Eiron
Exploring parallelization strategies for EIRENE

Oskar Lappi

November 23, 2022

Presentation outline

- Recap of the goals behind the Eiron project
- An EIRENE-Eiron comparison with some simple slab cases
- Some parallel algorithms for the neutral particle transport problem
- Performance comparisons

Recap: the problem

- EIRENE has a plethora of features, supporting a user community with multiple use-cases and integrations with external software
- EIRENE's computational capacity is being stretched in more challenging use cases, there is increasing demand for simulations at larger computational scale
- These users of EIRENE rely on current interfaces and behavior of EIRENE, so we cannot easily make breaking changes
- Making fundamental changes in order to improve performance and scalability is difficult to begin with, and the constraints of not breaking multiple legacy use cases while doing this makes the task more difficult

Recap: the project

- We want to try different approaches before settling on one for EIRENE, compare their performance and scalability
- We're building a modular software library with a simplified core toy model of the neutral particle transport problem and a very lean set of features compared to EIRENE
- We're using the library to create different solvers and comparing their performance and scalability
- We're calling this software library Eiron

EIRENE vs Eiron

Eirene ($E\iota\rho\eta\nu\eta$)

- Athenian goddess of peace
- Serious business

Eiron ($E\iota\rho\omega\nu$)

- Athenian comedy character
- Less serious business

Eiron is much more simple than Eirene

Eiron...

- only solves problems on 2D structured Cartesian grids
- has nearly no physics built into it
- only has one simple (hard-coded) track length estimator
- does not yet calculate tally variances

Eiron has more of a software engineering focus

Eiron...

- provides components that interface with each other, which allows us to compose them in different ways
- has multiple parallel implementations of a (simplified) neutral particle transport solver using OpenMP and MPI
- the design is modular and data driven, and should be flexible enough to implement a few different physics cases
- end-to-end tests are used to compare the equality of the parallel implementations
- will produce the same result independent of algorithm and number of threads (with threshold for floating point operations)

Comparing EIRENE & Eiron with a contrived 2D slab case

Because Eiron is mostly a performance study, there is not much in terms of validation. In lieu of that, I'll try to show that Eiron passes the eye test: similar cases produce results that look qualitatively similar between EIRENE and Eiron.

The goal here is really just to show that Eiron does the same work that EIRENE does (for these simple 2D slab cases) and that therefore the performance lessons we've learned from Eiron can be assumed to translate to EIRENE.

Comparing EIRENE & Eiron with a contrived 2D slab case

Let’s start with a constant angular distribution orthogonal to the line source, 1200000 particles, one terminating and one scattering collision process. The scale will be off by a large multiple, but that’s not important.
Eiron’s rates are ionization rate: 0.0076, scattering rate: 0.0056.

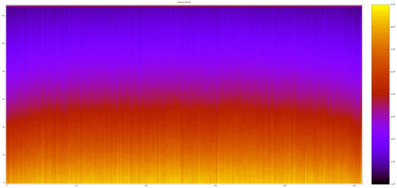


Figure: EIRENE

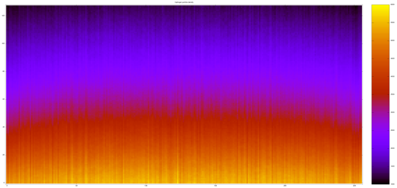


Figure: Eiron

Comparing EIRENE & Eiron with a contrived 2D slab case

Same case, slices in the x direction

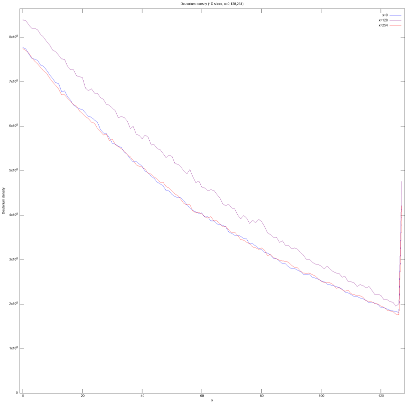


Figure: EIRENE

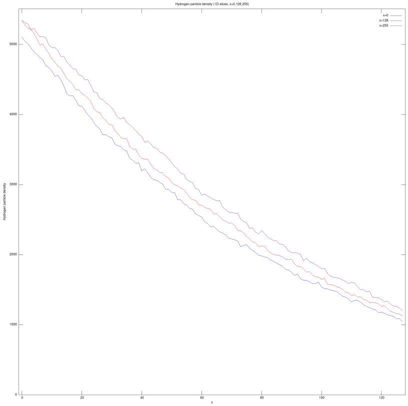


Figure: Eiron

Comparing EIRENE & Eiron with a contrived 2D slab case

Slices in the y direction

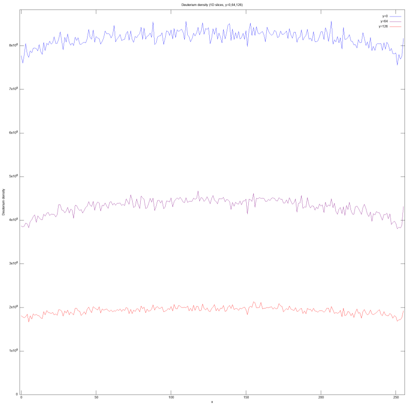


Figure: EIRENE

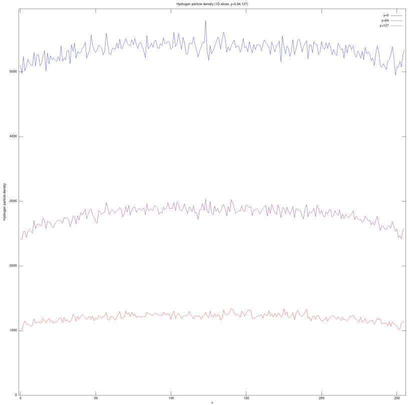


Figure: Eiron

Comparing EIRENE & Eiron with a contrived 2D slab case

Now let's see what happens when we keep the rates the same but switch to an isotropic angular distribution at the source.

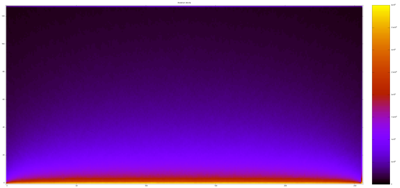


Figure: EIRENE

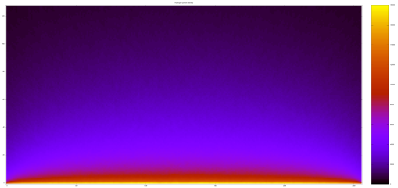


Figure: Eiron

Comparing EIRENE & Eiron with a contrived 2D slab case

Isotropic angular dist, slices in the x direction

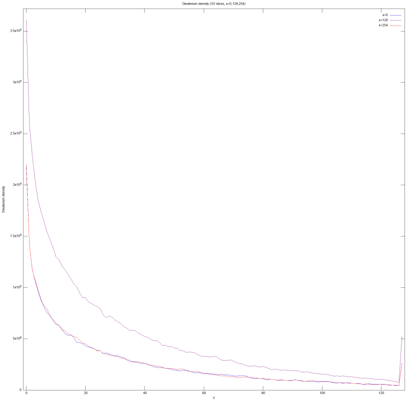


Figure: EIRENE

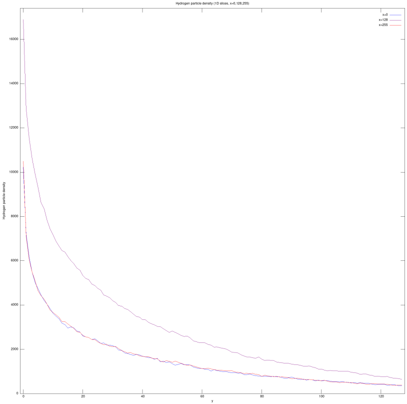


Figure: Eiron

Comparing EIRENE & Eiron with a contrived 2D slab case

Isotropic angular dist, slices in the y direction

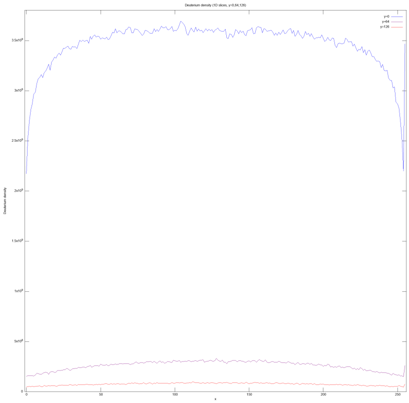


Figure: EIRENE

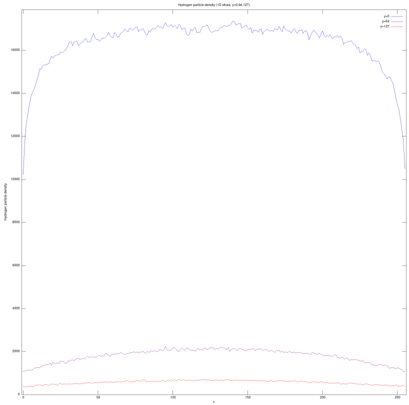


Figure: Eiron

Some data structures in Eiron

- Particle type enum:
  $\{ATOM|MOLECULE|TEST\_ION|PHOTON|ELECTRON|BULK\_ION\}$
- Chemical species properties:
  mass, ionization energy, charge, etc.
- Particle species:
  index{type, i}, species properties
- 2D plasma backgrounds:
  density, temperature, drift velocity
- Particles source:
  line segment, particle type
- Test particles:
  type, position, velocity, weight

Some data structures in Eiron

- Collision event type: enum designating collision resolution behavior $\{IONIZATION|ISOTROPIC\_SCATTERING|etc.\}$
- Collision rate model: enum designating a function $f : bgcell \to rate$
- Collision process:
  event type, incident bg spec., incident particle spec., new particle spec.
- Collision event:
  event type, updated particle
- Particle path:
  list of nodes (position, speed, weight)

Key operations

- Collision event resolution:
  particle, event_type, position → collision event
- Collision rate calculation:
  bg cell, rate model → rate
- Ray path discretization:
  ray → list of tuples (cell index, length)
- Line path discretization:
  line segment → list of tuples (cell index, length)

Some high-level components in Eiron

Actors

- Particle simulators

  task: Monte Carlo -simulation

  input: line sources, collision rate grids

  output: particle paths

- Tallier

  task: Scoring of tallies

  input: particle trajectory either as ray (on the fly)
  or line segments (tallying as a separate step)

  output: output tally grids

Message data

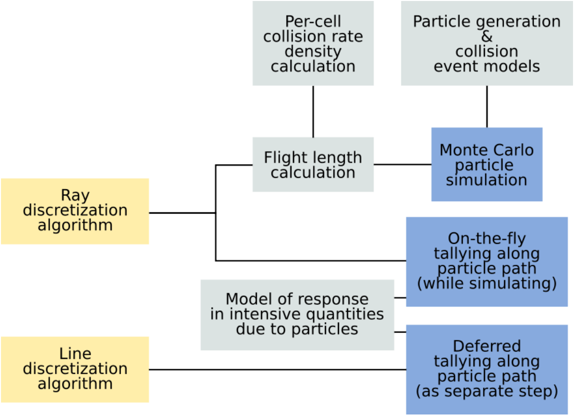
- Particle paths

  input and output message structure for the actors

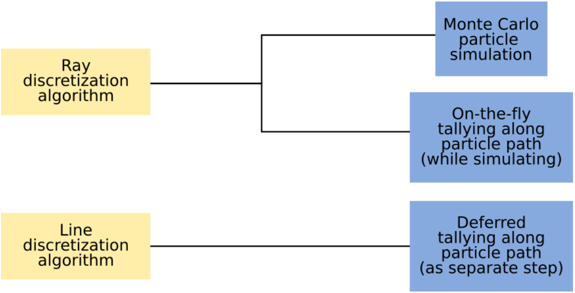
  a simulator produces them

  a tallier consumes them

Simplified structural view key components


Per-cell collision rate density calculation
Particle generation & collision event models
Flight length calculation
Monte Carlo particle simulation
Ray discretization algorithm
On-the-fly tallying along particle path (while simulating)
Model of response in intensive quantities due to particles
Line discretization algorithm
Deferred tallying along particle path (as separate step)

Simplified structural view key components



Ray discretization algorithm

Monte Carlo particle simulation

On-the-fly tallying along particle path (while simulating)

Line discretization algorithm

Deferred tallying along particle path (as separate step)

The components can be composed in many ways

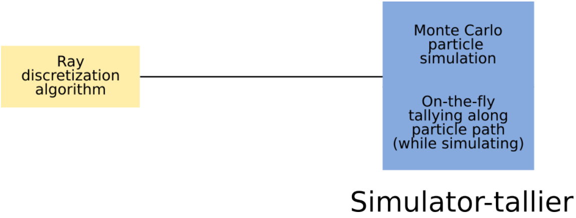

Ray discretization algorithm
Monte Carlo particle simulation
On-the-fly tallying along particle path (while simulating)
Simulator-tallier

Figure: Monolith

Parallelized using an OpenMP parallel for-loop over particles.

We've compared shared tallies with atomic adds and private tallies (each thread keeps its own tallies and they're reduced at the end).

The components can be composed in many ways

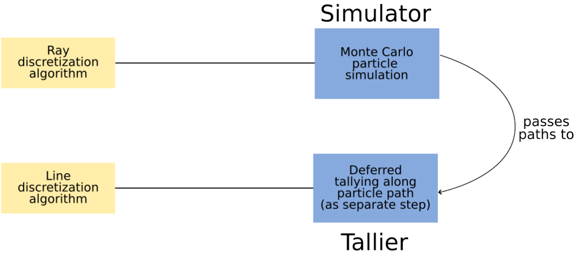

Simulator
Ray discretization algorithm
Monte Carlo particle simulation
passes paths to
Line discretization algorithm
Deferred tallying along particle path (as separate step)
Tallier

Figure: Deferred tallying

Parallelized using two separate OpenMP parallel for-loops over particles: one in the simulator, one in the tallier.

Again, we've compared shared tallies with atomic adds and private tallies (each thread keeps its own tallies and they're reduced at the end).

The components can be composed in many ways

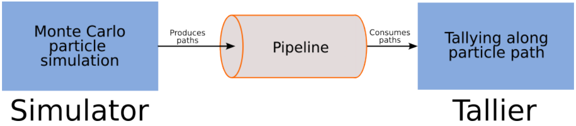


Figure: Pipeline execution

Parallelized using an OpenMP parallel for-loop over particles in simulator.

Parallelized using OpenMP tasks in tallier: one thread waits for paths, creates one tallying task per path.

The components can be composed in many ways

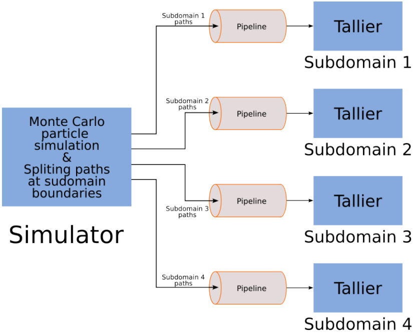


Figure: Pipeline with domain decomposed tallying

OpenMP-parallelism: same as last slide.

MPI-parallelism through domain decomposition.

The components can be composed in many ways

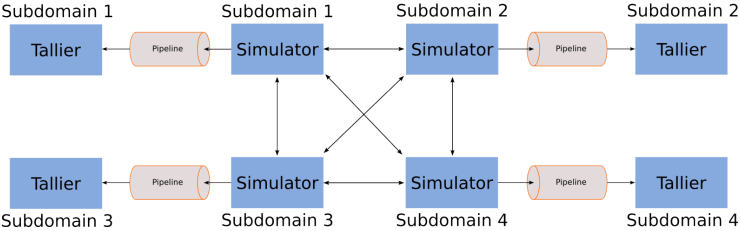


Figure: Domain-decomposed pipelines[1]

OpenMP-parallelism: same as before.

MPI-parallelism through domain decomposition and passing particle state between simulators.

[1] This setup has not been implemented yet

The components can be composed in many ways

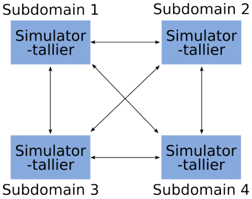


Figure: Domain-decomposed monolithic simulator-talliers[2]

OpenMP-parallelism: as with the monolithic setup.

MPI-parallelism through domain decomposition and passing particle state between simulators.

[2] This setup has not been implemented yet

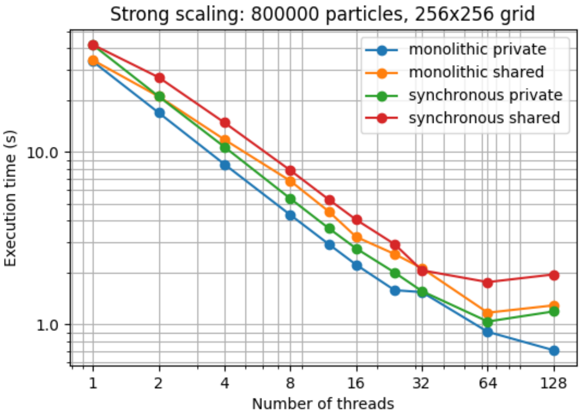
Benchmarks

We've run some benchmarks and tested the scalability of the implemented approaches.

The benchmarks were run with a simple setup: one test particle species, one scattering collision process and one ionizing collision process.

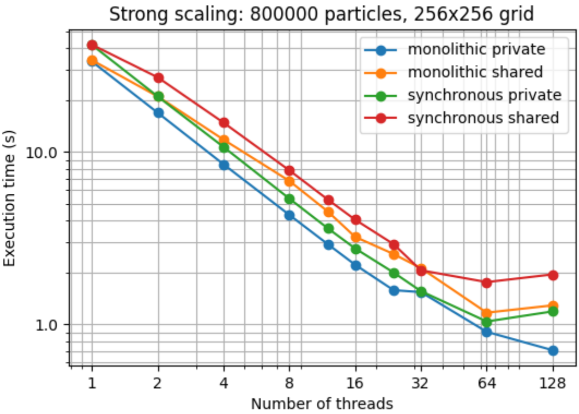
Remember that this is with a 2D structured grid, cell collision rates are precalculated, and no variances are being calculated.

Benchmark results and lessons learned



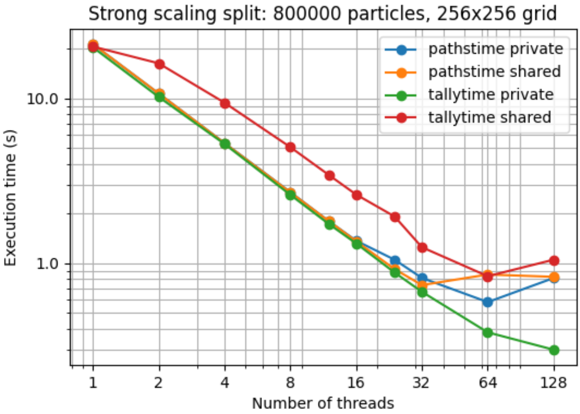
Strong scaling of the monolithic and deferred tallying designs, comparing shared tallies and atomic adds with private tallies and reduction (deferred tallying labeled synchronous here).

Benchmark results and lessons learned



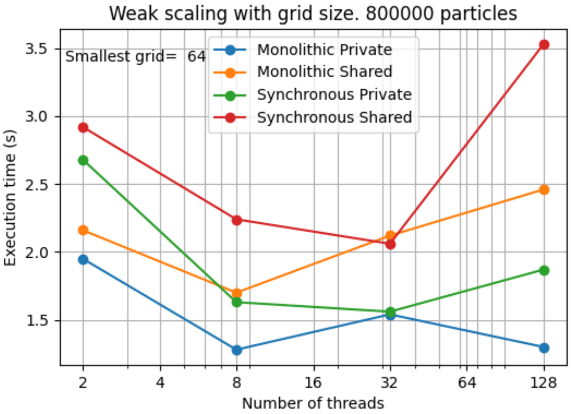
We see that using a shared tally grid is consistently slower by a constant factor up to 16 threads. The results after that also look bad for the shared tally grid approach, but this is a strong scaling case with a small grid, we can't read too much into the very end.

Results and lessons learned



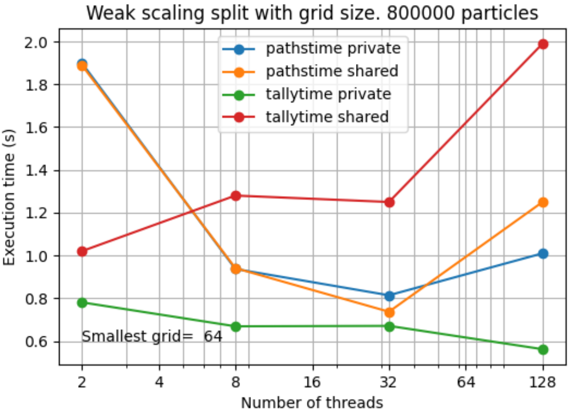
If we separate the time it takes to simulate from the time it takes to tally in the deferred case, we see that it is indeed tallying on a shared grid that's slow.

Results and lessons learned



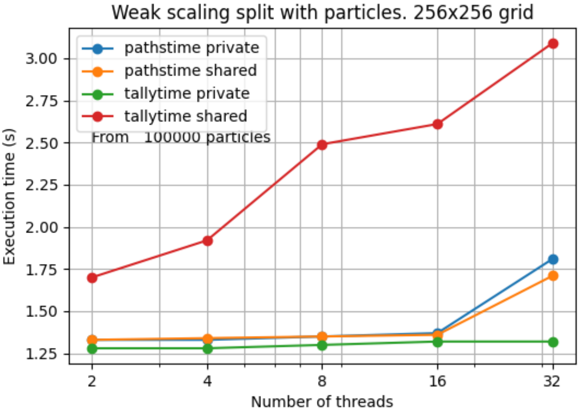
This weak scaling plot tells us that the shared grid implementations do indeed get slower as a function of threads, even if we try to keep the rate of synchronization low by increasing the grid size

Results and lessons learned



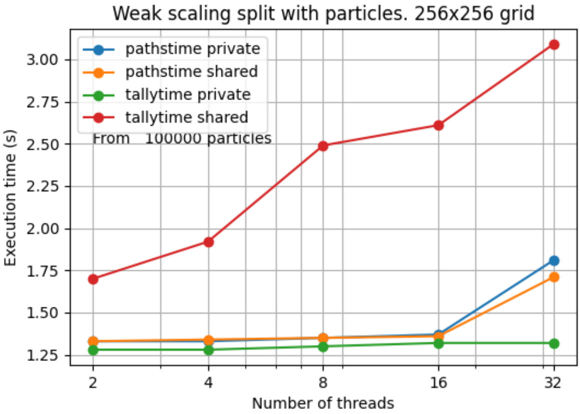
Focus on the red and green lines here, and the pattern becomes even more clear: the cost of synchronizing on a shared resource will increase with the number of threads.

Results and lessons learned



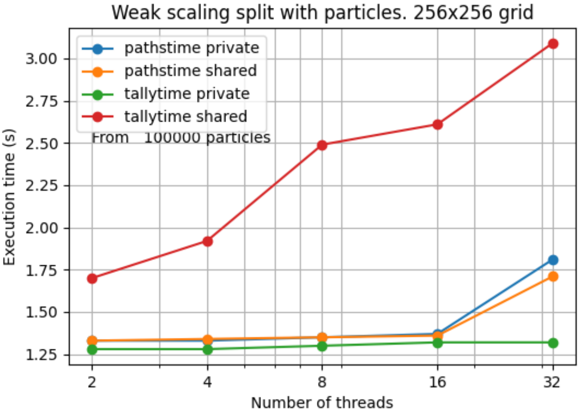
The effect is most clear when we scale the number of threads on a small grid and keep the particle count per thread constant. With an increase in the number of threads hitting a constant number of cache lines, the threads need to synchronize at an increasing rate.

Results and lessons learned



Using a private grid, threads will get the same number of particles and the same size grid at every scale in this plot, so runtime is expected to be constant.

Results and lessons learned



This is a classic trade-off between time and memory.
Shared grid → mem. use constant, time goes up with part. count.
Private grid → mem. use = O(n), time is constant.

Project status

Current tasks being worked on:

- Passing the simulation state of a particle from one simulator to another
- Managing particle generation and load balancing when the task of simulating one particle is shared by multiple simulators

Thank you

Questions?

Appendix: Storing particle paths to tally later

Using deferred tallying, we could output particle histories as point paths instead of the full tallies. Let's see what we could do with that.

- The user first produces one set of tallies using a separate tallying program, and then suddenly realizes that they need another tally that they did not specified, they call the same program again to produce it without having to simulate any of the particle dynamics or tallying anything else
- The user wants to produce a time evolved animation of the system. They write a script that cuts out a slice of a particle history between a start and end time. They use this script to produce new particle files corresponding to frames in the movie and runs a tallier program to produce tally files corresponding to the frames (or their deltas) and then visualizes the tally to render each frame

Appendix: deterministic random numbers

In order to get the same random numbers every time, regardless of the thread running the simulation and at which point of the particles history the thread started processing the particle, we create a new random number stream per particle (reseed at every particle). From this sequence of random number streams we can deterministically get the same random number from any thread with a two-part index: the random number stream id and the offset within the stream.

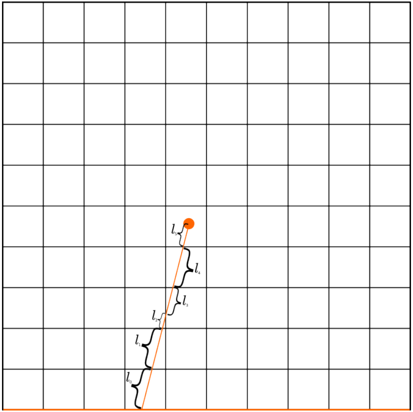
A thread will set its own random number stream position before it starts simulating a particle. When a thread serially moves to the next particle, it increments the stream id and sets the offset to 0. In OpenMP loops, the particle id space is partitioned between the threads, and they all set their streams accordingly. When particle simulation is distributed to multiple processes, the stream position is communicated along with the particle.

Appendix: deterministic random numbers

There are two important considerations when choosing the random number generator in this case:

1. The cost of reseeding must be low, this disqualifies e.g. the Mersenne Twister. Reseeding with a Mersenne Twister has a 35% overhead, because the initialization of the internal state array is a heavy operation (the state array is a few KB). Instead I'm using PCG, which is roughly 300 times quicker on my laptop.
2. The internal state of the PRNG should have a large enough domain, a 32 bit internal state is probably too small. The birthday collision probability for 32-bit numbers = 50% at a population of 77163 particles. For 64-bit values, the collision probability is 50% at 5 billion particles (short scale billions). And of course, with 32-bit numbers you can only have 2^{32} unique particle histories (~ 4 billion).

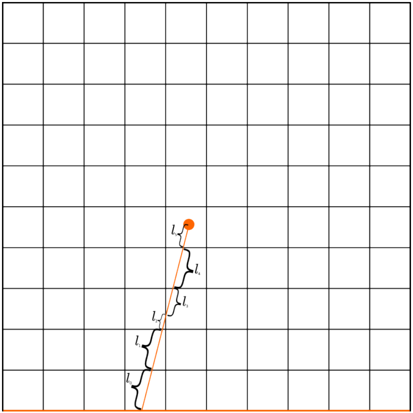
Appendix: Path discretization



$$\int_0^L f \; ds = \sum_{k \in grid} f_k * l_k$$

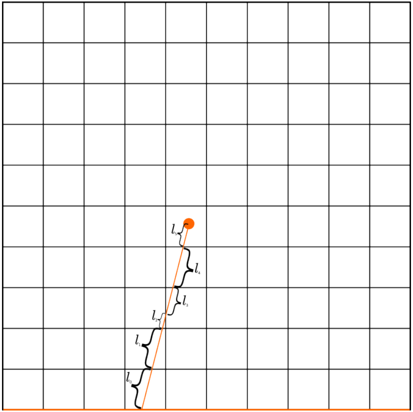
Let's go back to that integral. We're on a grid, so the integral is the sum of products of collision density and path length within each cell.

Appendix: Path discretization



In order to calculate the sum, we must find the length l_k within each cell k, I call this the particle path discretization problem. We also need the lengths for tallying.

Appendix: Path discretization



The problem is similar, but not equivalent, to line rasterization in computer graphics. We want to transform a line (p_0, p_1) or a ray $(p_0, \vec{v})$ to a list of tuples (cell_index, length).

Appendix: Path discretization

There are two separate path discretization methods:

- Ray discretization, when we're simulating and only have a starting point and direction
- Line segment discretization, when the path has been fixed, can be useful in deferred tallying

It's easy to conceive of a naive algorithm, let's say for a line segment:

```
1. Use a transformed space where the line segment's starting point < end point for both x and y
2. Set p = the starting point
3. While true:
    | Set cell = the cell surrounding p
    | Check for an intersection with the right and upper cell boundaries
    | if there is no intersection, calculate the length for cell as |end - p|, break loop
    | calculate the length for cell as |intersection - p|
    | Set p = intersection
```

We're using a faster but more convoluted algorithm for the 2D case, but this one generalizes nicely to 3D structured grids.

Appendix: Monolith vs deferred tallying solver

100000 particles, 2048^2 grid

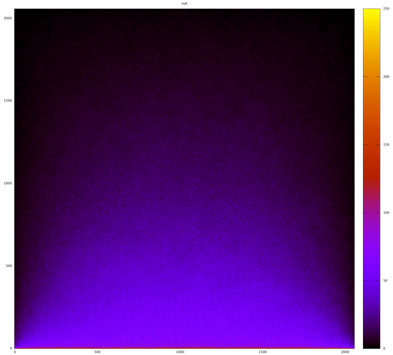


Figure: Monolith

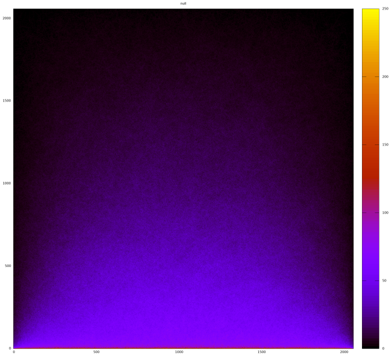


Figure: Deferred tallying

Appendix: Monolith vs deferred tallying solver

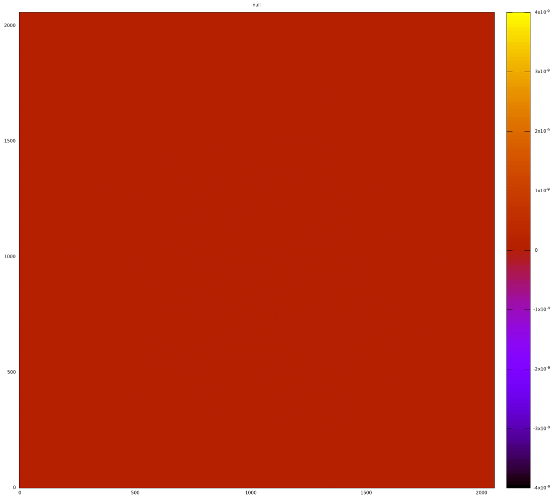


Figure: Diff of the two results, largest errors on the order of 10^{-9}

Appendix: Monolith vs pipeline solver

100000 particles, 256^2 grid

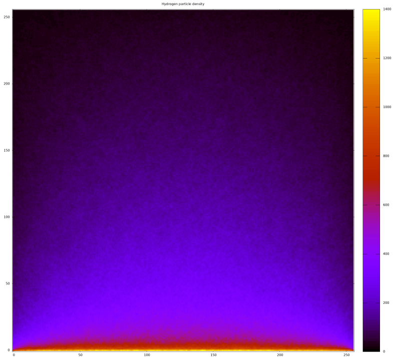


Figure: Monolith

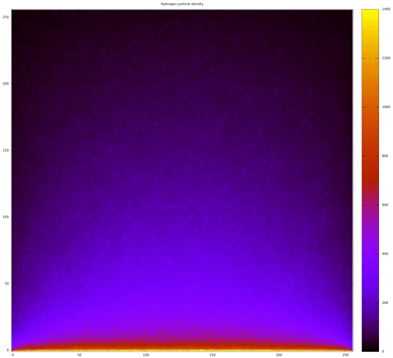


Figure: Pipeline: 1 sim, 2 talliers

Appendix: Monolith vs pipeline solver

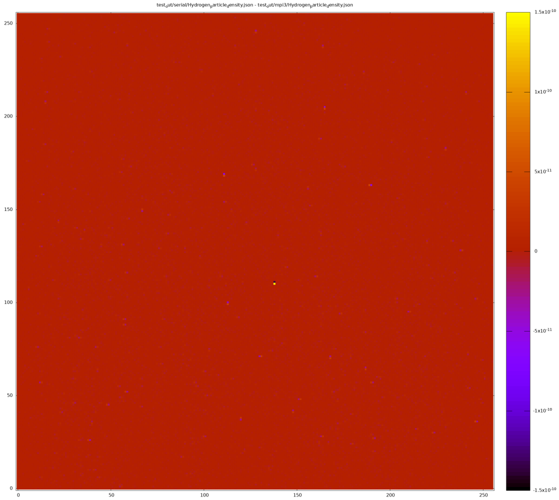


Figure: Diff of the two results, largest errors on the order of 10^{-10}