SWIFT: Strong scaling for particle-based simulations on more than 100’000 cores.

•••

Matthieu Schaller, Pedro Gonnet, Aidan B. G. Chalk & Peter W. Draper

Durham University, UK

PASC 2016 -- Lausanne, Switzerland -- 08 June 2016

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) and the `DiRAC` software team.

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

What we do and how we do it

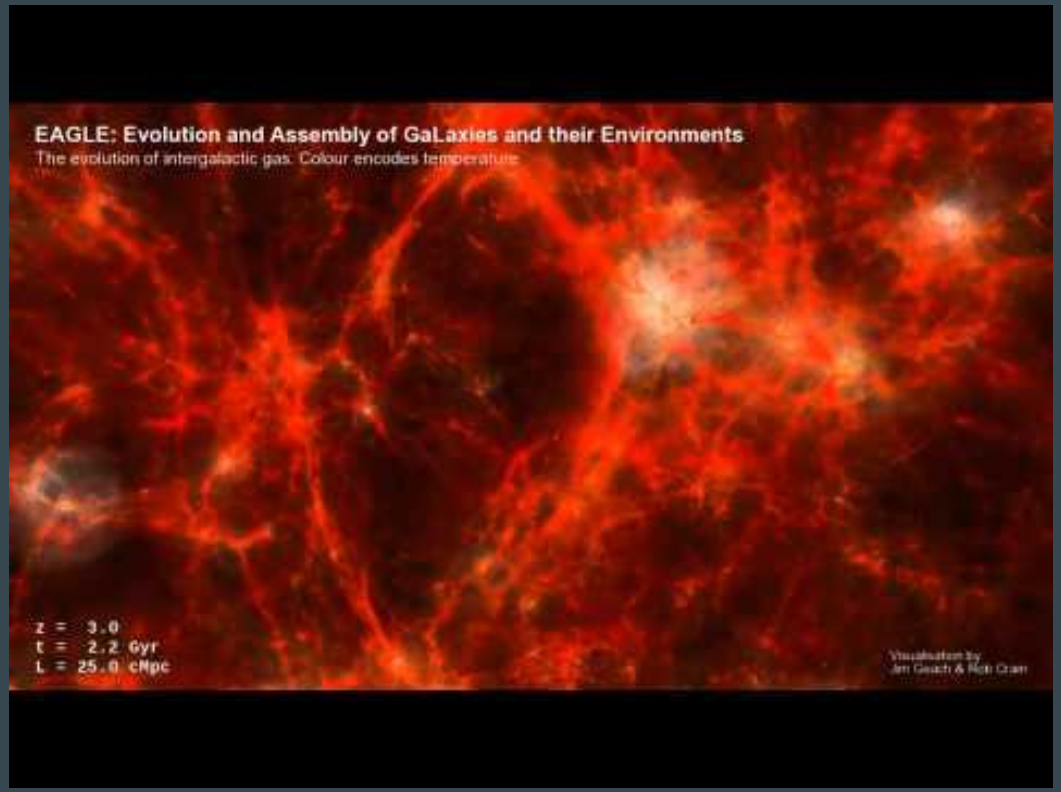
- Astronomy / Cosmology simulations of the formation of the Universe and galaxy evolution.
- `EAGLE` project[1]: 48 days of computing on 4096 cores. >500 TBytes of data products (post-processed data is public!). Most cited astronomy paper of 2015.
- Simulations of gravity and hydrodynamic forces with a spatial dynamic range spanning 6 orders of magnitude running for >2M time-steps.
- Most of it with the slightly outdated `MPI`-only `GADGET` code. Better scaling and performance required for the next generation runs.

One simulated galaxy out of the EAGLE virtual universe.

[1] www.eaglesim.org

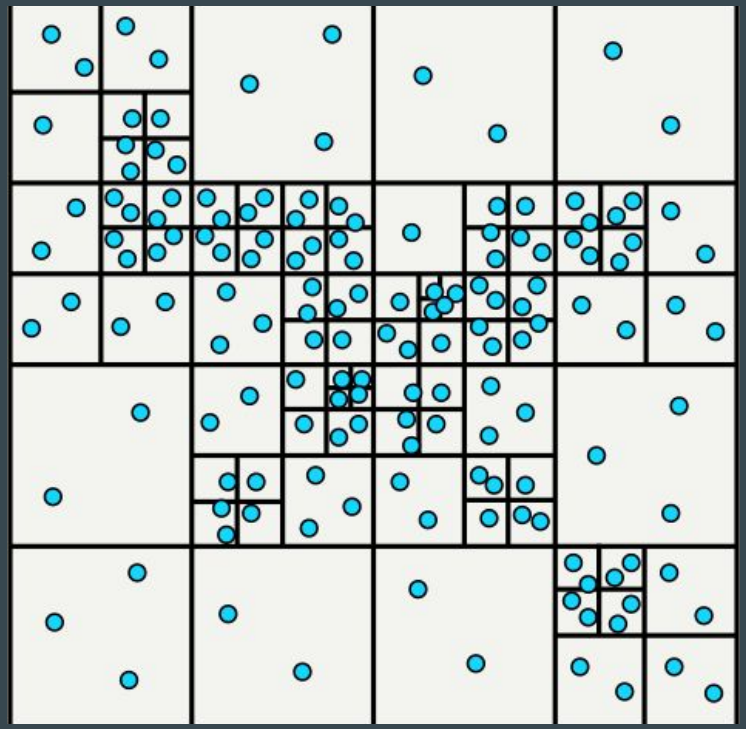
What we do and how we do it

- 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.



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).

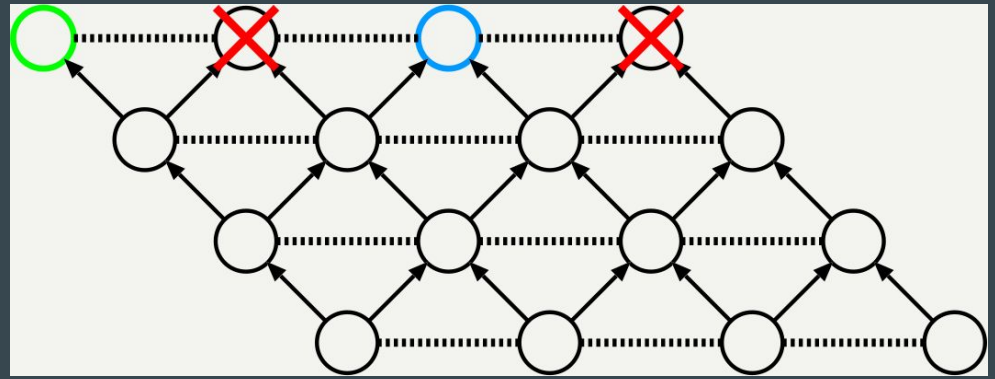


Task based parallelism

- 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 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.

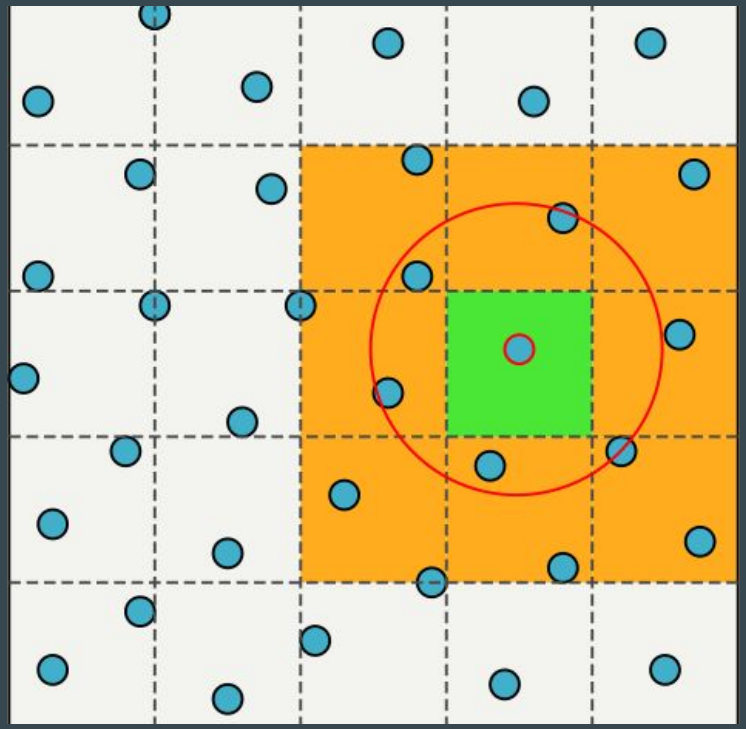


- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.

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

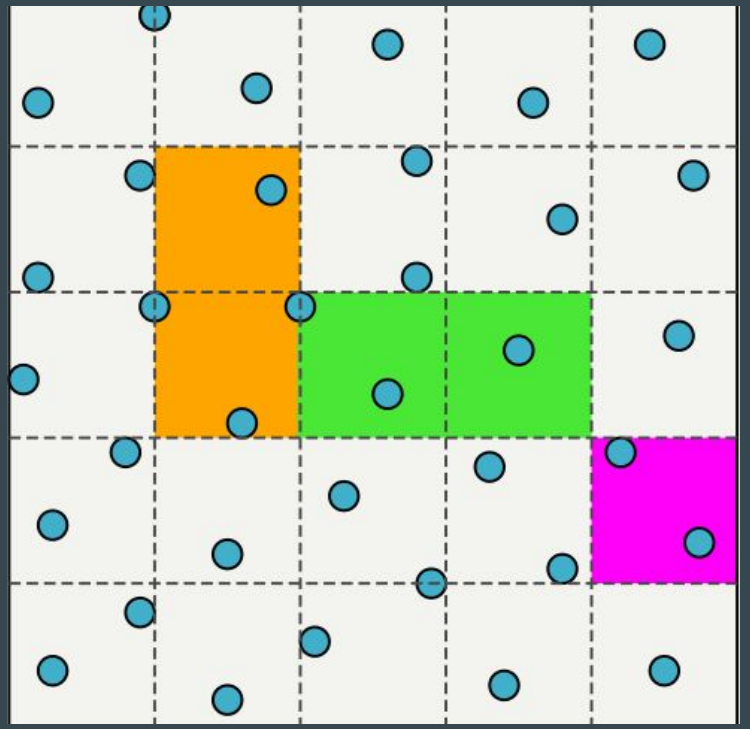
SPH scheme: Single-node parallelization

- 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.



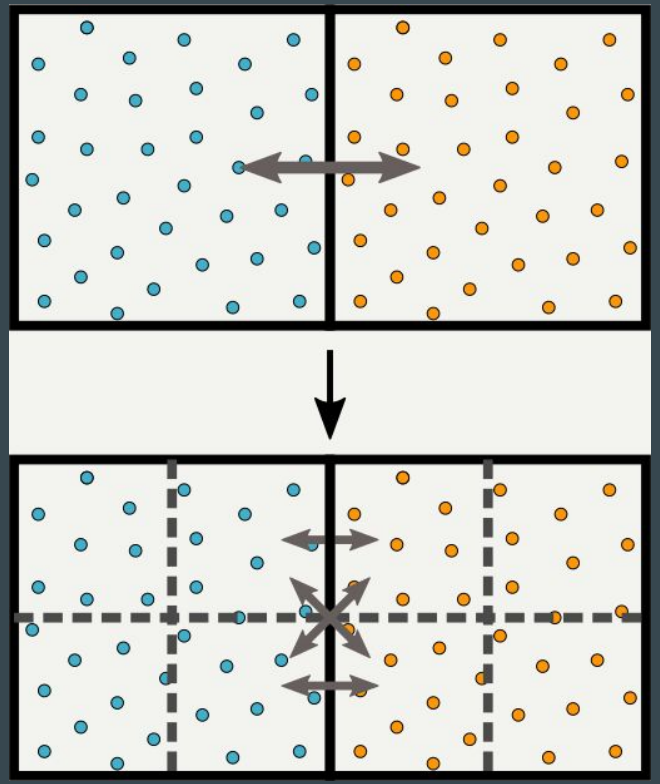
SPH scheme: Single-node parallelization

- 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.
- Amount of "work" per cell varies but order in which cells or pairs of cells is irrelevant.
  → Perfect for task-based parallelism.
  → Two tasks acting on the same cell *conflict*.
  → The tasks of the second loop *depend* on the tasks of the first loop.

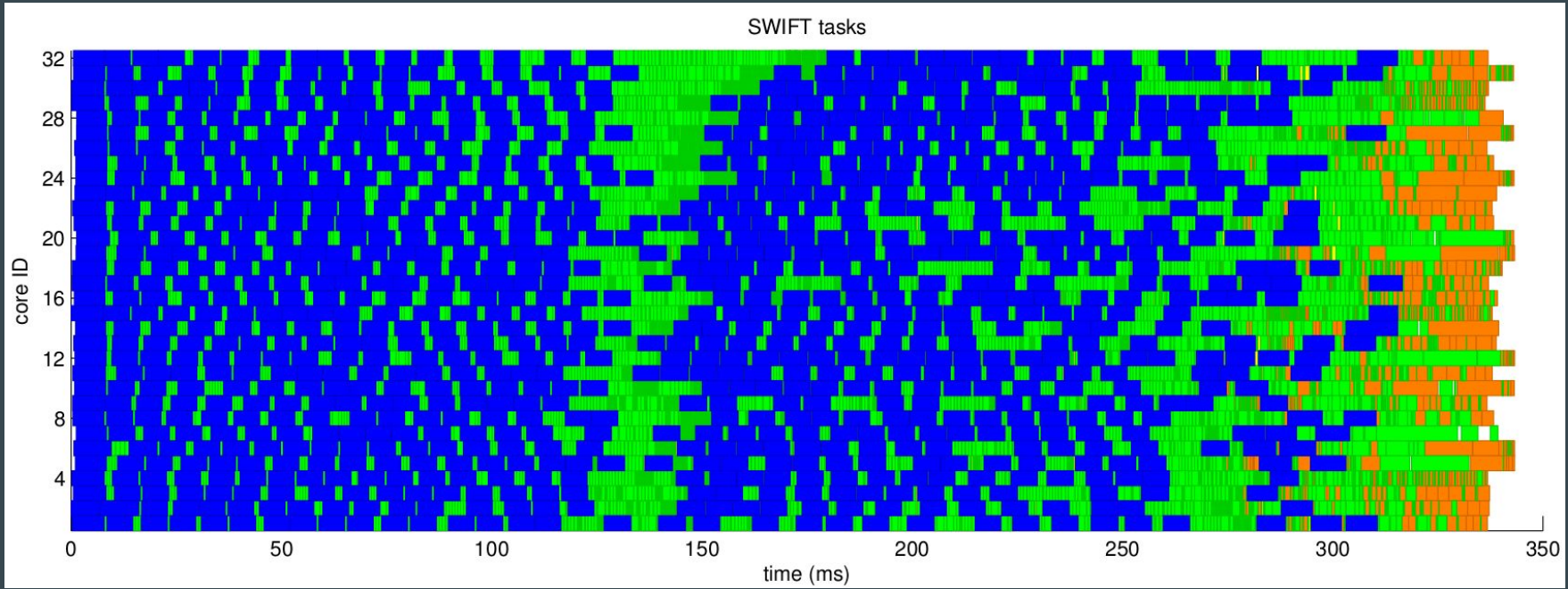


SPH scheme: Adaptive mesh and recursive scheme

- Tasks get split recursively when cells they act upon are too crowded.
- All the extra “sub-task” are automatically added to the task scheduler.
- Allows to keep a roughly constant amount of work per task.
- Tasks are kept at a size of a ≈ L1 cache.
  → Great for SIMD vectorization



SPH scheme: Single node parallel performance

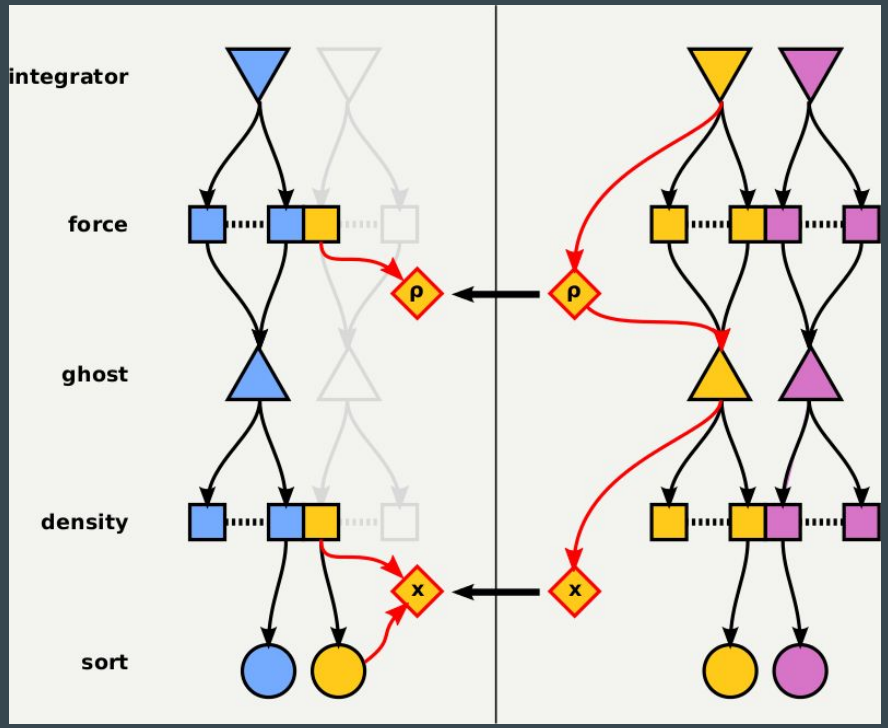


Task graph for one time-step. Colours correspond to different types of task. Almost perfect load-balancing is achieved on 32 cores.

How can this success be extended to clusters of many-core nodes ?

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.

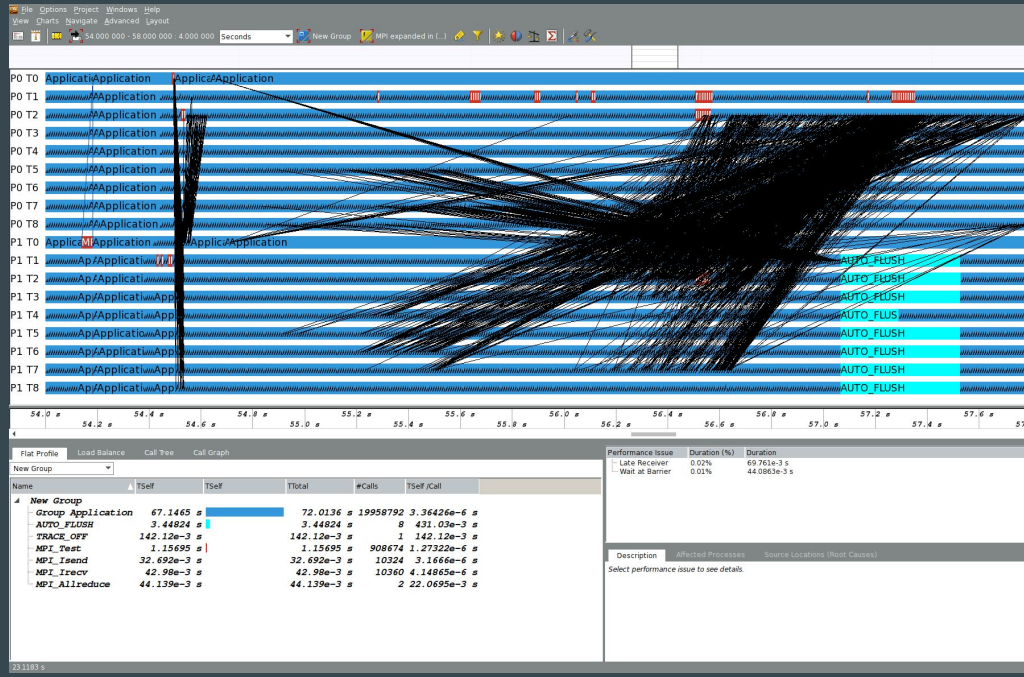
- Not all MPI implementations fully support the MPI v3.0 standard
  - Uncovered several bugs in different implementations providing `MPI_THREAD_MULTIPLE`.
  - e.g.: OpenMPI 1.10.x crashes when running on Infiniband!

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

- Most common analysis and benchmark tools do not support our approach!

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 nodes 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.

Flat Profile | Load Balance | Call Tree | Call Graph

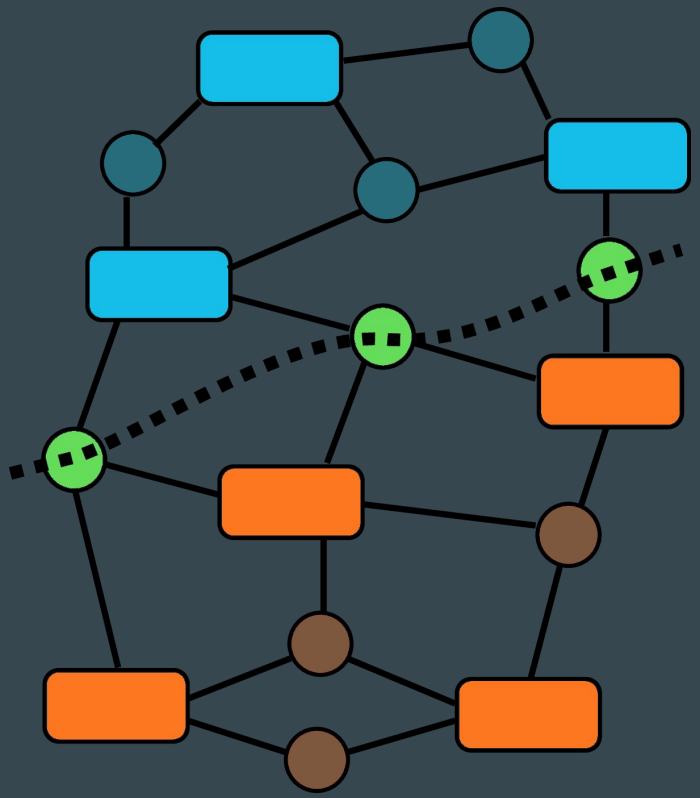
New Group

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

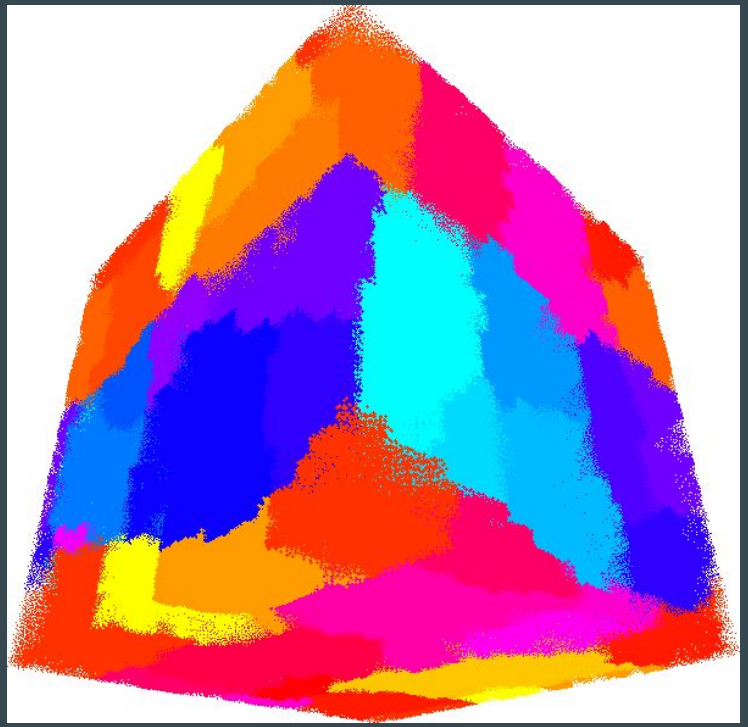
- For each task we compute the amount of work (=runtime) required.
- We can build a graph in which the simulation data are nodes and the tasks operation on the data are hyperedges.
- The task graph is split to balance the work (not the data!) using METIS.
- Tasks spanning the partition are computed on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



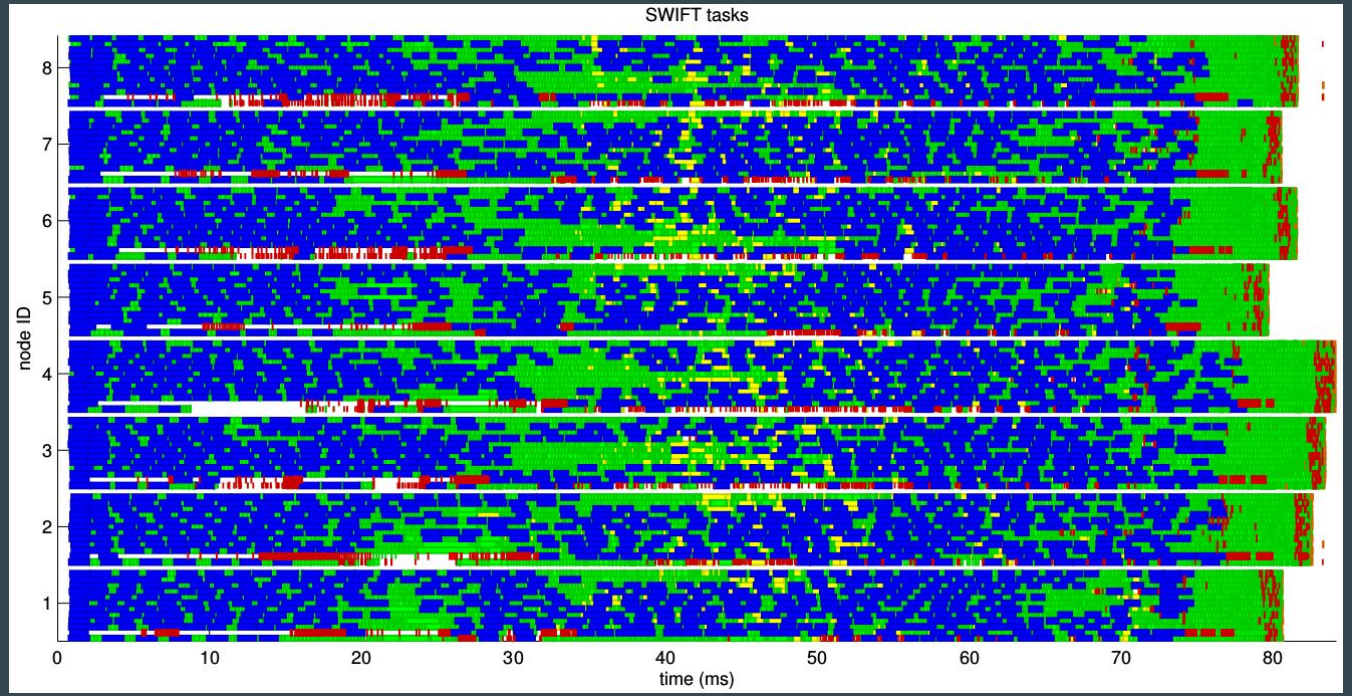
Domain decomposition

- 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.



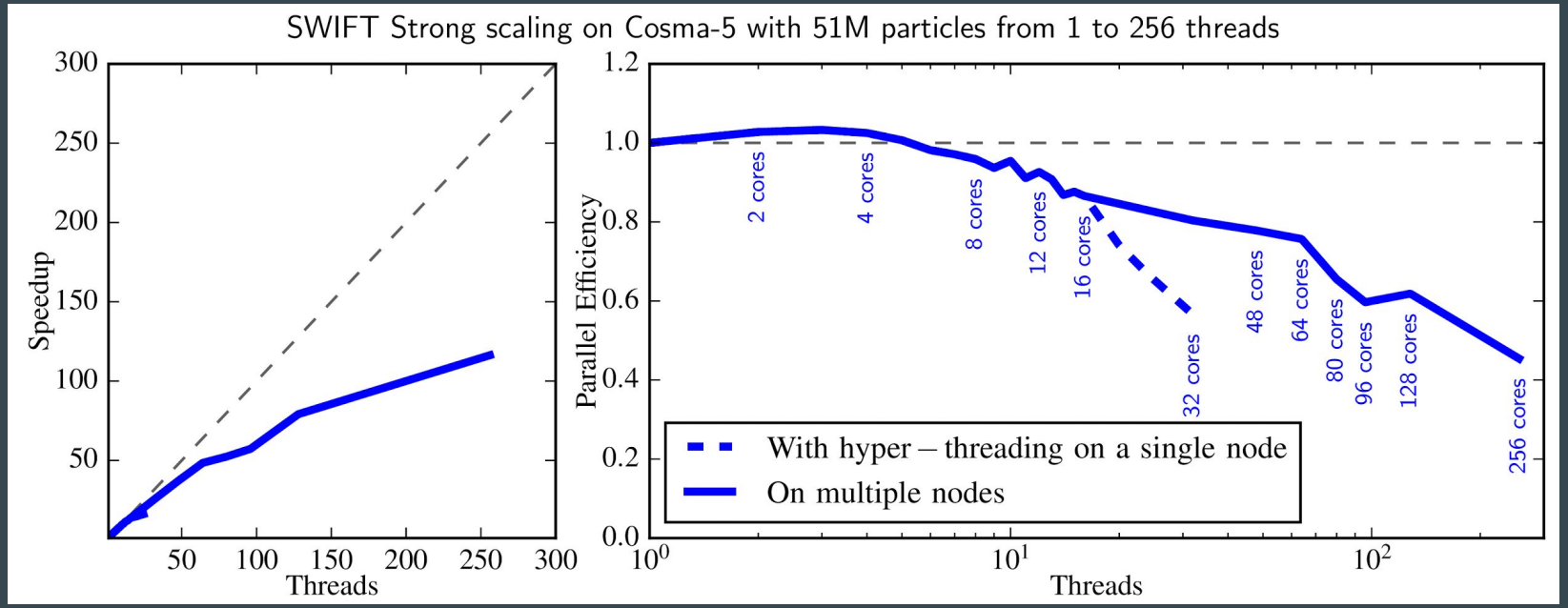
Multiple node parallel performance



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

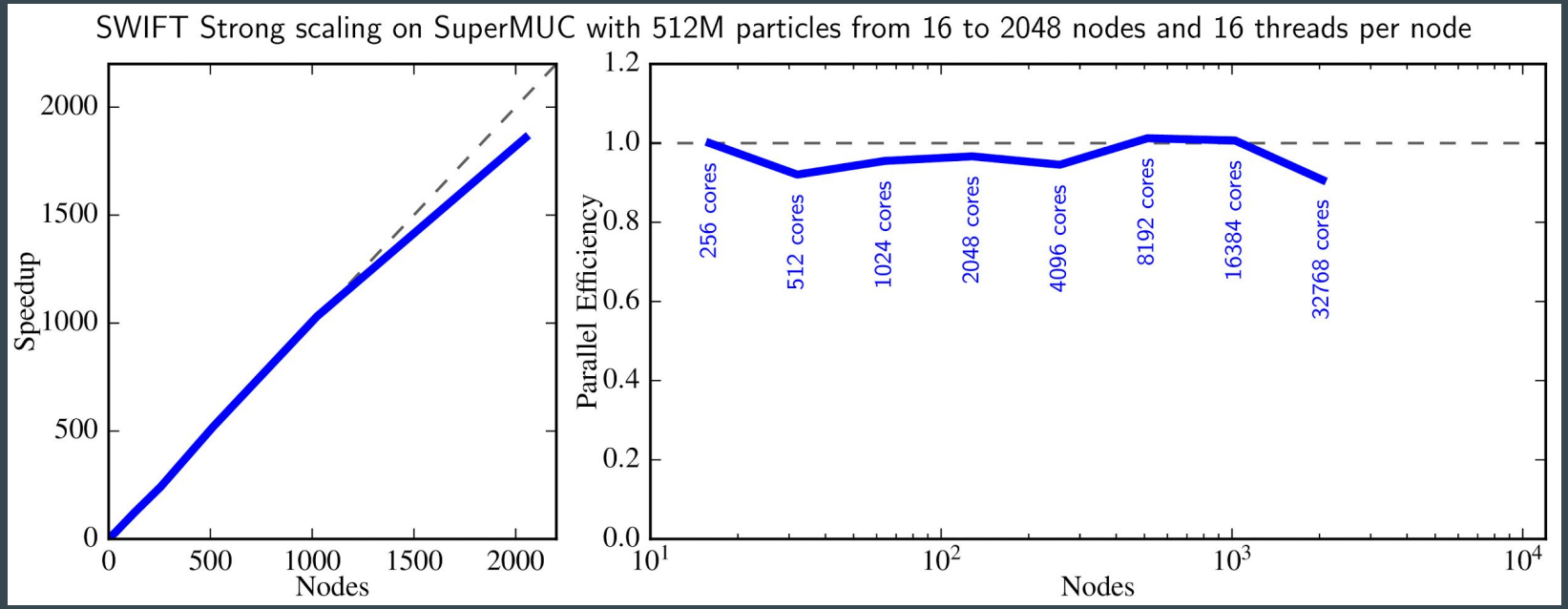
How does this perform on various architectures ?

Scaling results: DiRAC Data Centric facility “Cosma-5”



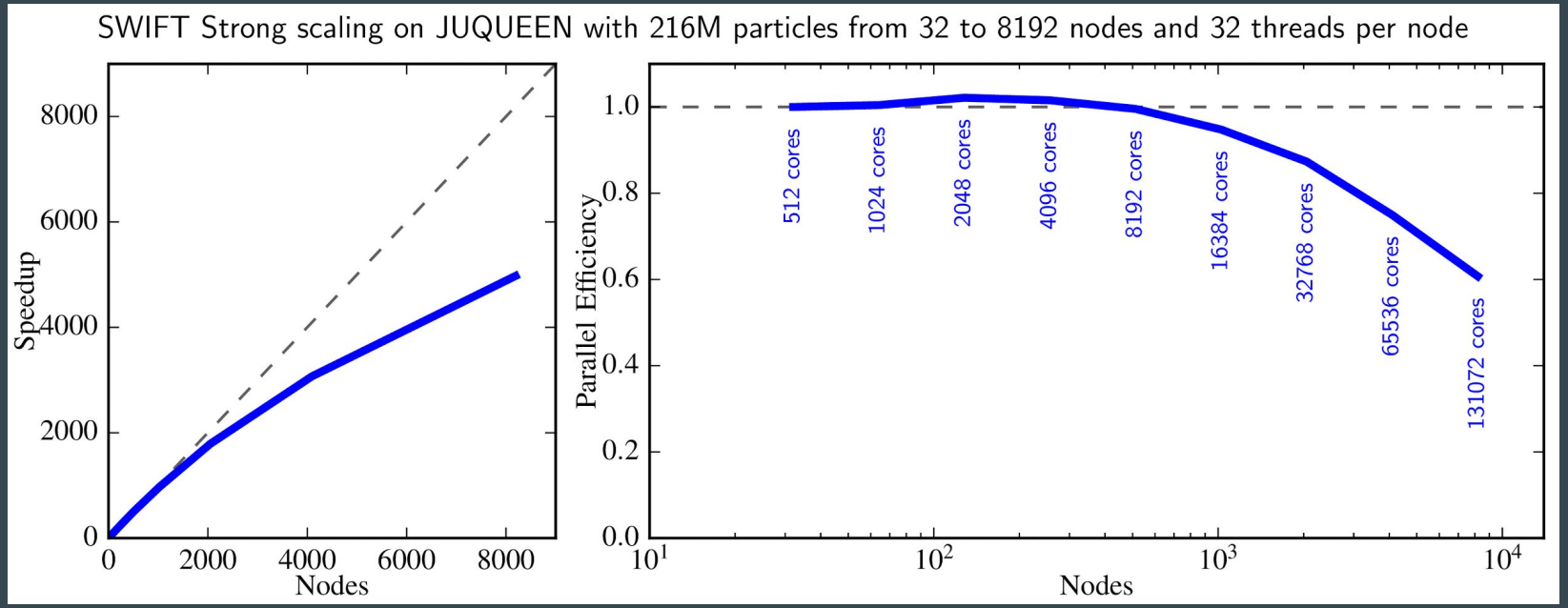
System: x86 architecture - 2 Intel Sandy Bridge-EP Xeon E5-2670 at 2.6 GHz with 128 GByte of RAM per node.

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.

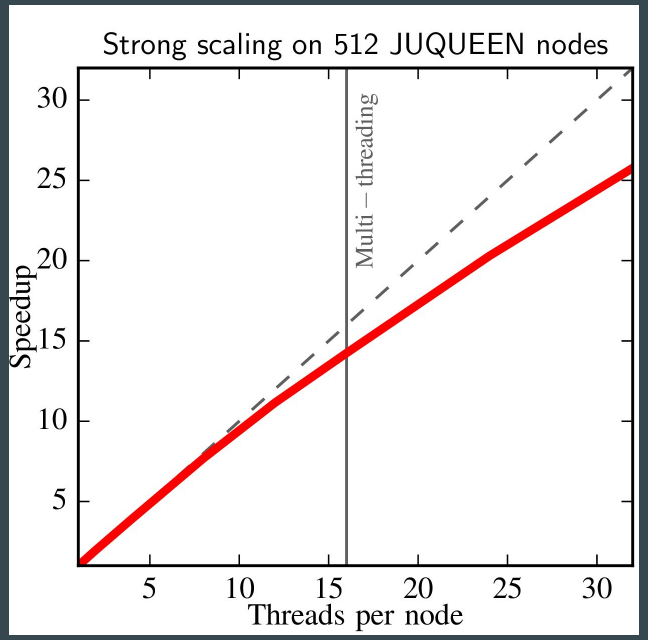
Scaling results: JUQUEEN (#11 in Top500)



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.

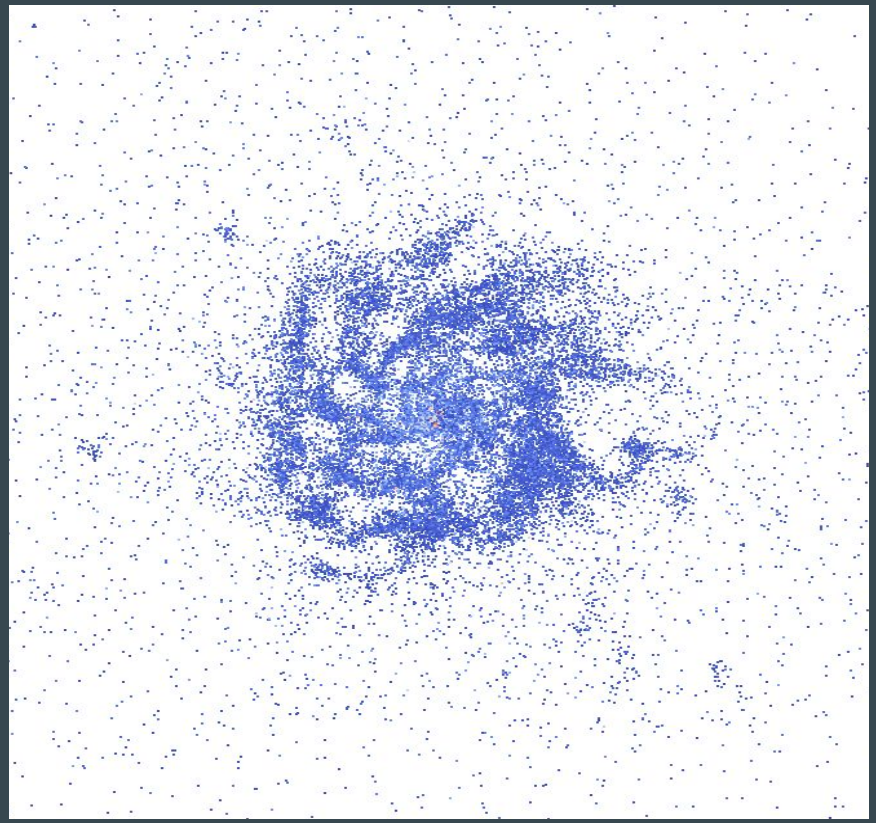


More on SWIFT

- Completely open-source software including all the examples and scripts.
- ~20’000 lines of C code without fancy language extensions.
- More than 10x 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!
- 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: Make use of the CPU vector units (SIMD) to gain extra speed.

www.swiftsim.org