INTEL® HPC DEVELOPER CONFERENCE

FUEL YOUR INSIGHT

SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication and Vectorization to achieve maximal HPC performance

Matthieu Schaller

Research Assistant

Institute for Computational Cosmology, Durham University, UK

November 2016

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 Ghent (Belgium), 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 = 19.8
t = 0.2 Gyr
L = 25.0 cMpc
Simulation by the EAGLE collaboration

What we do and how we do it

- Solve coupled equations of gravity and 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.

Very similar to molecular dynamics but requires two loops over the neighbours.

Challenges:

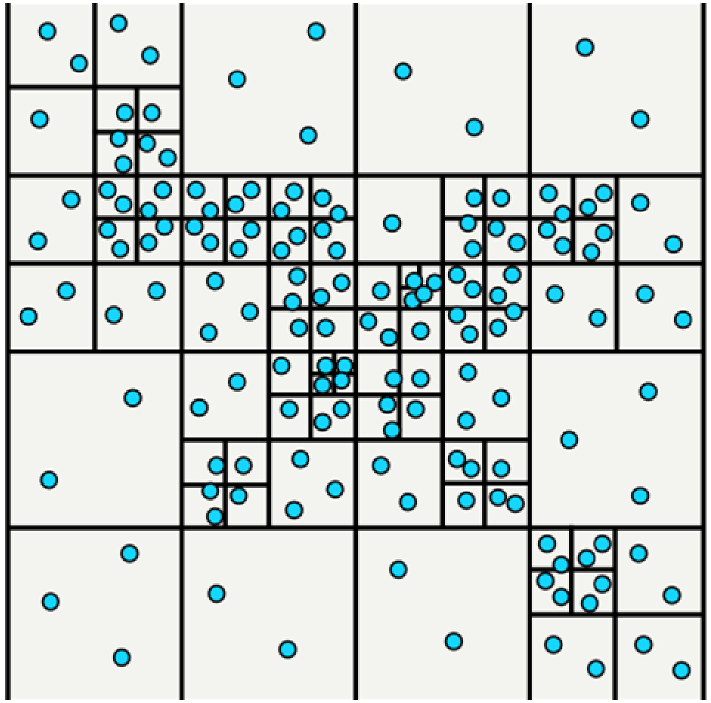
- Particles are unstructured in space, large density variations.
- Particles will move and the neighbour list of each particle evolves over time.
- Interaction between two particles is computationally cheap (low flop/byte ratio).

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.



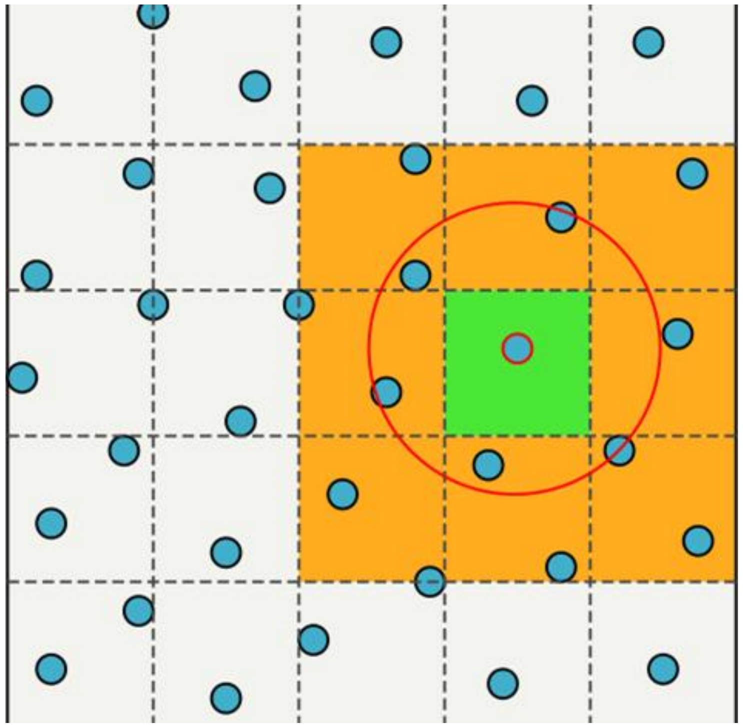
SPH scheme: The traditional method

```
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:

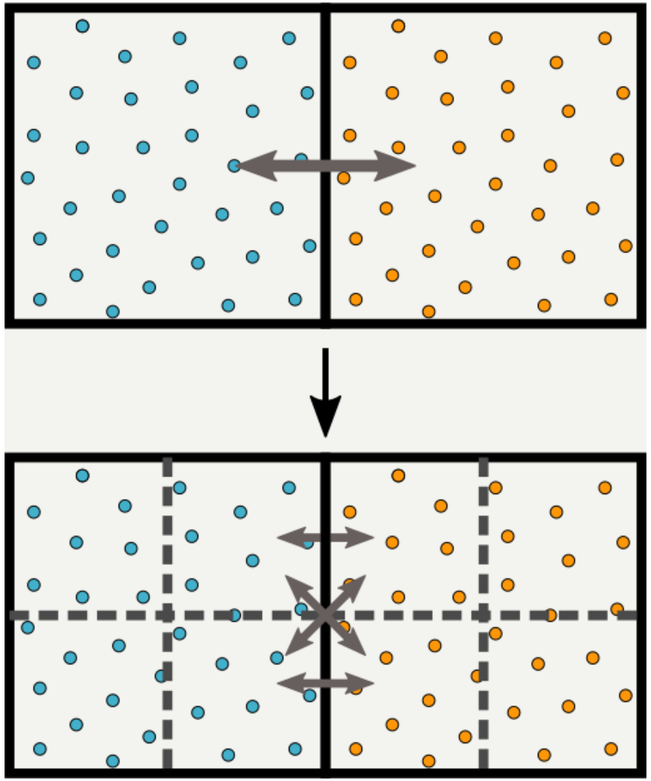
- Neighbour search is performed via the use of an adaptive grid constructed recursively until we get ~500 particles per cell.
- Cell spatial size matches search radius.
- Particles interact only with partners in their own cell or one of the 26 neighbouring cells



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
     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
} } } }
```

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

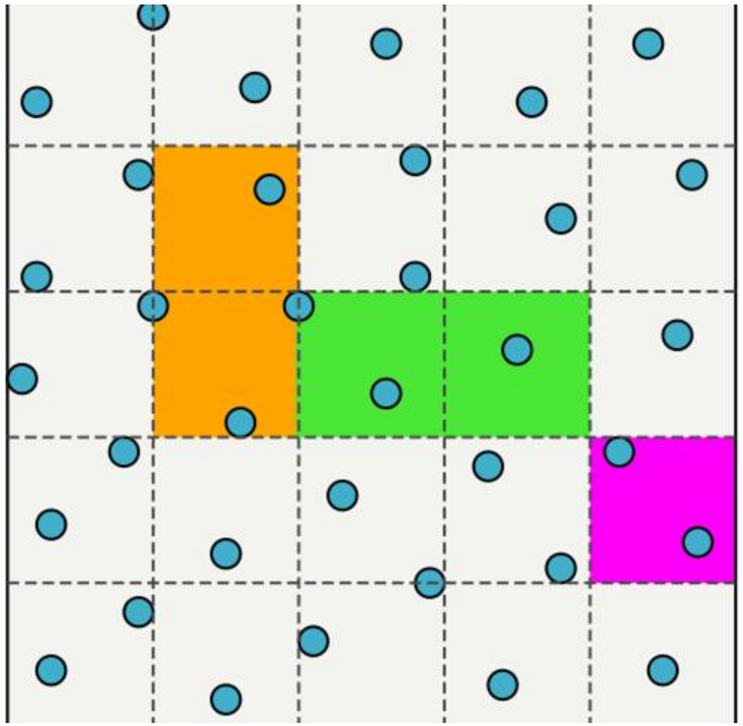
Single-node parallelisation

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.

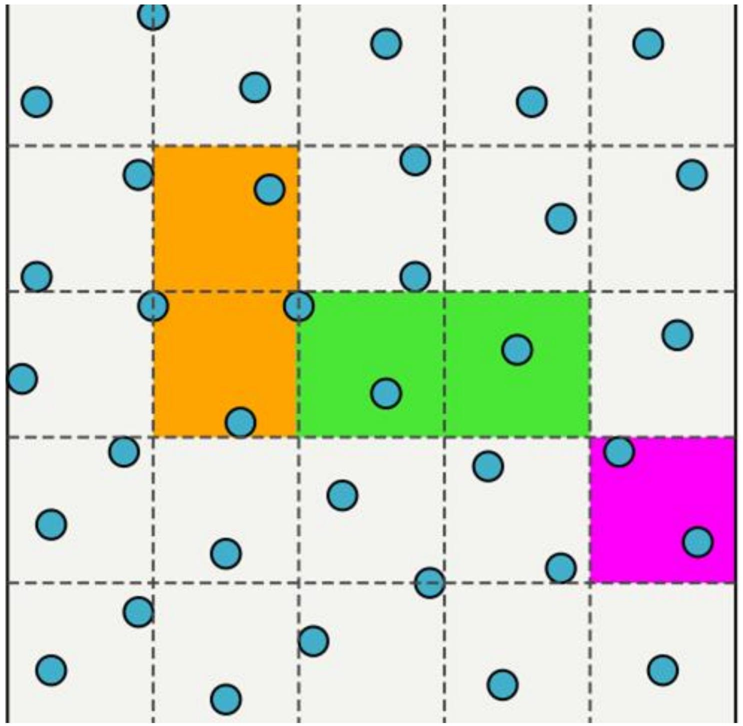


SPH scheme: Single-node parallelization

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

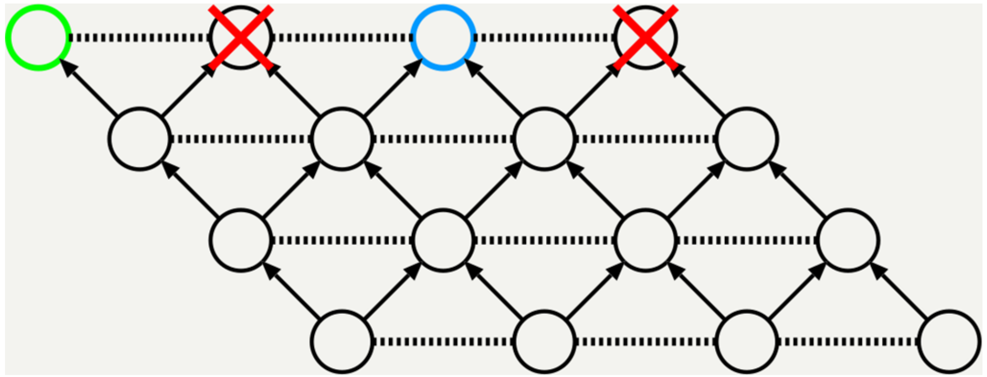
- -> No need to enforce and order.
- -> Only need to make sure we don't process pairs that use the same cell.
- -> Pairs could have vastly different runtimes since they can have very different particle numbers.

We need dynamic scheduling !



Task-base parallelism 101

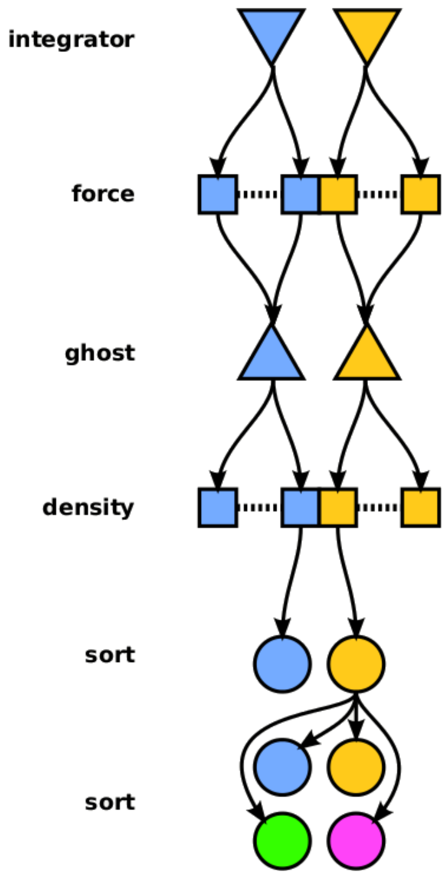
Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.



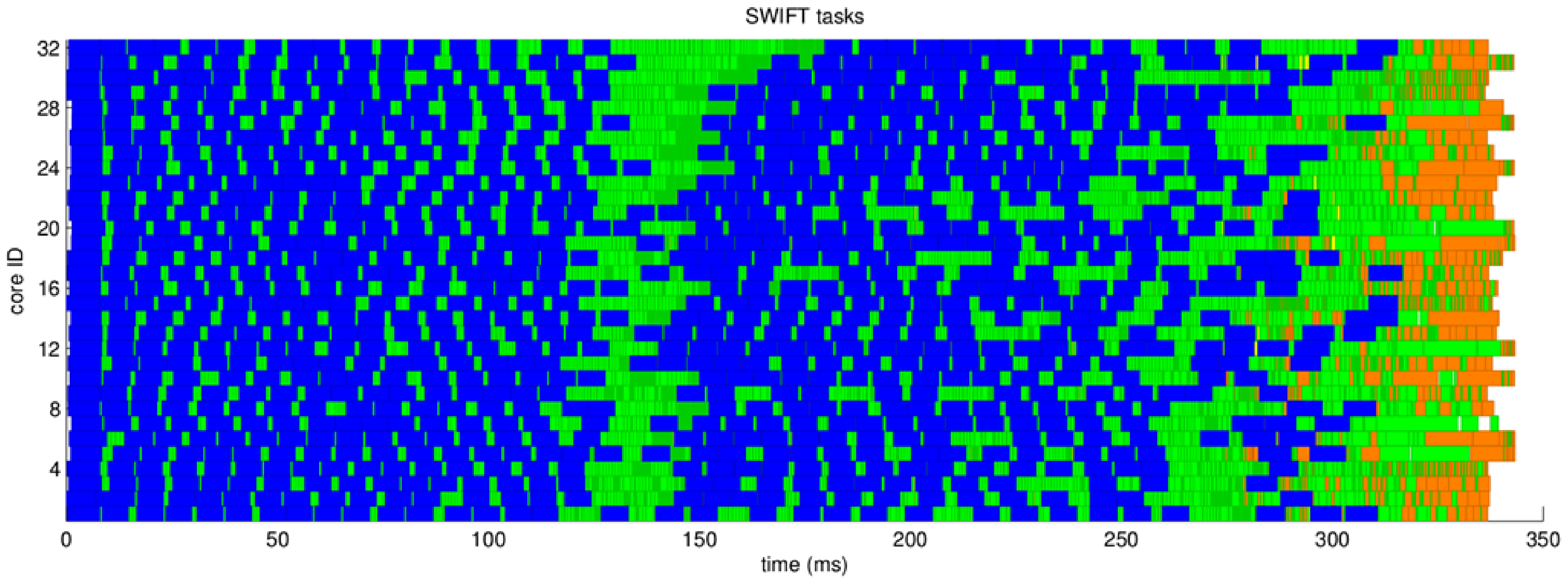
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  Which tasks it depends on,
  Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.
- We use our own (problem agnostic !)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

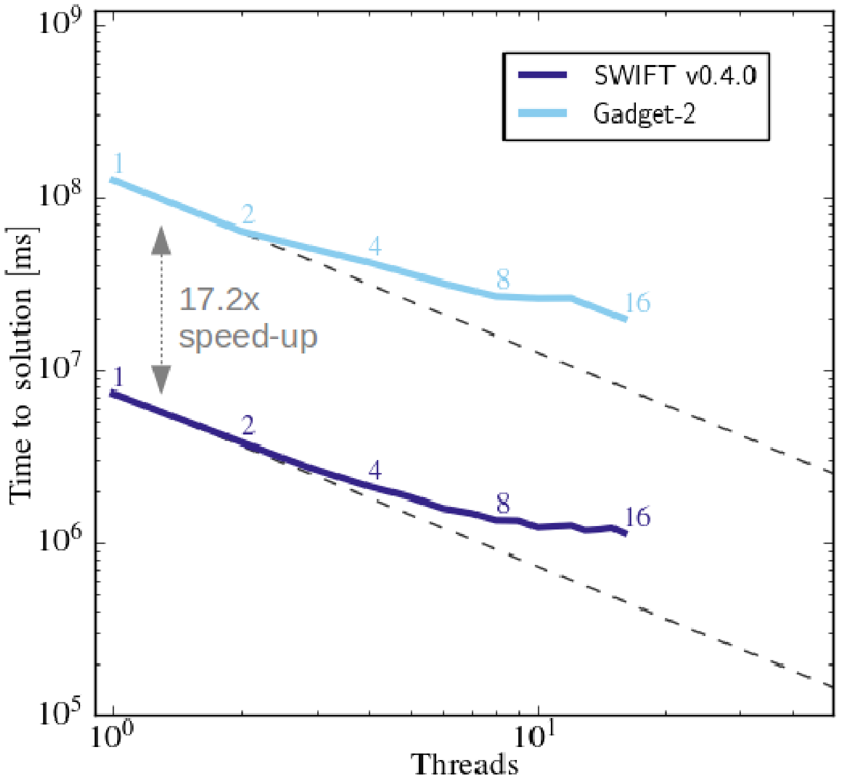


Task graph for one time-step. Colours correspond to different types of task. Almost perfect load-balancing is achieved on 32 cores.

Single node performance vs. Gadget

- Realistic problem (video from start of the talk)
- Same accuracy.
- Same hardware.
- Same compiler (no vectorization here).
- Same solution.

More than 17x speed-up vs. “industry standard” Gadget code.

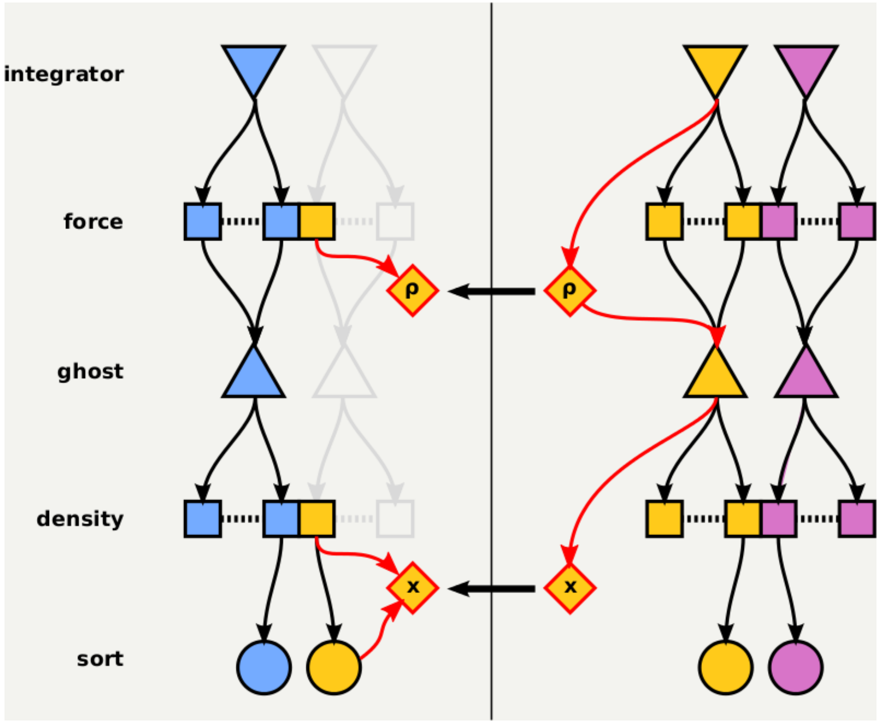


Multi-node parallelisation

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.

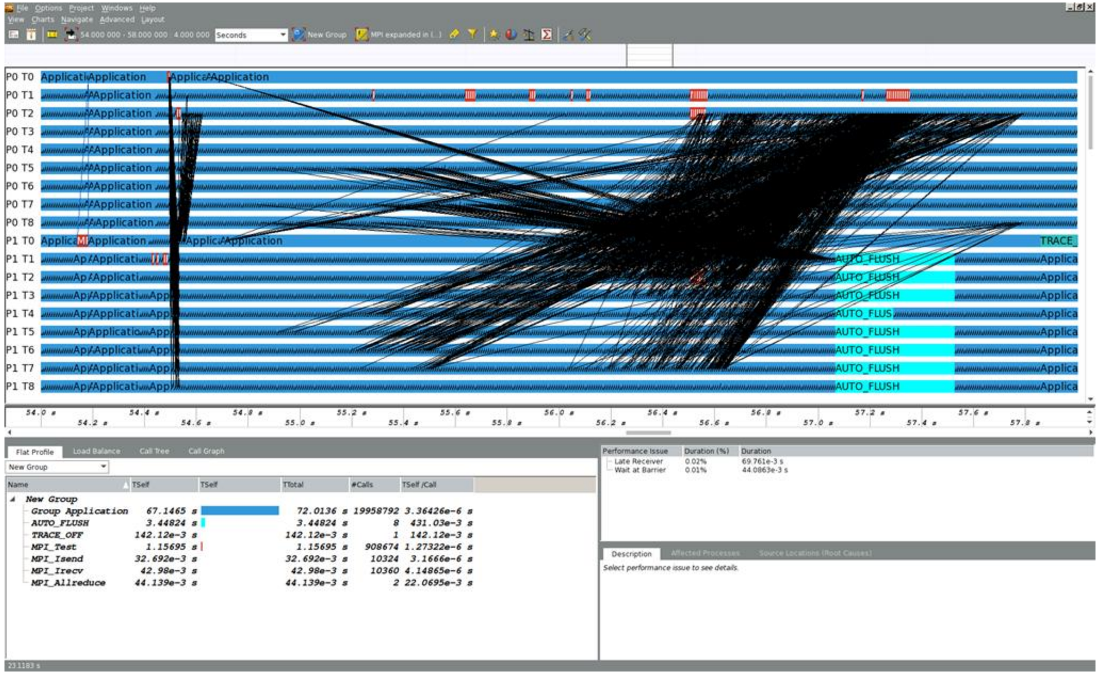
Not all MPI implementations fully support the MPI v3.0 standard.

- Uncovered several bugs in different implementations providing `MPI_THREAD_MULTIPLE`.
- e.g.: OpenMPI 1.x crashes when running on Infiniband!

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.



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

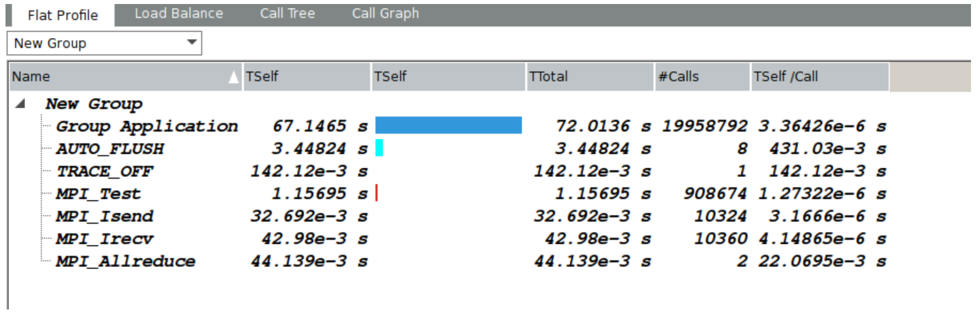
Performance Issue	Duration (%)	Duration
Late Receiver	0.02%	69.761e-3 s
Wait at Barrier	0.01%	44.0863e-3 s

Select performance issue to see details.

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.

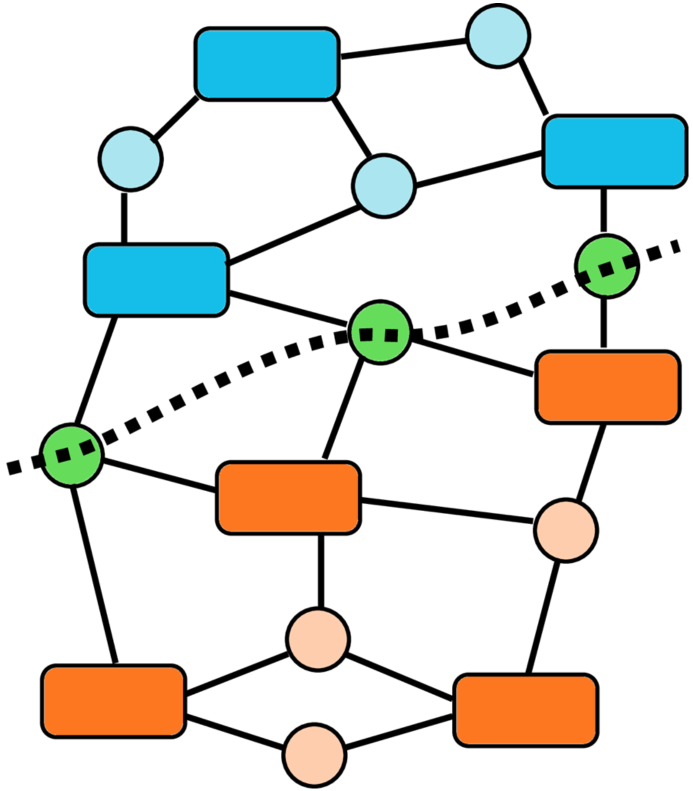


Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
New Group					
Group Application	67.1465 s		72.0136 s	19958792	3.36426e-6 s
AUTO_FLUSH	3.44824 s		3.44824 s	8	431.03e-3 s
TRACE_OFF	142.12e-3 s		142.12e-3 s	1	142.12e-3 s
MPI_Test	1.15695 s		1.15695 s	908674	1.27322e-6 s
MPI_Isend	32.692e-3 s		32.692e-3 s	10324	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 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.



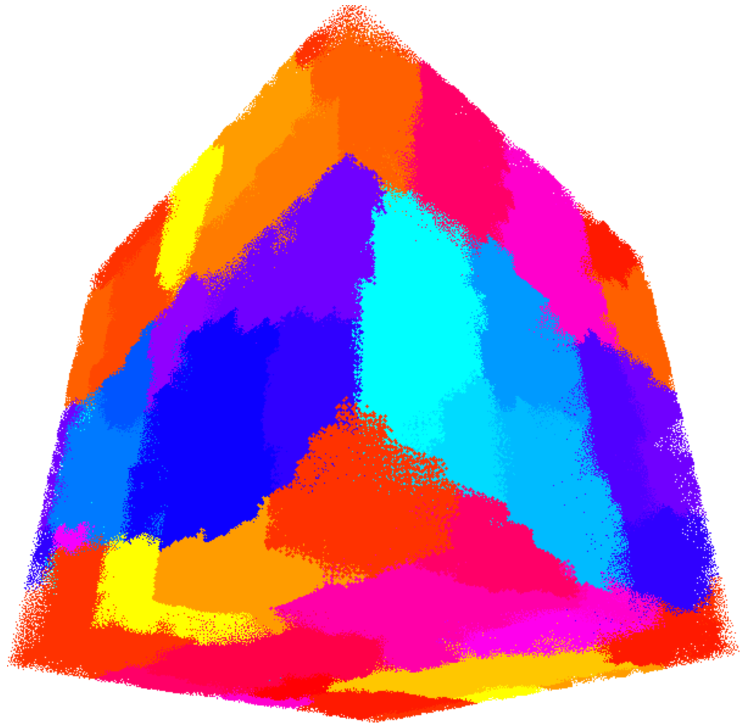
Domain decomposition

Domain geometry can be complex.

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

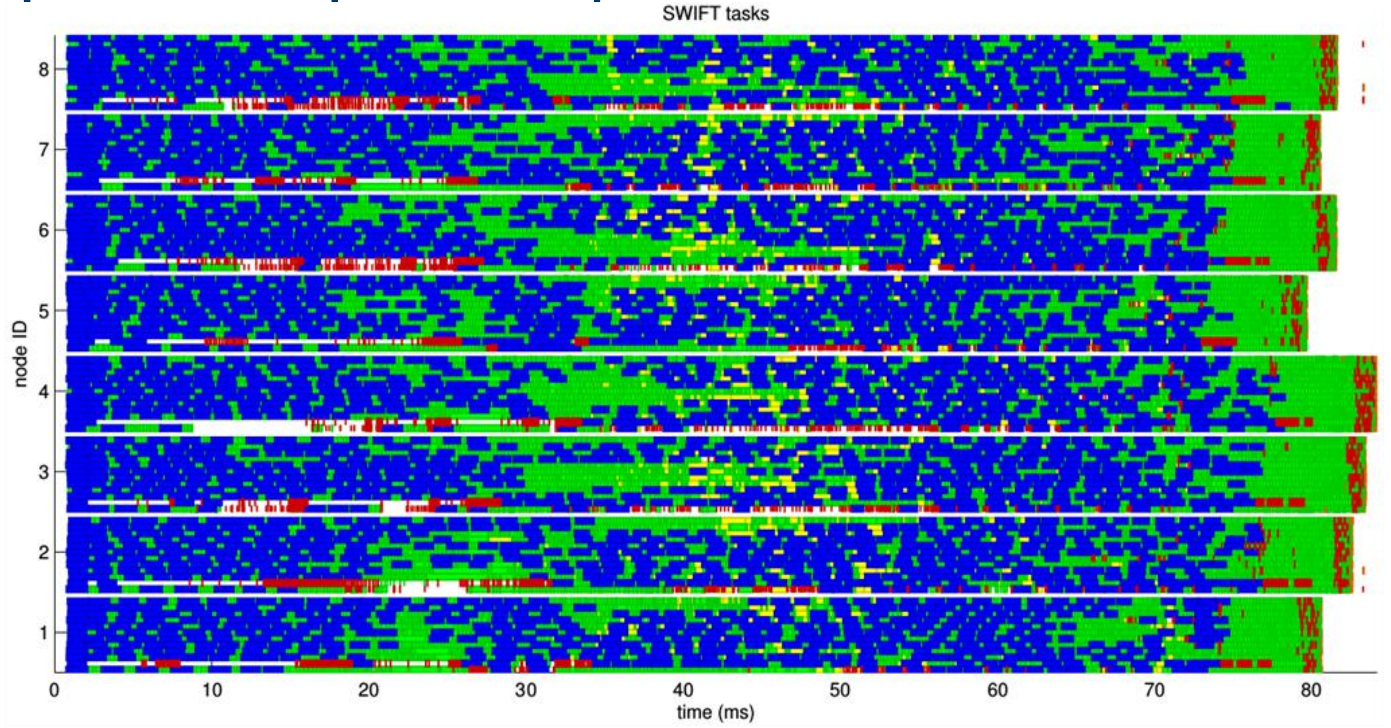
Domain shapes and computational costs evolve over the course of the simulation.

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



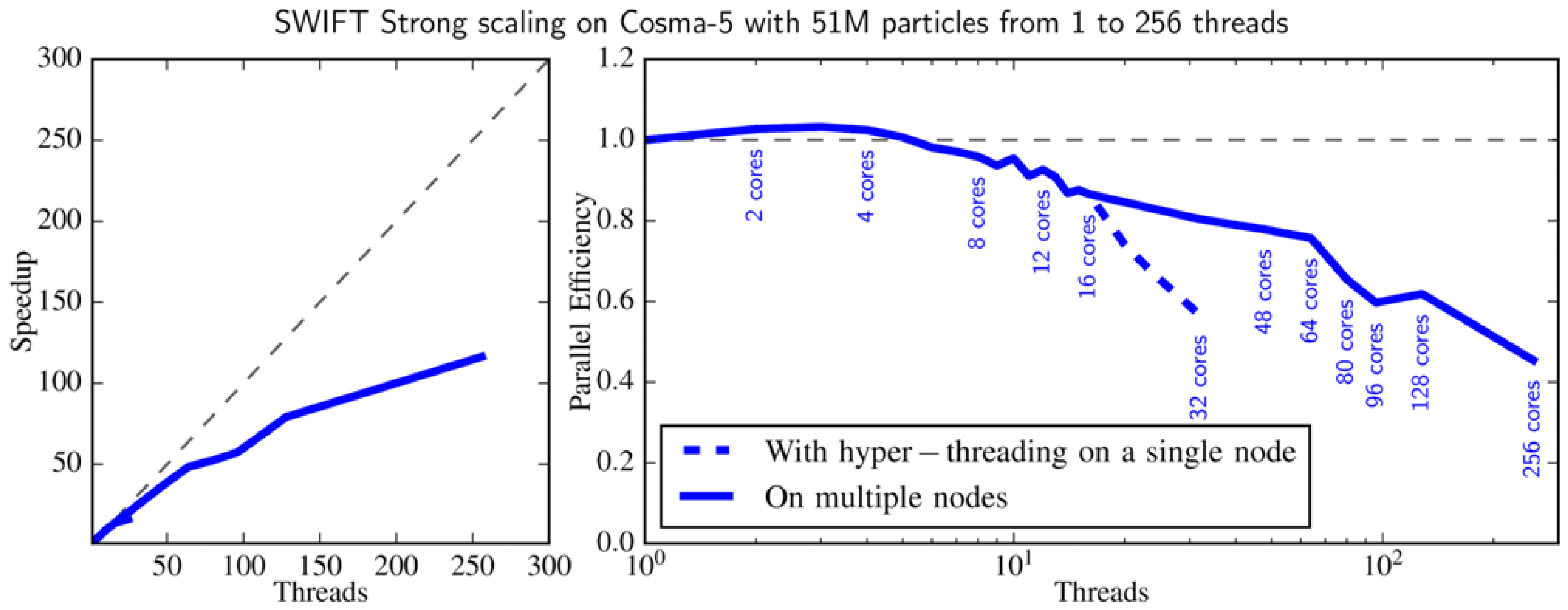
Particles coloured by the domain they belong to for a cosmological simulation.
The domains are un-structured.

Multiple node parallel performance



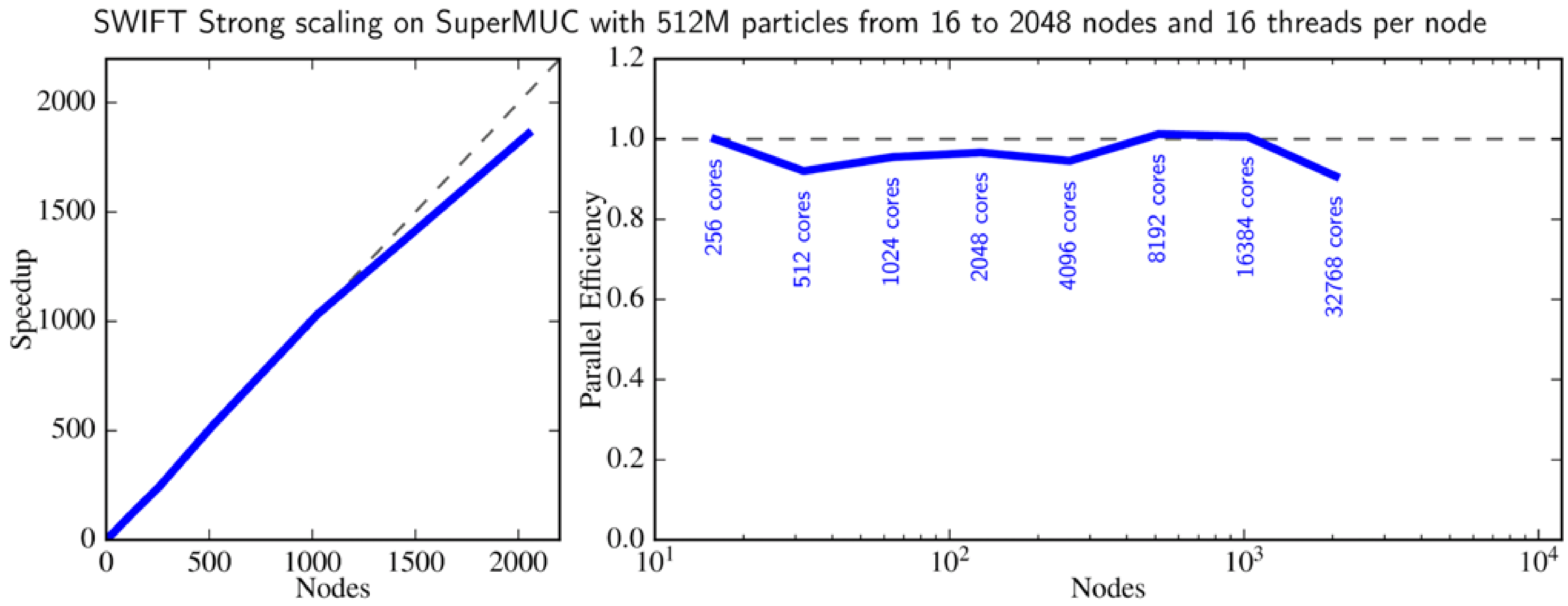
Task graph for one time-step. Red and yellow are MPI tasks. Almost perfect load-balancing is achieved on 8 nodes of 12 cores.

Scaling results: DiRAC Data Centric facility *Cosma-5*



System: x86 architecture - 2 Intel Sandy Bridge-EP Xeon E5-2670 at 2.6 GHz with 128 GByte of RAM per node.

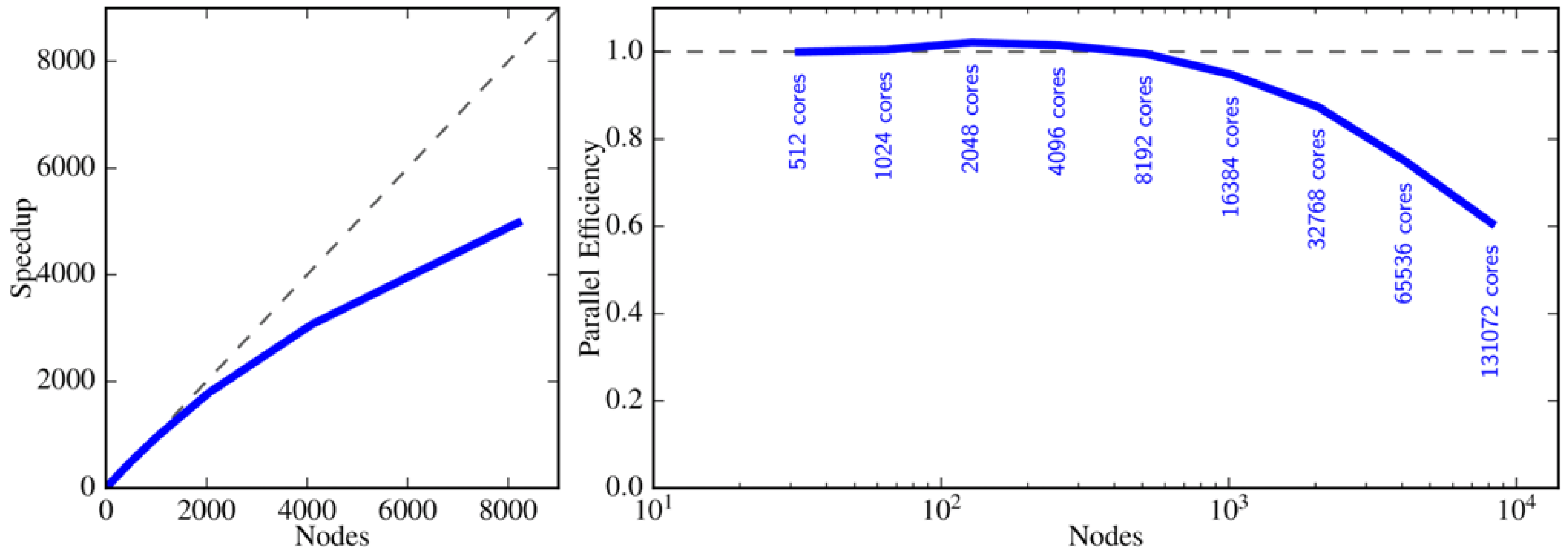
Scaling results: SuperMUC (#22 in Top500)



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

Scaling results: JUQUEEN (#11 in Top500)

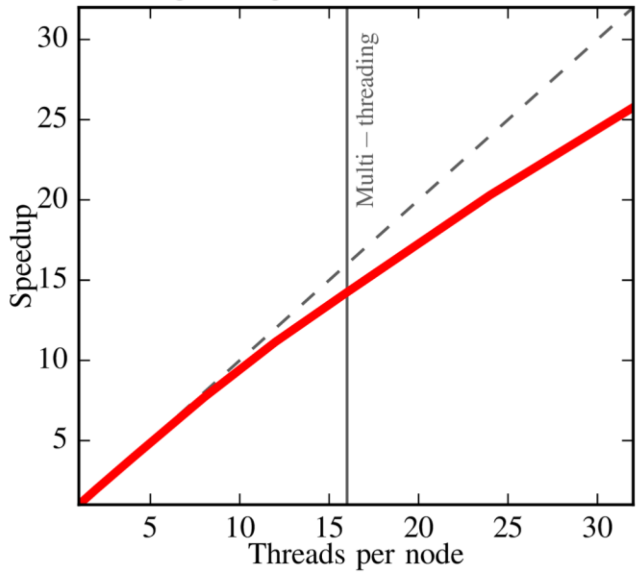
SWIFT Strong scaling on JUQUEEN with 216M particles from 32 to 8192 nodes and 32 threads per node



System: BlueGene Q - IBM PowerPC A2 processors running at 1.6 GHz with 16 GByte of RAM per node.

Scaling results

- 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

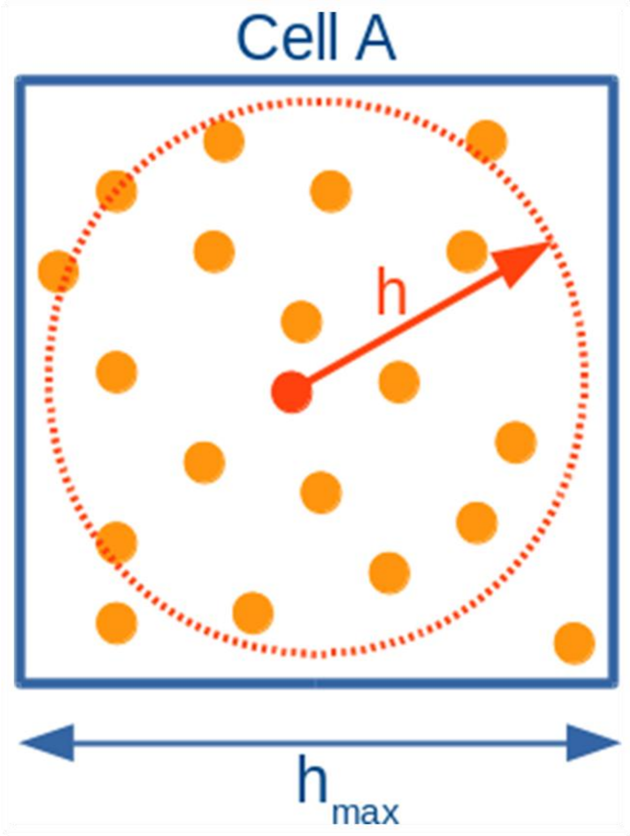
Explicit vectorization using intrinsics

Explicit vectorization of the core routines.

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....

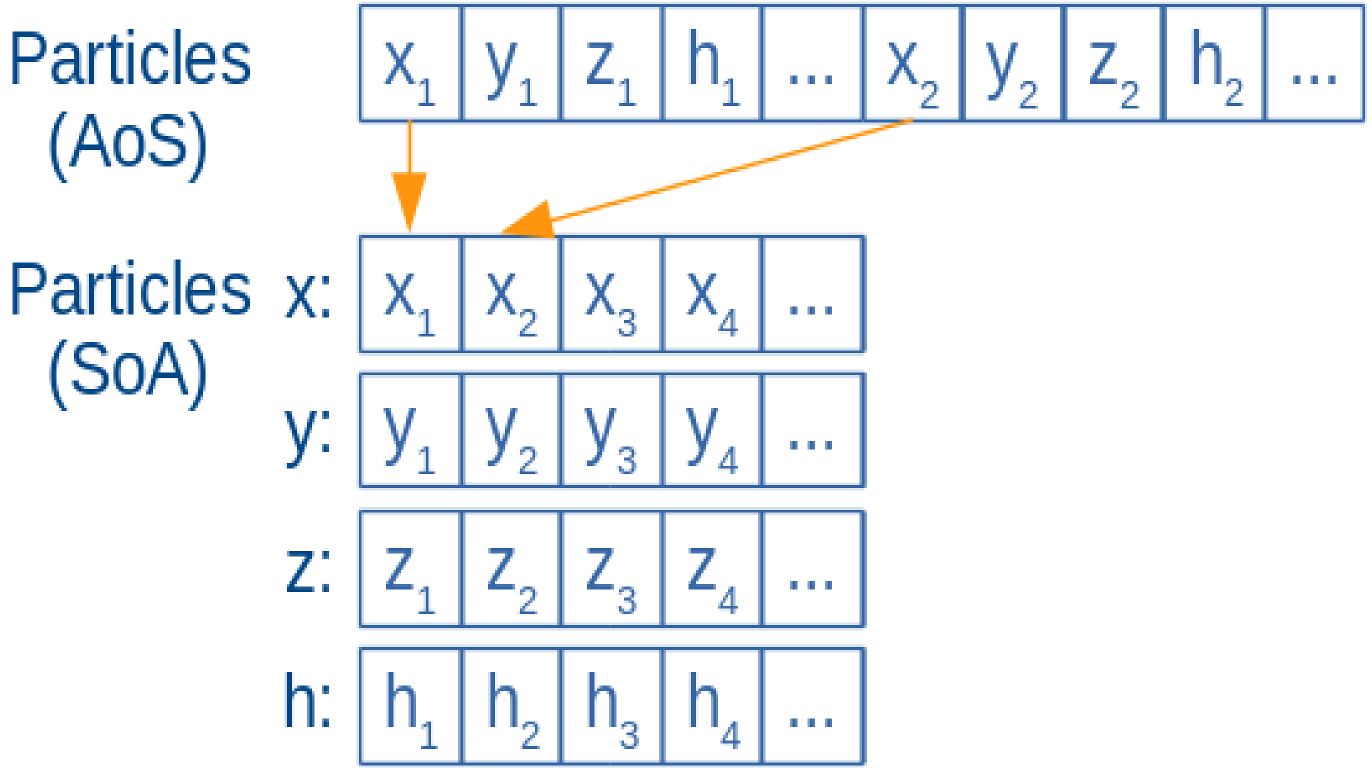
Need to manually implement a better solution

```
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);
  }
}
```

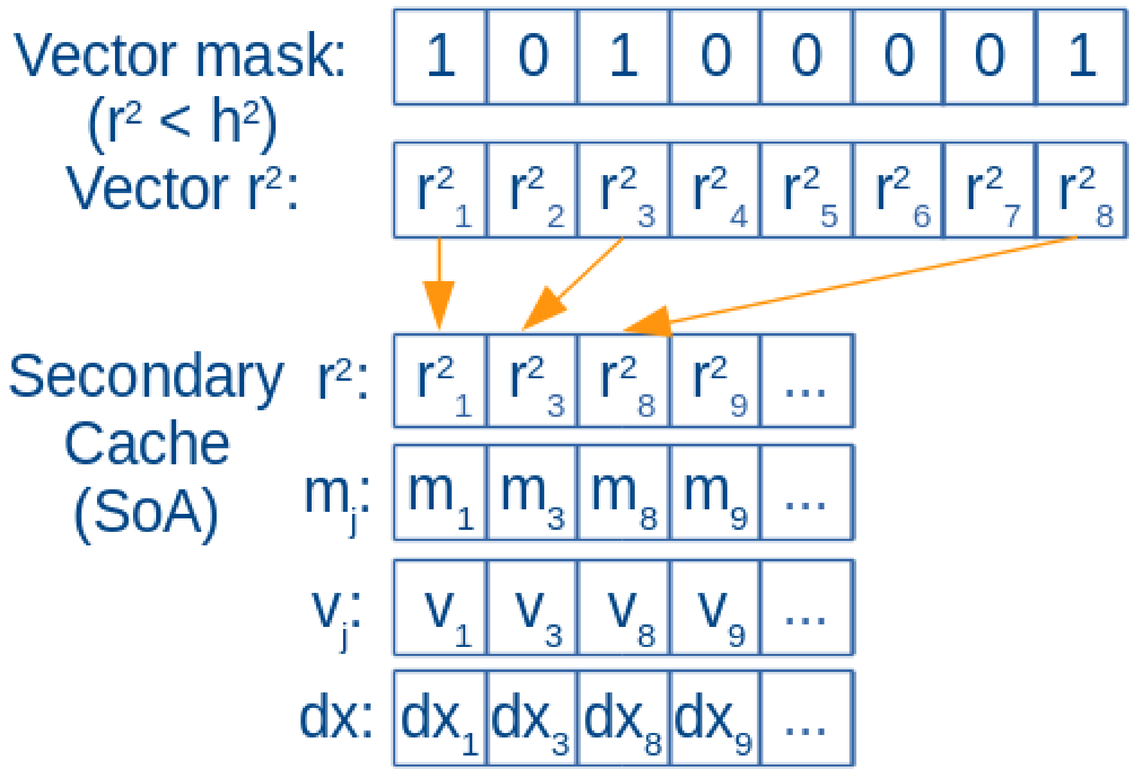
Explicit vectorization: strategy

- Use local particle cache
- Find particles that interact and store them in a secondary cache
- Calculate all interactions on a particle and store results in a set of intermediate vectors
- Perform horizontal add on intermediate vectors and update the particle with the result
- Process 2 vectors at a time when entering the interaction loop in order to overlap independent instructions
- Pad caches to prevent remainders and mask out the result

Step 1: Form a local cache of particles

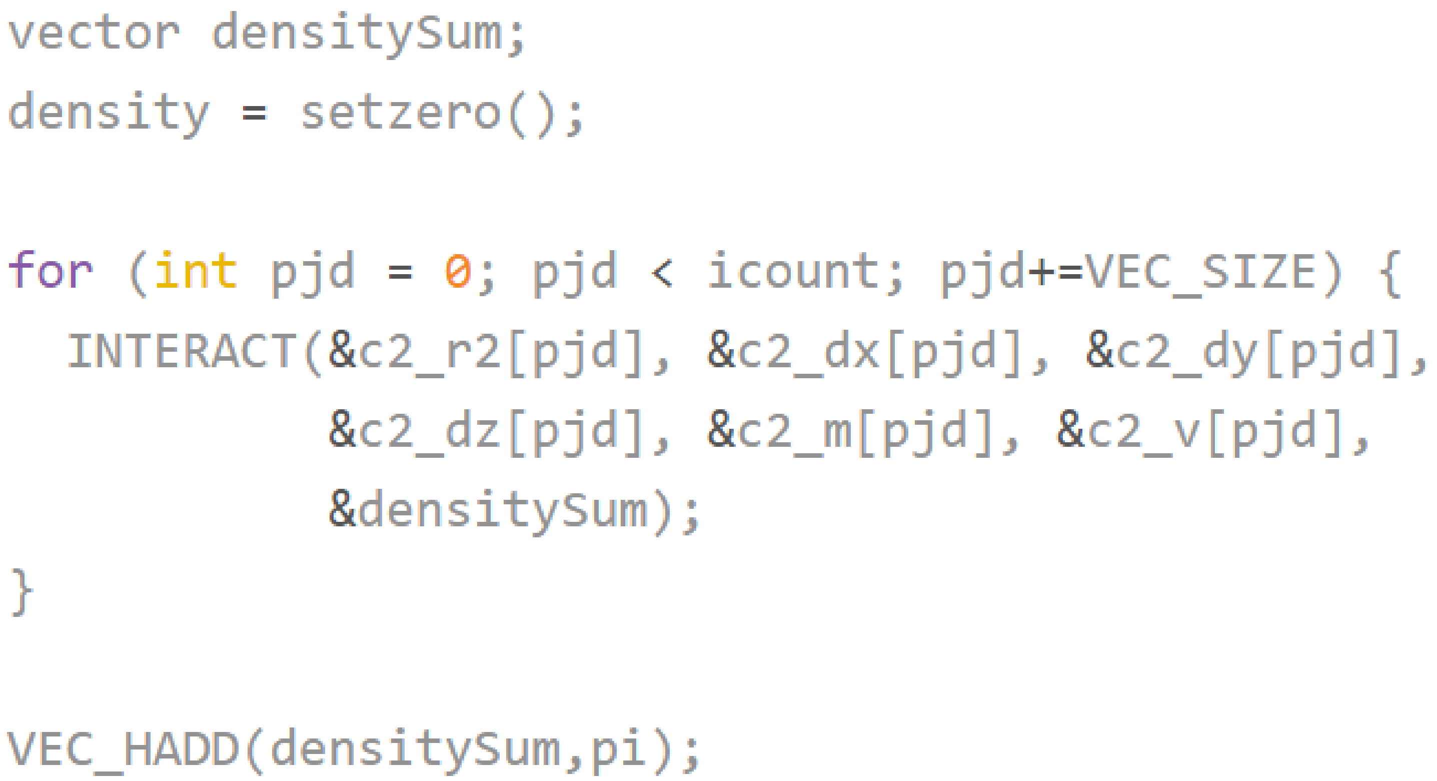

Particles (AoS)
x_1 y_1 z_1 h_1 ... x_2 y_2 z_2 h_2 ...
Particles (SoA)
x: x_1 x_2 x_3 x_4 ...
y: y_1 y_2 y_3 y_4 ...
z: z_1 z_2 z_3 z_4 ...
h: h_1 h_2 h_3 h_4 ...

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



Vector mask: ($r^2 < h^2$)
1 0 1 0 0 0 1
Vector r^2:
r^2_1 r^2_2 r^2_3 r^2_4 r^2_5 r^2_6 r^2_7 r^2_8
Secondary Cache (SoA)
r^2: r^2_1 r^2_3 r^2_8 r^2_9 ...
m_j: m_1 m_3 m_8 m_9 ...
v_j: v_1 v_3 v_8 v_9 ...
dx: dx_1 dx_3 dx_8 dx_9 ...

Step 3: Process all pairs in the 2nd cache

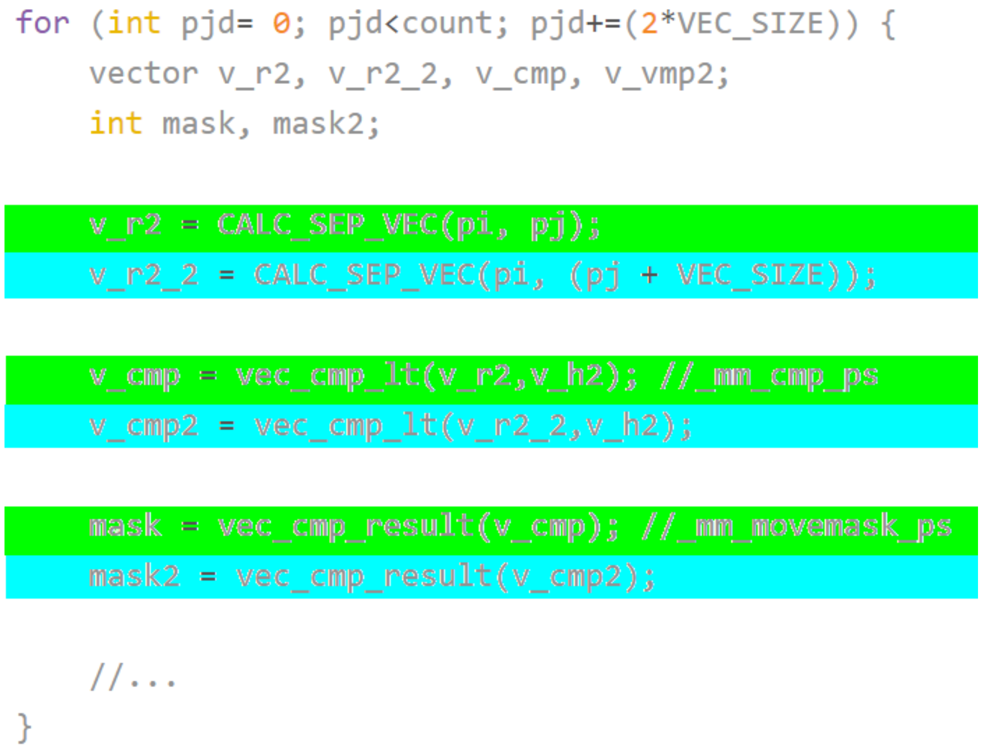


```
vector densitySum;
density = setzero();

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);
```

Improvements: Process two vectors at a time

Detailed vTune analysis showed limitations due to bubble forming in the pipeline and loads blocked by store forwarding.

Solution: Interleave operations from 2 vectors.



```
for (int pjd= 0; pjd<count; pjd+=(2*VEC_SIZE)) {
    vector v_r2, v_r2_2, v_cmp, v_vmp2;
    int mask, mask2;

    v_r2 = CALC_SEP_VEC(pi, pj);
    v_r2_2 = CALC_SEP_VEC(pi, (pj + VEC_SIZE));

    v_cmp = vec_cmp_lt(v_r2,v_h2); //_mm_cmp_ps
    v_cmp2 = vec_cmp_lt(v_r2_2,v_h2);

    mask = vec_cmp_result(v_cmp); //_mm_movemask_ps
    mask2 = vec_cmp_result(v_cmp2);

    //...
}
```

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

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

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.

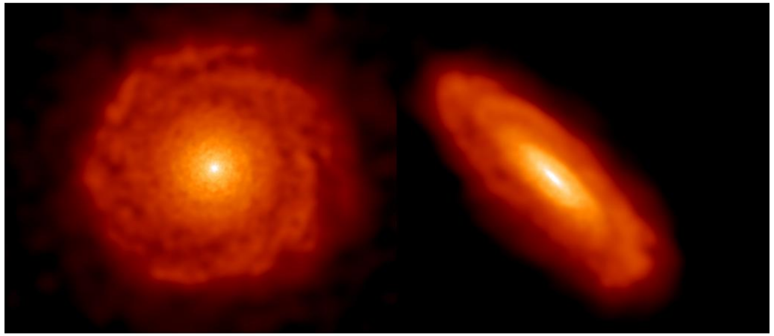
More on SWIFT

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.

INTEL® HPC DEVELOPER CONFERENCE
FUEL YOUR INSIGHT

Thank you for your time

Matthieu Schaller

www.swiftsim.org

www.intel.com/hpcdevcon