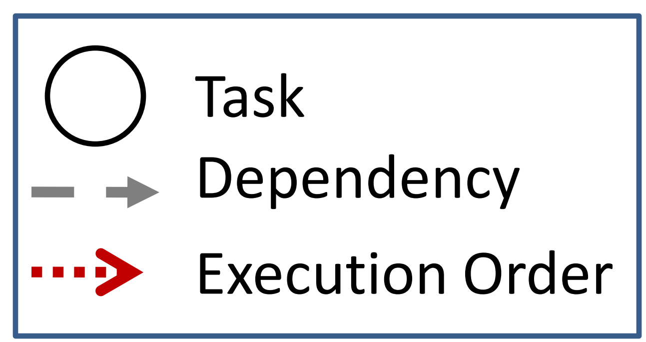


Task Graph Execution

1) Static: Predetermined order

- Tasks are Synchronized
- Higher waiting times





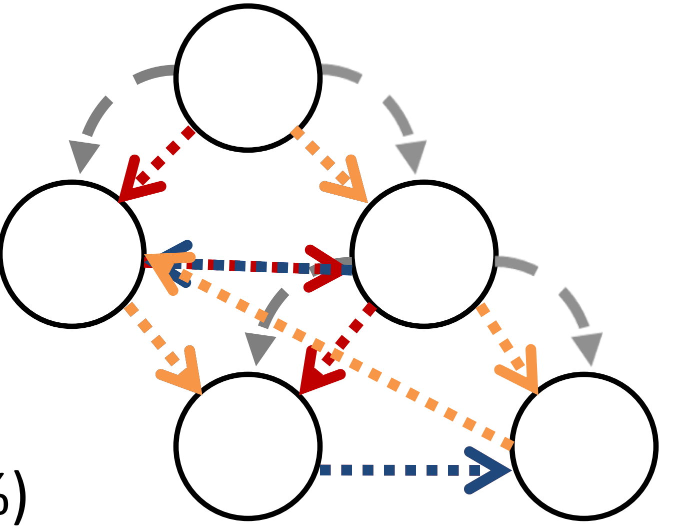
Task Graph Execution

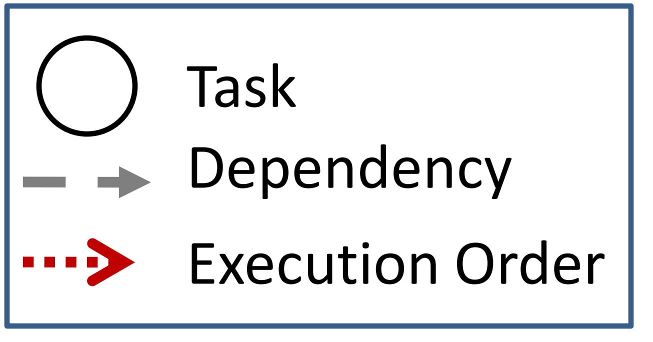
1) Static: Predetermined order

- Tasks are Synchronized
- Higher waiting times

2) Dynamic: Execute when ready

- Tasks are Asynchronous
- Lower waiting times (up to 25%)

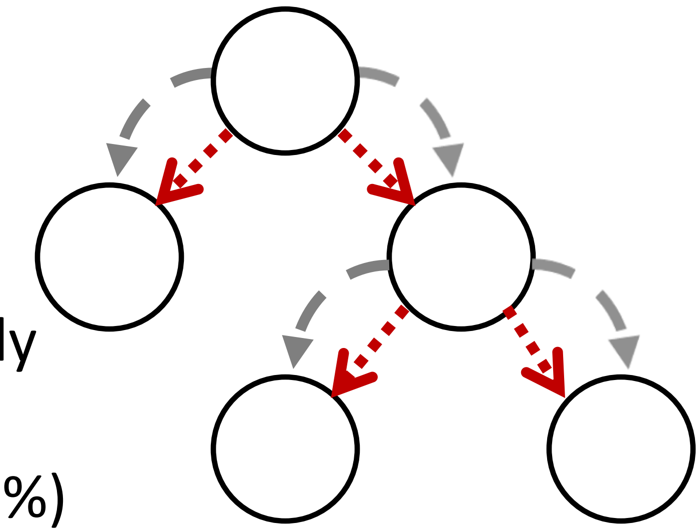




Task Graph Execution

1) Static: Predetermined order

- Tasks are Synchronized
- Higher waiting times

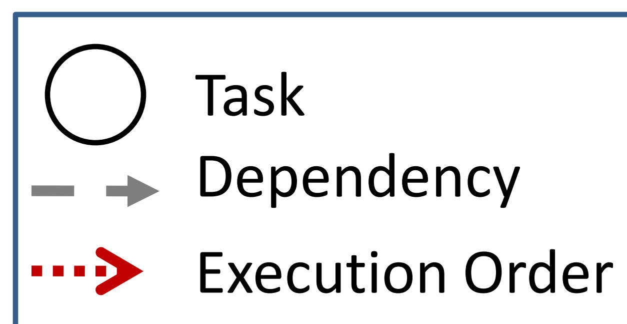


2) Dynamic: Execute when ready

- Tasks are Asynchronous
- Lower waiting times (up to 25%)

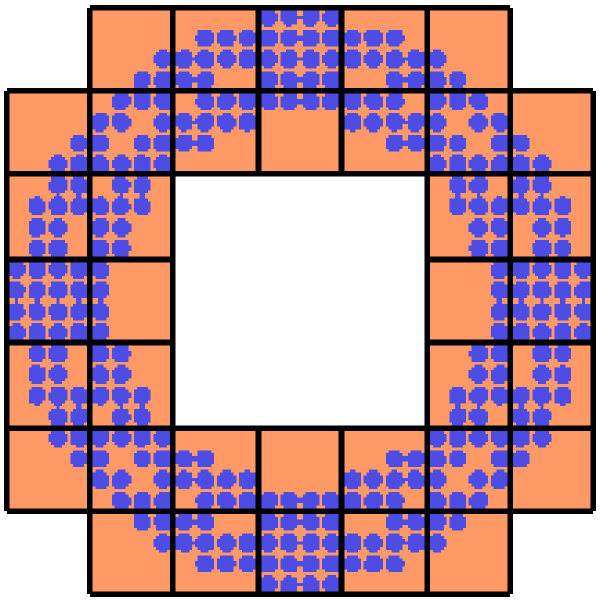
3) Dynamic Multi-threaded:

- Task-Level Parallelism
- Decreases Communication
- Decreases Load Imbalance



Tiled Regridding Algorithm

- Use fixed sized tiles
  - Occur at regular intervals
  - Can exploit regularity
    - Neighbor finding
    - Grid Comparisons



```
FOR each tile
    FOR each cell in tile
        IF cell has refinement flag
            patches.add(tile)
            BREAK
        END IF
    END FOR
END FOR
```

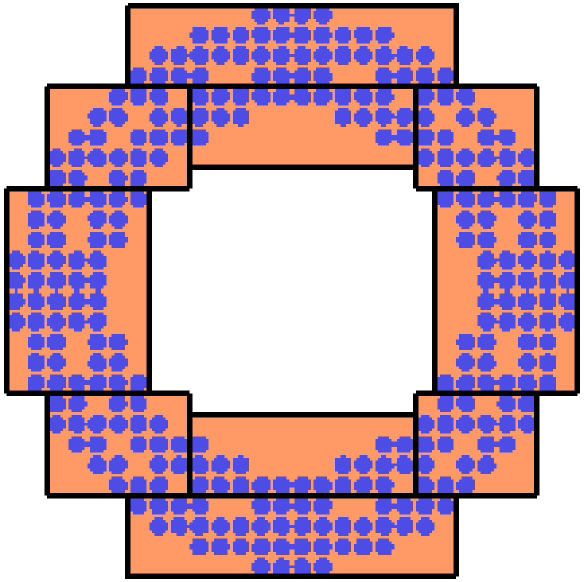
Trivial to paralleize

- **Computation: O(C/P)**
- **Communication: None!**
- **Faster than creating the flags list!**

Regridder Comparison

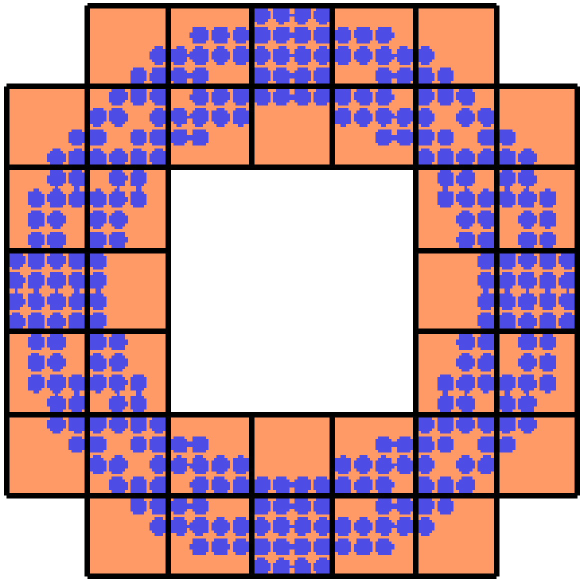
Berger-Rigoutsos

- Global algorithm
- Computation will not weak scale
- Communication will not weak or strong scale
- O(Patches) All reduces!
- Irregular patches
- Complex implementation

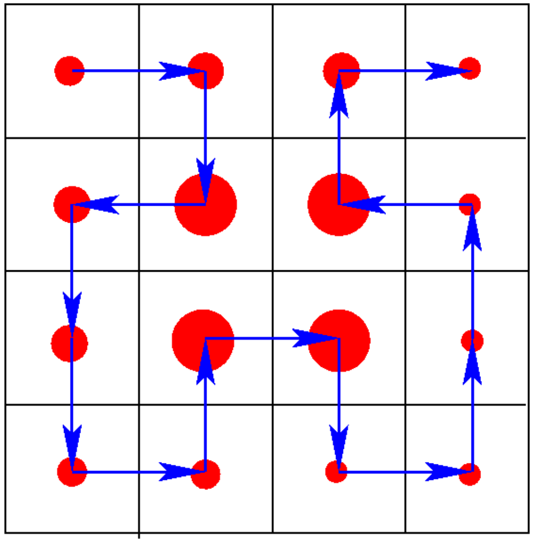


Tiled

- Local Algorthim
- Computation will weak & strong scale
- No communication
- Simple implementation
- Regular patches
- More Patches
- Over-refines



Uintah Load Balancing



- Assign Patches to Processors
  - Minimize Load Imbalance
  - Minimize Communication
  - Run Quickly in Parallel
- Uintah Default: Space-Filling Curves
  - ***O((N log N)/P + (N log^2 P)/P***
  - Luitjens, J., Berzins, M., and Henderson, T. Parallel space-filling curve generation through sorting: Research articles. *Concurr. Comput.: Pract. Exper. 19, 10 (2007), 1387–1402.*
- Support for Zoltan

In order to assign work evenly we must know how much work a patch requires

Cost Estimation: Performance Models

$E_{r,t}$: Estimated Time

G_r: Number of Grid Cells

P_r: Number of Particles

$$E_{r,t} = c_1 G_r + c_2 P_r + c_3$$

c_1, c_2, c_3 : Model Constants

- **Need to be proportionally accurate**
- **Vary with simulation component, sub models, compiler, material, physical state, etc.**

Can estimate constants using least squares at runtime

$$\begin{bmatrix} G_0 & P_0 & 1 \\ \dots & \dots & \dots \\ G_n & P_n & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} O_{0,t} \\ \dots \\ O_{n,t} \end{bmatrix}$$

$O_{r,t}$: Observed Time

What if the constants are not constant?

Cost Estimation: Fading Memory Filter

$E_{r,t}$: Estimated Time $O_{r,t}$: Observed Time α: Decay Rate

$$E_{r,t+1} = \alpha \, O_{r,t} + (1 - \alpha) \, E_{r,t}$$

$$= \alpha \underbrace{(O_{r,t} - E_{r,t})}_{\text{Error in last prediction}} + E_{r,t}$$

- No model necessary
- Can track changing phenomena
- May react to system noise
- Also known as:
  - Simple Exponential Smoothing
  - Exponential Weighted Average

Compute per patch

Cost Estimation: Kalman Filter, 0th Order

$E_{r,t}$: Estimated Time **$O_{r,t}$: Observed Time**

Update Equation: $E_{r,t+1} = E_{r,t} + K_{r,t} (O_{r,t} - E_{r,t})$

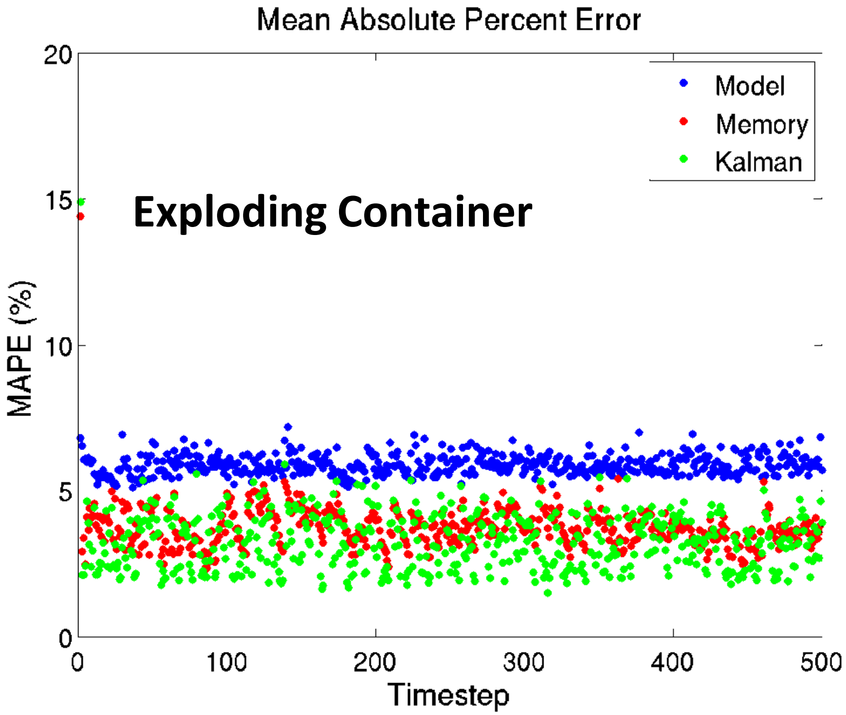
Gain: $K_{r,t} = M_{r,t} / (M_{r,t} + \sigma^2)$

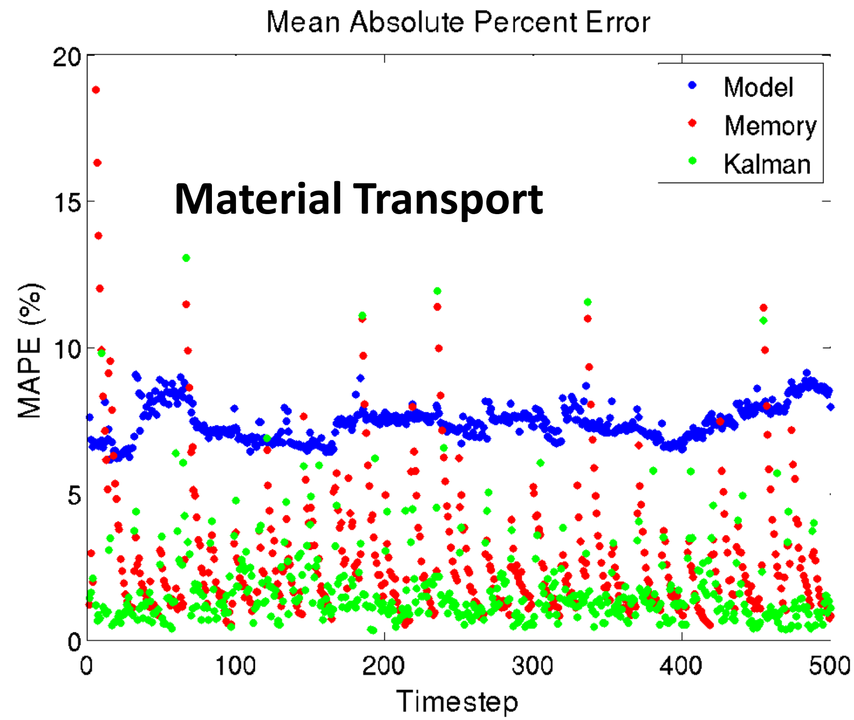
a priori cov: $M_{r,t} = P_{r,t-1} + \phi$

a posteri cov: $P_{r,t} = (1 - K_{r,t}) M_{r,t}$ $P_0 = \infty$

- Accounts for uncertainty in the model: ϕ
- Accounts for uncertainty in the measurement: σ^2
- No model necessary
- Can track changing phenomena
- May react to system noise
- **Faster convergence than fading memory filter**

Cost Estimation Comparison



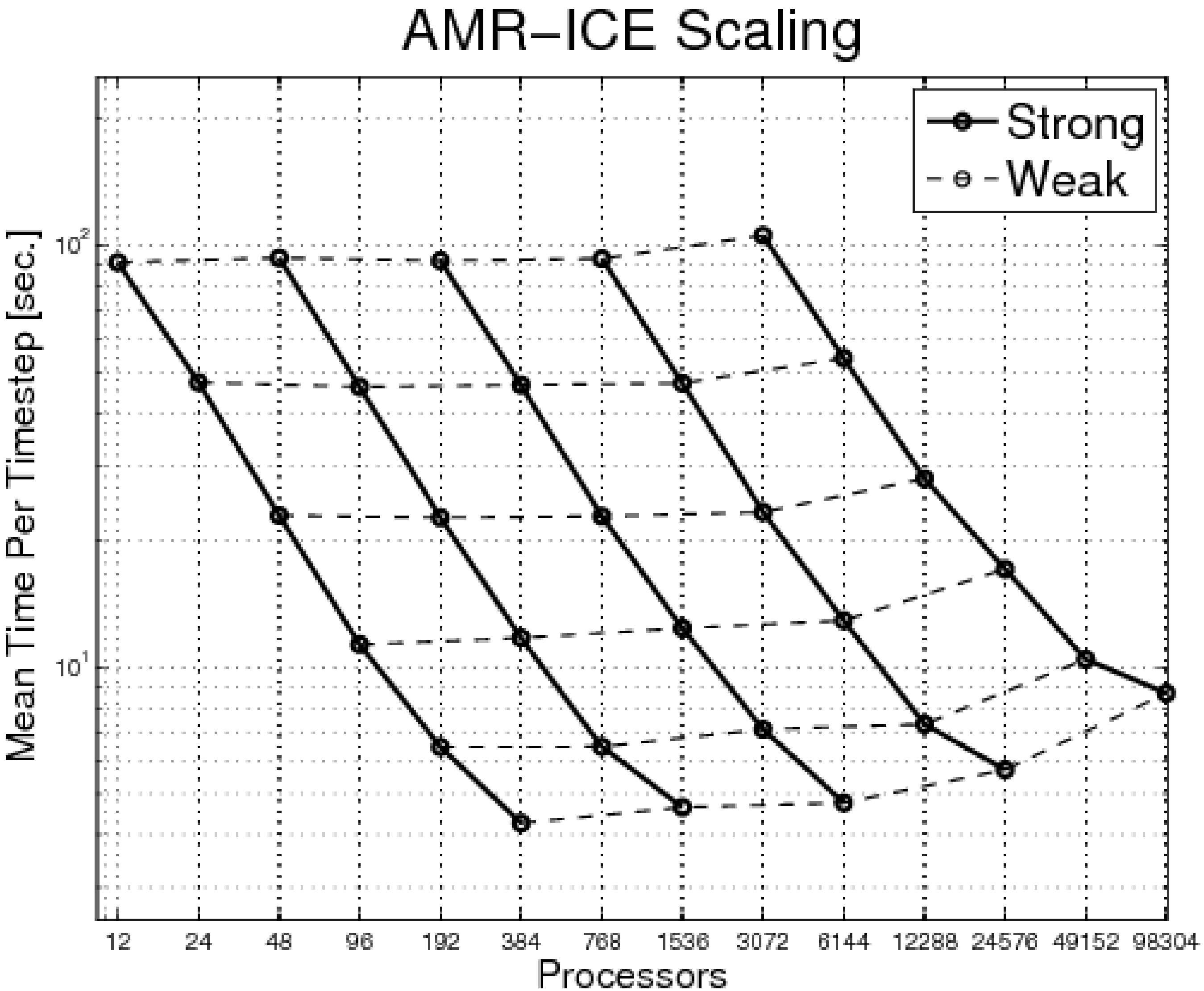


- Filters provide best estimate
- Filters can spike with system noise

	Ex. Cont.	M. Trans.
Model LS	6.08	7.63
Memory	3.95	3.10
Kalman	3.44	2.01

Justin Luitjens and Martin Berzins, Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework, Accepted in IPDPS 2010.

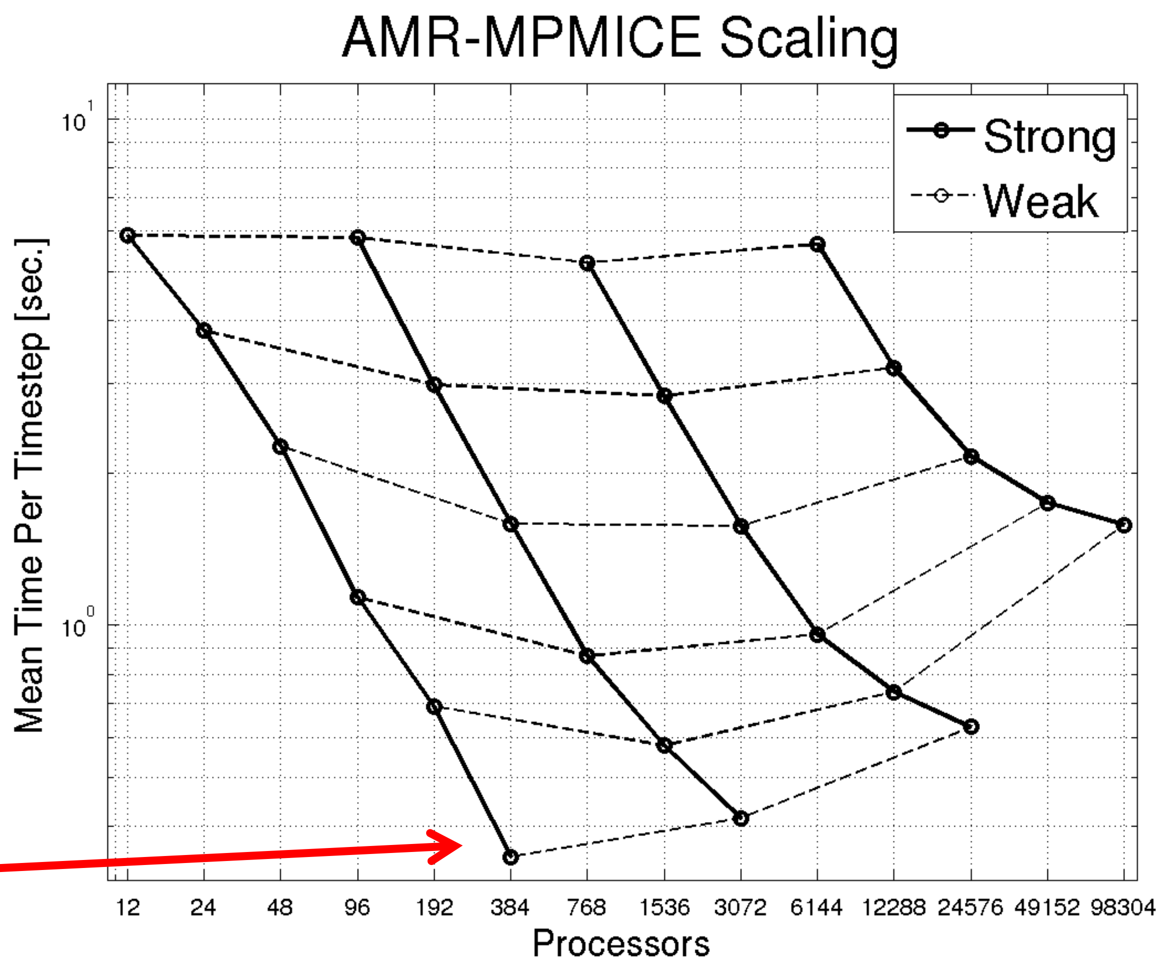
AMR ICE Scalability



AMR MPMICE Scalability

Decent MPMICE scaling

More work is needed



One 8^3 patch per processor

Problem: Exploding Container