SWIFT:

Task-based Calculation + Task-based MPI + Task-based I/O = Maximal Performance

Dr. Matthieu Schaller (on behalf of the SWIFT team)
11/11/2017

Intel® HPC Developer Conference 2017
Denver, Colorado, US

Durham University

This work is a collaboration between two departments at Durham University (UK):

- The Institute for Computational Cosmology,
- The School of Engineering and Computing Sciences,

with contributions from the astronomy group at the university of Ghent (Belgium), St-Andrews (UK), Lausanne (Switzerland)
as well as the DiRAC and STFC software team at Hartree (UK).

This research is partly funded by an Intel IPCC since January 2015.

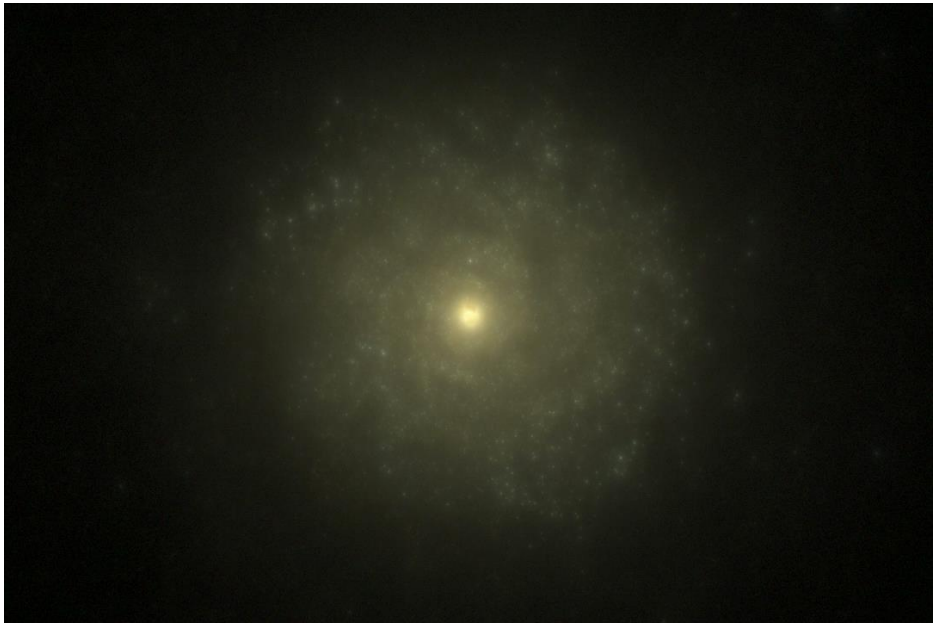
Durham University

Introduction

Durham University

What we do and how we do it

- Cosmological simulations of the formation of galaxies.
- EAGLE project: 48 days of computing on 4096 cores, most cited astronomy paper of 2015.
- Simulate gravity and hydrodynamics with particles.
- Large dynamic range and >2M time-steps.



HPC Challenges in 2017

- Exploit all 3 levels of parallelism (MPI, threads, vector instructions).
- Decrease the CPI as much as possible (high FLOP / Byte read).
- Hide inter-node communications.
- Fast i/o on parallel file systems with many other users.

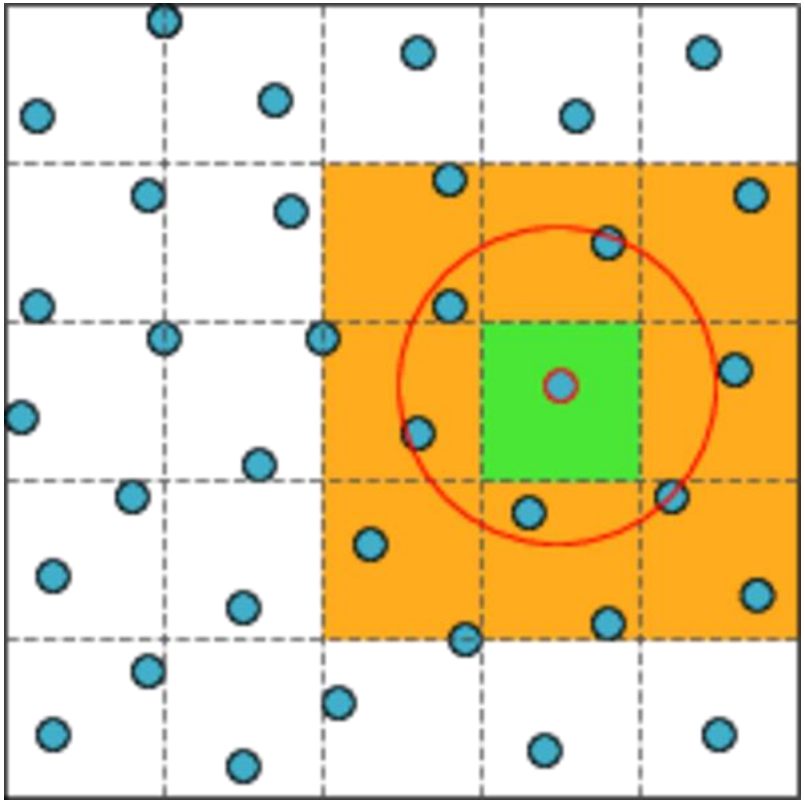
Challenges of our simulations

- Particles are unstructured in space.
  → No simple vectorizable pattern.
- Particles will move and the neighbours they interact with will change at every time-step.
  → No pre-computed list or mesh that can be re-used over and over.
- Particles have individual time-steps.
  → Hard to load balance steps with only a handful of particles.
- Interactions are computationally cheap.
  → Likely dominated by memory bandwidth.

SWIFT particle hydrodynamics

Lesson 1: Make things predictable.

- Neighbour search done using a grid with ~500 particles per cell.
- Cell size designed to match search radius.
- Particles only interact with same cell or 26 neighbouring cells.
- Can now list all interactions!



SWIFT particle hydrodynamics

```
for (int ci=0; ci < nr_cells; ++ci) {    // loop over all cells (>1 000 000)
   for(int cj=0; cj < 27; ++cj) {        // loop over all 27 cells neighbouring
cell ci

      const int count_i = cells[ci].count;  // Around 400-500
      const int count_j = cells[cj].count;

      for(int i = 0; i < count_i; ++i) {
         for(int j = 0; j < count_j; ++j) {

            struct part *pi = &parts[i];
            struct part *pj = &parts[j];

           INTERACT(pi, pj);   // symmetric interaction (low FLOPS)
} } } }
```

SWIFT particle hydrodynamics

```
for (int ci=0; ci < nr_cells; ++ci) {
   for(int cj=0; cj < 27; ++cj) {
```

Threads + MPI

```
      const int count_i = cells[ci].count;  // Around 400-500
      const int count_j = cells[cj].count;

      for(int i = 0; i < count_i; ++i) {
         for(int j = 0; j < count_j; ++j) {

            struct part *pi = &parts[i];
            struct part *pj = &parts[j];

           INTERACT(pi, pj);   // symmetric interaction (low FLOPS)
} } } }
```

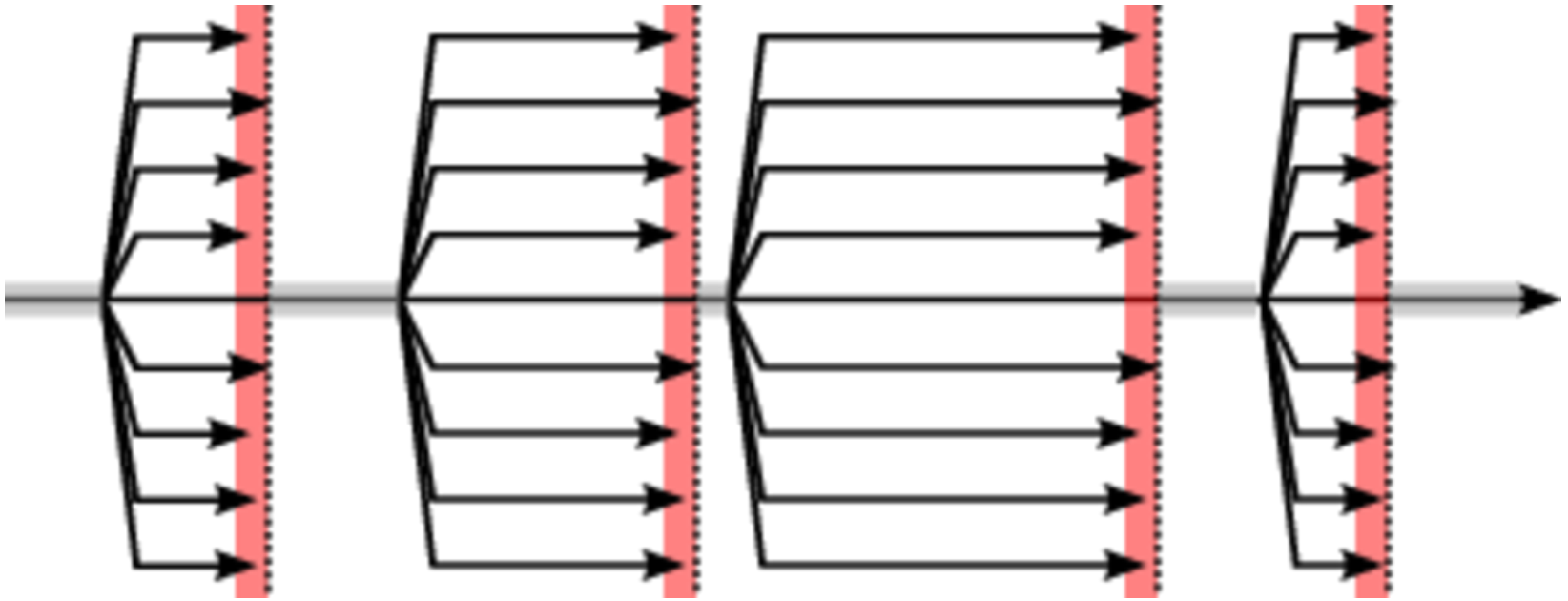
Vectorization

Single-node parallelisation

Durham University

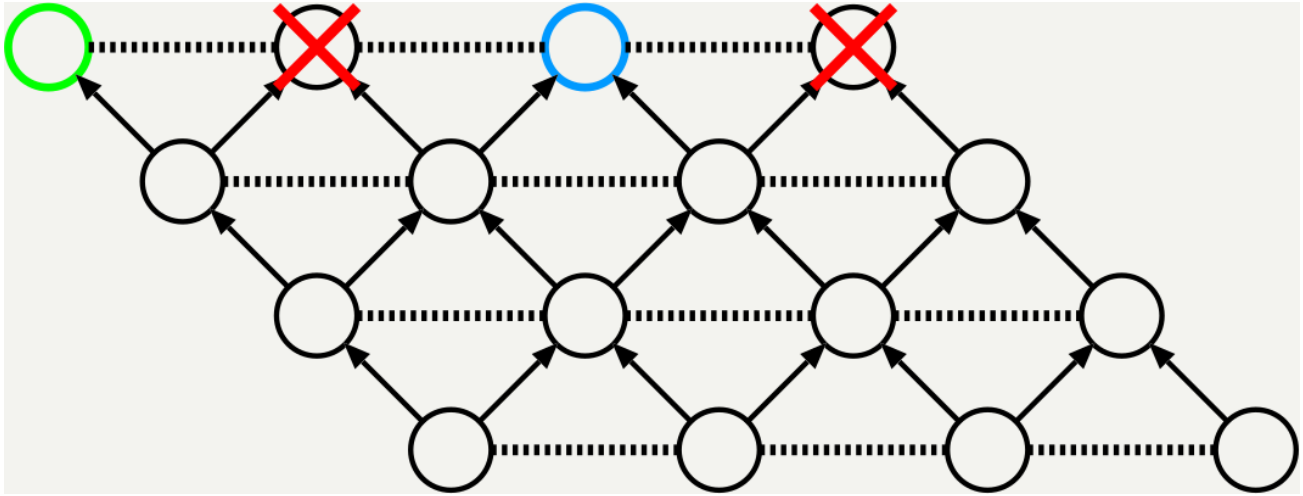
Traditional parallelisation pattern

"fork + join" model



Task-based parallelism

"Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing."



We use our own (problem agnostic !) Open-source library *QuickShed*.

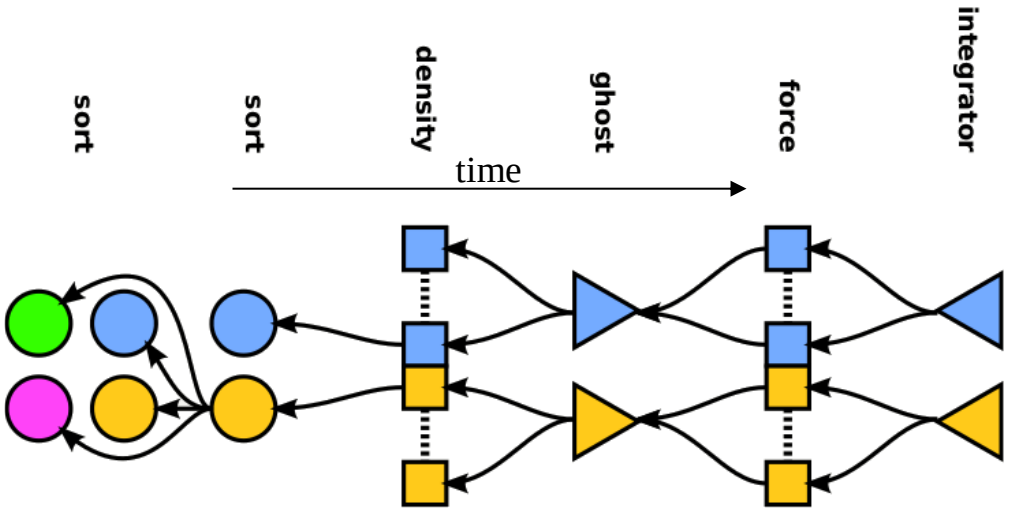
arxiv:1601.05384

Task-based parallelism

- Many libraries out there:
  - HPX
  - Charm++
  - HiHat
  - OmpSS
  - …
- Quickshed is a simple (3000 lines) C library which handles conflicts and dependencies, MPI communications and accelerators offloading.
- arxiv:1601.05384

Task-based parallelism

- Requires to re-write the software and not “just” add pragmas.
- Allows for simple (application scientist friendly!) code within tasks.
- Express the underlying algorithm as a graph.



Single node parallel performance

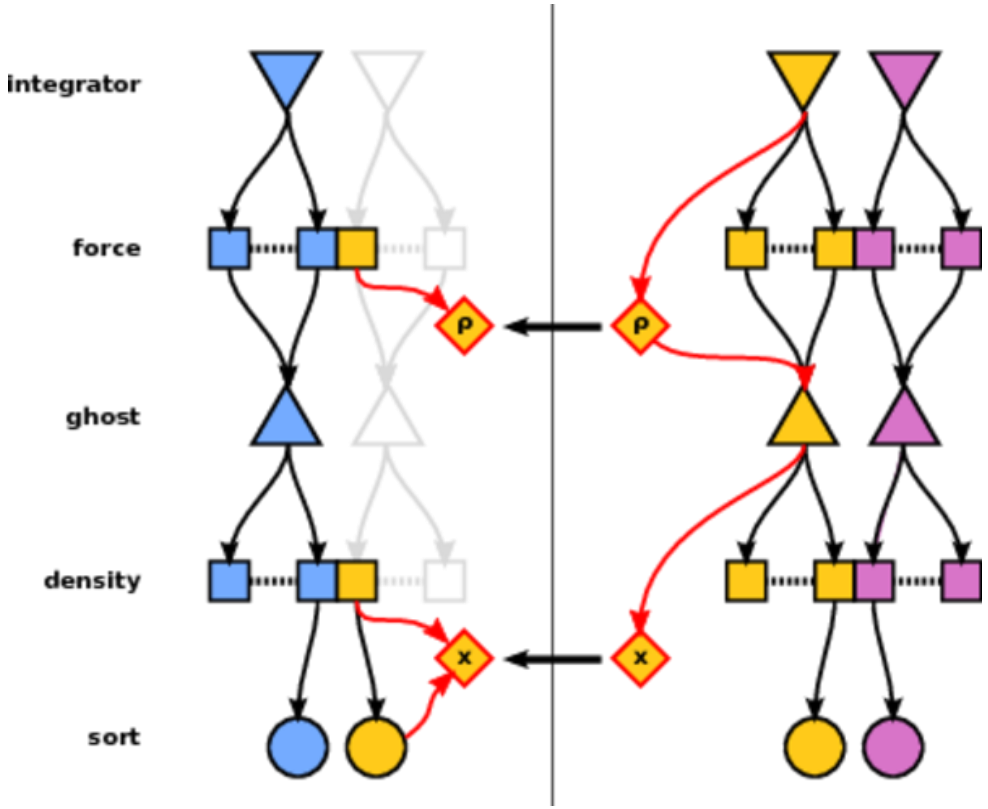
Almost perfect load-balancing on 16 cores.

Multi-node parallelism

Multi-node parallelism

- Can’t afford the usual “send then compute” pattern.
- Need the same flexibility as on one rank as tasks are executed in random order.
- Send small parts of the data within the tasking system.
  → Requires library that correctly implements `MPI_THREAD_MULTIPLE`.
  → Many experienced users and textbooks advise against this.
- If problem big enough, no need to care about latency or bandwidth!
- Very efficient use of the machine (network and FPUs at the same time!).

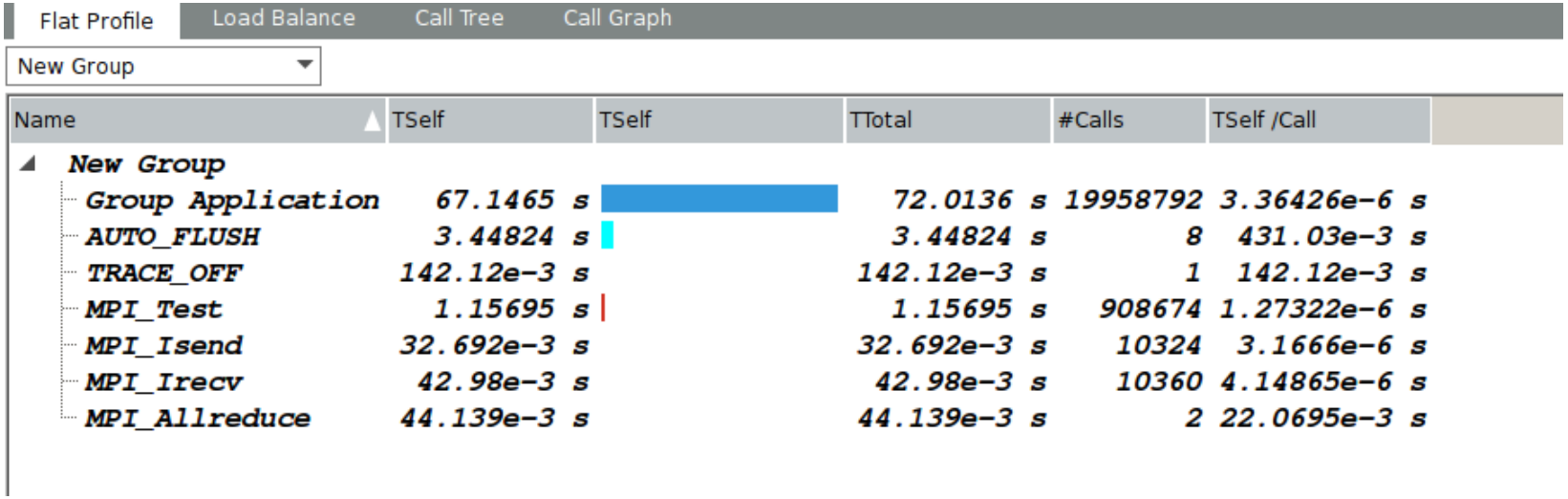
Multi-node parallelism



Some details

- Communication tasks do not perform any computation:
  - Call `MPI_Issend()` and `MPI_Irecv()` when enqueued.
  - Check communications have arrived with `MPI_Test()`.
- Maximal performance: increase `I_MPI_EAGER_THRESHOLD` to allow for most messages to be sent directly.
- Start the purely local tasks first, then the ones that depend on MPI data.

Multi-node parallelism



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
New Group					
Group Application	67.1465 s		72.0136 s	19958792	3.36426e-6 s
AUTO_FLUSH	3.44824 s		3.44824 s	8	431.03e-3 s
TRACE_OFF	142.12e-3 s		142.12e-3 s	1	142.12e-3 s
MPI_Test	1.15695 s		1.15695 s	908674	1.27322e-6 s
MPI_Isend	32.692e-3 s		32.692e-3 s	10324	3.1666e-6 s
MPI_Irecv	42.98e-3 s		42.98e-3 s	10360	4.14865e-6 s
MPI_Allreduce	44.139e-3 s		44.139e-3 s	2	22.0695e-3 s

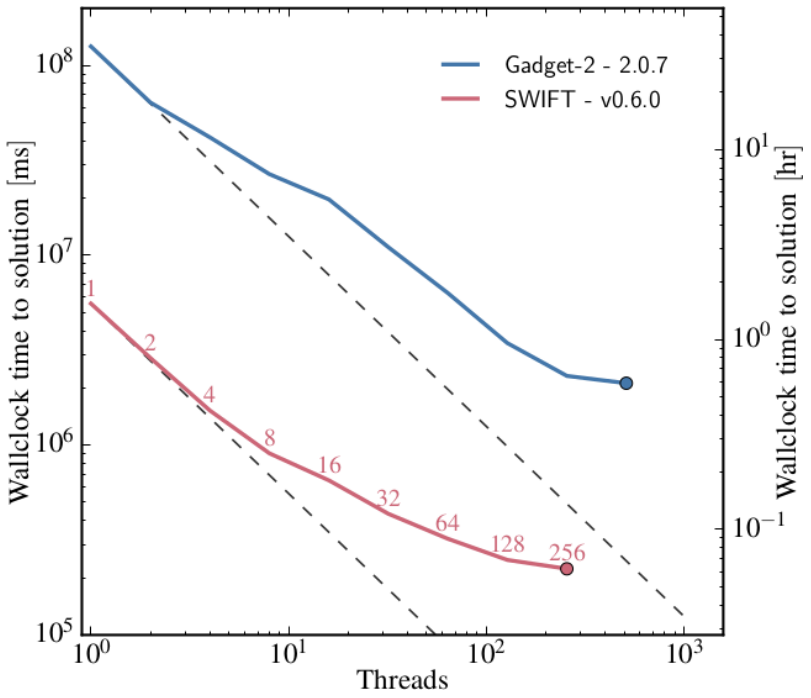
Intel ITAC output from two 36 cores Broadwell nodes.
→ More than 10’000 point-to-point communications over this time-step!
→ Almost 1’000’000 calls to `MPI_Test()`.

Performance vs. Gadget-2

- Strong scaling of realistic problem.

- Same accuracy.
- Same hardware.
- Same compiler.
- Same solution.

- 30.4x speed-up on 1 node!
- Code keeps scaling beyond 512 cores.



Quick I/O sneak-peak

Task-based i/o

- i/o is another bottleneck where the run halts and under-uses the cluster.
- Traditional pattern is to stop and dump everything.
- Huge load on the filesystem and network!
- Very inefficient! Very hard to achieve theoretical peak i/o especially on crowded systems.

Task-based i/o

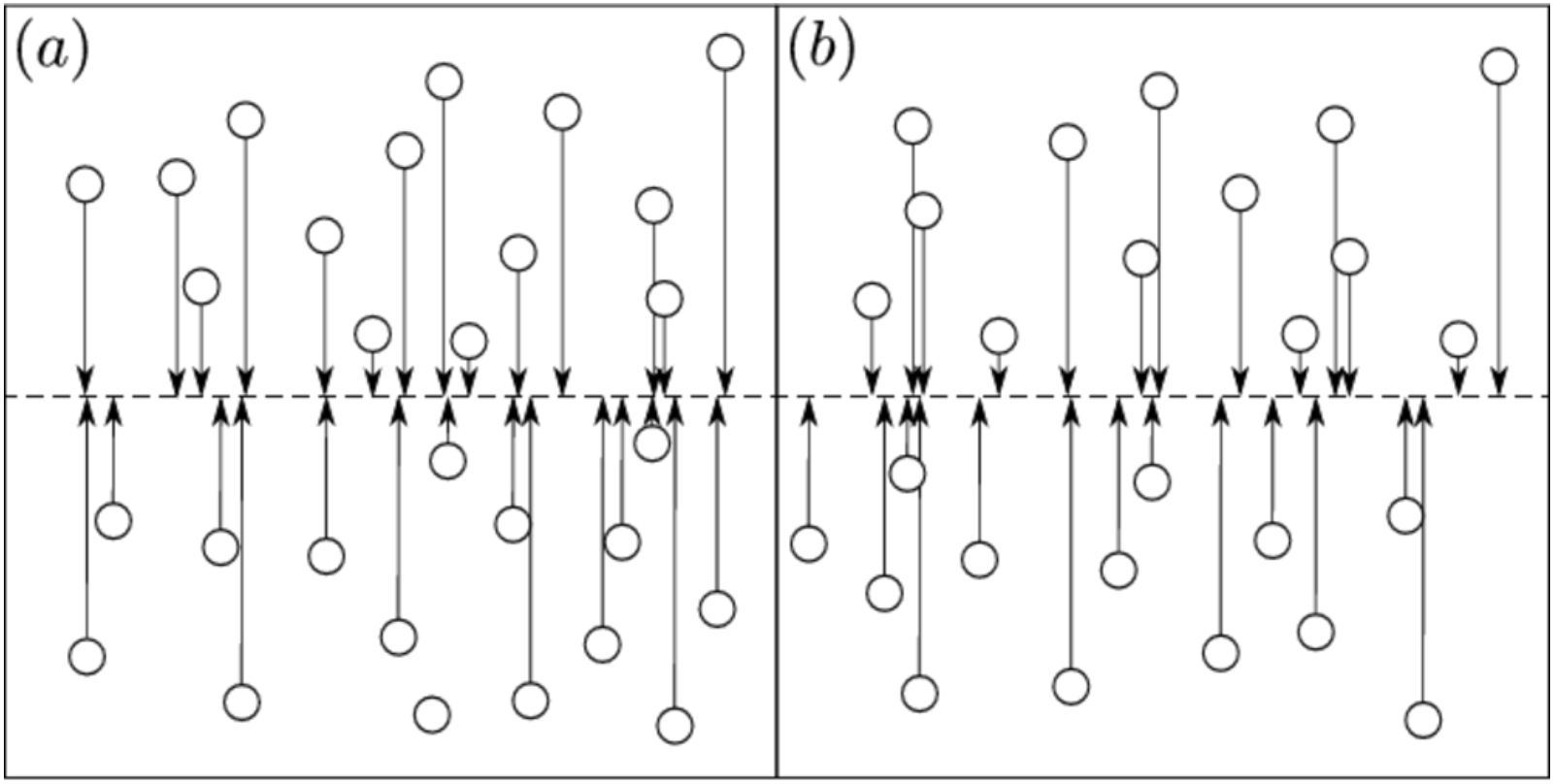
- Take advantage of the time-step hierarchy.
- Dump only parts of the data that vary rapidly (i.e. has changed a lot since last dump).
- Work on a cell-by-cell basis.
- Do this within the tasking system while the rest is happening.
  → No “stop and dump” pattern.
- Use memory-mapped files for maximal efficiency.

Vectorization

Vectorization

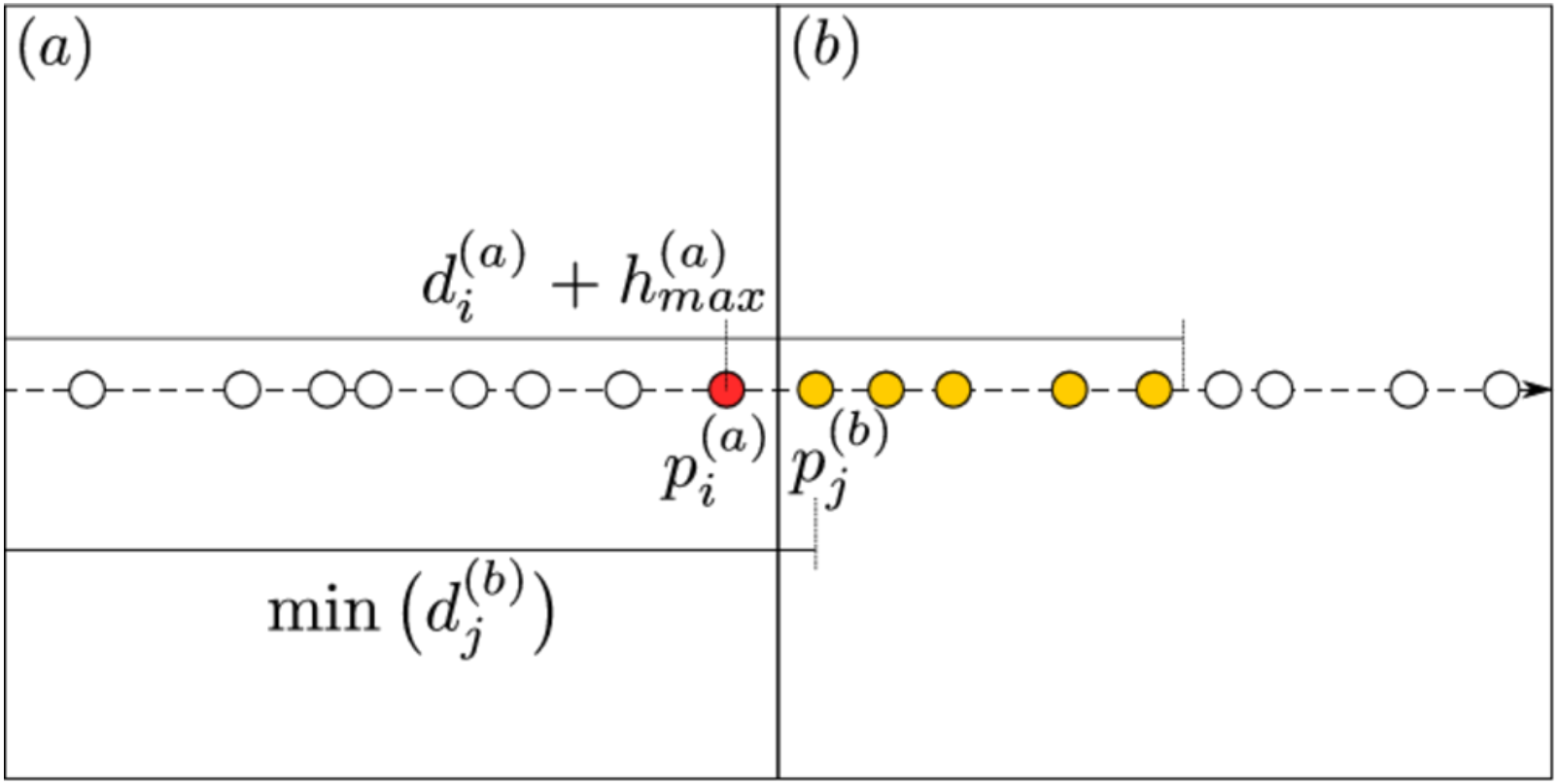
- Interact particles in one cell with particles in another cell.
- → naively N^2 algorithm.
-
- Need to construct local SoA cache for higher efficiency.
-
- Pre-sort particles on axis linking cells.
-
- Allows to only probe part of the interactions.
- → Reduce the number of false-positives!
-
- Need to use intrinsics as pattern too complex for auto-vectorizers.

Vectorization



Willis et al., Parco, 2017

Vectorization



Willis et al., Parco, 2017

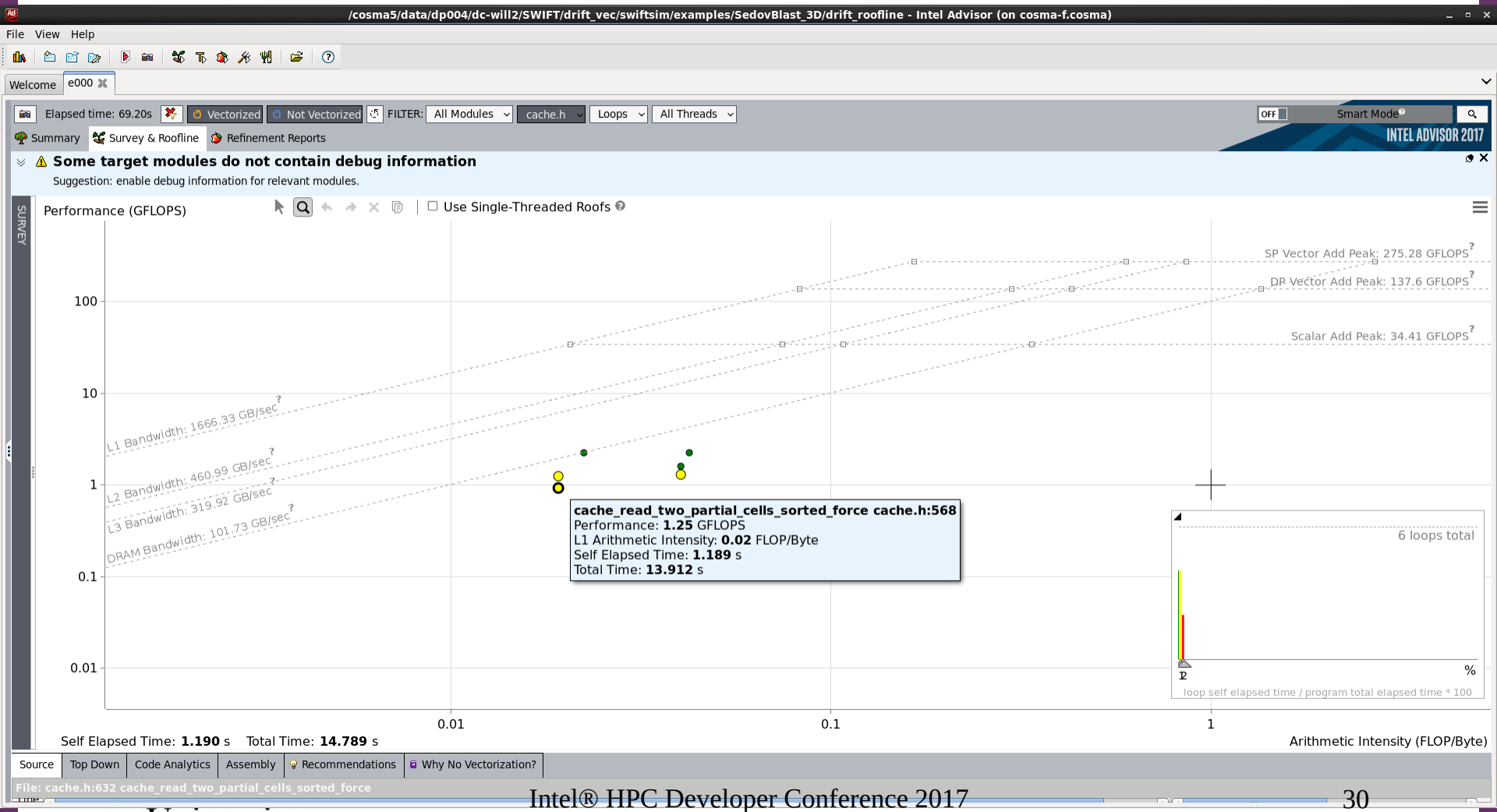
Vectorization

Machine Name	CFLAGS	Scalar Time [ms] (`-no-vec -no-simd`)	Vectorised Time [ms]	Speed-up
COSMA-5	`-xAVX`	0.56	0.25	2.24x
Hamilton	`-xCORE-AVX2`	0.49	0.20	2.43x
Kyll	`-xMIC-AVX512`	1.98	0.49	4.07x

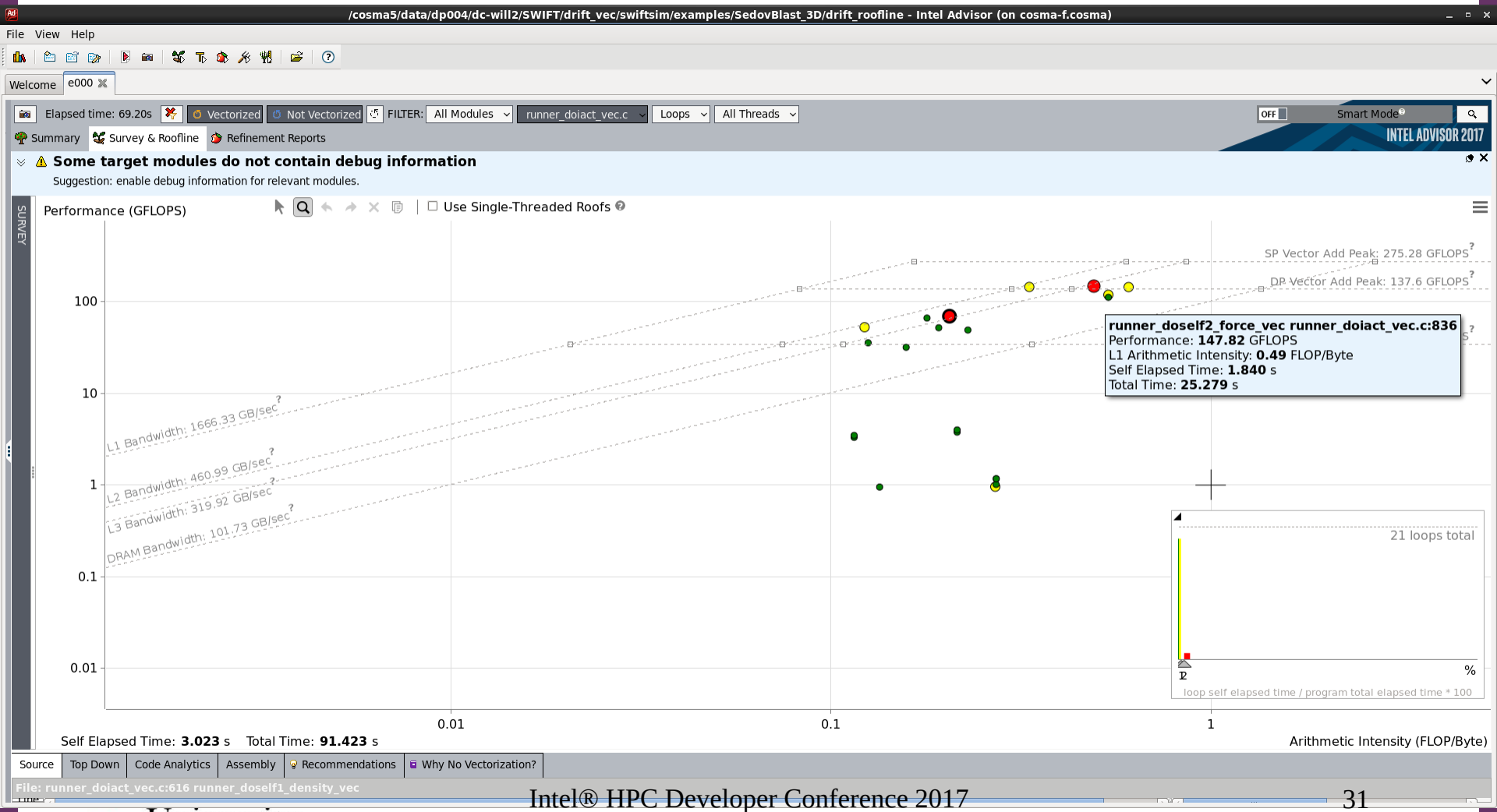
More than 12x faster than a “perfect” naive implementation.

Willis et al., Parco, 2017

Limitations – Memory bandwidth



Limitations – Memory bandwidth

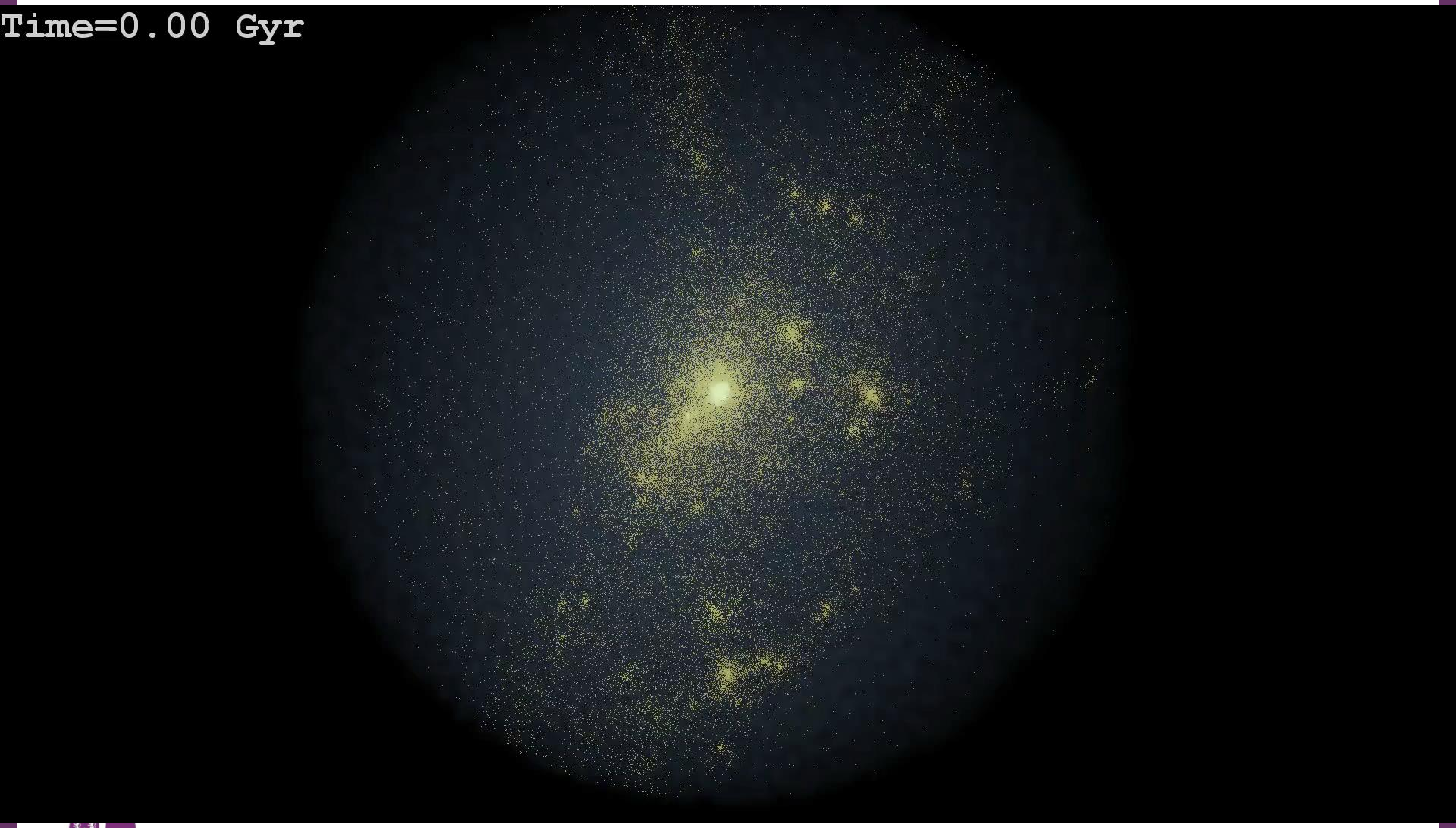


Conclusions & Take-home messages

Durham University

Conclusions

- New ideas can challenge pre-existing paradigms.
- Try new things.
- Clever algorithms are often better than vectorized naive code.
- Hiding communications, hiding i/o and a flexible scheduler are key to high performance on modern systems.



Durham University