SWIFT: Scheduling tasks efficiently on 256 cores - The KNL challenge

•••

Matthieu Schaller, Pedro Gonnet, Aidan B. G. Chalk, James S. Willis,
Peter W. Draper & *SWIFT* team

Durham University, UK

IPCC-EMEA fall meeting 2016 -- Toulouse, France -- 26 October 2016

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

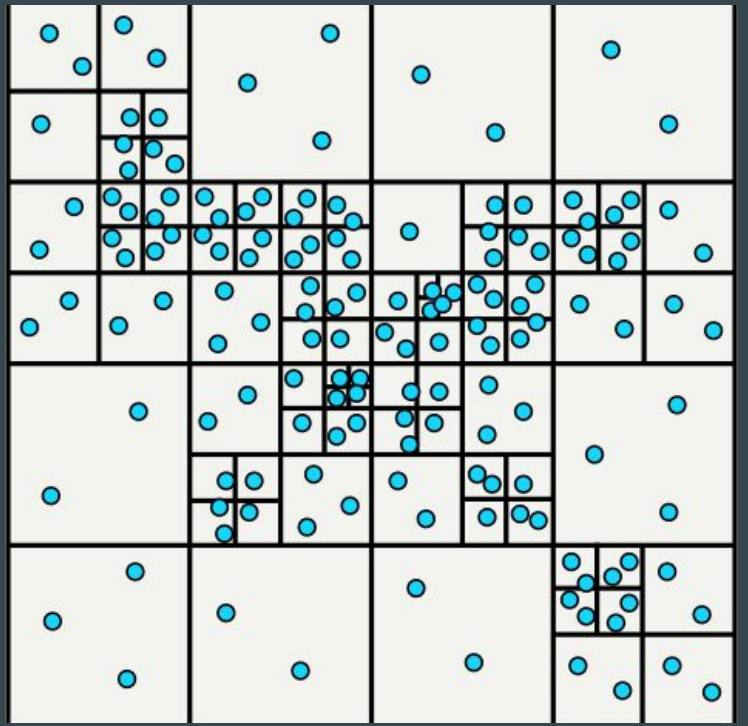
- The Institute for Computational Cosmology,
- The School of Engineering and Computing Sciences,

with contributions from the astronomy group at the university of Ghent (Belgium), St-Andrews (UK), Lausanne (Switzerland), Perth (Australia), Paris (France) and the DiRAC software team.

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

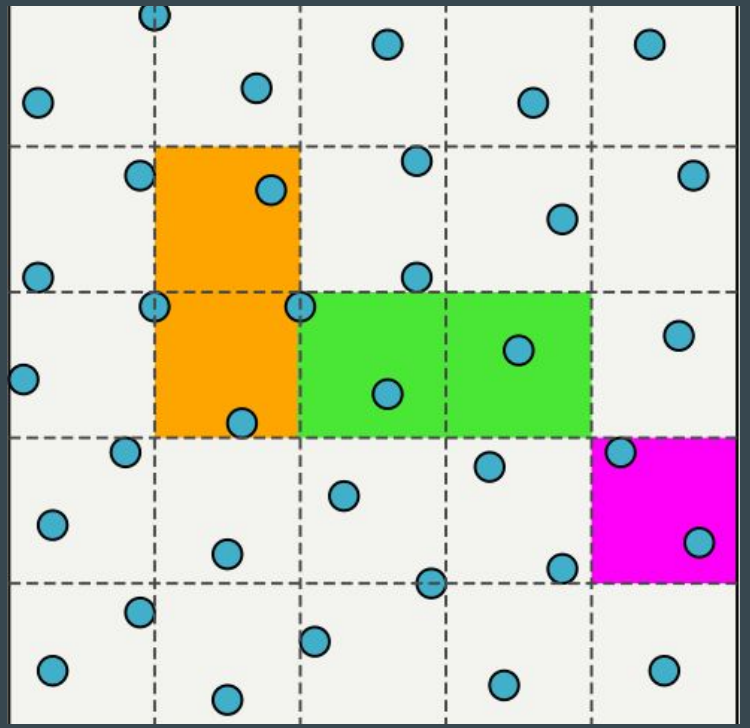
SPH scheme: The problem to solve

- 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).



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.

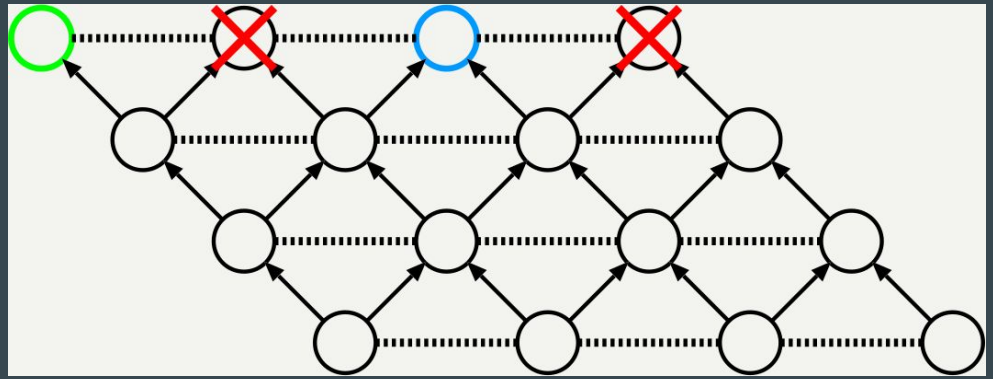


Task based parallelism

- 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.

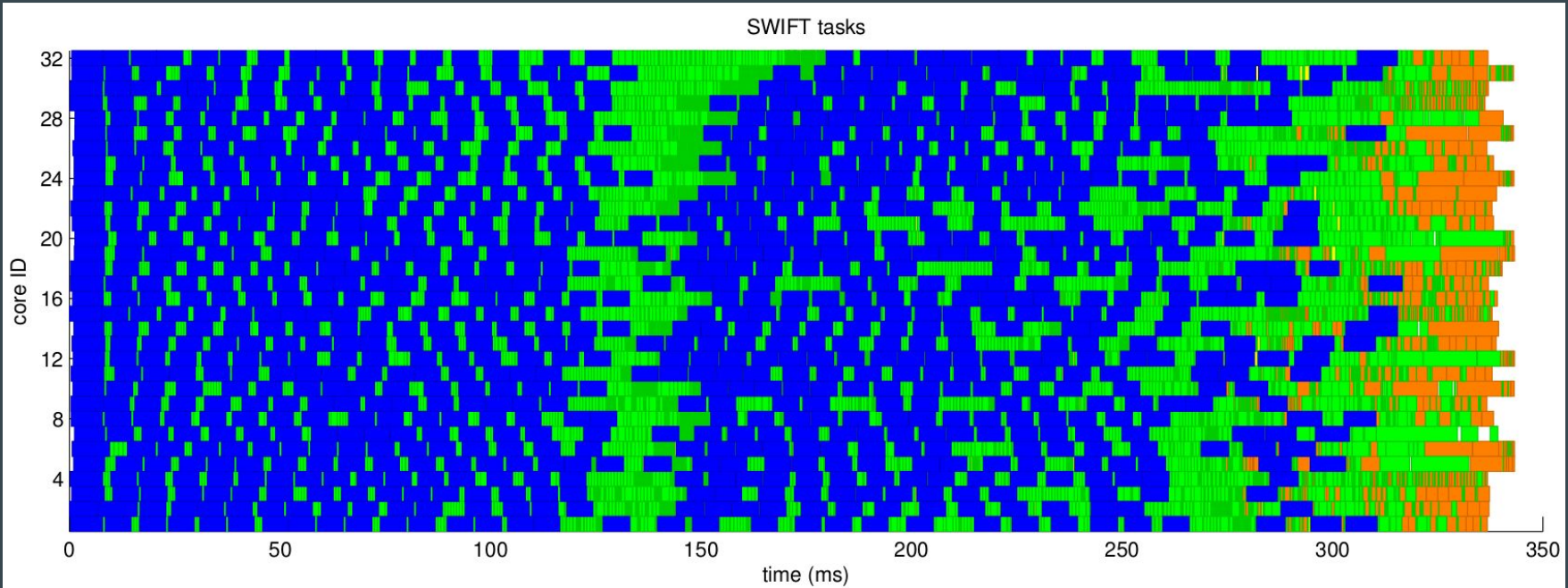
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.



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

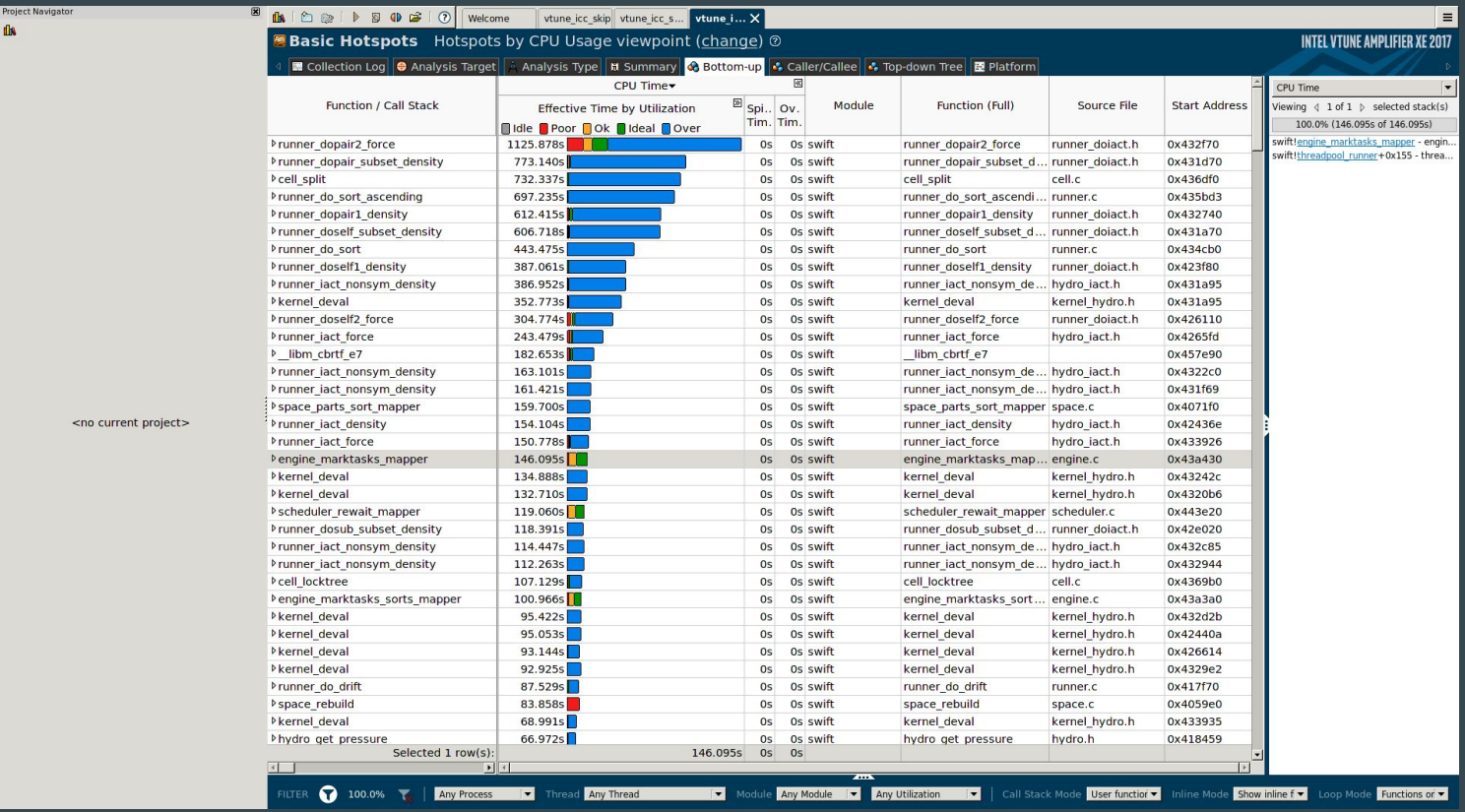
- We use our own Open-source library `QuickSched` (`arXiv:1601.05384`)

SPH scheme: Single node parallel performance



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

SPH scheme: Single node parallel performance



16 cores Haswell node.

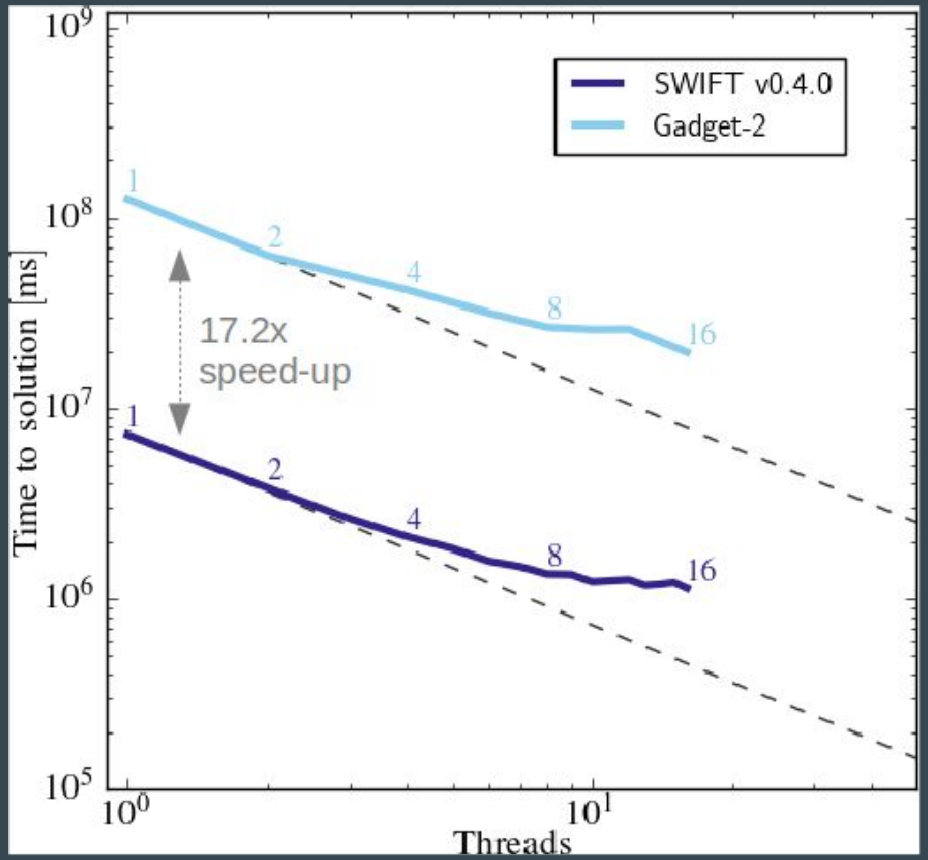
- *Blue: 16 cores*
- *Green: 5-15 cores*
- *Yellow: 3-4 cores*
- *Red: 1-2 cores*

Comparison to Gadget

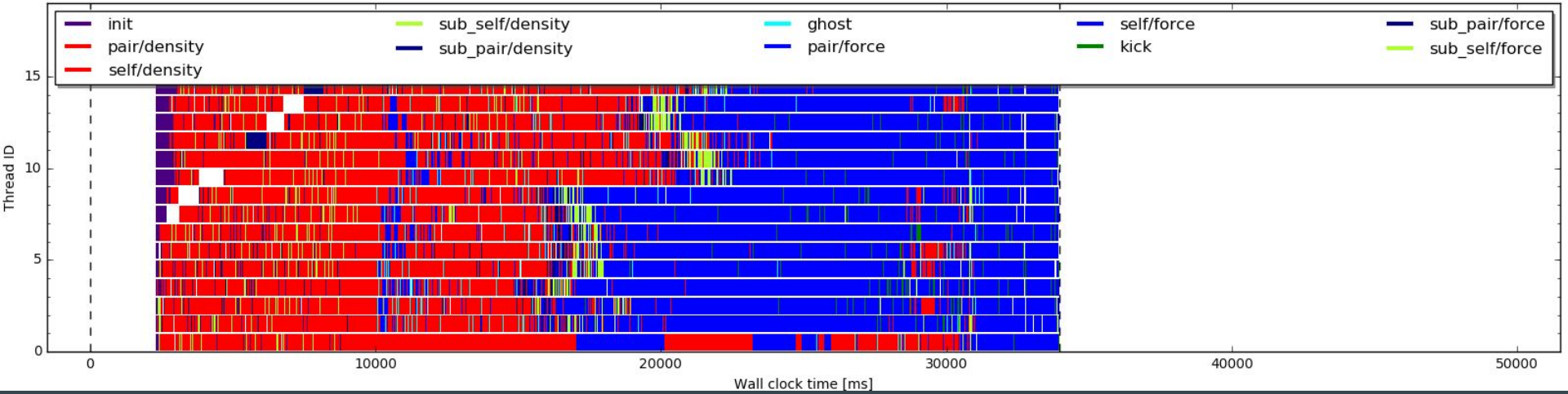
- Realistic problem (see James' talk)
- Same accuracy.
- Same hardware.
- Same compiler.
- Same solution.

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

Note that vectorization is switched off here.



Challenges for KNL



Pathological case. Large non-tasked sections at the start of the time-step.

Challenges for KNL

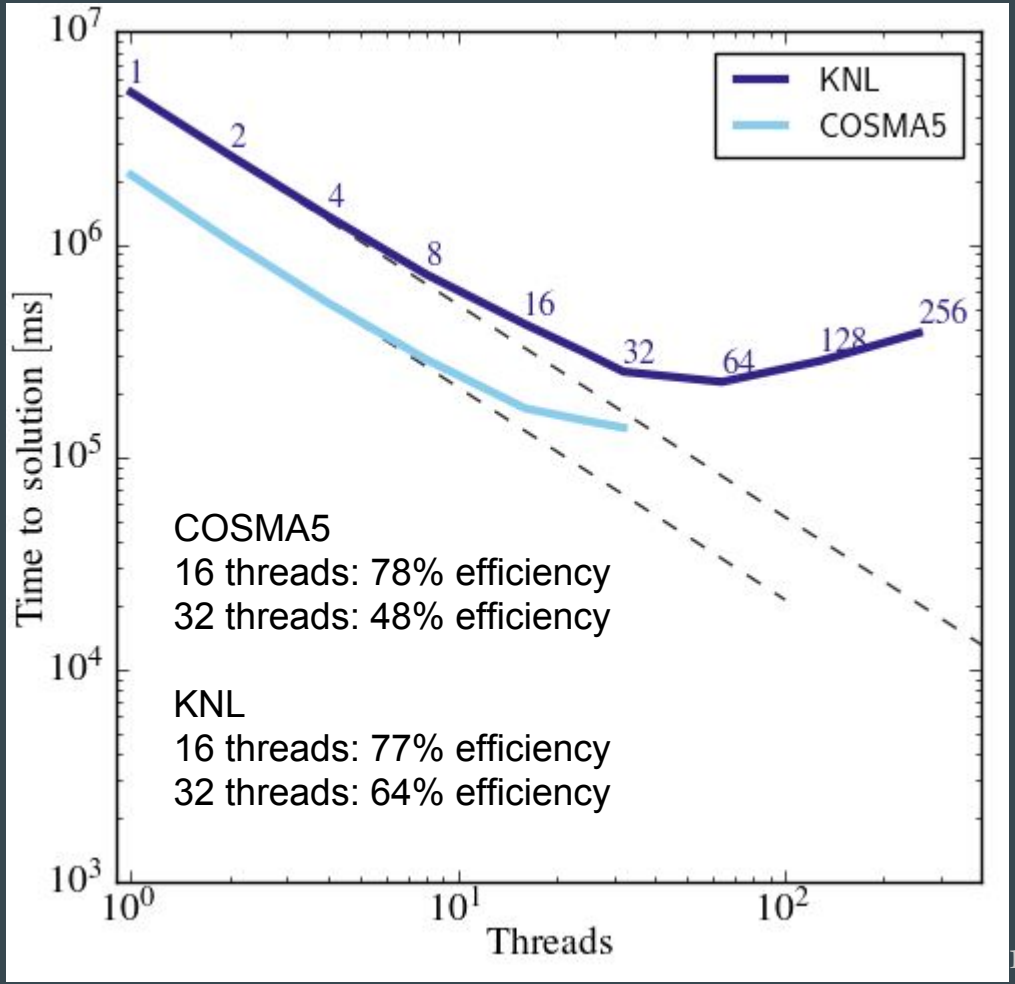
Cosma-5: Sandy Bridge 16 cores at 2.6GHz

Messages:

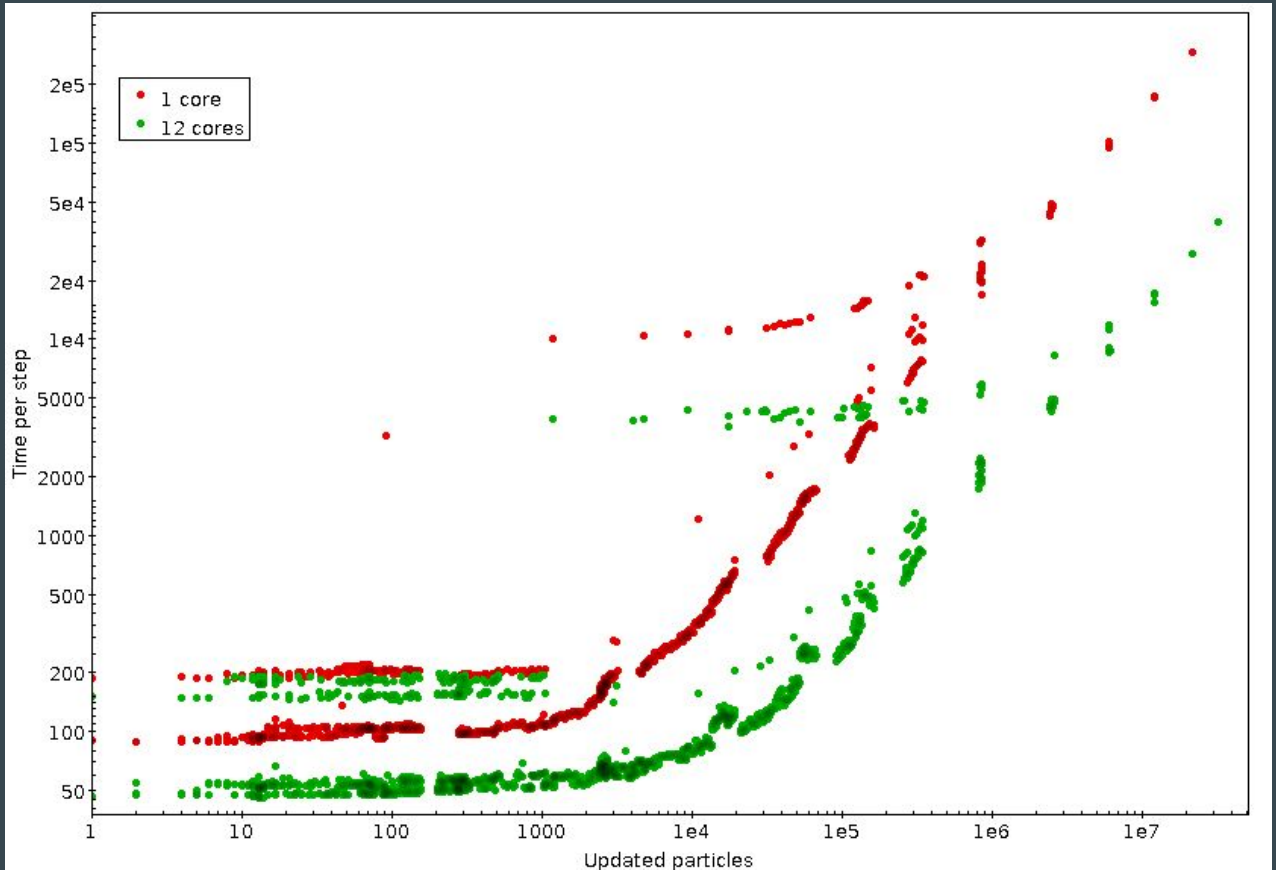
- No good scaling on hyperthreads (all architectures)
- KNL never faster than Sandy Bridge
- Need to improve scaling performance

Note that vectorization is switched off here.

KNL in cache mode and quadrant mode. Pinning left to the OS.



Another point of view



Parallelize the rest - Threadpools

Non-parallel sections of the code

- i/o
- Domain decomposition
- Some aspects of mesh construction
- Marking tasks as active or inactive (loop over task list)
- Scheduling the tasks (loop over task list)
- Reduction of time-step (loop over cells)

In summary:

- Physics perfectly load balanced, scales well
- Bottlenecks down to “maintenance” that is run serially in between time steps

Quick and dirty solution: OpenMP

- Could use OpenMP for these simple loops.
- As always comes with some issues:
  - No fine control over threads
  - Hard to interface with the rest of the task-based system (pthread)
  - Overheads

Quick and dirty solution: OpenMP

- Could use OpenMP for these simple loops.
- As always comes with some issues:
  - No fine control over threads
  - Hard to interface with the rest of the task-based system (pthread)
  - Overheads

Instead implement a lightweight pool of threads and mapper functions

Threadpool: Main idea

- Create a set of additional (p)threads.
- Make the threads wait at a barrier.
- Create a “map” function to be called on a chunk of the arrays.
- When required unleash the the threads on chunks of the array, each calling the map function.

Essentially a lightweight version of OpenMP

Threadpool: Example for user

```
void stats_collect_part_mapper(void *map_data, int nr_parts, void
*extra_data) {

  struct part* particles = (struct part*) map_data;

  /* Loop over particles */
  for (int k = 0; k < nr_parts; k++) {

    Stats.energy += particles[k].energy // for example
  }
}

// Elsewhere...
threadpool_map(threadpool, stats_collect_part_mapper, s->parts,
               s->nr_parts, sizeof(struct part), 10000);
```

Threadpool: Under the hood

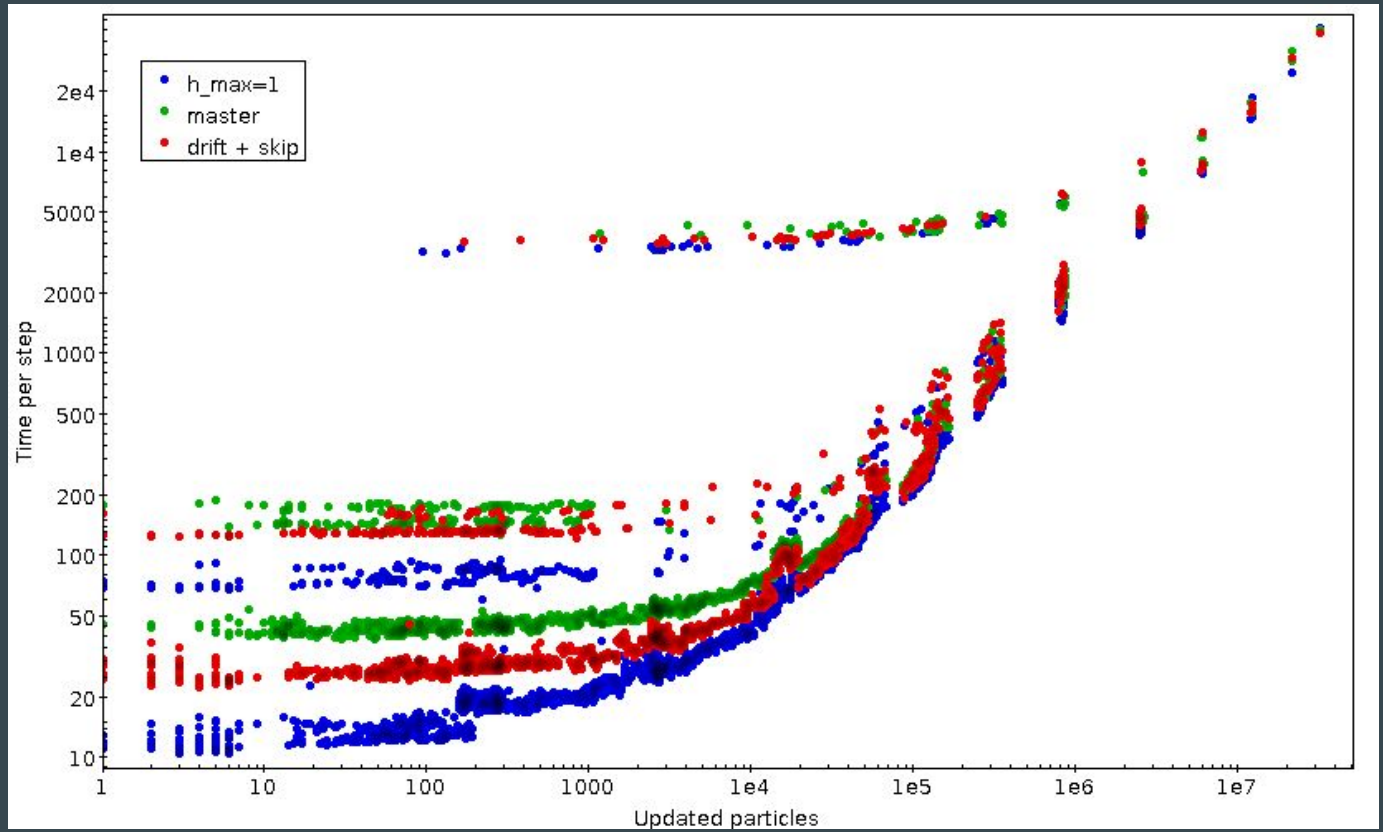
```
while (1) {
    /* Desired chunk size. */
    size_t chunk_size =
        (tp->map_data_size - tp->map_data_count) / (2 * tp->num_threads);


    /* Get a chunk and check its size. */
    size_t task_ind = atomic_add(&tp->map_data_count, chunk_size);
    if (task_ind >= tp->map_data_size) break;
    if (task_ind + chunk_size > tp->map_data_size)
    chunk_size = tp->map_data_size - task_ind;

    /* Call the mapper function. */
    tp->map_function((char *)tp->map_data + (tp->map_data_stride *
                     task_ind), chunk_size, tp->map_extra_data);
}
```

Results

Back to the time per particles



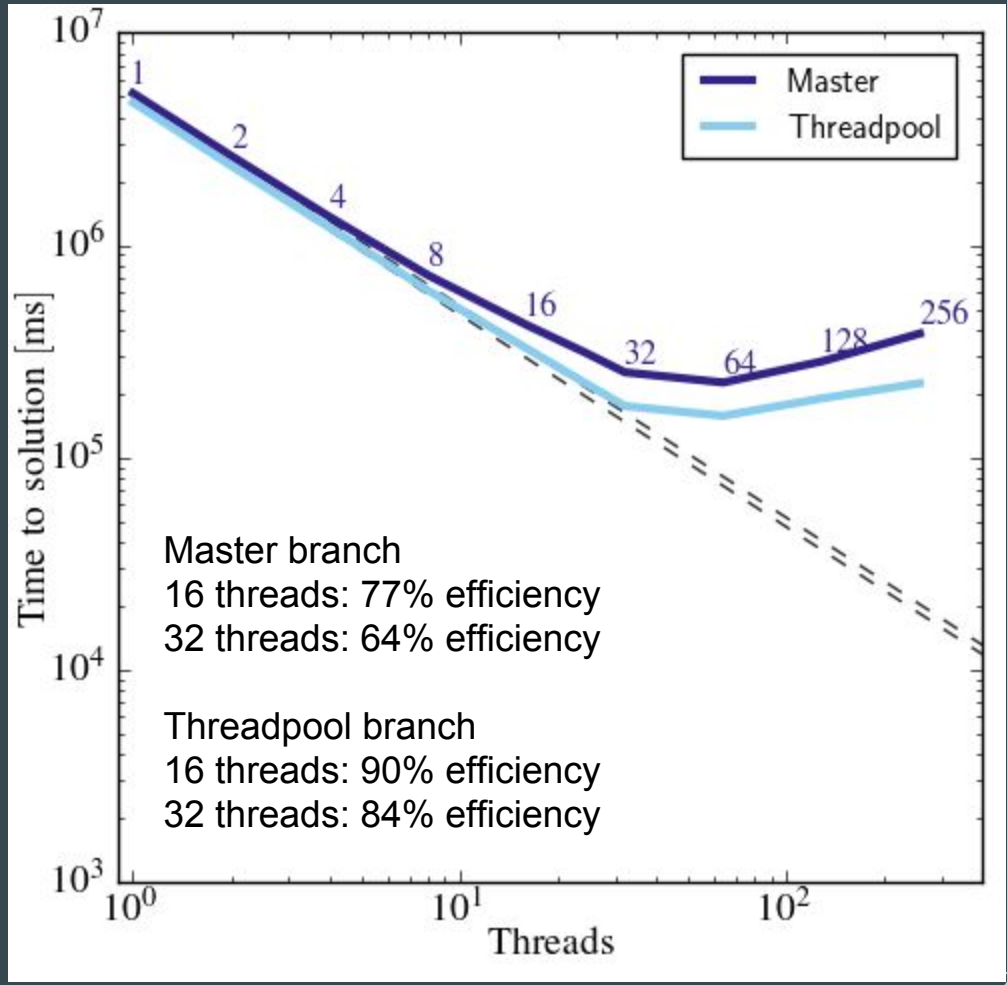
Results

Showing KNL only here.

Better efficiency. Achieve good scaling up to 32 cores. Still struggling to go to 64 cores.

Also improved performance on 1 core. Lowered time to solution.

Regular Xeon also benefits from solution (not shown here). Now 23x vs. Gadget instead of 17x on 16 cores !



Future work

Some more improvements

Reduce the number of scalar sections (Amdahl's law):

- Efficient i/o
- Fully parallel tree construction
- More parallel domain decomposition

Other implementation "details" :

- Make the threadpool a tiny bit smarter to reduce the need for magical constants

Open issues

Need better linux kernel ? From vTune analysis:

Function	CPU Time
runner_dopair2_force	5190.928s
runner_do_sort_ascending	3938.856s
runner_dopair1_density	3136.735s
runner_dopair_subset_density	2163.073s
runner_doself_subset_density	1989.453s
runner_doself1_density	1963.123s
runner_do_sort	1816.893s
runner_doself2_force	1811.123s
runner_iact_force	1000.722s
__do_softirq	958.811s
runner_iact_nonsym_density	939.821s
space_parts_sort_mapper	799.911s
cell_split	787.111s
[...]	

Linux 3.10.0-327.36.2.el7.x86_64

Conclusions

- KNL is hard (and not just because of AVX-512)
- Parallelisation of *all* parts of the code is necessary
- Using a very lightweight threadpool mechanism to parallelize simple section helps.
- Gains also apply to Xeon
- Is the current linux kernel up-to-date for KNL ?