The cosmological hydrodynamical code **SWIFT**

Matthieu Schaller

Institute for Computational Cosmology, Durham University, UK

6th July 2017

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), Lausanne (Switzerland) and the DiRAC software team.

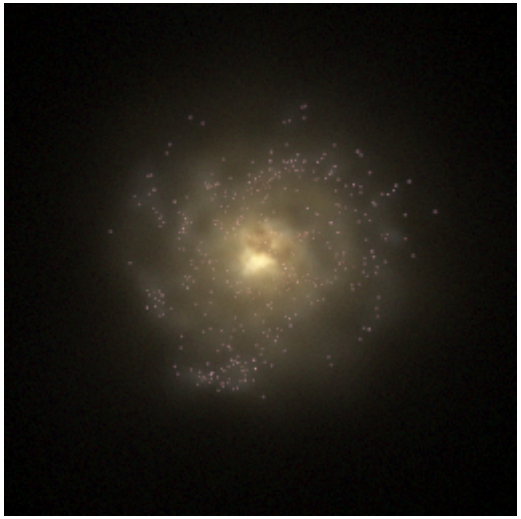
This research is partly funded by an Intel IPCC since January 2015.

Introduction

The problem to solve

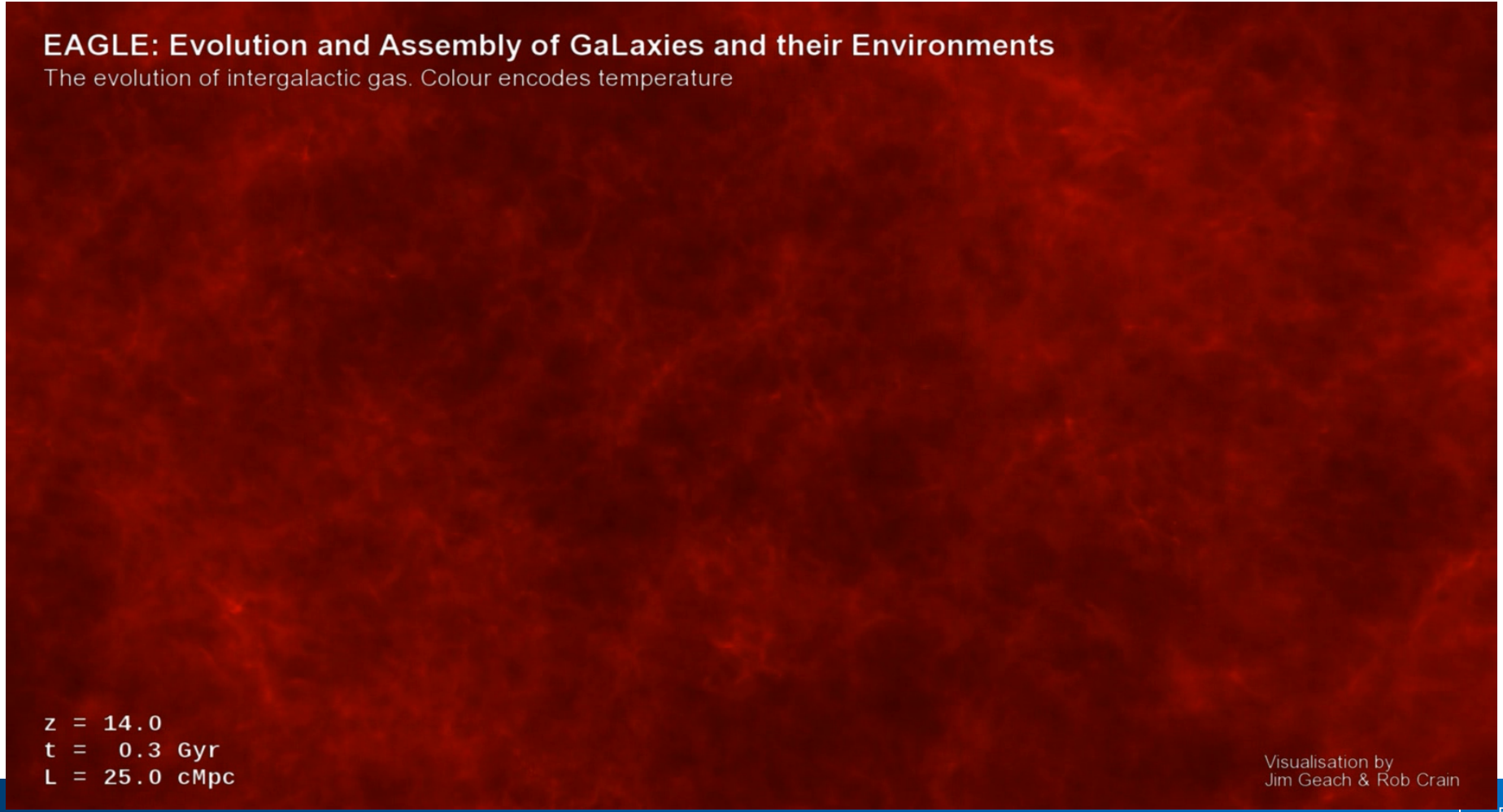
What we do and how we do it

- 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

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.

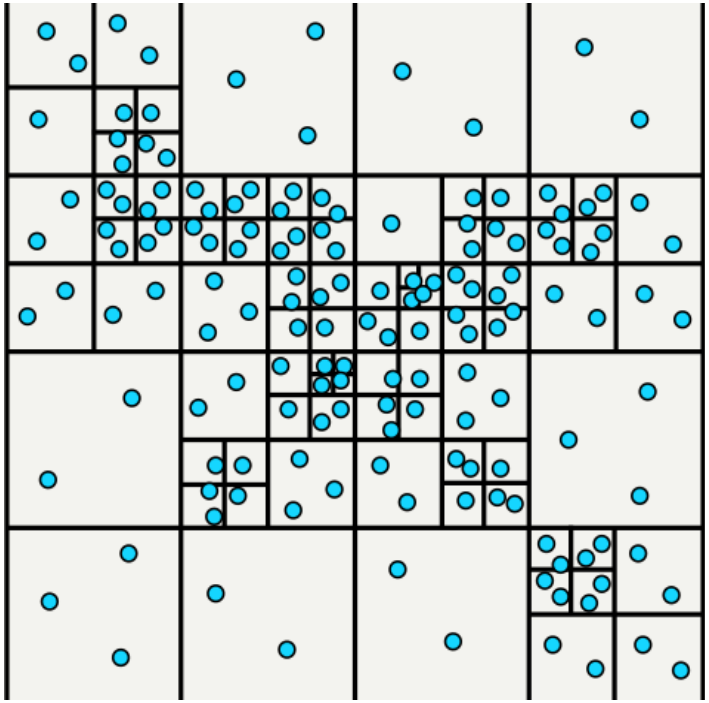
Challenges:

- 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).

SPH scheme: The traditional method

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

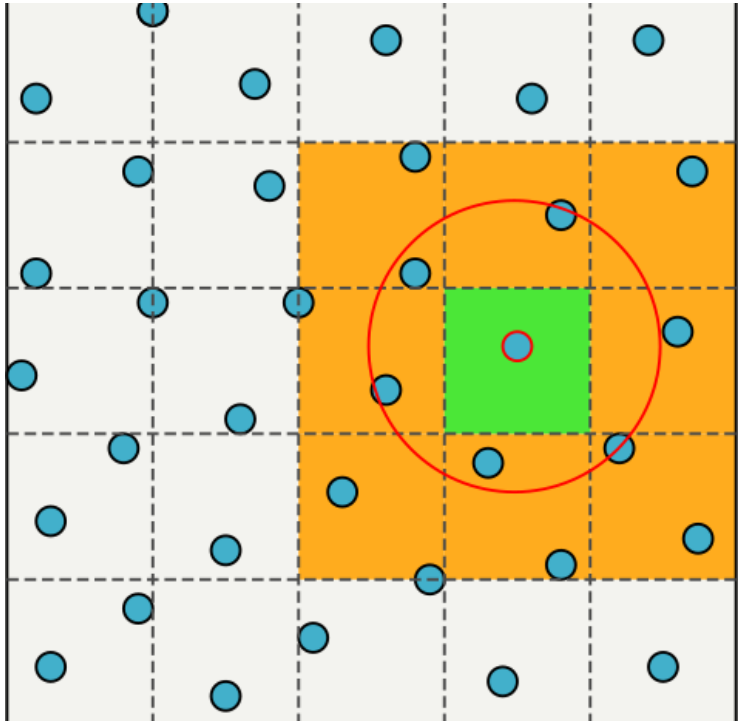
- MPI-only code.
- Neighbour search based on oct-tree.
- Oct-tree implies “random” memory walks
  - Lack of predictability.
  - Nearly impossible to vectorize.
  - Very hard to load-balance.



SPH scheme: The **SWIFT** way

Need to make things regular and predictable:

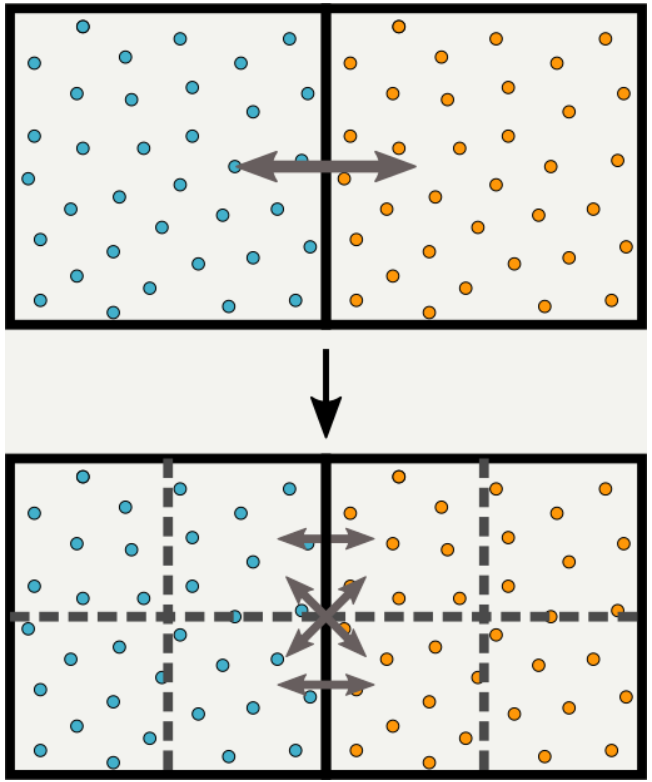
- Neighbour search is performed via the use of an adaptive grid constructed recursively until we get ~500 particles per cell.
- Cell spatial size matches search radius.
- Particles interact only with partners in their own cell or one of the 26 neighbouring cells



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

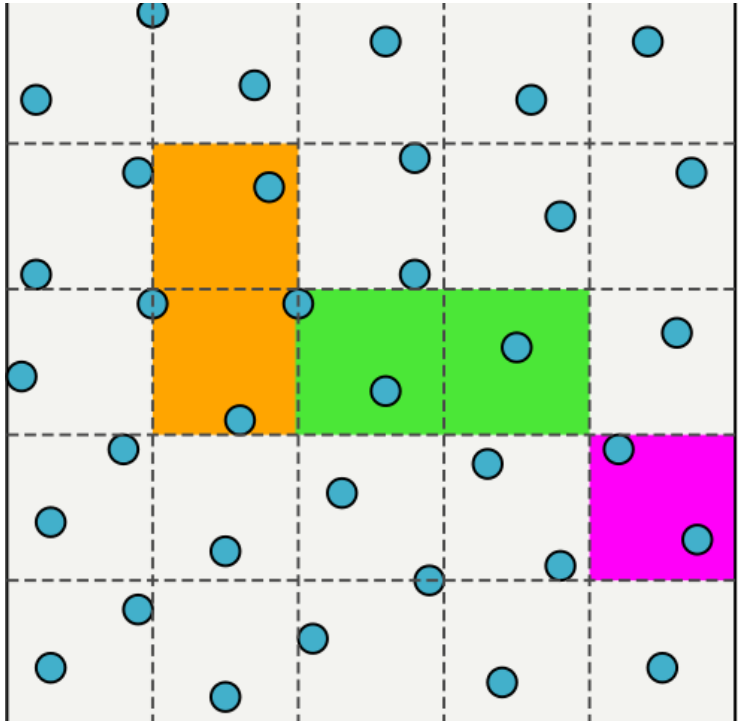
Single-node parallelisation

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.

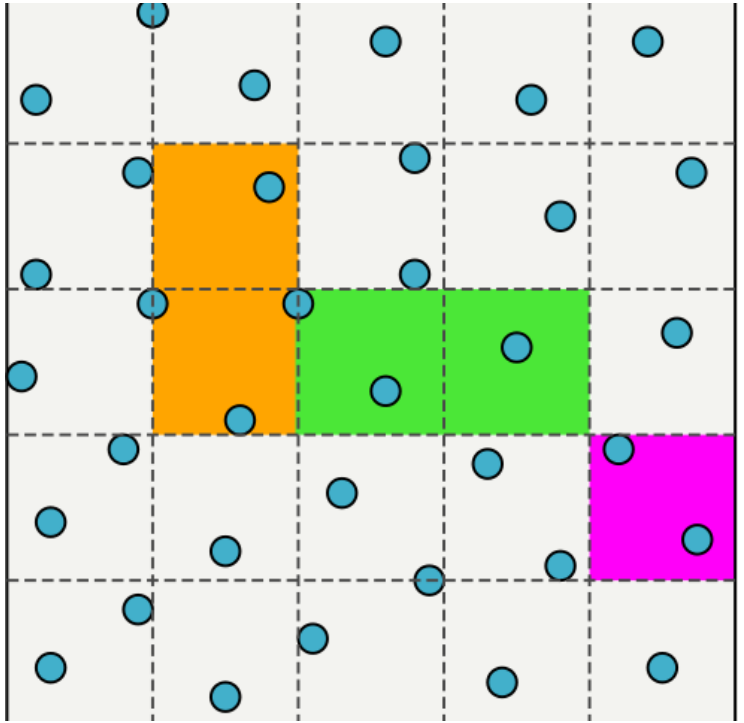


SPH scheme: Single-node parallelization

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.

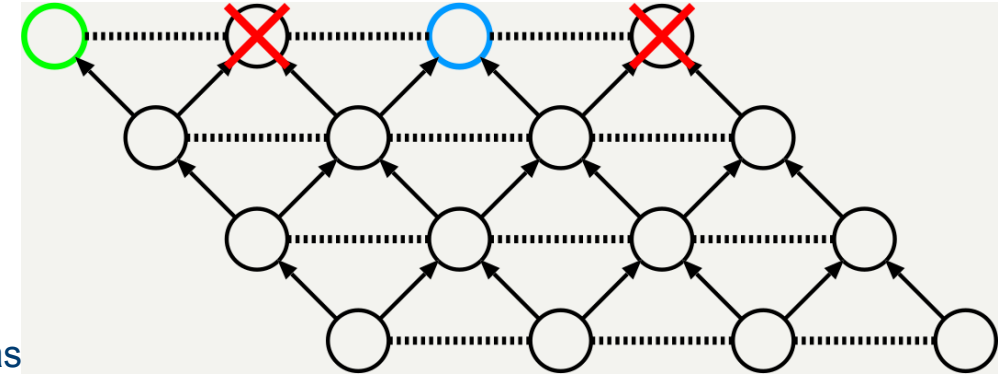
We need dynamic scheduling !



Task-base parallelism 101

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.

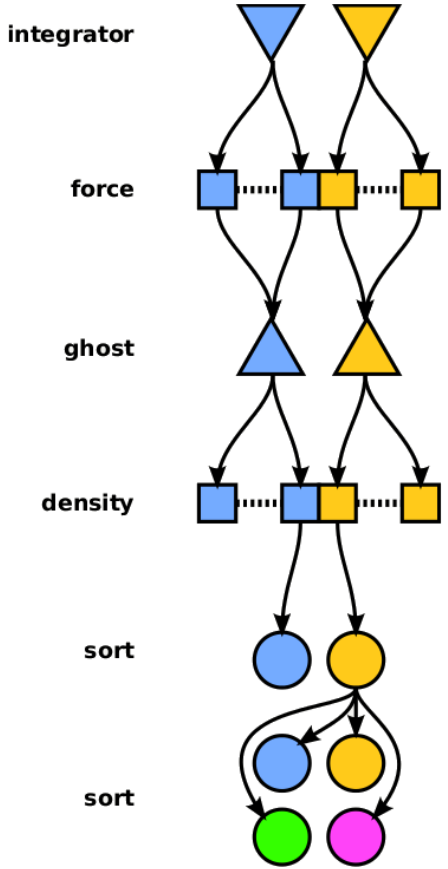
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  Which tasks it depends on,
  Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



- We use our own (problem agnostic !) Open-source library `QuickSched` (`arXiv:1601.05384`)

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.



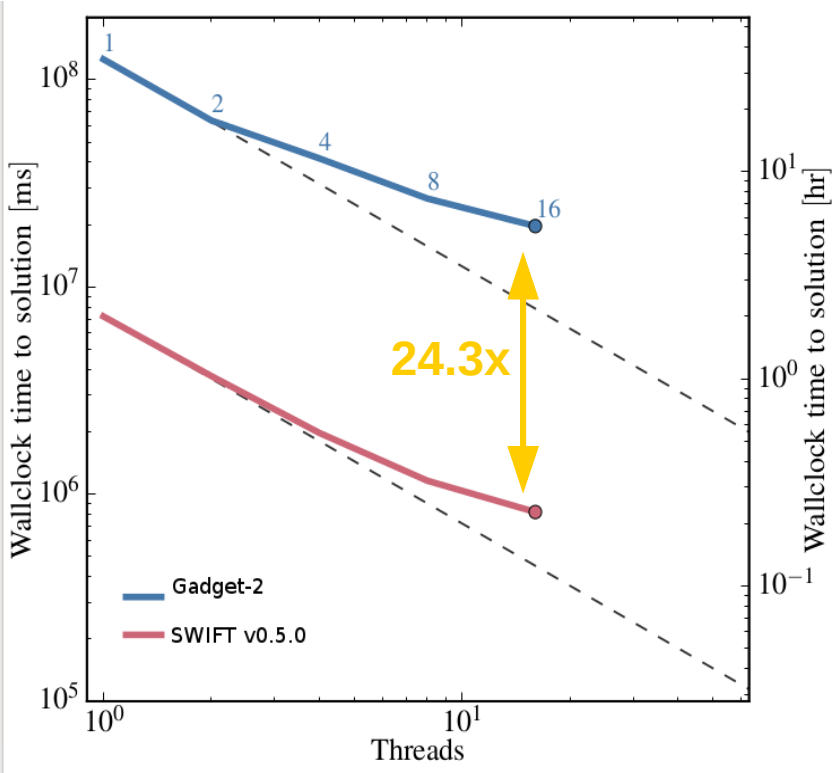
SPH scheme: Single node parallel performance

Task graph for one time-step.
Colours correspond to different types of task. Almost perfect load-balancing is achieved on 32 cores.

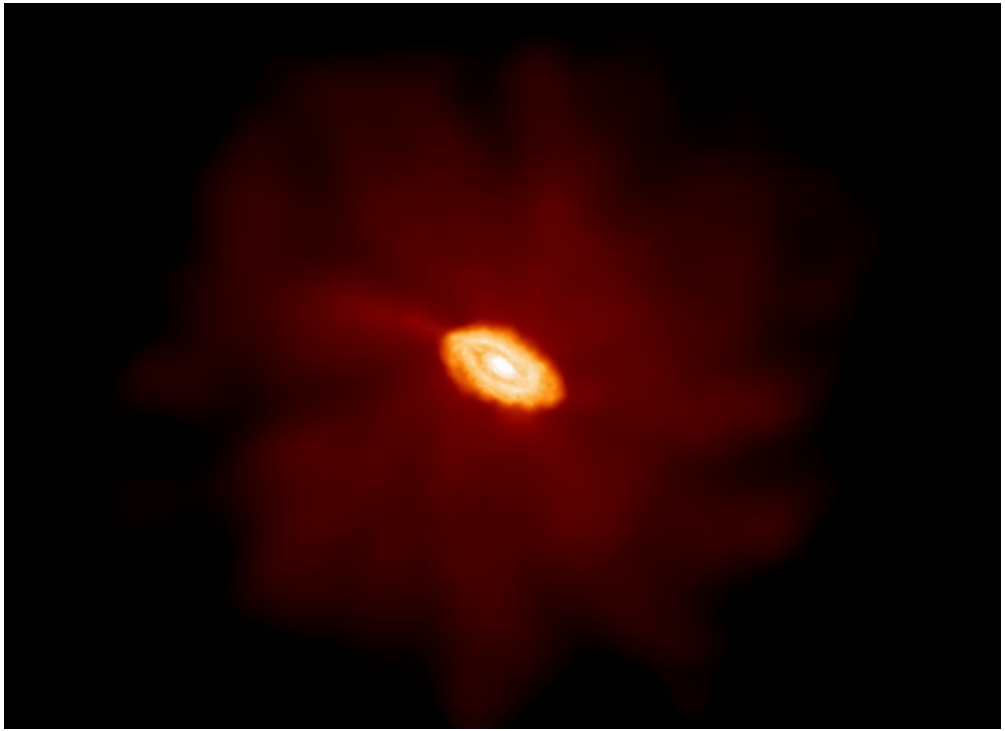
Single node performance vs. Gadget

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

More than 24x speed-up vs. “industry standard” Gadget code.



Result: Formation of a galaxy on a KNL



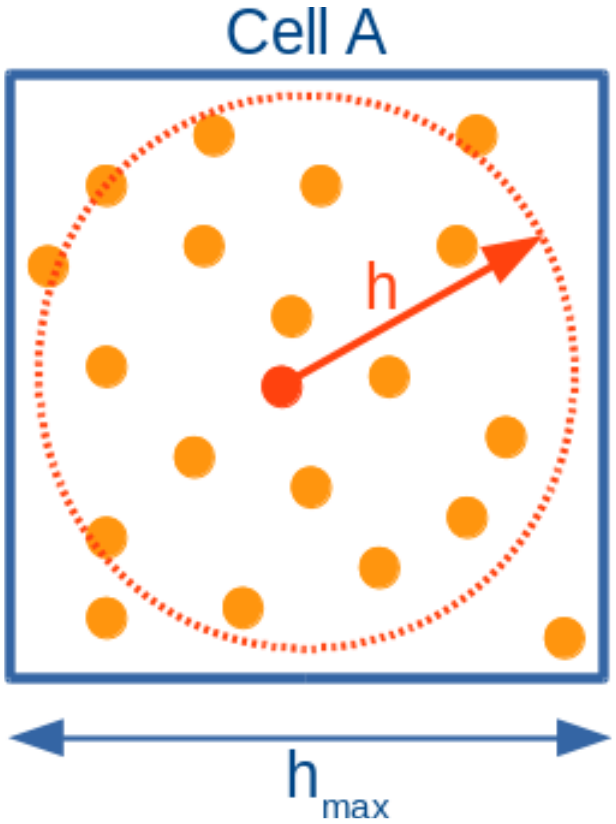
Credit: S. Arridge

SIMD parallelisation

Explicit vectorization using intrinsics

Explicit vectorization of the core routines.

Example of a task interacting all particles within one cell.



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.

Brute-force implementation

- Very simple to write
- Compilers can in principle “auto-vectorize” the whole problem.

```
for (int i = 0; i < ci->count; ++i) {

  hig2 = hi * hi * kernel_gamma2;
  for (int j = 0; j < ci->count; ++j) {

    hjg2 = hj * hj * kernel_gamma2;
    /* Check that particle doesn't interact with itself */
    if (pi == pj) continue;

    /* Pairwise distance */
    r2 = 0.0f;
    for (int k = 0; k < 3; k++) {
      dxi[k] = pi->x[k] - pj->x[k];
      r2 += dxi[k] * dxi[k];
    }

    /* Update pi? */
    if (r2 < hig2) INTERACT(r2, dxi, hi, hj, pi, pj);

    /* Update pj? */
    if (r2 < hjg2) INTERACT(r2, -dxi, hj, hi, pj, pi);
  }
}
```

Brute-force implementation

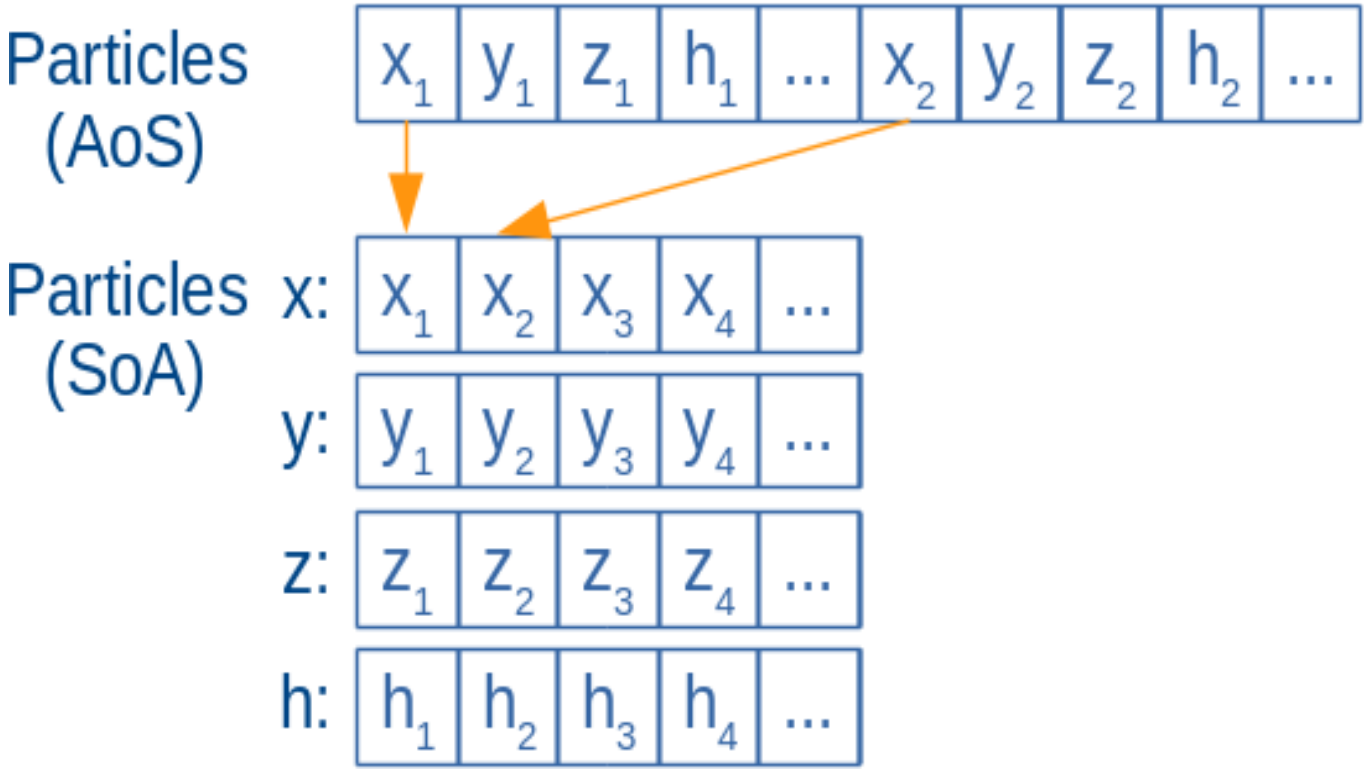
- Very simple to write
- Compilers can in principle “auto-vectorize” the whole problem.

... But most pairs of particles will not interact....

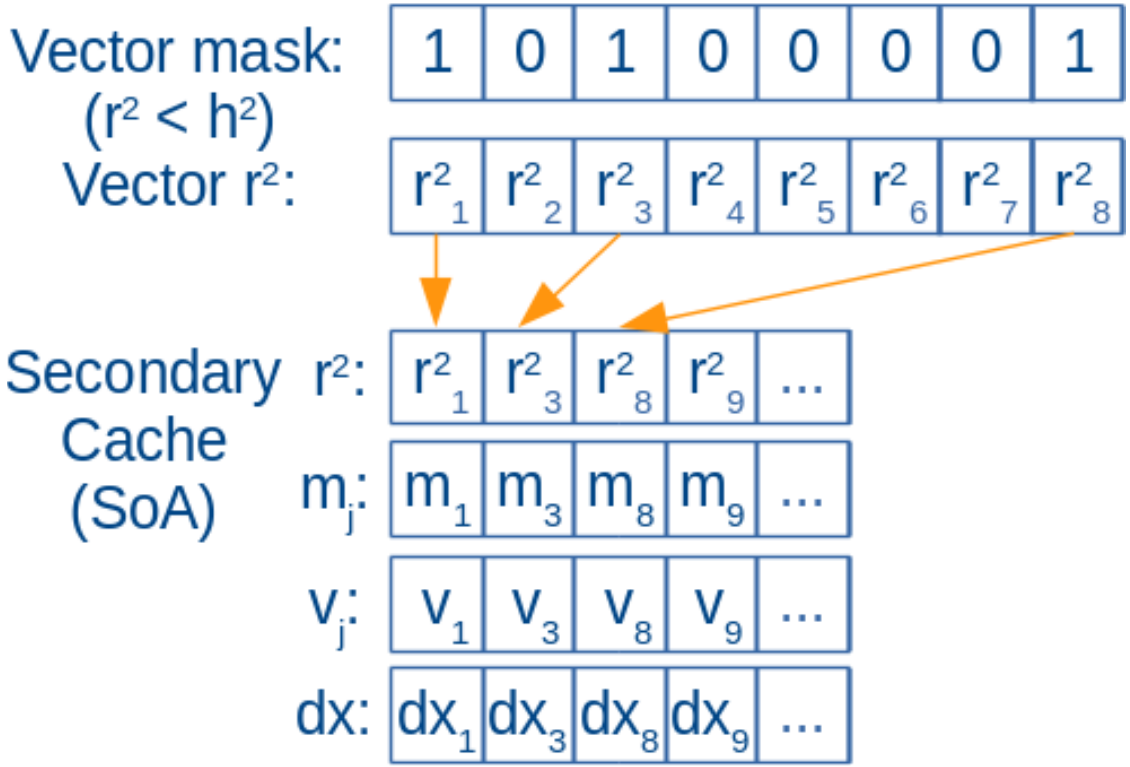
Need to manually implement a better solution

```
for (int i = 0; i < ci->count; ++i) {

  hig2 = hi * hi * kernel_gamma2;
  for (int j = 0; j < ci->count; ++j) {

    hjg2 = hj * hj * kernel_gamma2;
    /* Check that particle doesn't interact with itself */
    if (pi == pj) continue;

    /* Pairwise distance */
    r2 = 0.0f;
    for (int k = 0; k < 3; k++) {
      dxi[k] = pi->x[k] - pj->x[k];
      r2 += dxi[k] * dxi[k];
    }

    /* Update pi? */
    if (r2 < hig2) INTERACT(r2, dxi, hi, hj, pi, pj);

    /* Update pj? */
    if (r2 < hjg2) INTERACT(r2, -dxi, hj, hi, pj, pi);
  }
}
```

Step 1: Form a local cache of particles



Step 2: Find pairs and pack them in a 2nd cache



Step 3: Process all pairs in the 2nd cache

```
vector densitySum;
density = setzero();

for (int pjd = 0; pjd < icount; pjd+=VEC_SIZE) {
  INTERACT(&c2_r2[pjd], &c2_dx[pjd], &c2_dy[pjd],
           &c2_dz[pjd], &c2_m[pjd], &c2_v[pjd],
           &densitySum);
}

VEC_HADD(densitySum,pi);
```

Vectorization results

CFLAGS	Speed-up over naïve brute force	Speed-up over best serial version
-O3 **-xAVX**	**2.93x**	1.94x
-O3 **-xCORE-AVX2**	**3.64x**	2.74x
-O3 **-xMIC-AVX512**	**4.37x**	2.80x

Xeon ~ 2011

Xeon ~ 2014

KNL 2016
Xeon 2017

Better than the factor of 2x obtained from the auto-vectorizer

In the scalar case, there is a faster algorithm with the comparison shown here for fairness

Software Development Best Practices

A few tips from experience

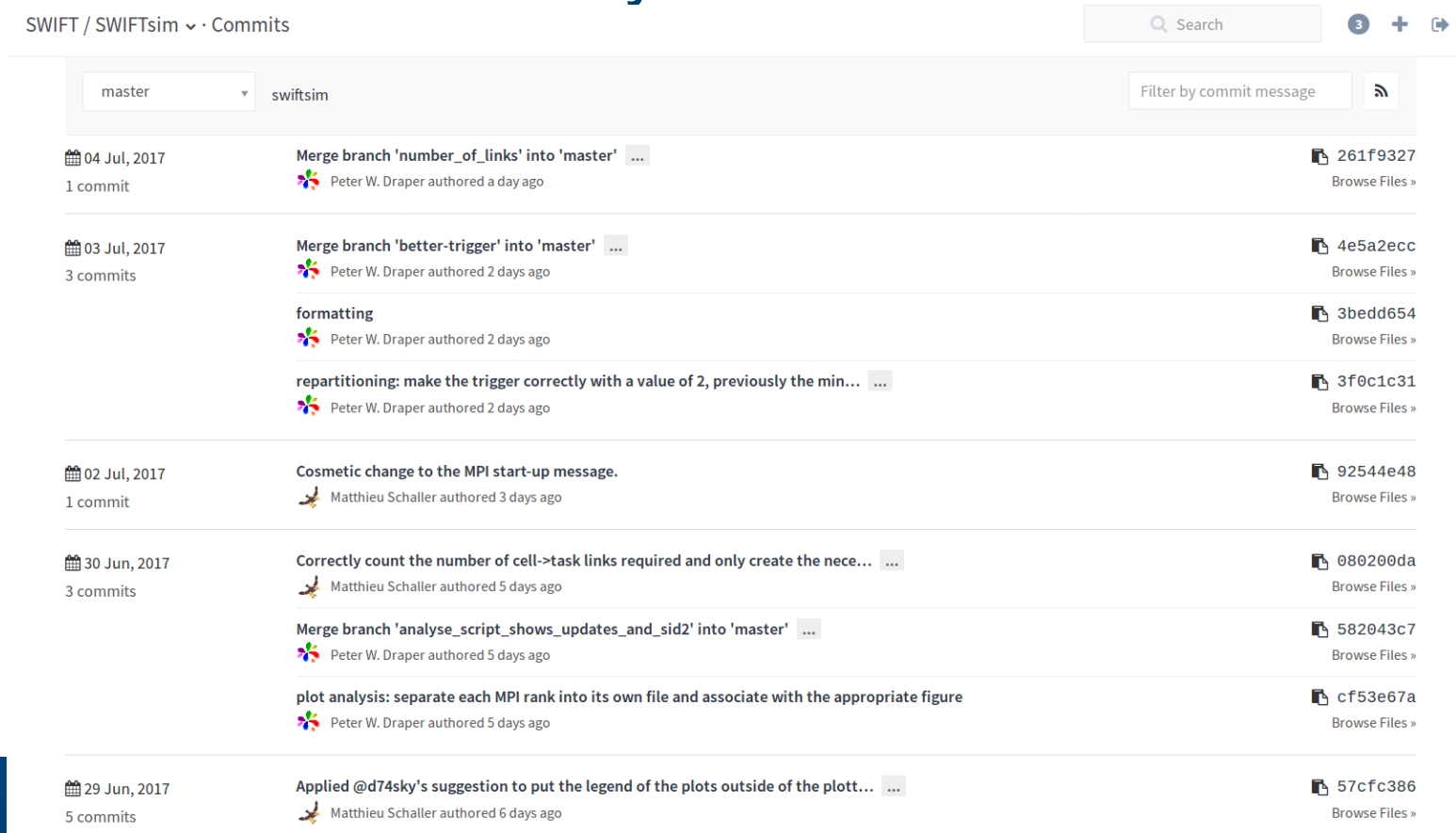
Use a version control system and repository

- Allows to store your work at many stages.
- Allows to roll back to older versions.
- Allows to search for the point where bugs were introduced.
- Allows branching/forking.

Examples: git, svn, mercurial,...

Many online platforms offer free services

Use a version control system



Use automated (unit) tests

- Allows to track unspotted bugs.
- Allows to track regression.
- Secures the stability of the software.
- Run daily or at each commit.

Examples: jenkins, travis,…

Again, many online platforms linked to repositories

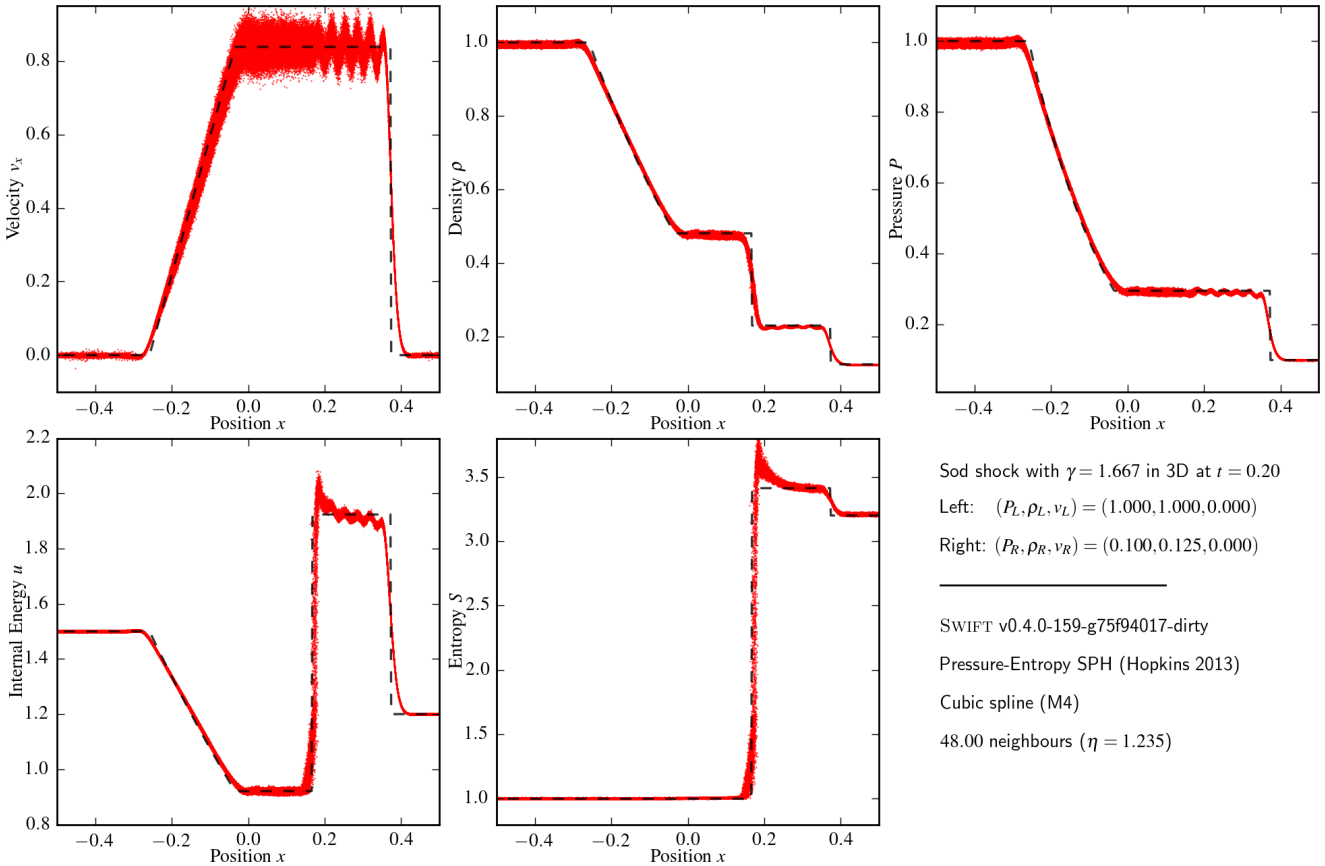
Use automated (unit) tests

- Allows to track unspotted bugs.
- Allows to track regression.
- Secures the stability of the software.
- Run daily or at each commit.

Examples: jenkins, travis,…

Again, many online platforms linked to repositories

Use automated (unit) tests



Sod shock with $\gamma = 1.667$ in 3D at $t = 0.20$

Left: $(P_L, \rho_L, v_L) = (1.000, 1.000, 0.000)$

Right: $(P_R, \rho_R, v_R) = (0.100, 0.125, 0.000)$

SWIFT v0.4.0-159-g75f94017-dirty

Pressure-Entropy SPH (Hopkins 2013)

Cubic spline (M4)

48.00 neighbours ($\eta = 1.235$)

Documentation

- Necessary to keep track of code.
- Make sure *you* understand what you wrote.
- Very important when working in teams.
- Many formats and automated software.

Examples: doxygen, readthedocs,...

Again, many online platforms linked to repositories

Documentation

```
/**
 * @brief Constructs the top-level pair tasks for the first hydro loop over
 * neighbours
 *
 * Here we construct all the tasks for all possible neighbouring non-empty
 * local cells in the hierarchy. No dependencies are being added thus far.
 * Additional loop over neighbours can later be added by simply duplicating
 * all the tasks created by this function.
 *
 * @param e The #engine.
 */
void engine_make_hydroloop_tasks(struct engine *e) {...}
```

```
/**...
__attribute__((always_inline)) INLINE static int cell_need_rebuild_for_pair(
    const struct cell *ci, const struct cell *cj) {

  /* Is the cut-off radius plus the max distance the parts in both cells have */
  /* moved larger than the cell size ? */
  /* Note ci->dmin == cj->dmin */
  return (kernel_gamma * max(ci->h_max, cj->h_max) + ci->dx_max_part +
              cj->dx_max_part >
          cj->dmin);
}
```

Conclusions

And take-away messages

More on SWIFT

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

~30’000 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

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

~30’000 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.

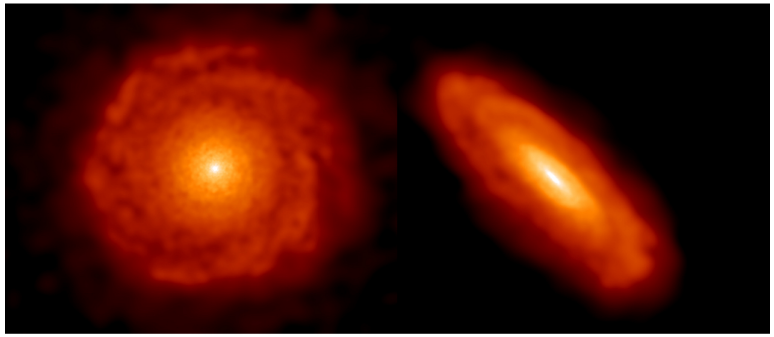
More on SWIFT

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.

Future: Addition of more physics to the code.

Future: Parallelisation of i/o.

Thank you for your time

Matthieu Schaller

www.swiftsim.org