Using Task-Based Parallelism, Asynchronous MPI and Dynamic Workload-Based Domain Decomposition to Achieve Near-Perfect Load-Balancing for Particle-Based Hydrodynamics and Gravity

Matthieu Schaller

Leiden Observatory, Netherlands

with

Stefan Arridge, Josh Borrow, Richard Bower, Aidan Chalk (Hartree Centre), Peter Draper, Pedro Gonnet (Google), Loic Hausamman (EPFL), Yves Revaz (EPFL), Bert Vandenbroucke (St. Andrews), James Willis

This work is a collaboration between 2 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 St-Andrews (UK), Dublin (Ireland), Leiden (Netherlands) Lausanne (Switzerland) as well as the DiRAC and CSCS software teams.

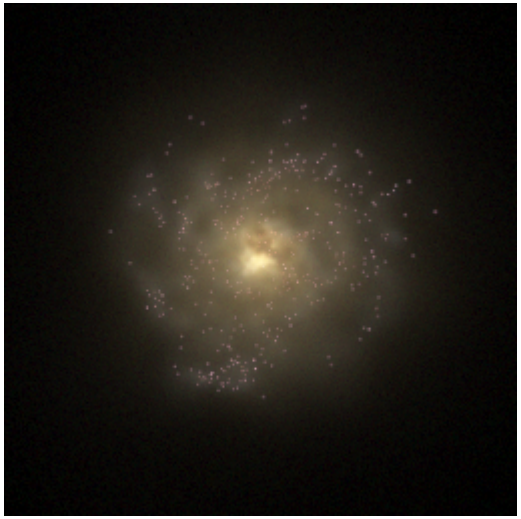
This research is partly funded by an Intel IPCC since January 2015.

Introduction

The problem to solve

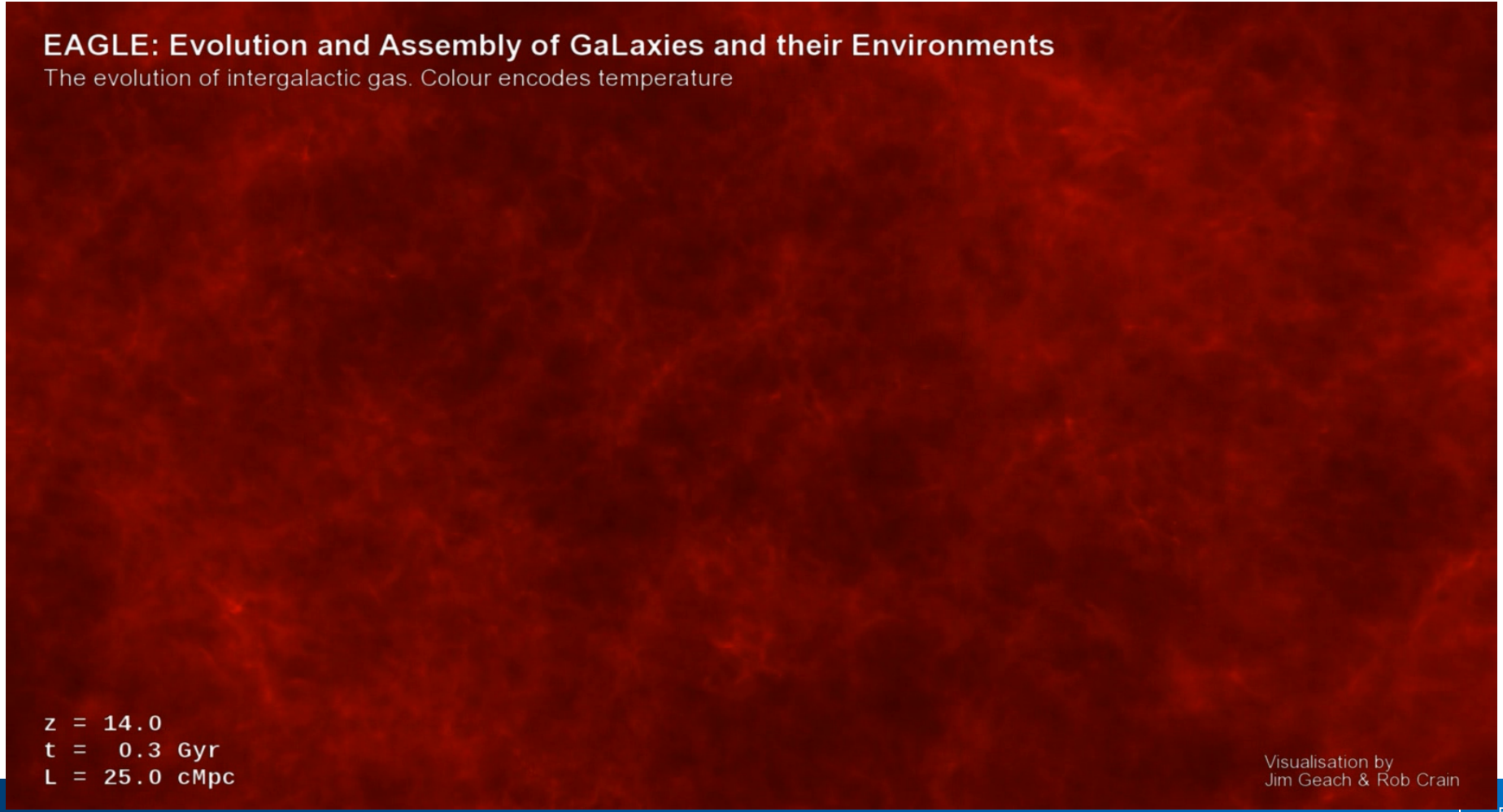
What we do and how we do it

- Astronomy / Cosmology simulations of the formation of the Universe and galaxy evolution.
- EAGLE project[1]: 48 days of computing on 4096 cores. >500 TBytes of data products (post-processed data is public!). Most cited astronomy paper of 2015 (out of >26000).
- Simulations of gravity and hydrodynamic forces with a spatial dynamic range spanning 6 orders of magnitude running for >2M time-steps.



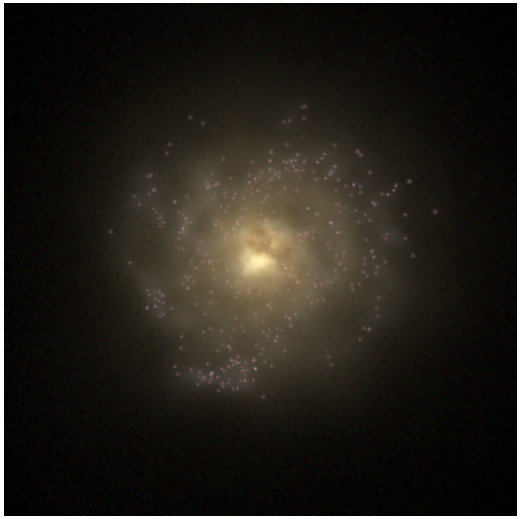
One simulated galaxy out of the EAGLE virtual universe.

1) www.eaglesim.org


EAGLE: Evolution and Assembly of GaLaxies and their Environments
The evolution of intergalactic gas. Colour encodes temperature
z = 14.0
t = 0.3 Gyr
L = 25.0 cMpc
Visualisation by
Jim Geach & Rob Crain

What we do and how we do it

- Solve coupled equations of gravity and hydrodynamics using SPH (Smoothed Particle Hydrodynamics).
- Consider the interaction between gas and stars/black holes as part of a large and complex *subgrid* model.
- Evolve multiple matter species at the same time.
- Large density imbalances develop over time:
  → Difficult to load-balance.



One simulated galaxy out of the EAGLE virtual universe.

SPH scheme: The problem to solve

For a set of N ($>10^9$) particles, we want to exchange hydrodynamical forces between all neighbouring particles within a given (time and space variable) search radius. Large density imbalances develop over time.

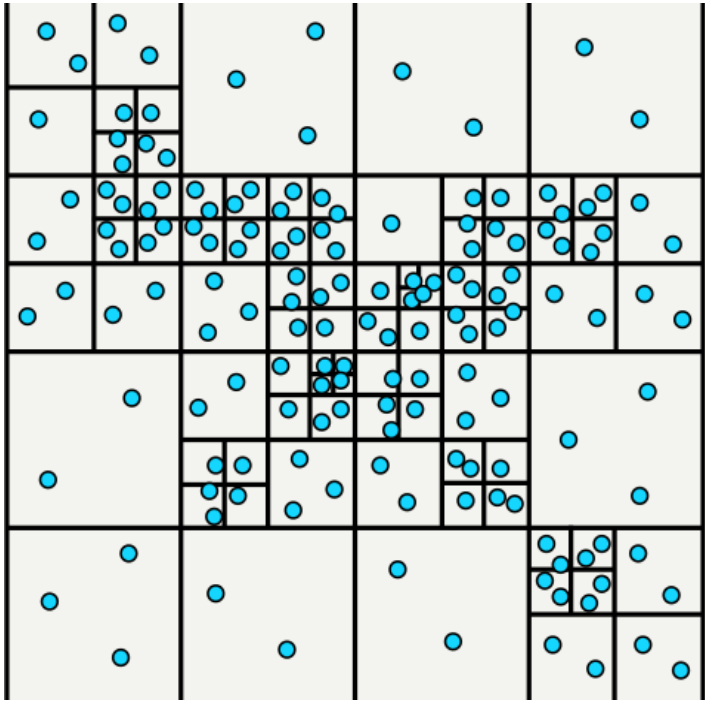
Challenges:

- Particles are unstructured in space, large density variations.
- Particles will move and the neighbour list of each particle evolves over time.
- Interaction between two particles is computationally cheap (low flop/byte ratio).
- Individual time-steps for each particle.

SPH scheme: The traditional method

The “industry standard” cosmological code is GADGET (Springel et al.1999, Springel 2005).

- MPI-only code.
- Neighbour search based on oct-tree.
- Oct-tree implies “random” memory walks
  - Lack of predictability.
  - Nearly impossible to vectorize.
  - Very hard to load-balance.



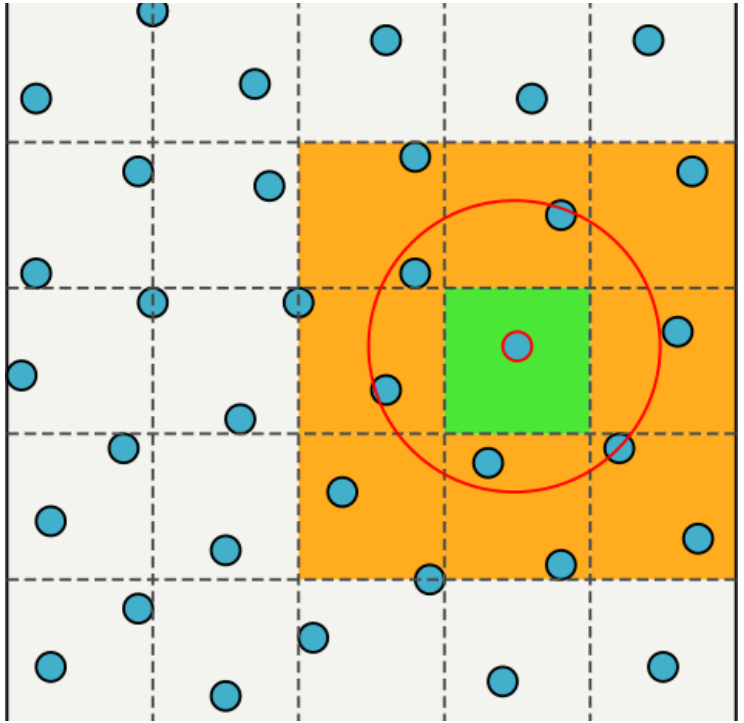
SPH scheme: The traditional method

```
for (int i=0; i<N; ++i) {   // loop over all particles

     struct part *pi = &parts[i];

     list = tree_get_neighbours(pi->position, pi->search_radius); // get a list of ngbs

     for(int j=0; j < N_ngb; ++j) {  // loop over ngbs

          const struct part *pj = &parts[list[j]];

          INTERACT(pi, pj);
} }
```

SPH scheme: The **SWIFT** way

Need to make things regular and predictable:

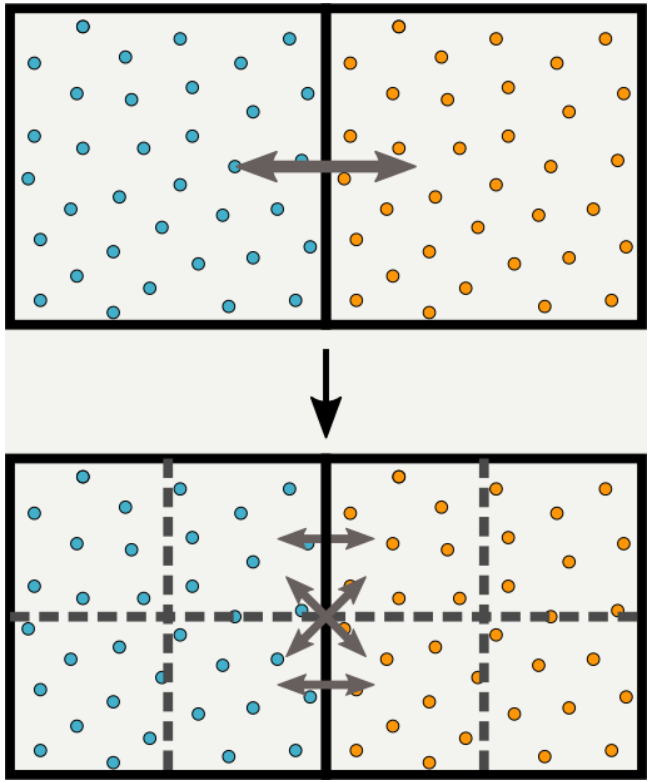
- Neighbour search is performed via the use of an adaptive grid constructed recursively until we get ~500 particles per cell.
- Cell spatial size matches search radius.
- Particles interact only with partners in their own cell or one of the 26 neighbouring cells



SPH scheme: The **SWIFT** way

Retain the large fluctuations in density by splitting cells:

- If cells have ~400 particles they fit in the L2 caches.
- Makes the problem very local and fine-grained.



SPH scheme: The **SWIFT** way

```
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
} } } }
```

SPH scheme: The **SWIFT** way

Threads + MPI

```
for (int ci=0; ci < nr_cells; ++ci) {    // loop over all cells
     for(int cj=0; cj < 27; ++cj) {      // loop over all 27 cells neighbouring cell ci
-------------------------------------------------------------------------------
          const int count_i = cells[ci].count;
          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
} } } }
```

Vectorization

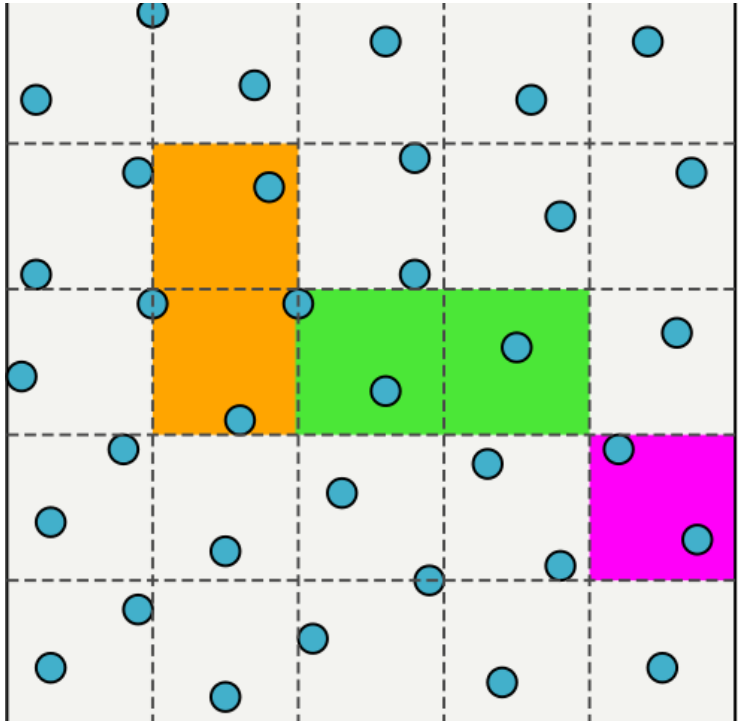
Single-node parallelisation

Task-based parallelism

SPH scheme: Single-node parallelization

No need to process the cell pairs in any specific order:

- -> No need to enforce and order.
- -> Only need to make sure we don’t process pairs that use the same cell.
- -> Pairs could have vastly different runtimes since they can have very different particle numbers.

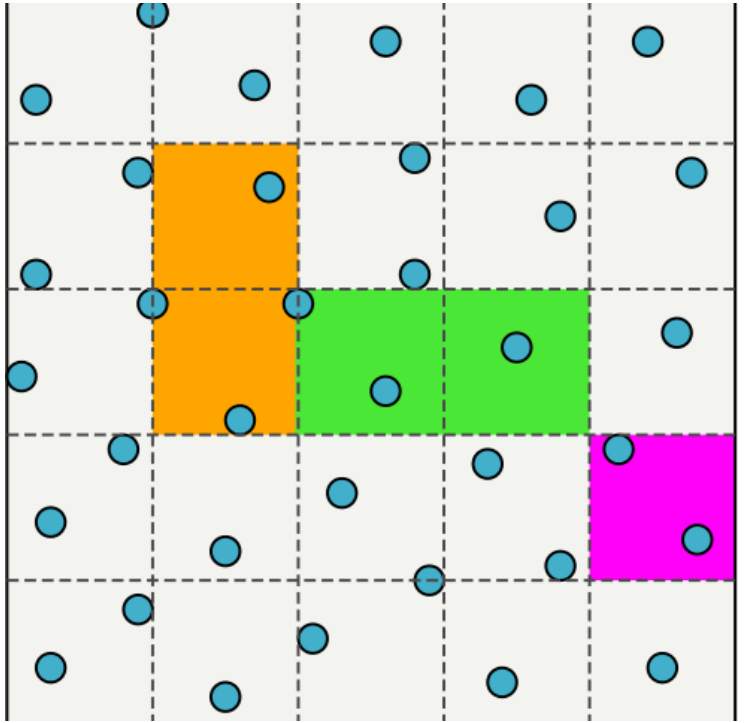


SPH scheme: Single-node parallelization

No need to process the cell pairs in any specific order:

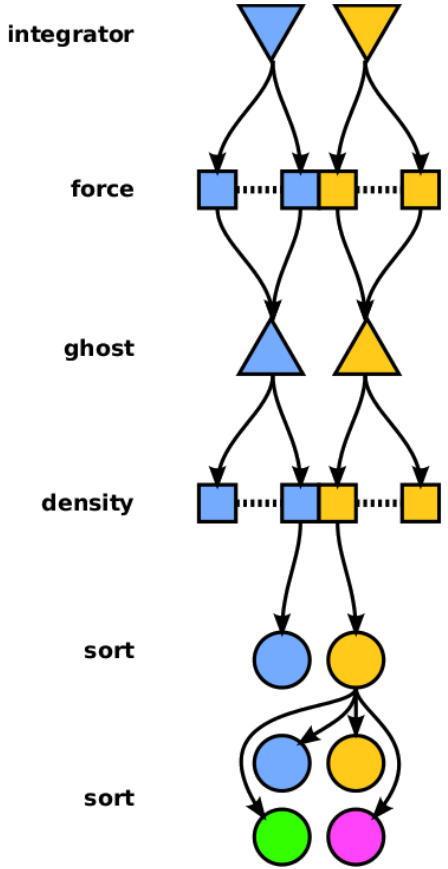
- -> No need to enforce and order.
- -> Only need to make sure we don’t process pairs that use the same cell.
- -> Pairs could have vastly different runtimes since they can have very different particle numbers.

We need dynamic scheduling !



Task-base parallelism for SPH

- For two cells, we have the task graph shown on the right.
- Arrows depict dependencies, dashed lines show conflict.
- Ghost tasks are used to link tasks and reduce the number of dependencies.



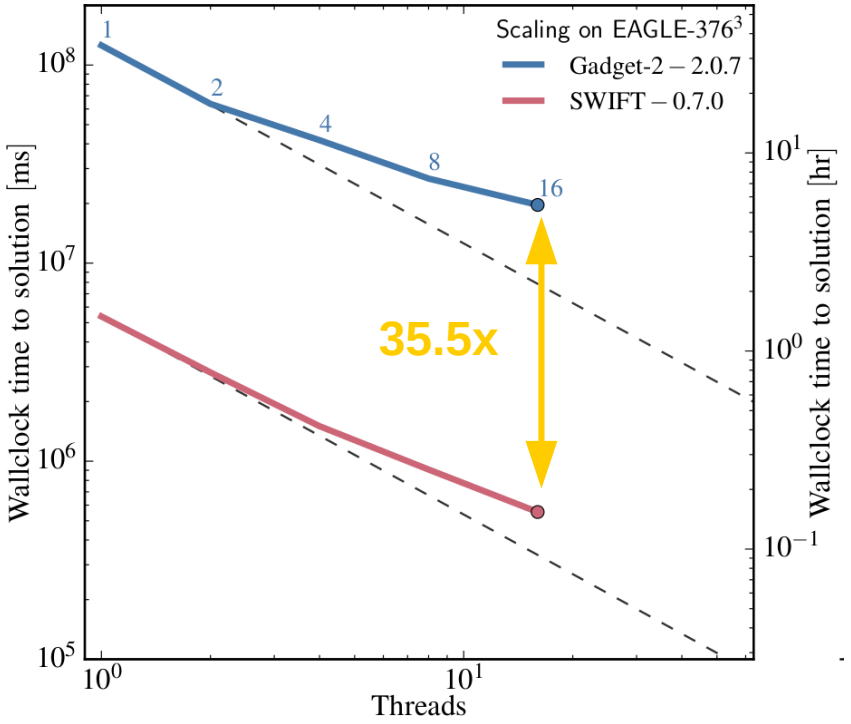
SPH scheme: Single node parallel performance

Task graph for one time-step.
Colours correspond to different types of task. Almost perfect load-balancing is achieved on 16 cores.

Single node performance vs. Gadget

- Realistic problem (video from start of the talk)
- Same accuracy.
- Same hardware.
- Same compiler.
- Same solution.

More than 30x speed-up vs. “industry standard” Gadget code.

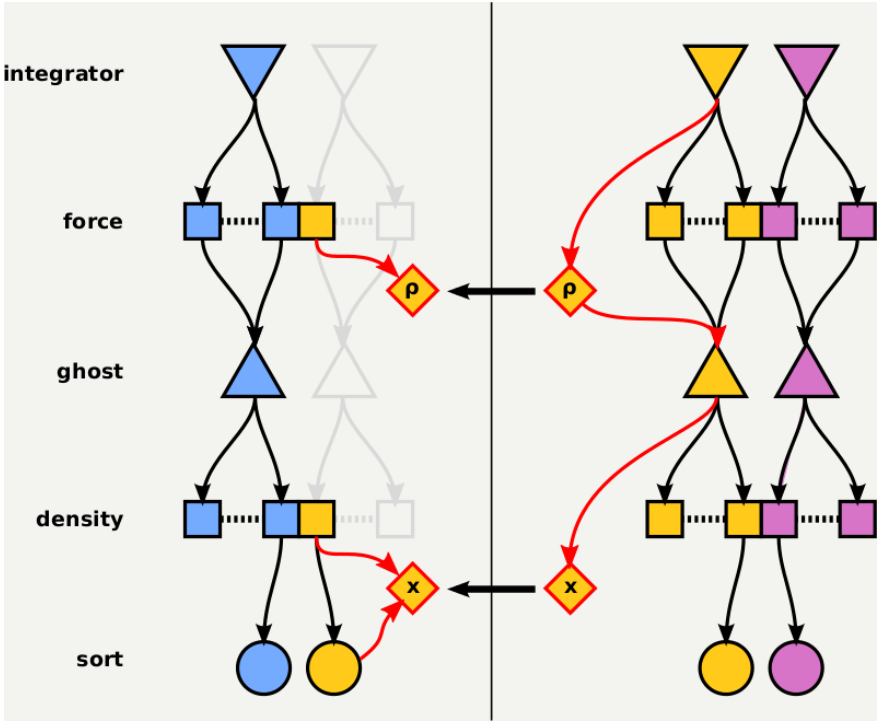


Multi-node parallelisation

Asynchronous MPI communications

Asynchronous communications as tasks

- A given rank will need the cells directly adjacent to it to interact with its particles.
- Instead of sending all the “halo” cells at once between the computation steps, we send each cell individually using MPI asynchronous communication primitives.
- Sending/receiving data is just another task type, and can be executed in parallel with the rest of the computation.
- Once the data has arrived, the scheduler unlocks the tasks that needed the data.
- No global lock or barrier !



Asynchronous communications as tasks

Communication tasks do not perform any computation:

- Call `MPI_Isend()` / `MPI_Irecv()` when enqueued.
- Dependencies are released when `MPI_Test()` says the data has been sent/received.

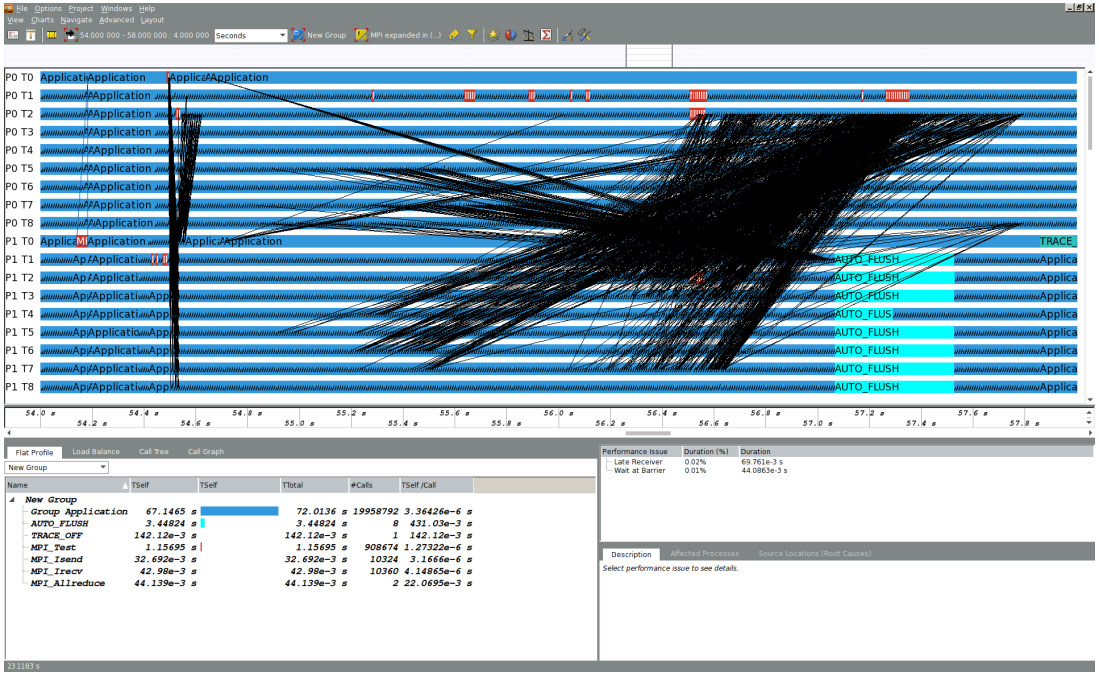
Not all MPI implementations fully support the MPI v3.0 standard.

- Uncovered several bugs in different implementations providing `MPI_THREAD_MULTIPLE`.
- e.g.: - OpenMPI 1.x – 2.0 crashes when running on Infiniband.
  - Intel-MPI 2017 crashes when running on OmniPath.

Most experienced MPI users will advise *against* creating so many send/recv.

Asynchronous communications as tasks

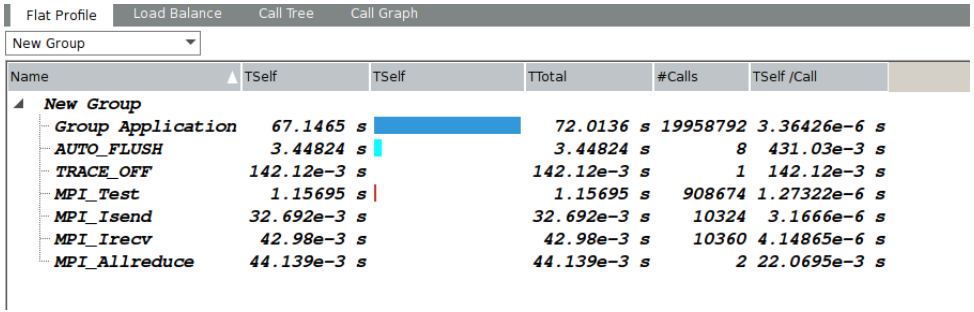
- Message size is 5-10kB.
- On 32 ranks with 16M particles in 250’000 cells, we get ~58’000 point-to-point messages *per time-step*!
- Relies on `MPI_THREAD_MULTIPLE` as all the local threads can emit sends and receives.
- Spreads the load on the network over the whole time-step.
  → More efficient use of the network!
  → Not limited by bandwidth.



Intel ITAC output from 2x36-cores Broadwell nodes. Every black line is a communication between two threads (blue bands).

Asynchronous communications as tasks

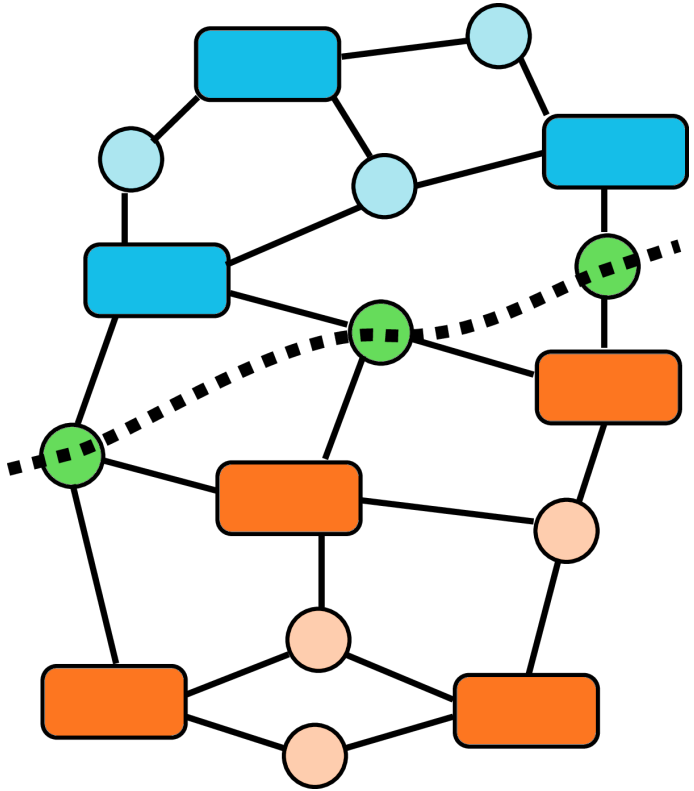
- Message size is 5-10kB.
- On 32 ranks with 16M particles in 250’000 cells, we get ~58’000 point-to-point messages *per time-step*!
- Relies on `MPI_THREAD_MULTIPLE` as all the local threads can emit sends and receives.
- Spreads the load on the network over the whole time-step.
  → More efficient use of the network!
  → Not limited by bandwidth.



Intel ITAC output from 2x36-cores Broadwell nodes. >10k point-to-point communications are reported over this time-step.

Domain decomposition

- For each task we compute the amount of work (=runtime) required.
- We can build a graph in which the simulation data are nodes and the tasks operation on the data are hyperedges.
- The task graph is split to balance the work (not the data!) using the METIS library.
- Tasks spanning the partition are computed on both sides, and the data they use needs to be sent/received between ranks.
- Send and receive tasks and their dependencies are generated automatically.



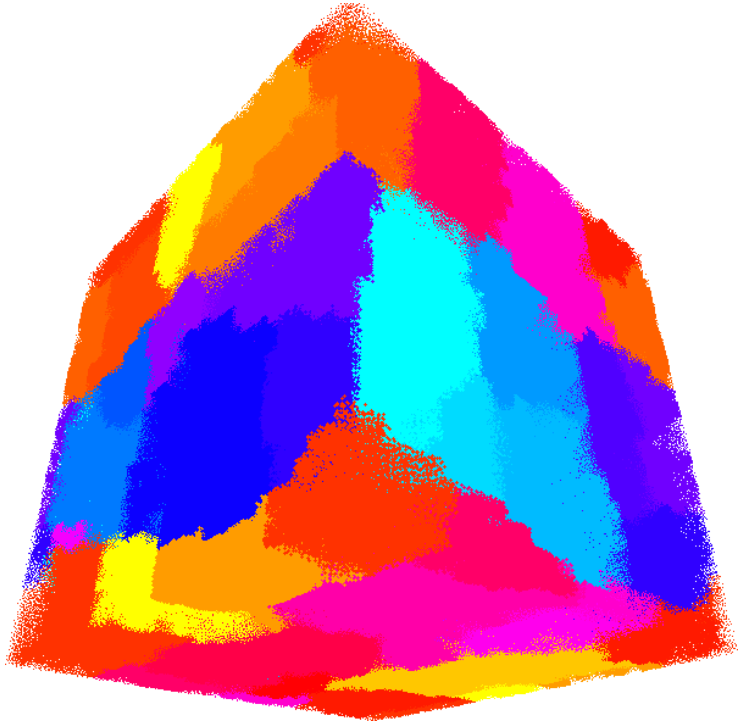
Domain decomposition

Domain geometry can be complex.

- No regular grid pattern.
- No space-filling curve order.
- Good load-balancing by construction.

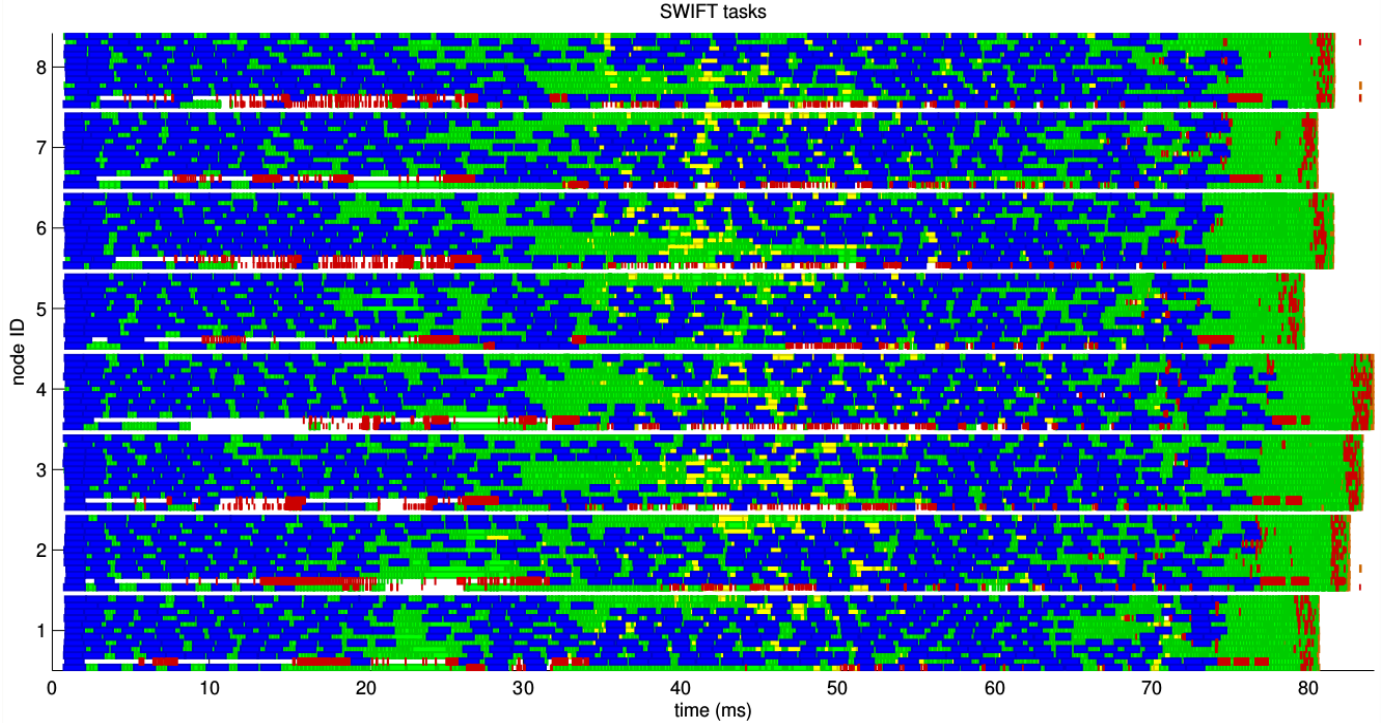
Domain shapes and computational costs evolve over the course of the simulation.

- Periodically update the graph partitioning.
- May lead to large (unnecessary?) re-shuffling of the data across the whole machine.



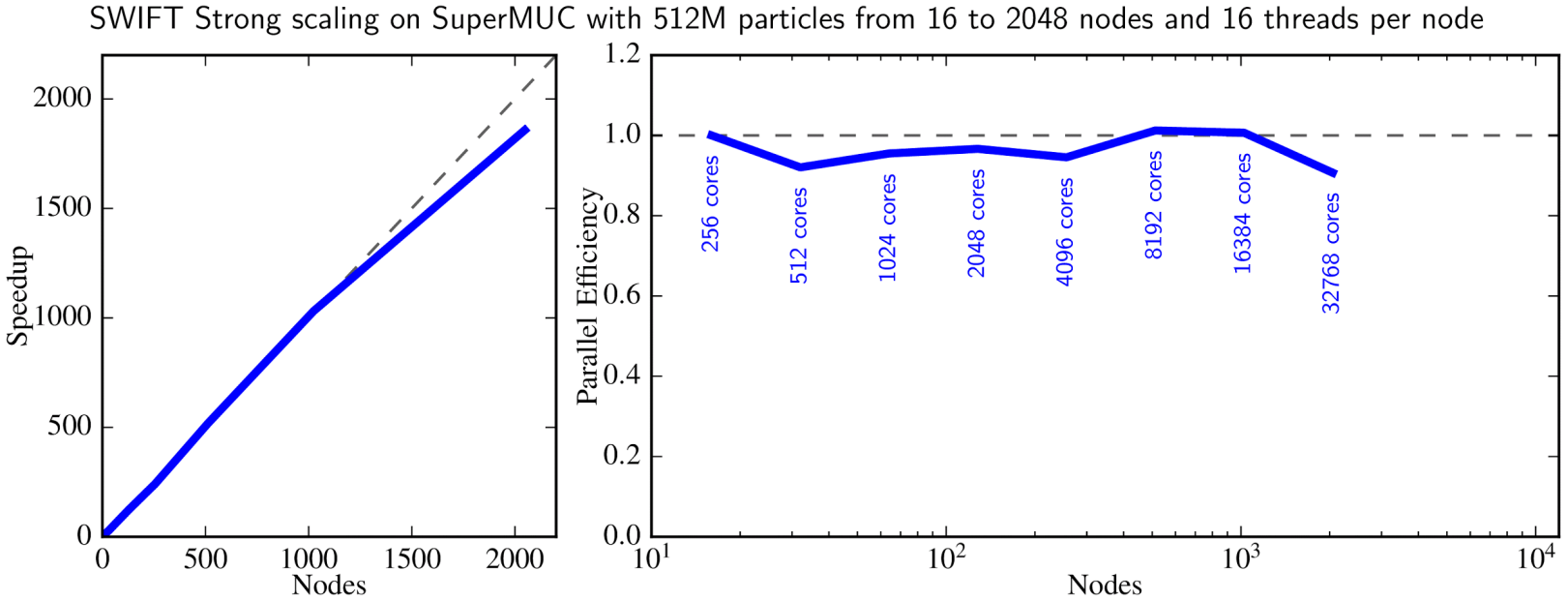
Particles coloured by the domain they belong to for a cosmological simulation.
The domains are un-structured.

Multiple node parallel performance



Task graph for one time-step. Red and yellow are MPI tasks. Almost perfect load-balancing is achieved on 8 nodes of 12 cores.

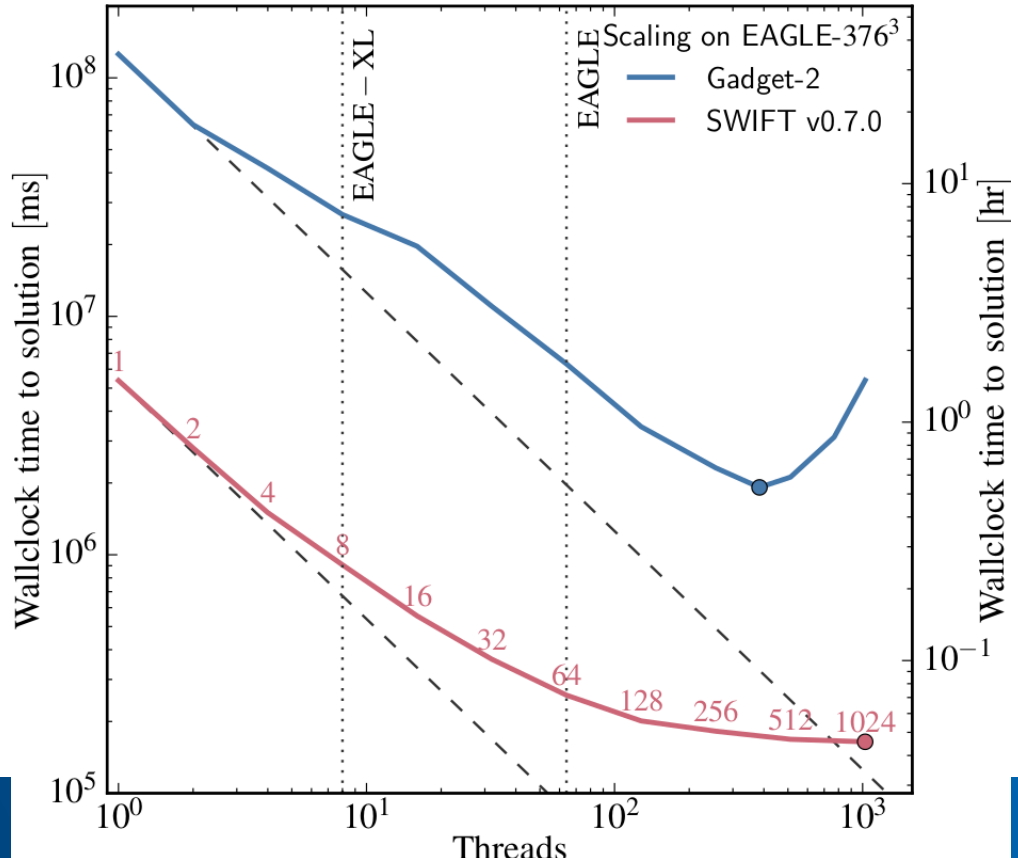
Scaling results: SuperMUC



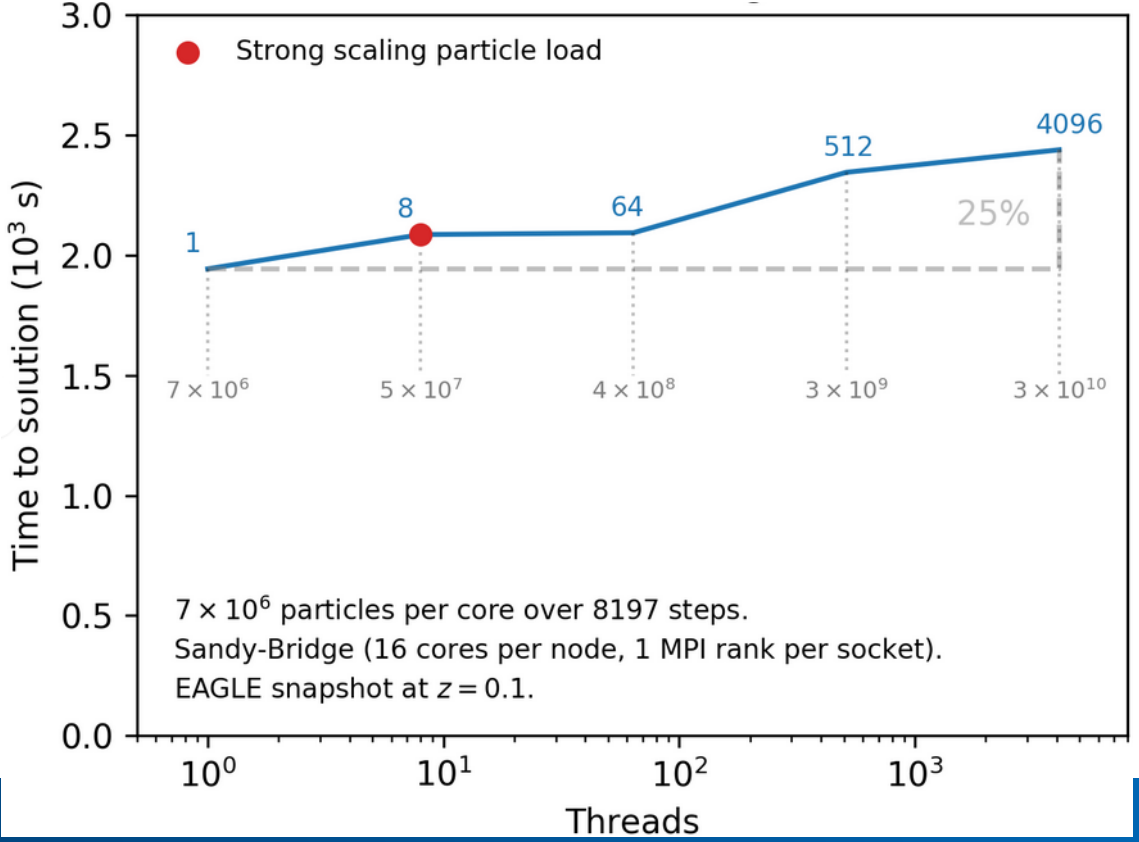
System: x86 architecture - 2 Intel Sandy Bridge Xeon E5-2680 8C at 2.7 GHz with 32 GByte of RAM per node.

Schaller+2016, PASC

Scaling results: Realistic case (multi-dt)

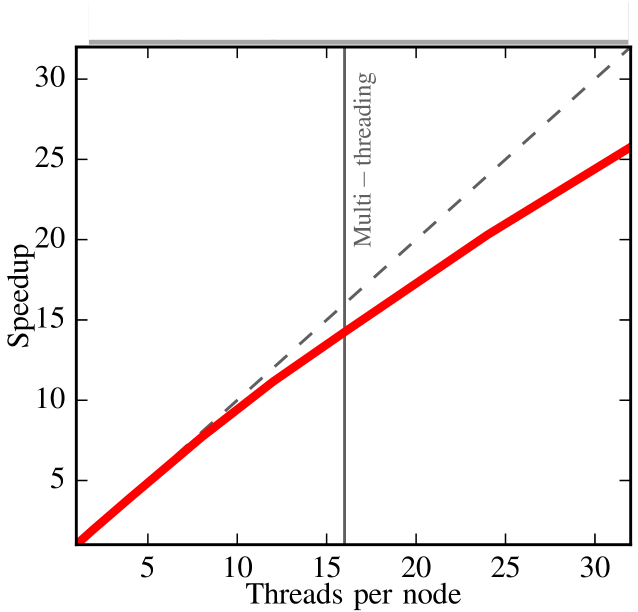


Scaling results: Weak scaling



Scaling results: “strong scaling” within node

- Almost perfect *strong*-scaling performance on a cluster of many-core nodes when increasing the number of threads per node (fixed #MPI ranks).
- Clear benefit of task-based parallelism and asynchronous communication.
- Future-proof! As the thread/core count per node increases, so does the code performance.
- Why?
  → Because we don’t rely on MPI for intra-node communications.



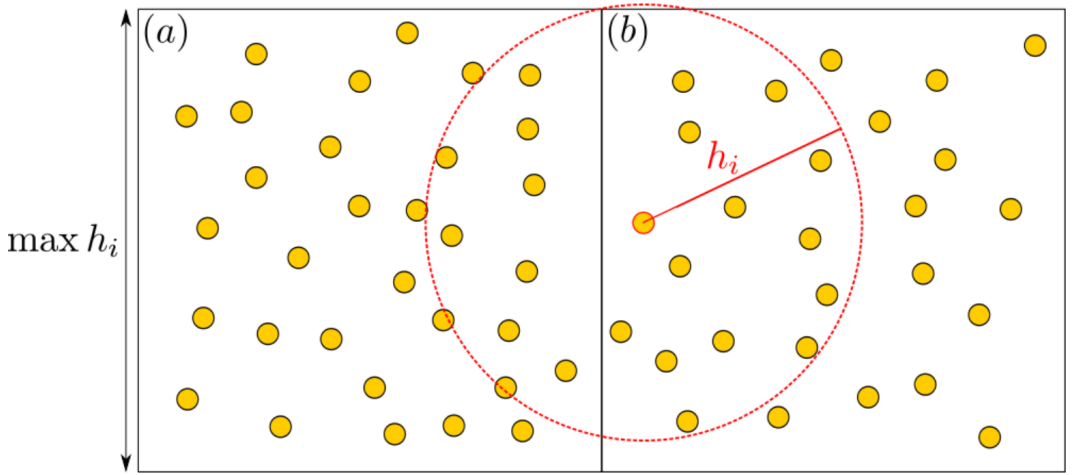
SIMD parallelisation

Explicit vectorization using intrinsics

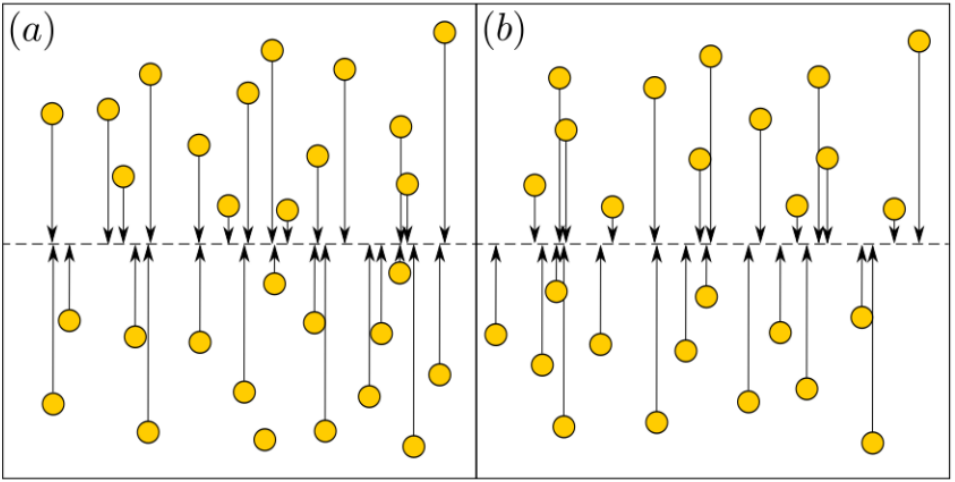
Explicit vectorization of the core routines.

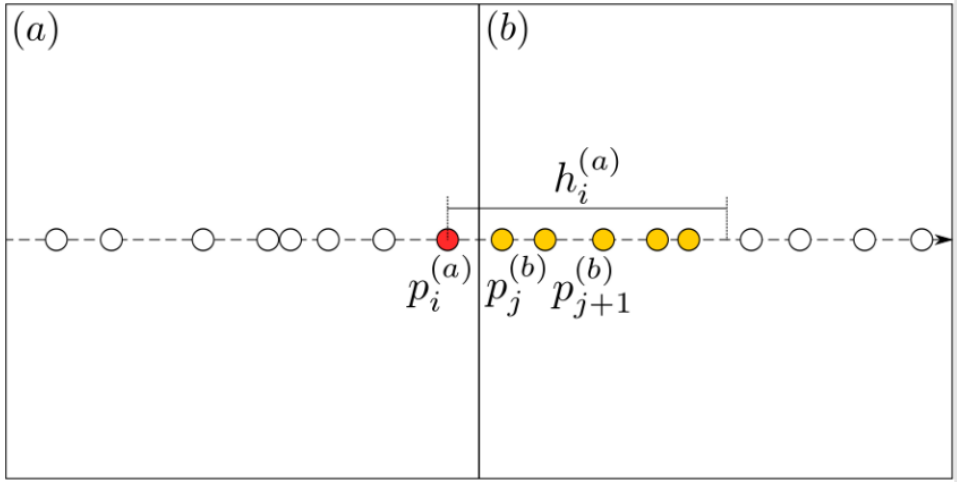
Thanks to our task-based parallel framework:

- No need to worry about MPI
- No need to worry about threading or race conditions
- Full problem holds in L2 cache.

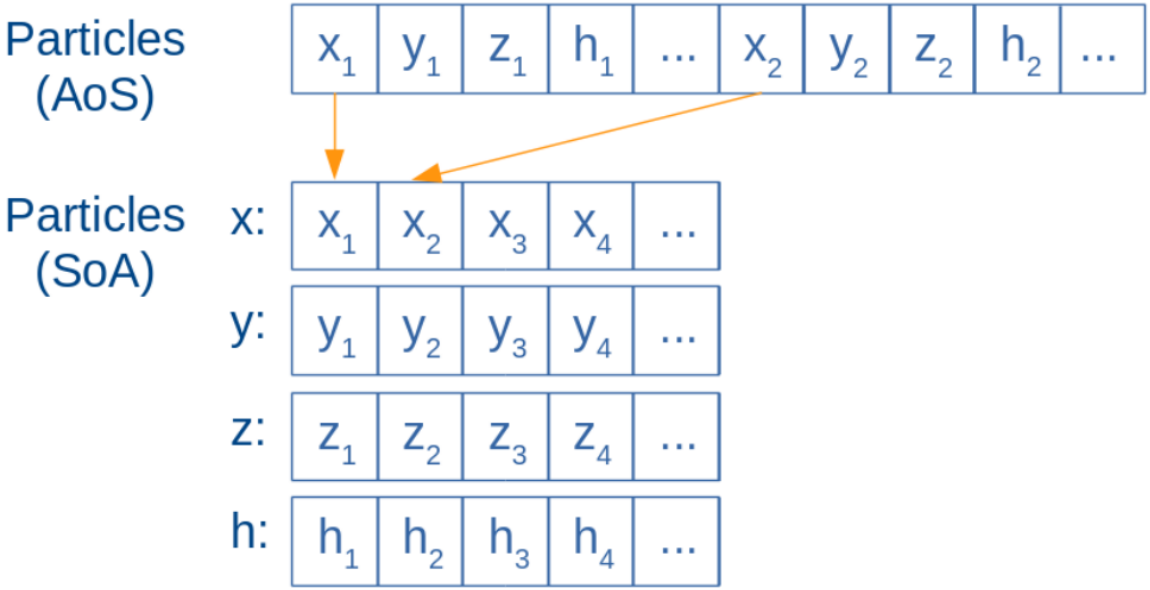


Sorted interactions (pseudo-Verlet list)

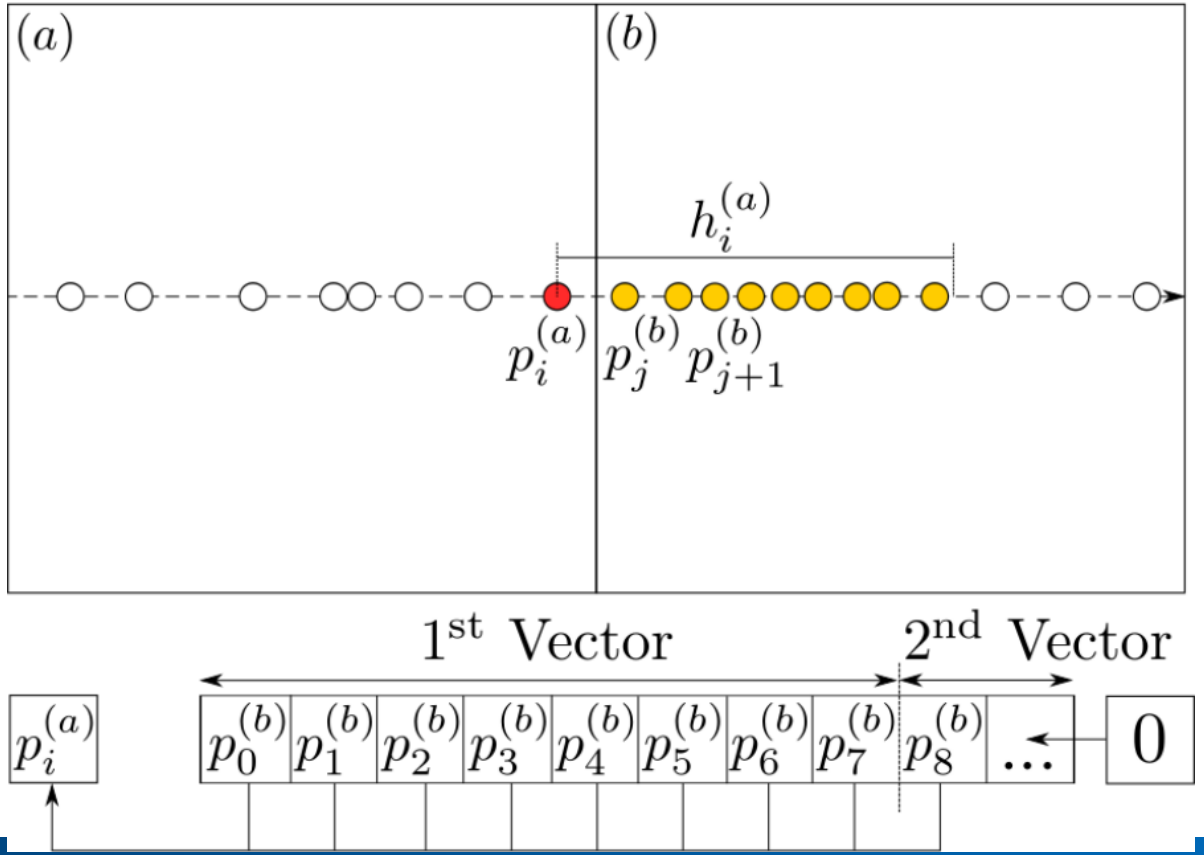


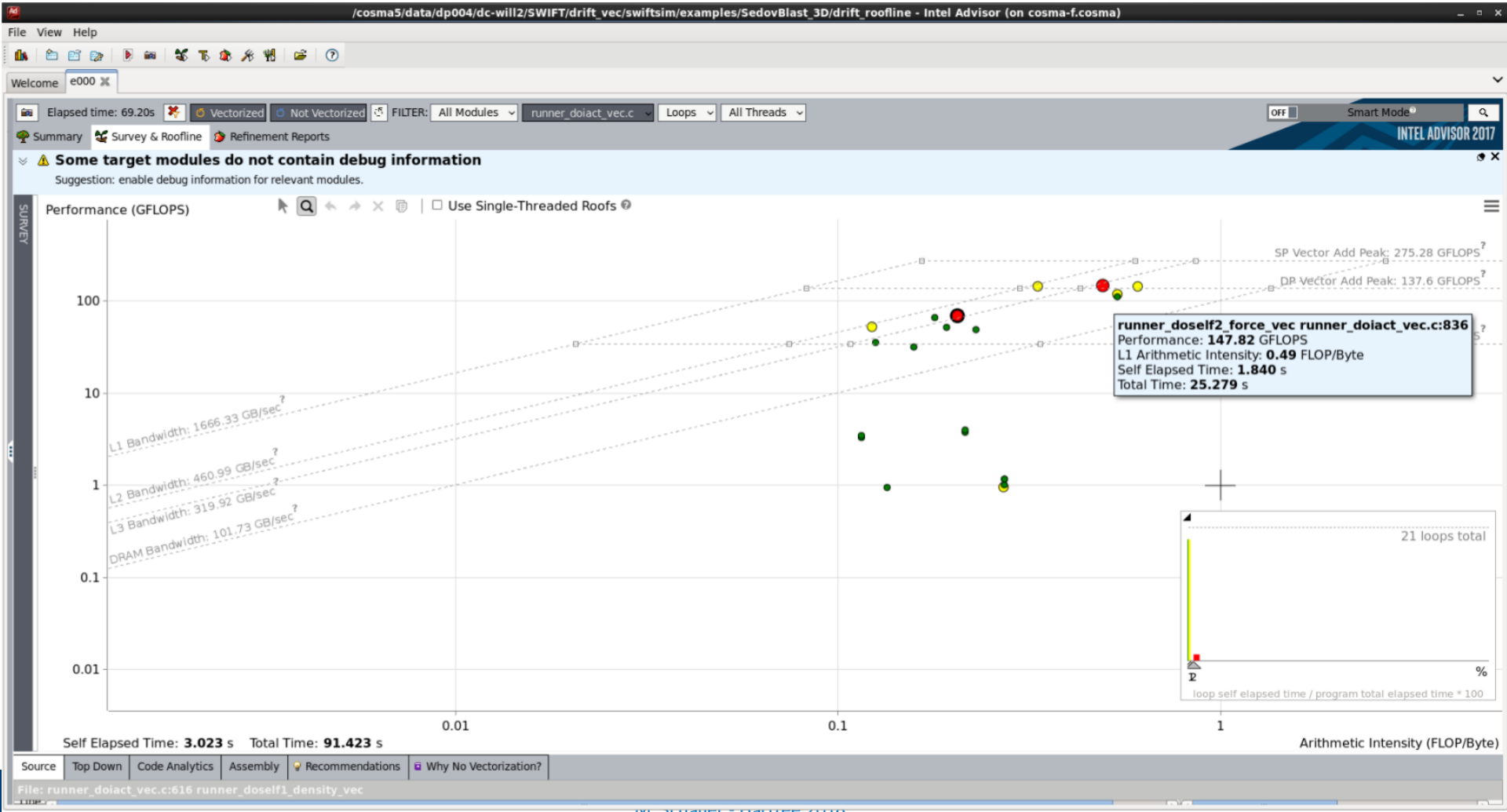


Sorted interactions with local cache



Sorted interactions with local cache




/cosma5/data/dp004/dc-will2/SWIFT/drift_vec/swiftsim/examples/SedovBlast_3D/drift_roofline - Intel Advisor (on cosma-f.cosma)
File View Help
Welcome
e000
Elapsed time: 69.20s
Vectorized
Not Vectorized
FILTER: All Modules
runner_doiact_vec.c
Loops
All Threads
Smart Mode
INTEL ADVISOR 2017
Summary
Survey & Roofline
Refinement Reports
Some target modules do not contain debug information
Suggestion: enable debug information for relevant modules.
SURVEY
Performance (GFLOPS)
Use Single-Threaded Roofs
SP Vector Add Peak: 275.28 GFLOPS
DP Vector Add Peak: 137.6 GFLOPS
L1 Bandwidth: 1666.33 GB/sec
L2 Bandwidth: 460.99 GB/sec
L3 Bandwidth: 319.92 GB/sec
DRAM Bandwidth: 101.73 GB/sec
runner_doself2_force_vec runner_doiact_vec.c:836
Performance: 147.82 GFLOPS
L1 Arithmetic Intensity: 0.49 FLOP/Byte
Self Elapsed Time: 1.840 s
Total Time: 25.279 s
21 loops total
%
loop self elapsed time / program total elapsed time * 100
100
10
1
0.1
0.01
0.01
0.1
1
Self Elapsed Time: 3.023 s
Total Time: 91.423 s
Arithmetic Intensity (FLOP/Byte)
Source
Top Down
Code Analytics
Assembly
Recommendations
Why No Vectorization?
File: runner_doiact_vec.c:616 runner_doself1_density_vec

Conclusions

And take-away messages

More on SWIFT

Completely open-source software including all the examples and scripts.

~40’000 lines of C code without fancy language extensions.

More than 20x faster than the *de-facto* standard `Gadget` code on the same setup and same architecture. Thanks to:

- Better algorithms
- Better parallelisation strategy
- Better domain decomposition strategy

Fully compatible with `Gadget` in terms of input and output files.

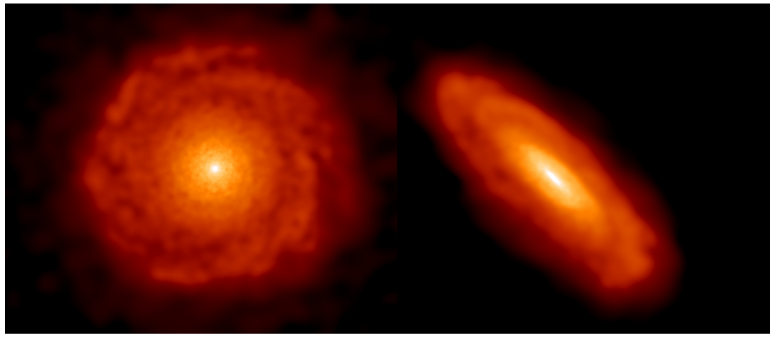
More on SWIFT

Gravity solved using a FMM and mesh for periodic and long-range forces.

Gravity and hydrodynamics are solved *at the same time* on the same particles as different properties are updated. No need for an explicit lock.

I/O done using the (parallel) HDF5 library, currently working on a continuous asynchronous approach.

Task-based parallelism allows for *very* simple code within tasks.
→ Very easy to extend with new physics without worrying about parallelism.



Conclusion and Outlook

Collaboration between Computer scientists and physicists works!

Successfully decomposed the parallelization in three separate problems.

Developed usable simulation software using state-of-the-art paradigms.

Great strong-scaling results up to >100’000 cores.