SWIFT 2 :

Keeping the Good,

Discussing the Bad,

Removing the Ugly

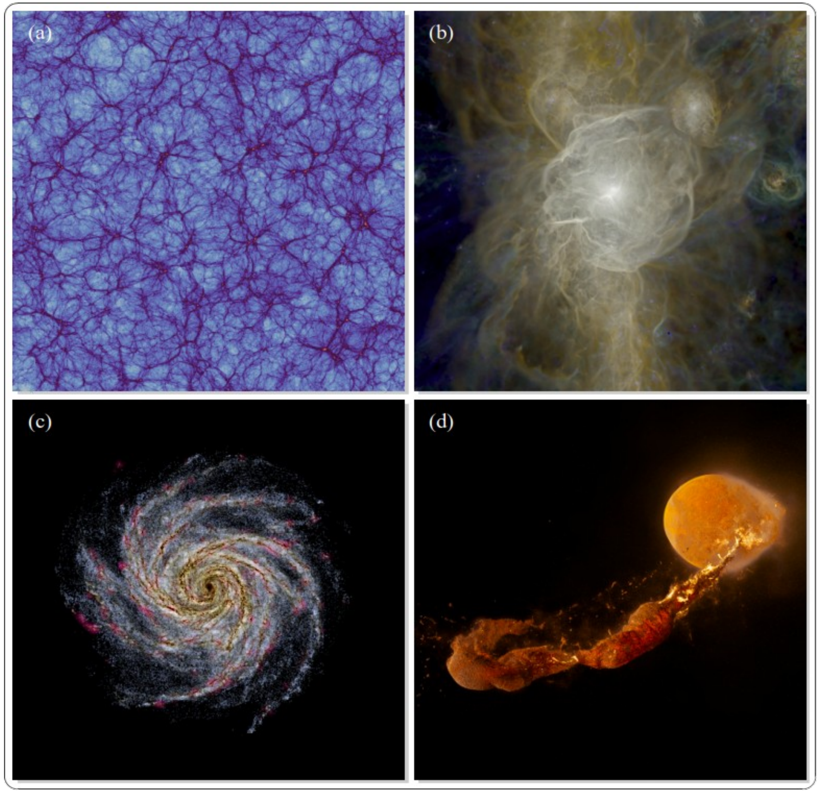
Cristian Barrera-Hinojosa, **Mladen Ivkovic,** *Pawel Radtke, Tobias Weinzierl*
Durham University
PAX-HPC project meeting, Lancaster
22. April 2024

What it Does

Cosmological & Astrophysical simulations:

- Hydrodynamics
- Gravity and Dark Matter
- Planetary science
- Neutrinos
- Radiative transfer and cooling
- Sub-grid models
- And much more!

Visit us at swiftsim.com



Schaller et al. 2024

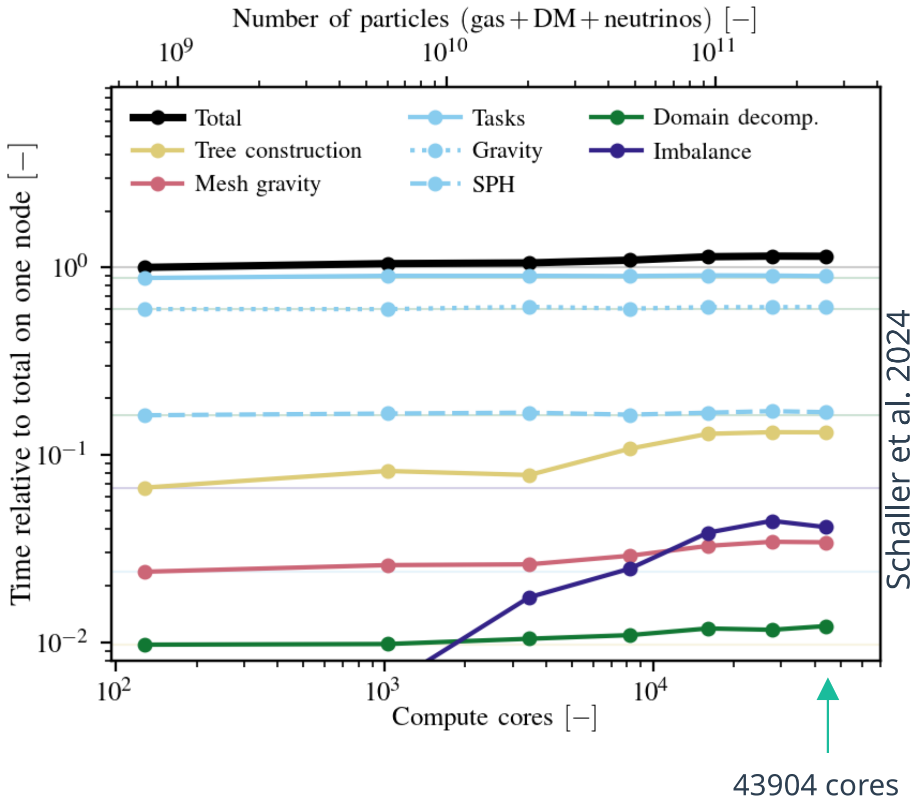
Under the Hood

- Particle methods to solve the physics
  - several flavours for almost all physics
- Written in C
- Paralellism:
  fine-grained interdependent tasking with own scheduler based on pthreads (based on QuickSched library)
  - Permits Asynchronous MPI communications
  - Permits domain decomposition based on work, not data

SWIFT: The Good

The fine-grained tasking approach is key to SWIFT's successes:

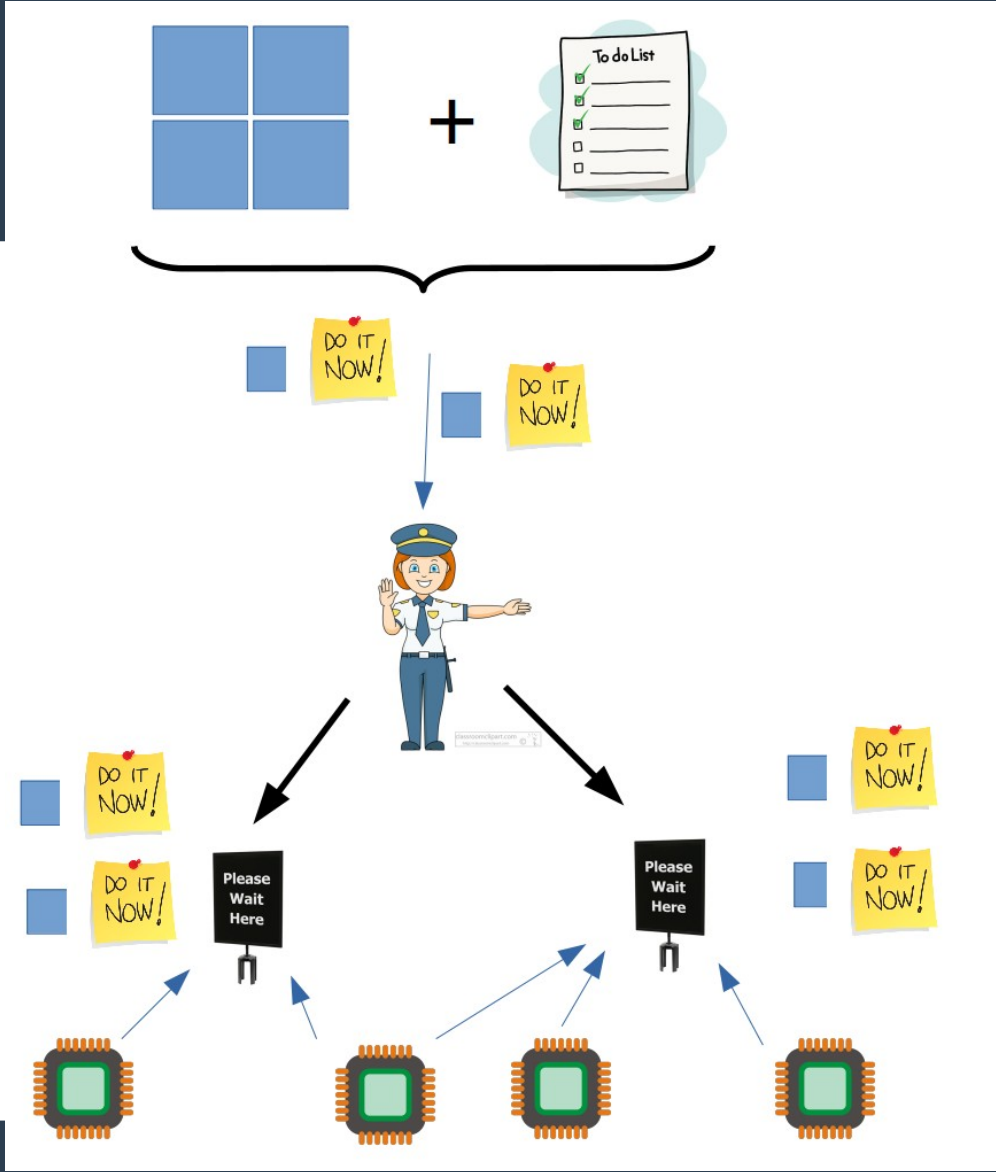
- Largest ever moon formation simulations
- Largest cosmological hydrodynamical simulation (by particle number):
  - 128×10^9 hydro particles
  - 128×10^9 gravity particles
  - 10^9 neutrino particles
- Remarkable weak scaling



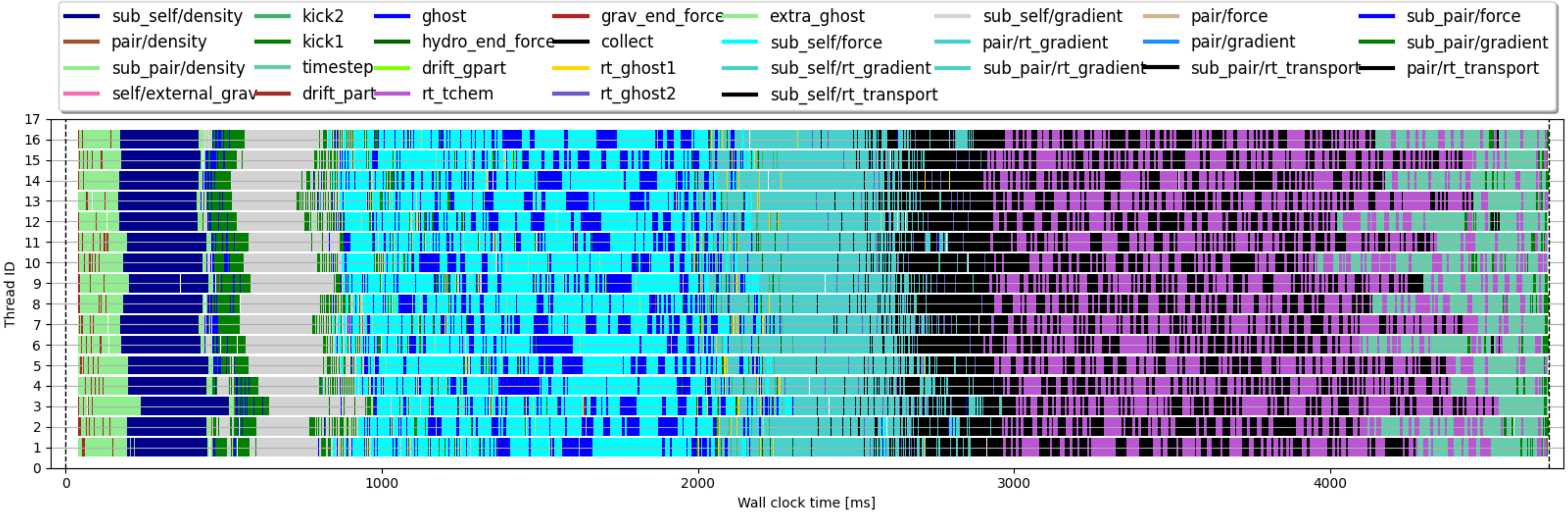
SWIFT: The Not-So-Good

- **Clearly SWIFT is doing a couple of things right.**
- **What can and needs to be improved upon?**
  - We need to look into how SWIFT does things internally, in particular how the fine-grained tasking and scheduler work.

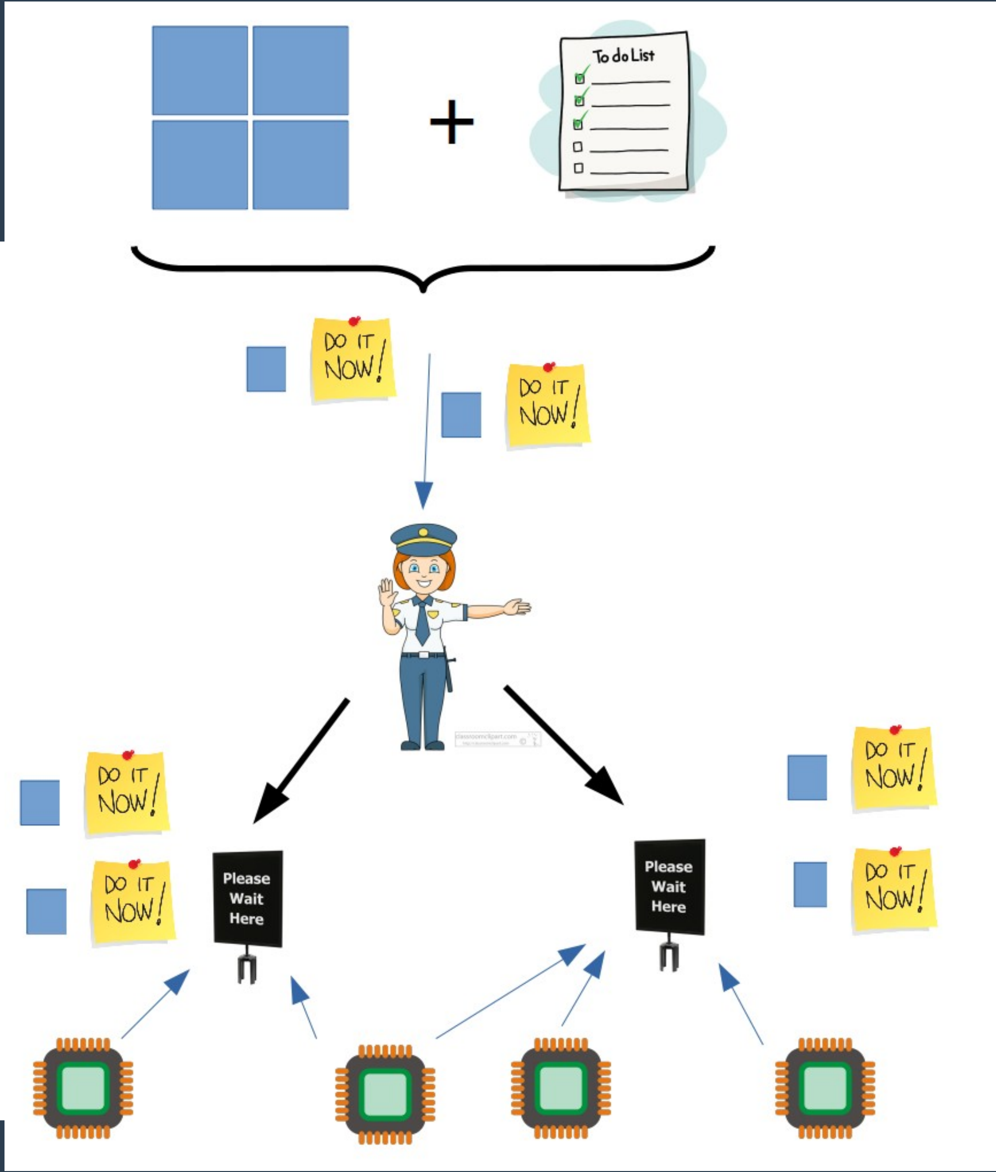
Task-Based Parallelism



Task-Based Parallelism: How it looks like in practice



Task-Based Parallelism



Task-Based Parallelism: Dependencies

Task:

+

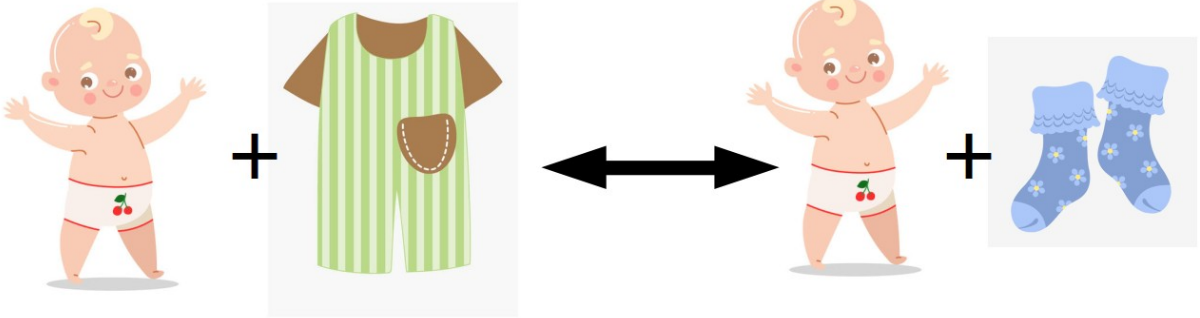
Dependency:

→

Task-Based Parallelism: Conflicts

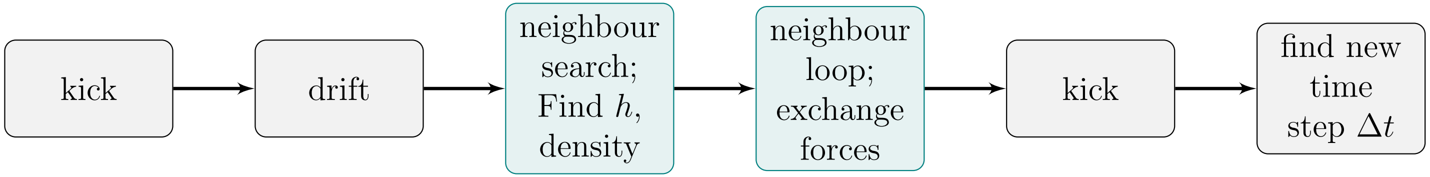
Task:

Conflicts:

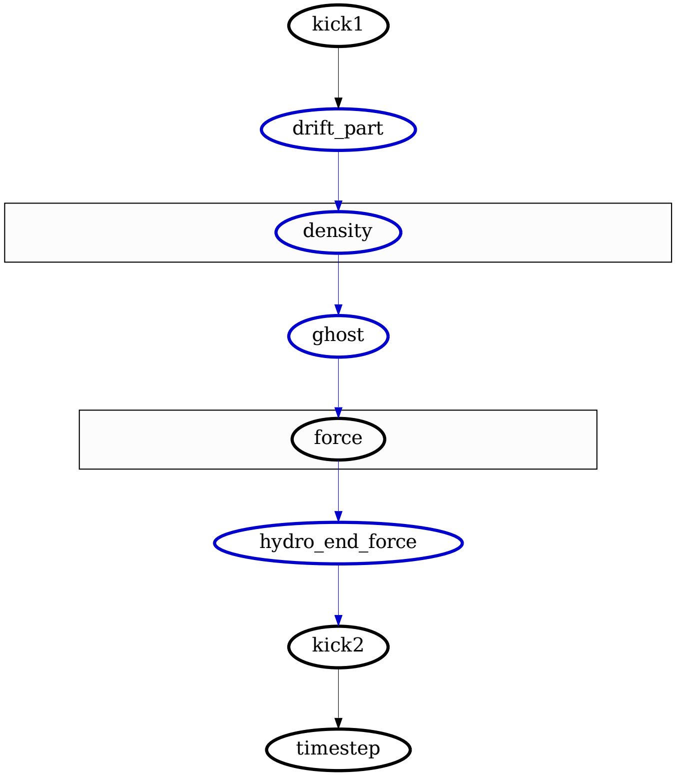


The Dependency Graph As Algorithm Steps

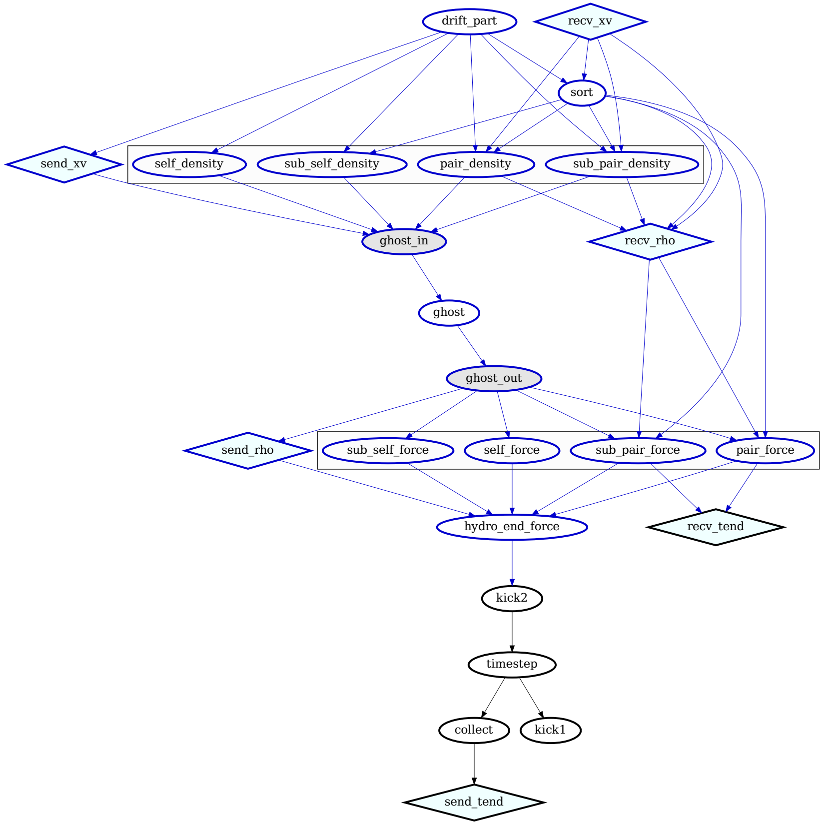
A single SPH step for each particle needs the following order of operations:



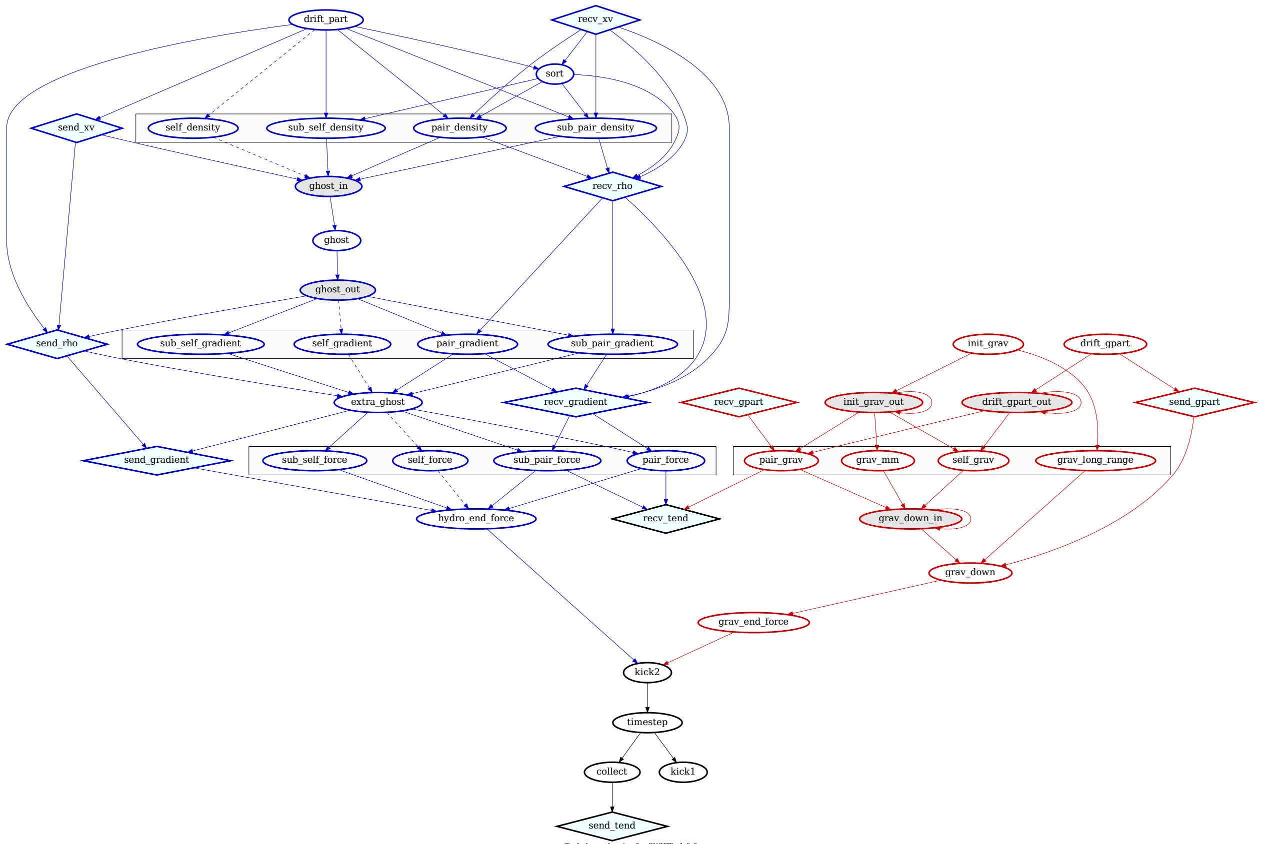
The Dependency Graph As Algorithm Steps



The Dependency Graph: In Reality

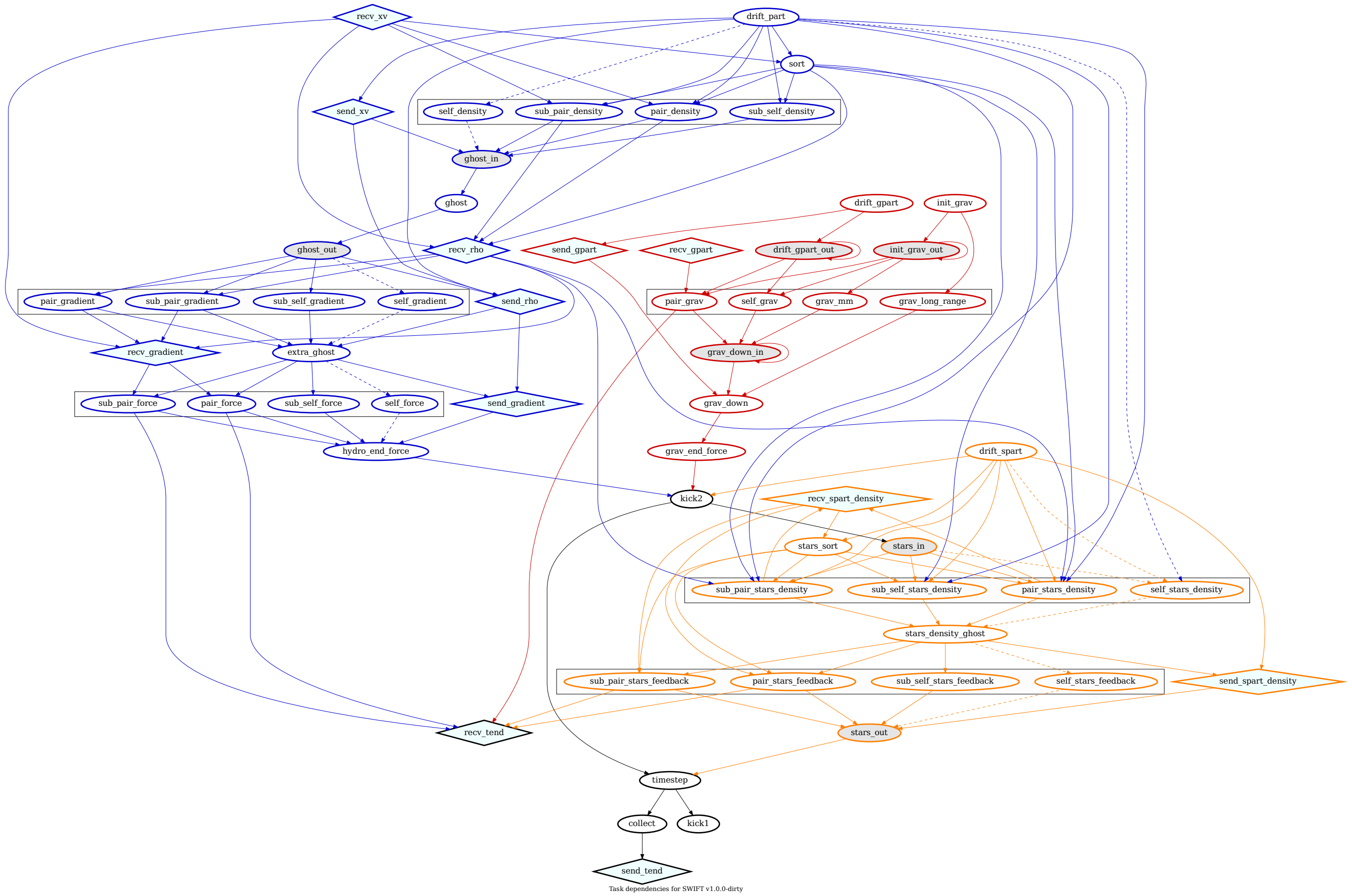


Adding Gravity

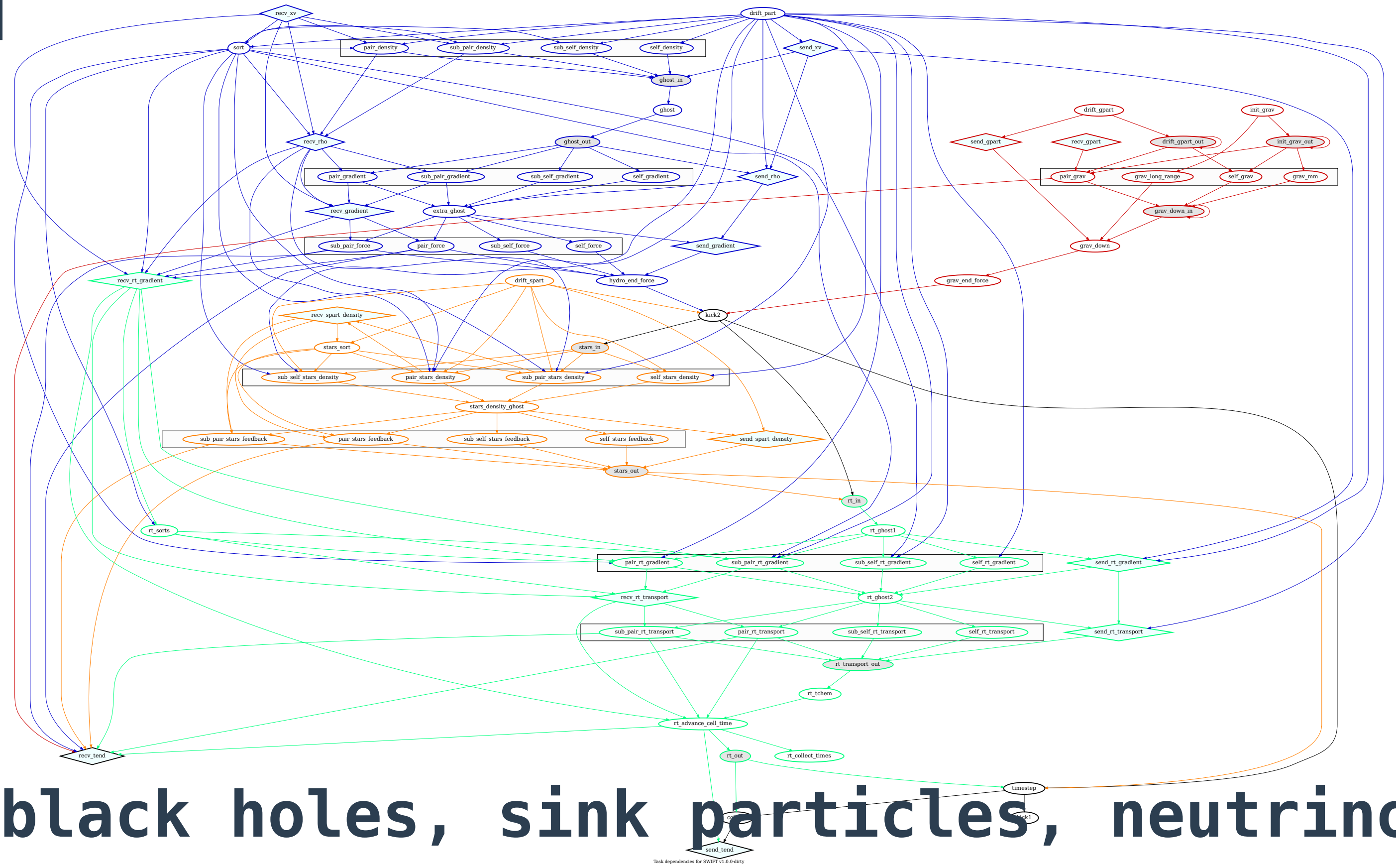


Task dependencies for SWIFT v1.0.0

Adding Stars and Stellar Feedback



Adding Radiative Transfer



… and black holes, sink particles, neutrinos, MHD...

How Is It Done?

- **Task creation:**

  engine_maketasks.c: ~5k lines of

```
if (condition A) {
    TA = create_task(A);
}
if (condition B) {
    TB = create_task(B);
}
if (condition A && condition B &&
    condition C) {
    create_dependency(TA, TB);
}
```

- **Task activation:**

  cell_unksip.c: ~3.5k lines of

```
if (condition A) {
    activate_task(TA);
}
if (condition B) {
    activate_task(TB);
}
```

All of this needs to be done manually.

To Make Matters Worse

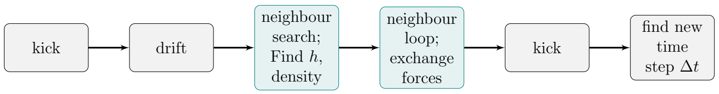
The dependency graph doesn't show the full picture.

Discussing the Bad

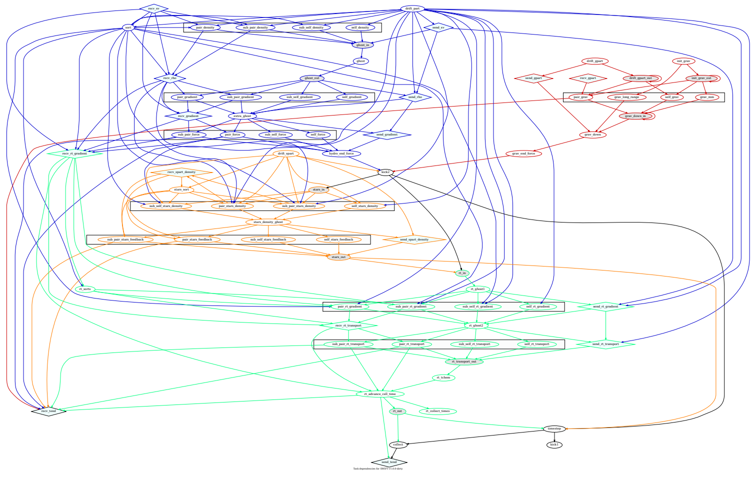
- **The current tasking system is deeply embedded into SWIFT**
  - Adding new physics to SWIFT is **tricky, convoluted, and very time consuming**.
  - There are **countless pitfalls and edge cases** that are nearly impossible to predict and hard to diagnose and debug.
  - This means that physicists will have to spend a lot of time not doing physics. :(
- **The current tasking system is not future-proof**
  - Only supports CPU tasking, **no GPUs** (yet).

How do we fix that?

- **We need to replace the engine.**
- **Goals:**
  - Keep fine-grained tasking.
  - Separation of concerns:
    users specify this:



Not this:



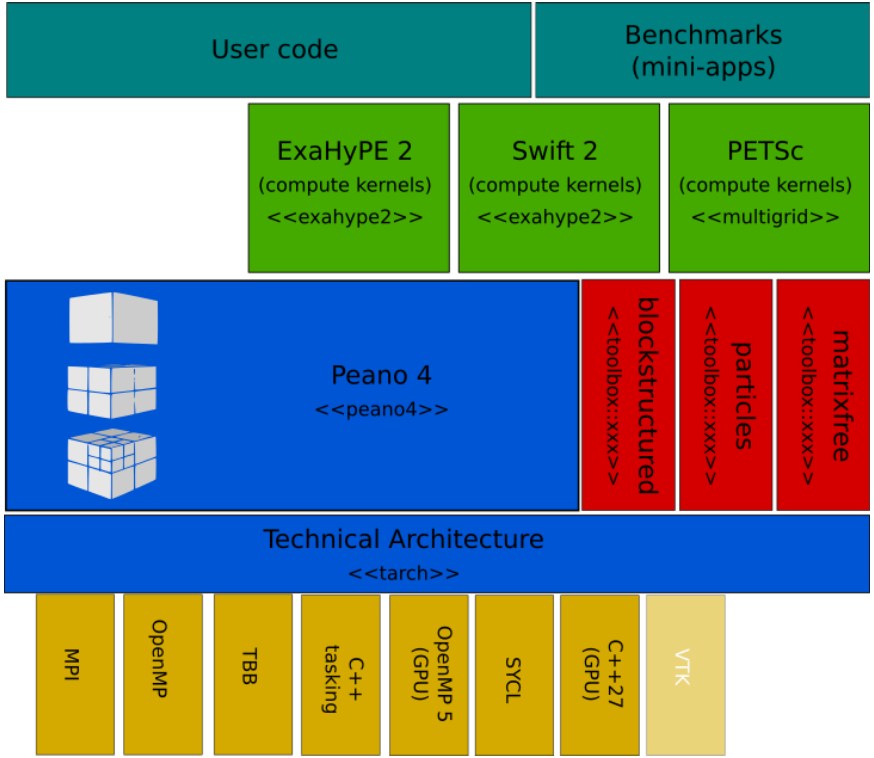
Goals (cont.)

Dependency graph is ***generated***, not ***written*** by devs

- Users can focus on the equations they want solved
- We can worry about (and play with) the underlying framework
  - Precise parallelisation strategy: Which scheduler to use? What to solve on GPUs? Which MPI communication strategy to use?
  - Data management: What to store as AoS, what as SoA? What precision to use?
  - How to group together function calls into tasks?
  - We can even go so far as to design a set of tests and benchmarks that will tell you the best configuration for your problem and your machine.

How?

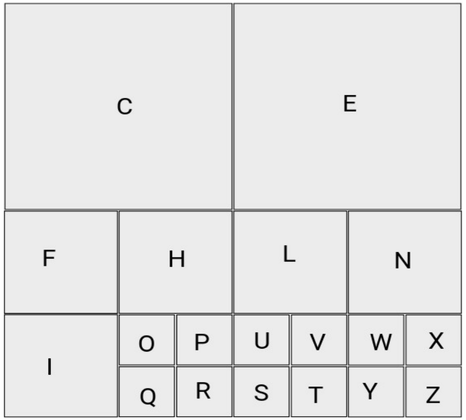
- Place SWIFT in Peano4 framework
  - Peano4 provides parallelisation, domain decomposition, optimization
  - SWIFT 2 extension provides framework to adopt Swift kernels (physics)

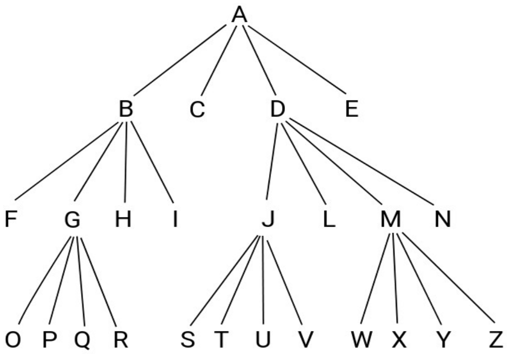


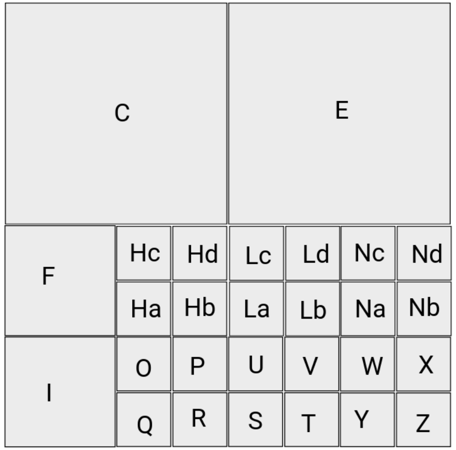
http://www.peano-framework.org

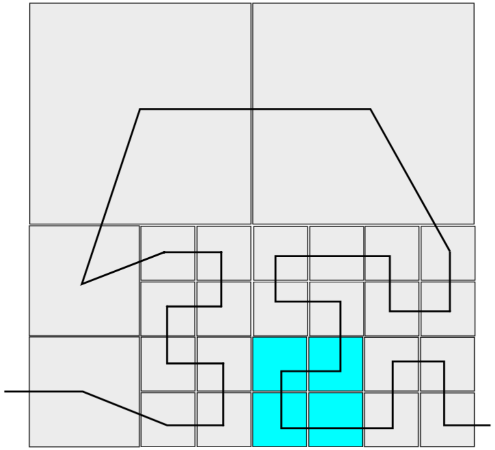
What Peano Gives Us

- Adaptive Mesh Refinement
- Tree Traversals along Peano Space Filling Curve









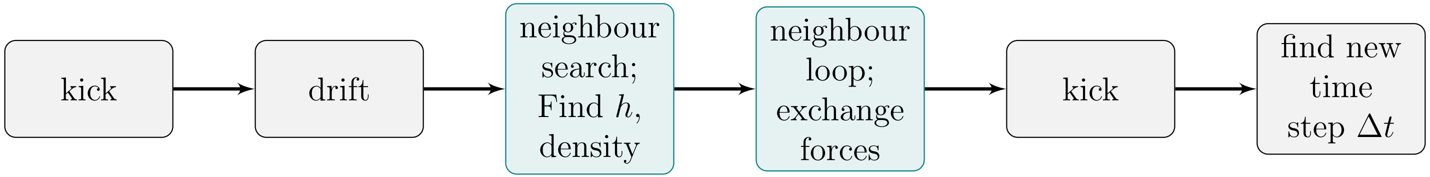
Weinzierl et al. 2019

How It Works

- **Particles are stored in a dual tree:**
  - Both in cells and on vertices
- **Peano provides top-down grid traversals.**
  - Users can’t touch that.
- **During the traversal, events are triggered.**
  - vertex/cell touched for the first time during traversal.
  - Cell can be worked on.
  - vertex/cell touched for the last time during traversal.
- **We attach whatever we need done to these events.**

How It Works

- **Main Idea:**
  - Translate Algorithm steps onto grid traversals using these events.
  - One algorithm step corresponds to one grid traversal.



Example

- **Touch vertex first time:**
  - Do something on all particles assigned to this vertex
- **Cell can be worked on:**
  - Do a particle-particle interaction loop
- **Touch vertex last time:**
  - Do something on all particles assigned to this vertex

What It Looks Like In Practice

- **Step 1: Define a particle type**

```
class Particle():
  self.data.add_attribute( dastgen2.attributes.Double("mass") )

  self.data.add_attribute( dastgen2.attributes.Double("density" ) )

  self.data.add_attribute( dastgen2.attributes.Double("pressure") )

  self.data.add_attribute( dastgen2.Peano4DoubleArray("v","Dimensions") )

  # etc …
```

What It Looks Like In Practice

- **Step 2: Define the life cycle of your particle**

```
kick    = AlgorithmStep( ... )
drift   = AlgorithmStep( ... )
density = AlgorithmStep(
            name               = "Density",
            touch_vertex_first_time_kernel = "functionPrepareDensity(particles);" ,
            cell_kernel =                    "densityInteraction(particles);",
            touch_vertex_last_time_kernel =  "functionEndDensity(particles);",
          )
```

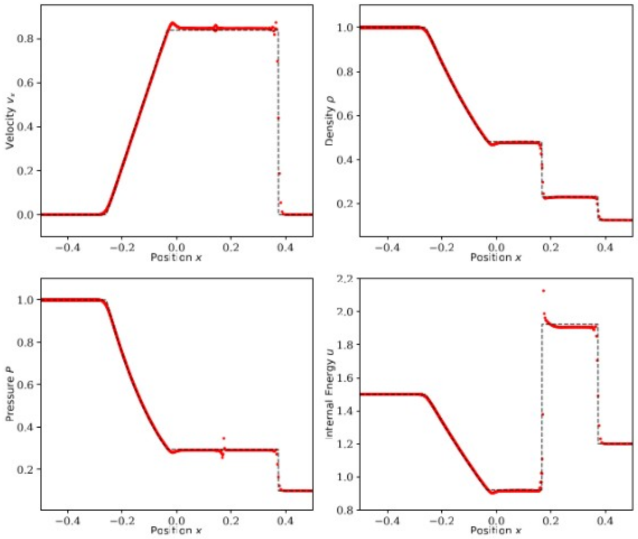
- **then add it to your particle:**

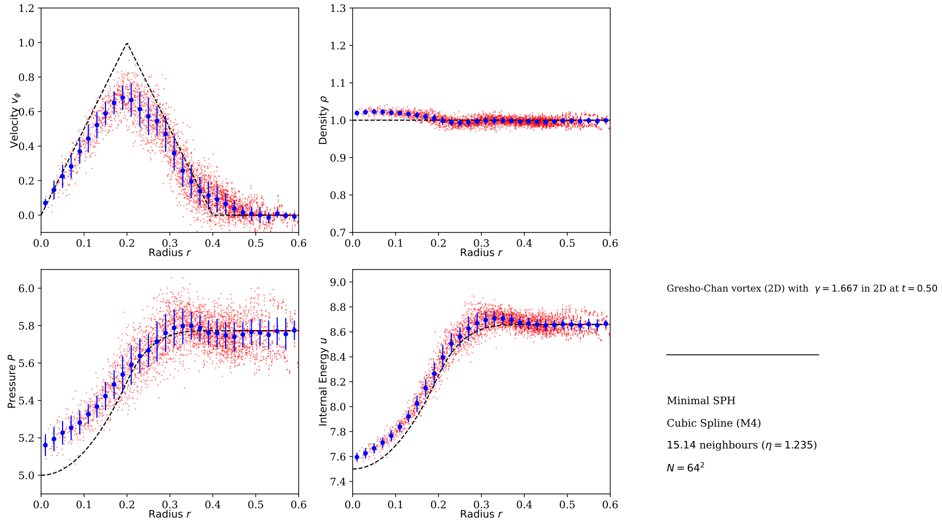
```
particle.algorithm_steps = [kick, drift, density, force, timestep]
```

- **And Peano4/Swift 2 does the rest for you!**

Current State Of Affairs

- Bare-bones SPH implementation is present and running





Automatic Runtime Dependency Checks

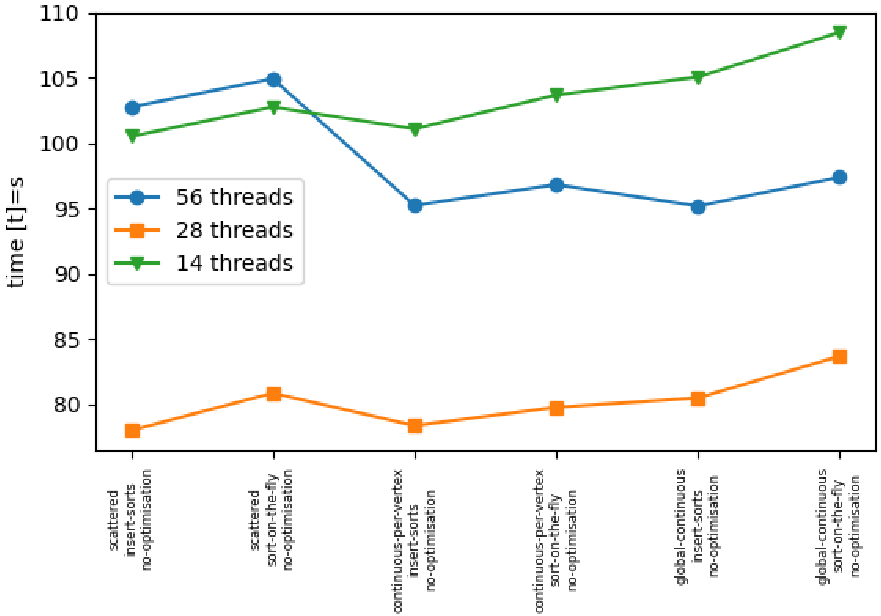
- In Debug mode, we can keep track of each stage of the particle during a simulation step

	Touch Vertex First Time	Cell Kernel	Touch Vertex Last Time
AlgorithmStep 1	1	1	1
AlgorithmStep 2	1	0	0
...			
AlgorithmStep N	0	0	0

- Verify on-the-fly that dependencies are satisfied: Nothing done too early, nothing done too late.
- These checks are automatically generated for you!

Storage Management Experiments

- **Store particles**
  - Globally, randomly on heap
  - Globally, contiguous
  - Per-vertex, contiguous
- **Particle sorting:**
  - On-the-fly, or in additional step
- **Outcome:**
  - Sorting comes at noticeable expense
  - For large thread counts, sorting gives speedup, as nasty memory access is avoided



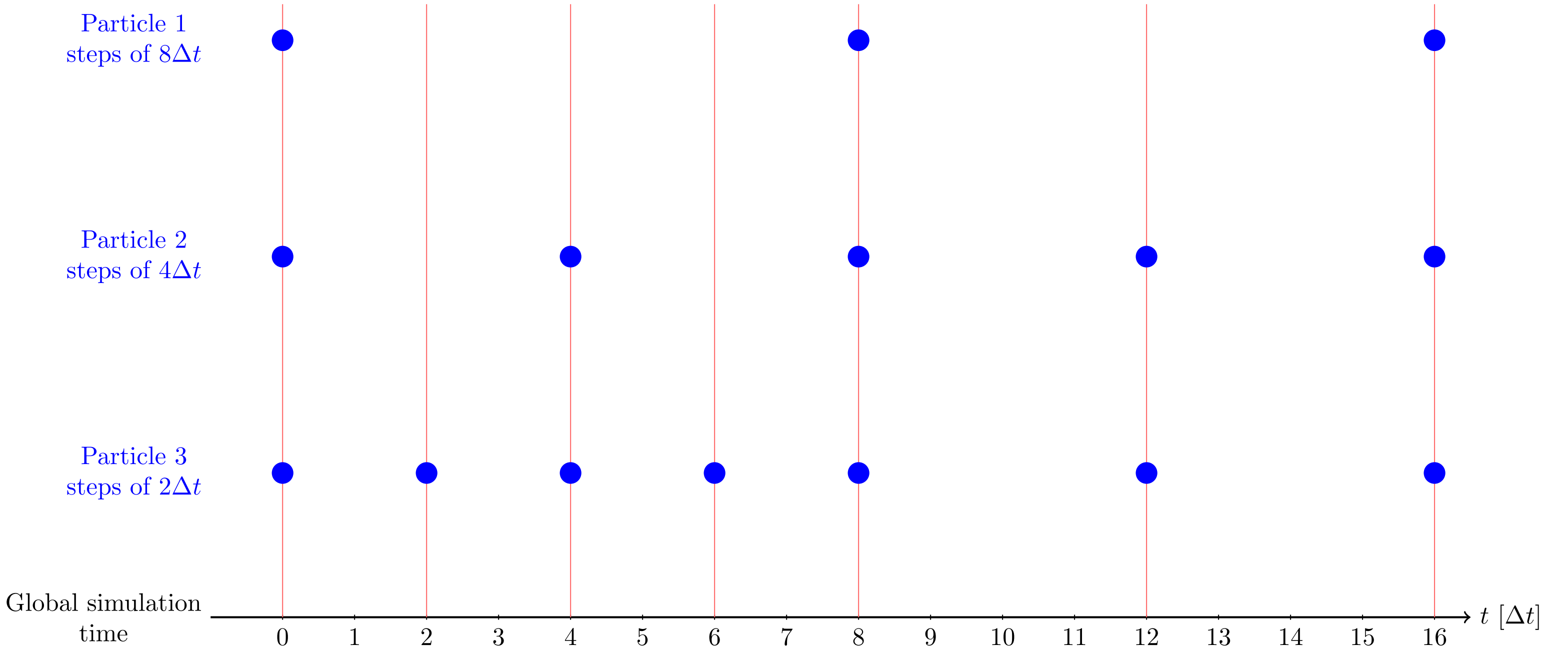
Outlook

- **Currently in progress and planned:**
  - A wider suite of benchmarks, testing different scenarios
  - Performance analysis and optimization
  - Compiler extension to allow memory compression via C++ annotations
  - Adaptive and individual time step sizes
  - Additional physics, additional particle methods...

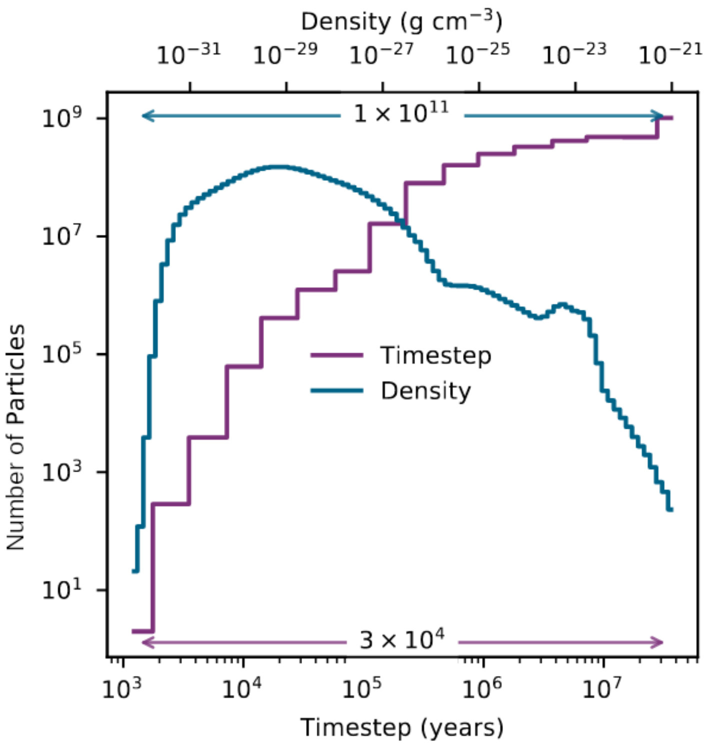
Final Slide

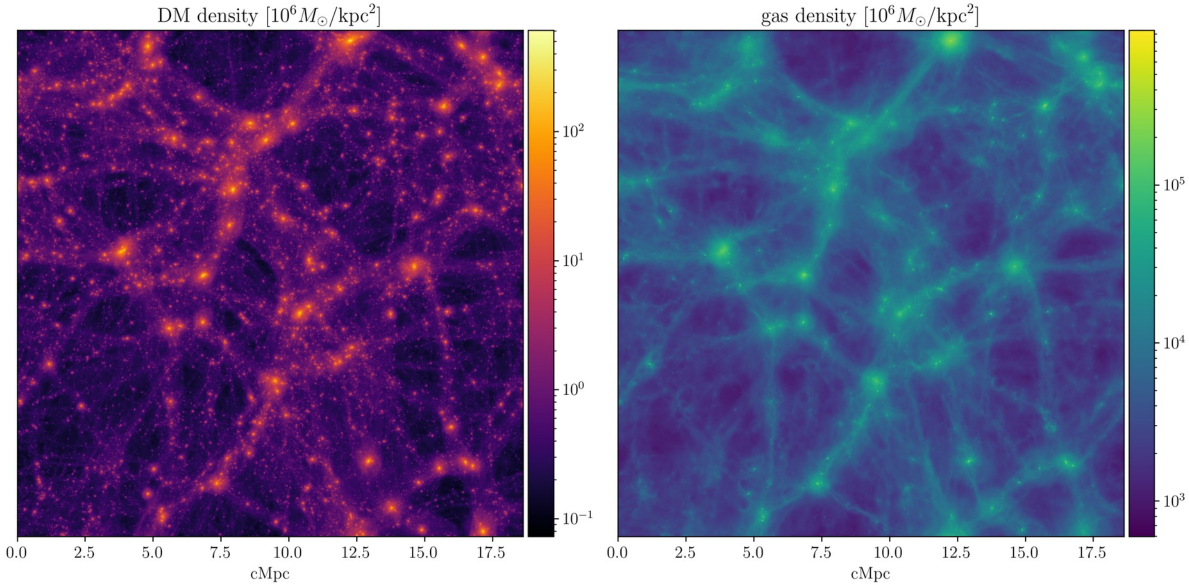
- Final Slide

Individual Timestepping



Individual Timestepping





Borrow et al. 2018

Data-Based Parallelism