Eiron: A toy model of EIRENE for performance studies

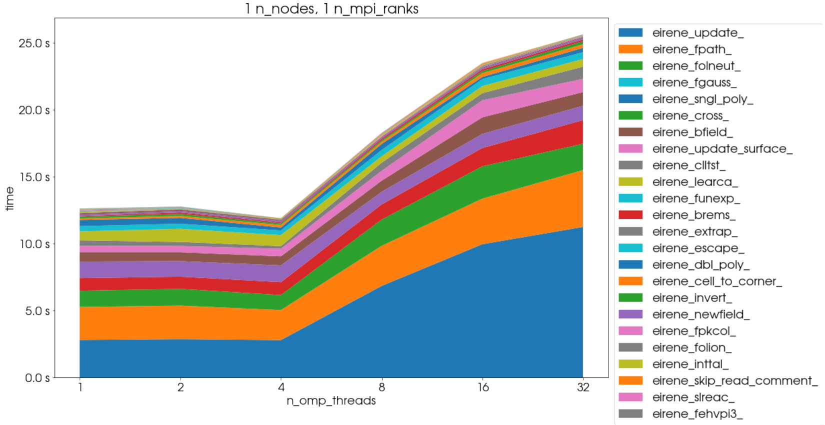
Oskar Lappi

November 9, 2021

Problem

- EIRENEs core computational loop was originally designed to be serial
- Existing MPI-parallelization exhibits good runtime scalability, but memory usage also scales linearly with number of processes
- Existing OpenMP-parallelization exhibits poor scalability. Scales to ca. 4 threads, after that scaling is negative: more threads $\implies$ longer runtime.

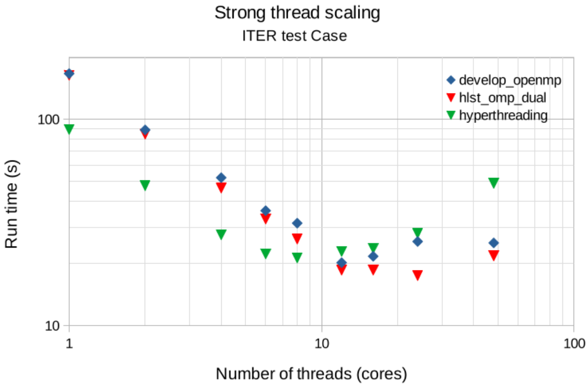
OpenMP scalability



2D-D slab sample, develop_openmp branch, -DOPENMP=ON

NOTE: x-axis log-scale, y-axis linear scale

OpenMP scalability, Huw Leggate



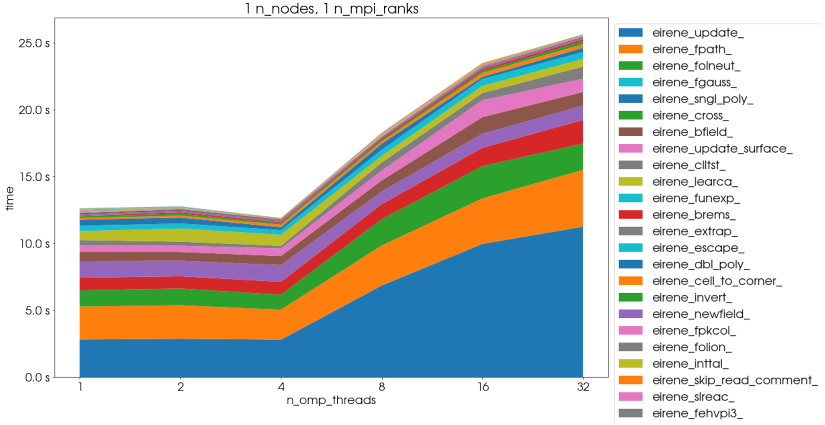
ITER test case, develop_openmp branch
NOTE: log-log plot

OpenMP scalability

Measurements are not directly comparable, we used different test cases and machines. Haven't found Huw's old reports of the 2D-D slab case

E.g. Mahti, the machine I used, uses AMD Rome CPUs, which have a core complex with 4 cores that share a cache. This implies the bottleneck for the 2D-D slab case is main memory access on Mahti.

Where is the bottleneck



- Most of the time in EIRENE is spent writing responses in *eirene_update*
- Next most time in *eirene_fpath*

Plan

- Create a toy model with the same overarching computational structure as EIRENE but without requirements for physical correctness
- Using the toy model, produce a parallelization strategy that scales
- The design can then be transferred to EIRENE

Eiron: the toy model

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

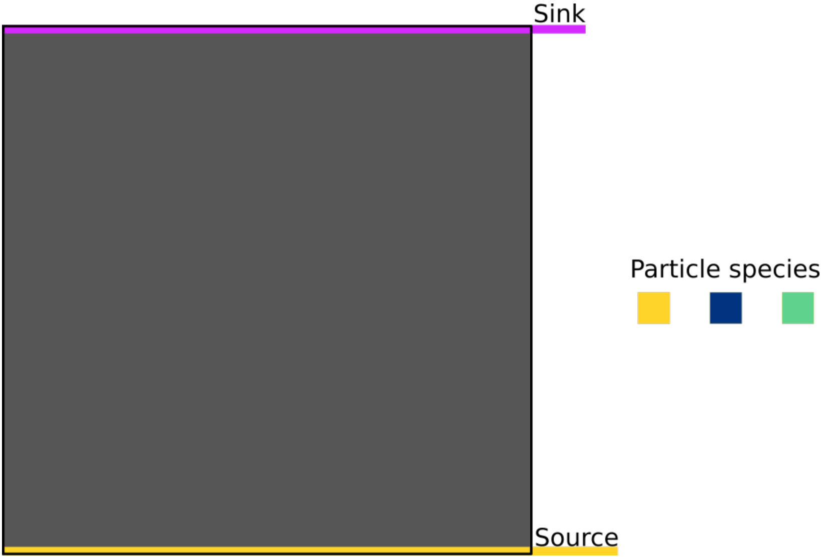
- Athenian goddess of peace

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

- Athenian comedy character

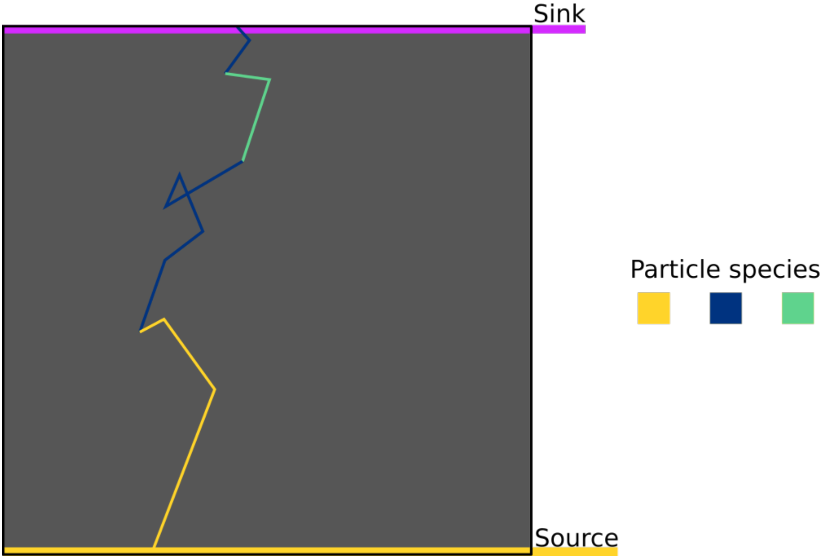
Eiron

Scores particles traveling in polyline paths on a 2D Cartesian grid



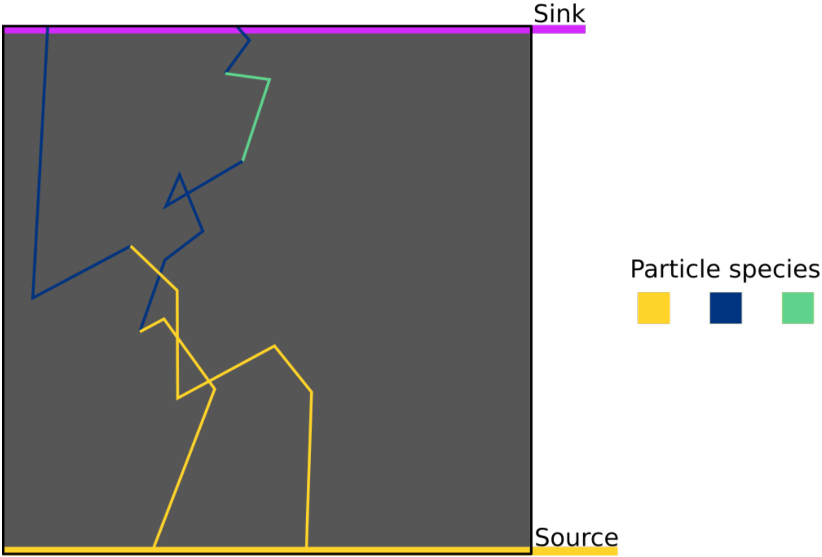
Eiron

Scores particles traveling in polyline paths on a 2D Cartesian grid



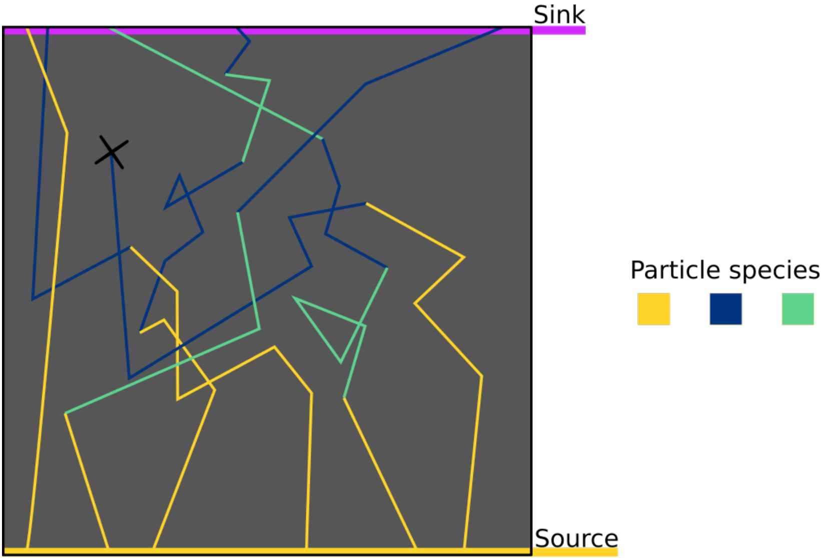
Eiron

Scores particles traveling in polyline paths on a 2D Cartesian grid



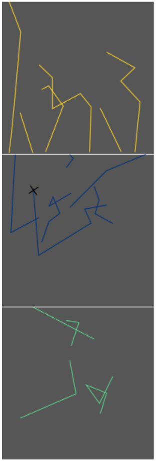
Eiron

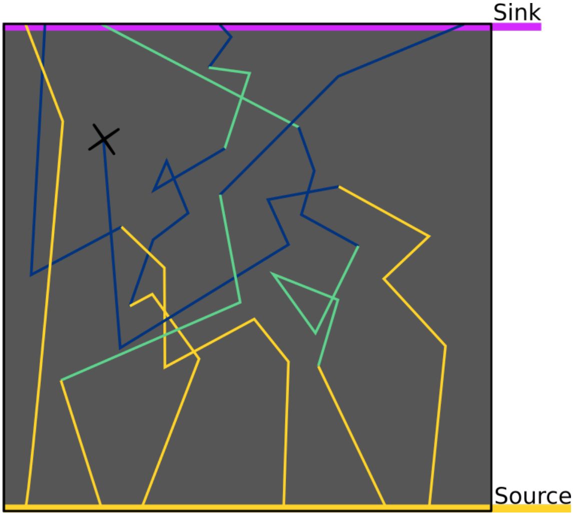
Scores particles traveling in polyline paths on a 2D Cartesian grid



Eiron

Scored tallies written to separate memory buffers per species (and tally)





Eiron

Requirements

- Record multiple quantities to a cartesian grid consisting of the combined contribution of many polyline particle paths
- Grid is indexed by (in rising order of stride): subspecies, x, y species, tally
- Particles may change species along their path

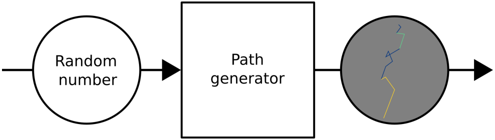
Main components

- Path generator
- Response tallier

The project is not at a state where I can show any concrete measurements, however we can already use these component definitions to explore and analyze different designs.

Component: Path generator

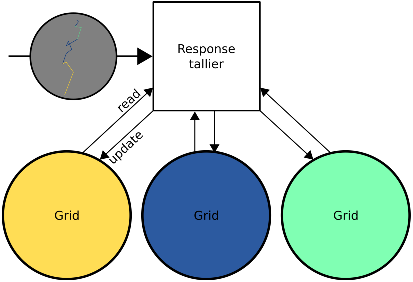
- Generates particles from line sources
- Follows the particle, generating collisions with background particles



Squares	components (procedures/actors)
Circles	state
Arrows	data flow

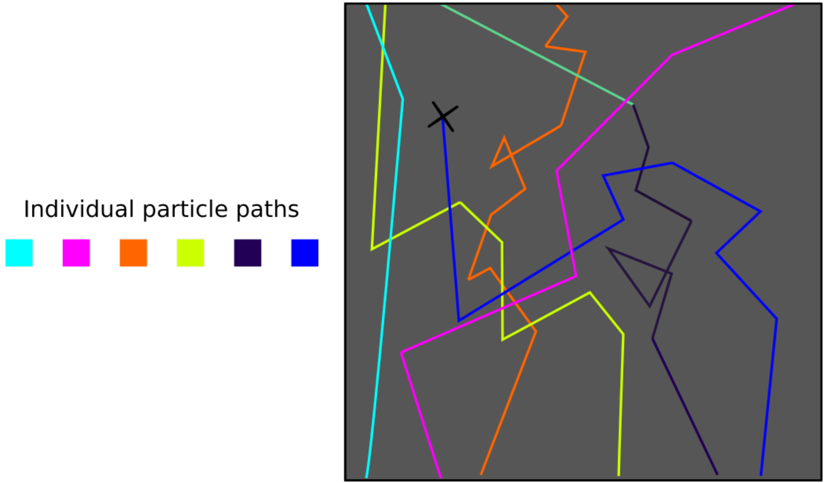
Component: Tallier

- Calculates contribution of particle-background interactions to each cell
- Writes contributions to grid. $grid[i] \leftarrow grid[i] + x$

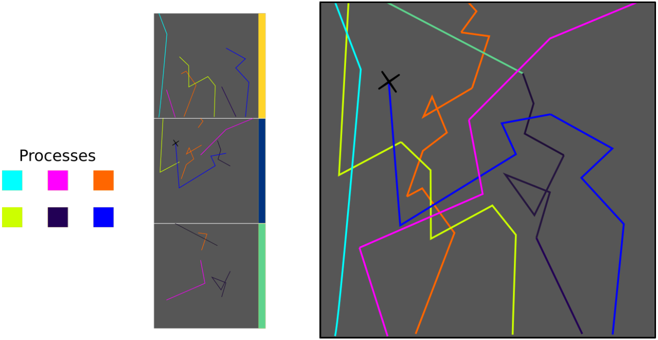


Squares	components (procedures/actors)
Circles	state
Arrows	data flow

Existing parallelization strategy, by particle

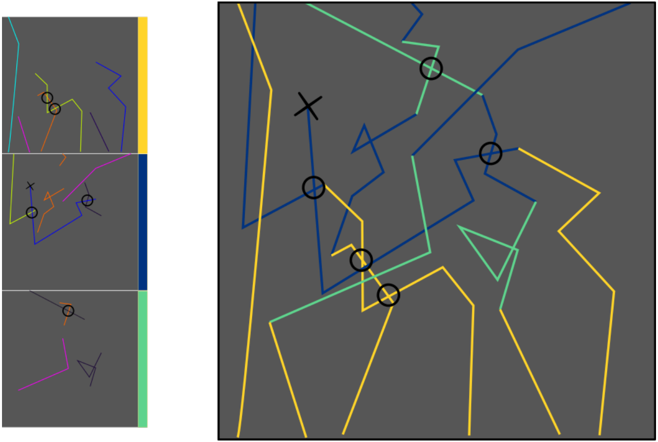


Existing parallelization strategy, by particle



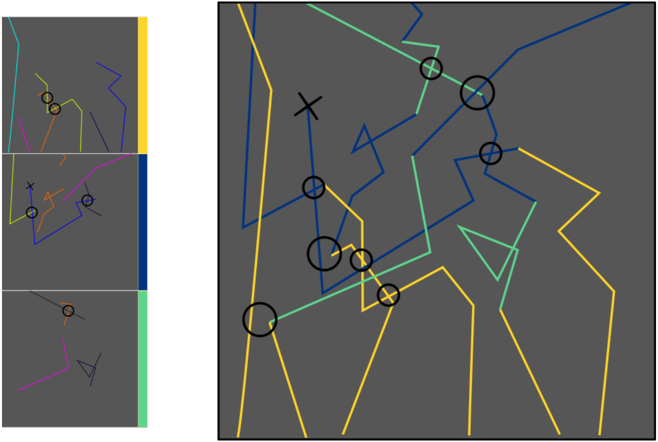
Issue: processes write to the same memory regions. They share mutable state.

Existing parallelization strategy, by particle



Intersecting paths $\implies$ data dependence $\implies$ synchronization necessary for correctness

Existing parallelization strategy, by particle

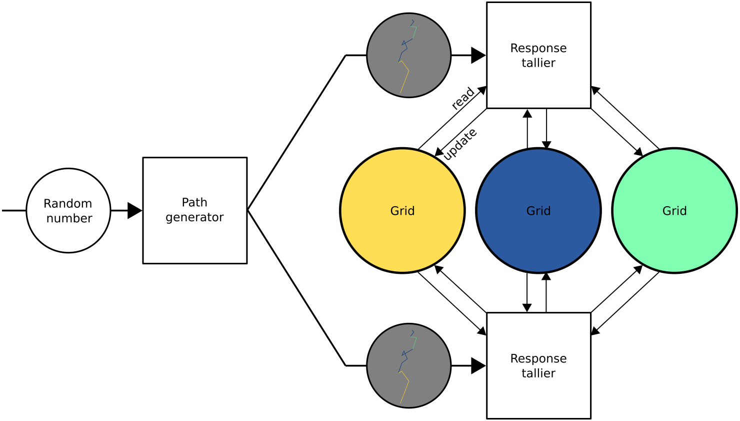


Paths map to same cache line $\implies$ false sharing $\implies$ cache invalidation

Existing parallelization strategy, by particle

Data-flow: shared mutable state

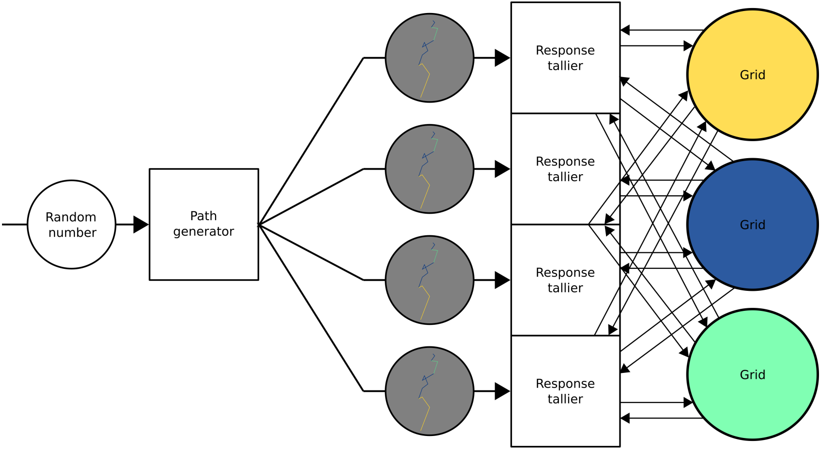
If we parallelize the tallying by particles, talliers will need concurrent access to the same memory resource. This concurrent access requires synchronization and causes cache invalidation.



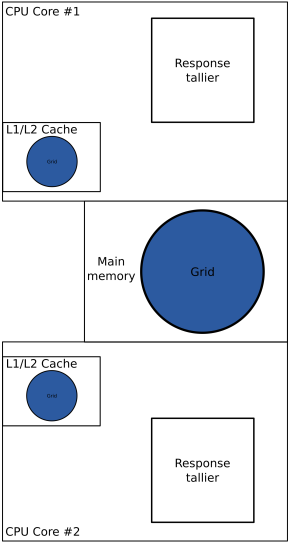
Existing parallelization strategy, by particle

Challenge: shared mutable state

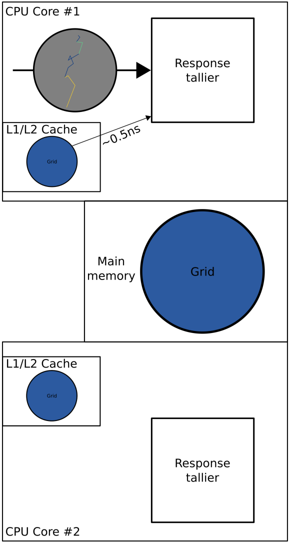
Adding more processes that write tallies increases the probability that two processes share a cache line at any moment in time $\implies$ the cost of synchronization increases with the number of threads.



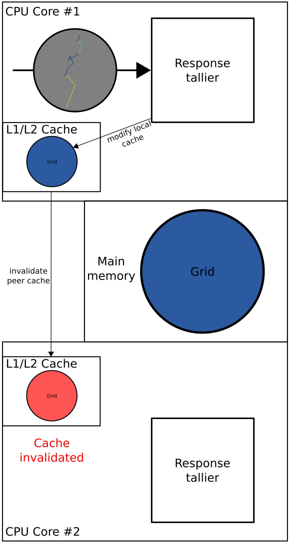
Shared mutable state, hardware view



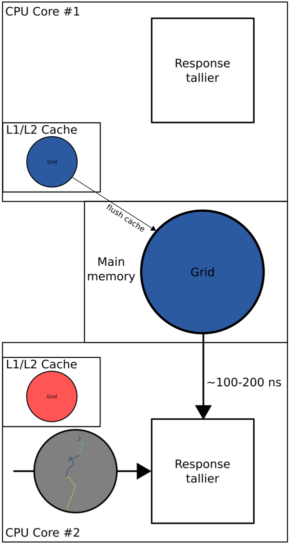
Shared mutable state, hardware view



Shared mutable state, hardware view

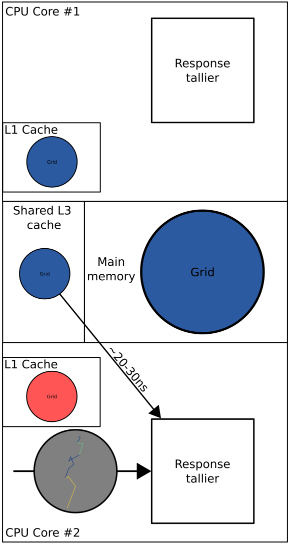


Shared mutable state, hardware view



200X slower than best case scenario

Shared mutable state, hardware view



40X slower than best case scenario

Existing parallelization strategy, by particle

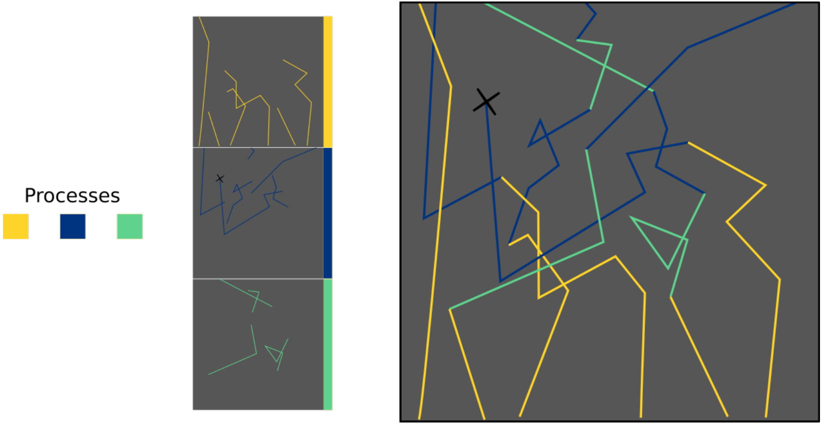
Summary

- Access to the grid is the bottleneck
- If the bottleneck has negative scaling, so does the entire system
- Need to find a strategy that does not produce negatively scaling bottlenecks

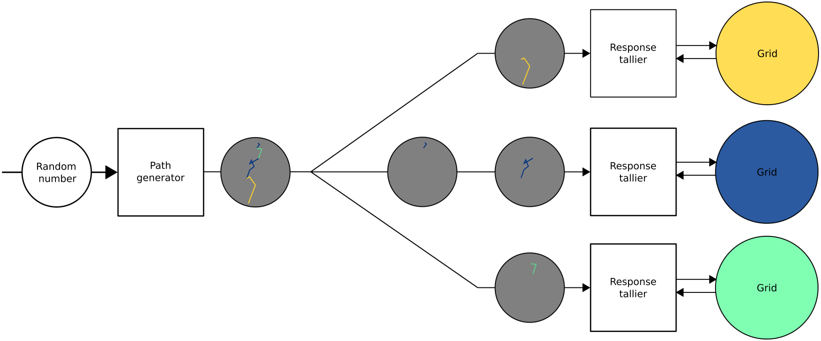
Alternative parallelization strategy, by memory region

Particles have data dependencies, but luckily, all data dependencies are local to a grid cell $\implies$ operations on disjoint regions of memory are independent.

E.g. every cell with the same species and tally index is an independent region of memory.

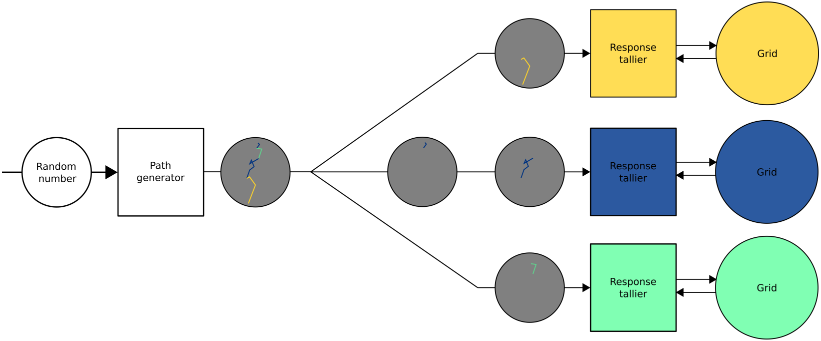


Alternative parallelization strategy, by memory region



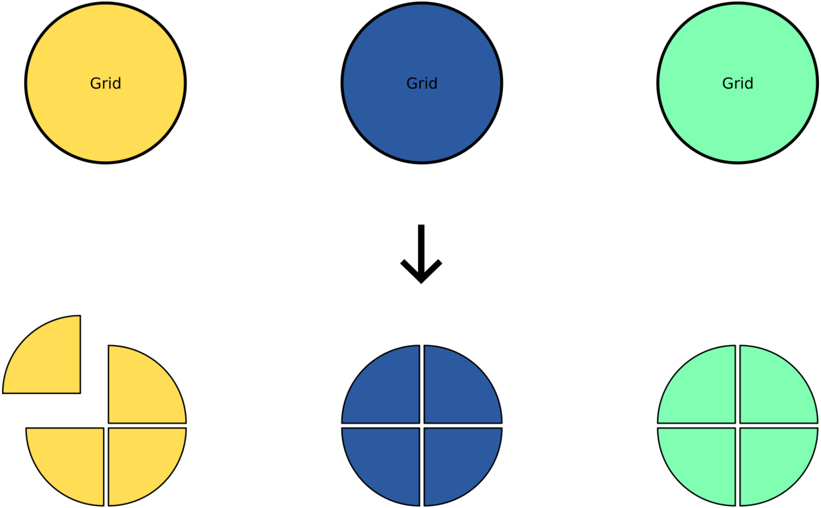
Idea: split particle path into sub-paths by species

Alternative parallelization strategy, by memory region



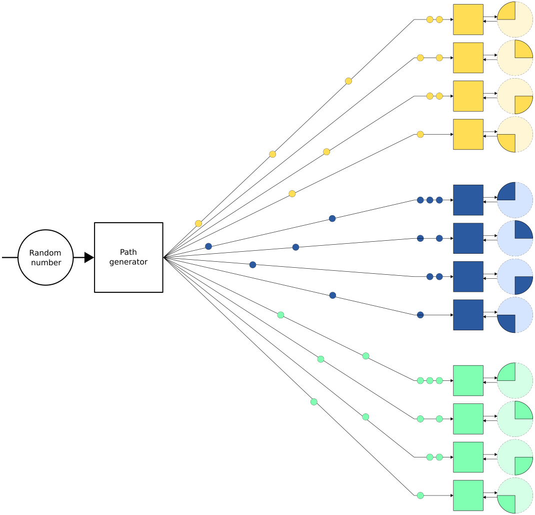
Idea: split particle path into sub-paths by species

Alternative parallelization strategy, by memory region



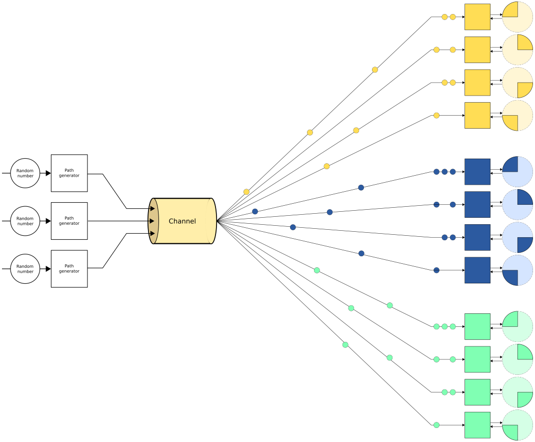
Add in domain decomposition

Alternative parallelization strategy, by memory region



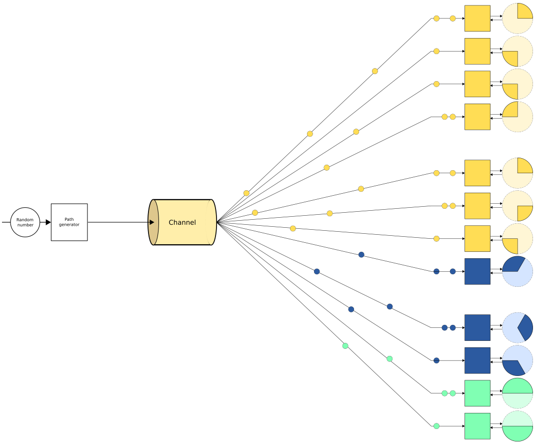
Idea: split particle path into sub-paths by species *and subdomain*

Alternative parallelization strategy, by memory region



If we conceptualize data flow from path generation to tallying in the form of a channel, we can scale up whichever of the two stages is the bottleneck

Alternative parallelization strategy, by memory region



We can also load balance proportional to particle species predominance. Either by adaptive grid decomposition or by duplicating busy memory regions.

Alternative parallelization strategy, by memory region

Summary

- Data dependencies are local $\implies$ can organize tallying into independent tasks, one process per task
  - workloads can be balanced by varying the size of the task or by assigning multiple processes to tasks
  - suitable for dynamic load balancing (more processes can be spawned)
- Polylines fit into relatively small memory buffers
  - suitable for network transmission
  - this architecture can be used to create a distributed program
  - MPI is a better fit for this design than OpenMP

Alternative parallelization strategy, by memory region

- The general concepts which inform the design are well understood and similar designs have a proven track record
- Result of a performance comparison can be predicted to favor this design over the old one almost certainly
- However, the hardest part of a design is the implementation. By implementing this design in Eiron first, we will experience the challenges of implementing it first-hand. The solutions to these challenges in the Eiron implementation can then be used to guide the implementation of the design in Eirene.

Eiron, design considerations

- A modular design is important, so that different axes of parallelization can be explored
- Creating a library allows for multiple binaries to be created, which allows for more flexibility in the architecture of the system
- Both test-suites and benchmark-suites should be created, to allow for easy comparison of designs
  - APIs need to be designed with testing and performance measurements in mind

Eiron, project status

What's been done

- A modular design of logical components that compose nicely
- A functioning serial tallier
- A functioning particle sources (generates starting points of particle paths)
- Particle modeling in progress
- Collision probability modeling in progress
- Collision result modeling in progress

What's yet to be started

- tally modeling
- line integrals
- decomposition along species/tallies
- domain decomposition
- work scheduling/coordination *much work here*

Thank you

Questions?