Performance Optimization of Smoothed Particle Hydrodynamics for Multi/Many-Core Architectures

Dr. Fabio Baruffa
Leibniz Supercomputing Centre
fabio.baruffa@lrz.de

MC² Series: Colfax Research Webinar, http://mc2series.com
March 7th, 2017

Work contributors

Dr. Fabio Baruffa
Sr. HPC Application Specialist
Leibniz Supercomputing Centre

- Member of the IPCC @ LRZ
- Expert in performance optimization and HPC systems

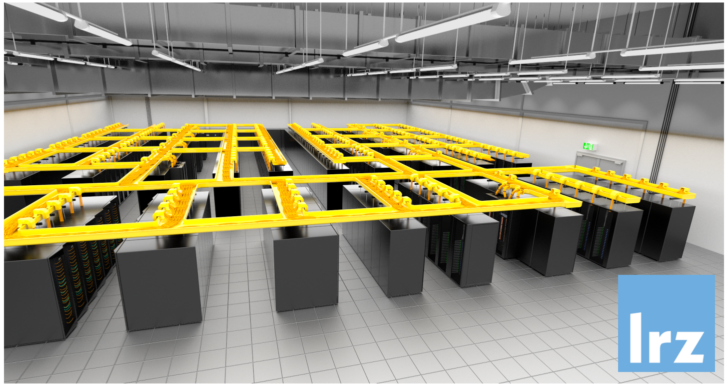
Dr. Luigi Iapichino
Scientific Computing Expert
Leibniz Supercomputing Centre

- Member of the IPCC @ LRZ
- Expert in computational astrophysics and simulations

Email contacts: fabio.baruffa@lrz.de luigi.iapichino@lrz.de

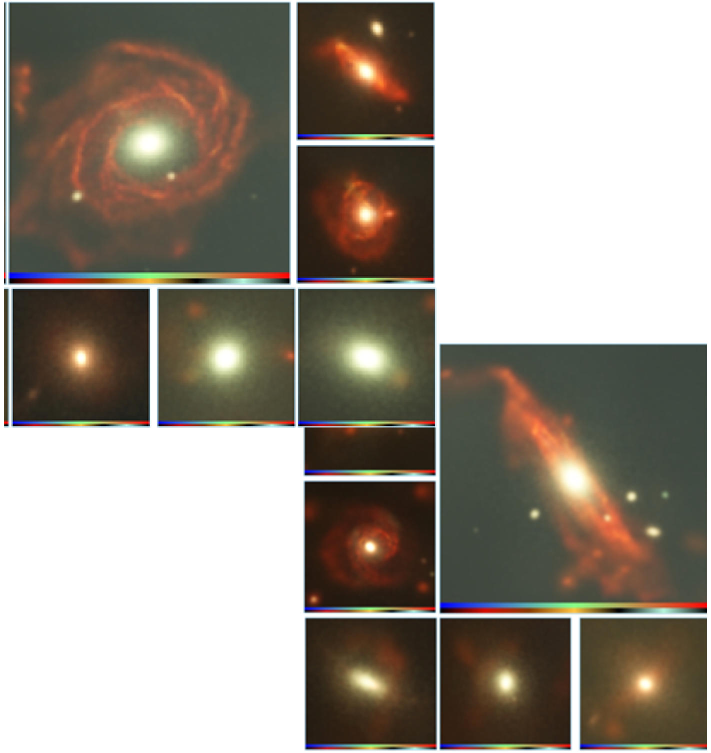
Intel® Parallel Computing Centers (IPCC)

- The IPCCs are an Intel initiative for code modernization of technical computing codes.
- The work primary focus on code optimization increasing parallelism and scalability on multi/many core architectures.
- Currently ~70 IPCCs are funded worldwide.
- Our target is to prepare the simulation software for new platforms achieving high nodel-level performance and multi-node scalability.



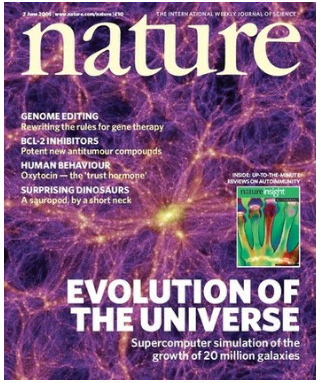
Outline of the talk

- Overview of the code: P-Gadget3 and SPH.
- Challenges in code modernization approach.
- Multi-threading parallelism and scalability.
- Enabling vectorization through:
  - Data layout optimization (AoS → SoA).
  - Reducing conditional branching.
- Performance results and outlook.



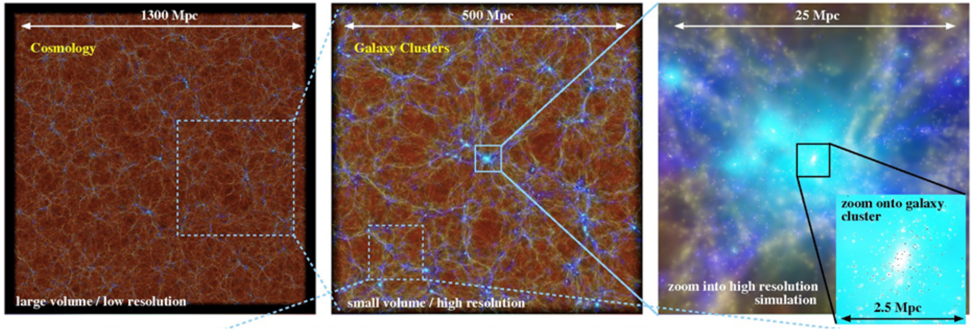
Gadget intro

- Leading application for simulating the formation of the cosmological large-scale structure (galaxies and clusters) and of processes at sub-resolution scale (e.g. star formation, metal enrichment).



- Publicly available, cosmological TreePM N-body + SPH code.

- Good scaling performance up to O(100k) Xeon cores (SuperMUC @ LRZ).



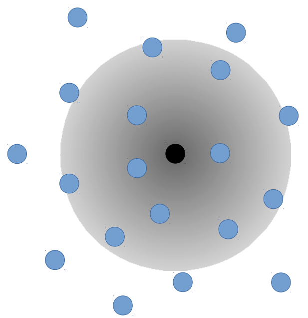
Smoothed particle hydrodynamics (SPH)

- SPH is a Lagrangian particle method for solving the equations of fluid dynamics, widely used in astrophysics.

- It is a mesh-free method, based on a particle discretization of the medium.

- The local estimation of gas density (and all other derivation of the governing equations) is based on a kernel-weighted summation over neighbor particles:

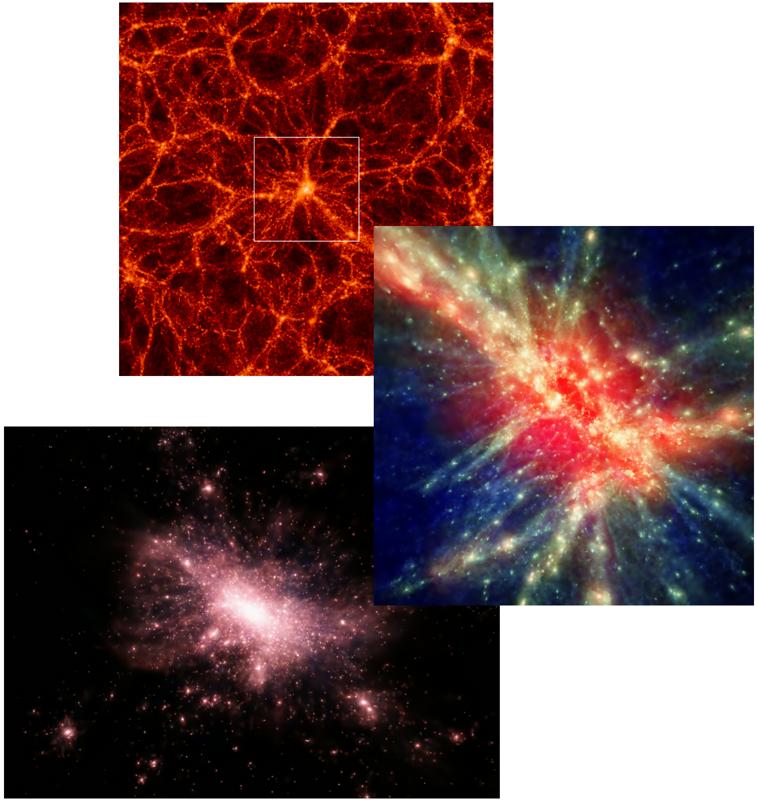
$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_j)$$



Gadget features

The code can be run at different levels of complexity:

- N-Body-only (a.k.a. dark matter) simulations.
- N-Body + gas component (SPH).
- Additional physics (sub-resolution) modules: radiative cooling, star formation,...
- More physics → more memory required per particles (up to ∼ 300B / particle).



Features of the code

- Gadget has been first developed in the late 90s as serial code, has later evolved as an MPI and a hybrid code.
- After the last public release Gadget-2, many research groups all over the world have developed their own branches.
- The branch used for this project (P-Gadget3) has been used for more than 30 research papers over the last two years.
- The code have ~200 files, ~400k code lines, extensive use of #IFDEF, ext. libs (fftw,hdf5).

Basic principles of our development

- We perform code modifications which are minimally invasive.

- Our intention is to ensure:
  - Portability on all modern architectures (Intel® Xeon/MIC, Power, GPU,...);
  - Readability for non-experts in HPC;
  - Consistency with all the existing functionalities.

- The domain scientists have to be able to modify the code without coping with performance questions.

Code modernization approach

- Scalar optimization: compiler flags, data casting, precision consistency.
- Vectorization: prepare the code for SIMD, avoid vector dependencies.
- Memory access: improve data layout, cache access.
- Multi-threading: enable OpenMP, manage scheduling and pinning.
- Communication: enable MPI, offloading computation.

https://software.intel.com/en-us/articles/what-is-code-modernization; colfaxresearch.com

Code modernization approach

- Scalar optimization: compiler flags, data casting, precision consistency.
- Vectorization: prepare the code for SIMD, avoid vector dependencies.
- Memory access: improve data layout, cache access.
- Multi-threading: enable OpenMP, manage scheduling and pinning.
- Communication: enable MPI, offloading computation.

Preparation for the next generation processors and efficient usage of the current hardware

https://software.intel.com/en-us/articles/what-is-code-modernization; colfaxresearch.com

Target architectures for our project

Intel® Xeon processor

- E5-2650v2 Ivy-Bridge (IVB) @ 2.6 GHz, 8-cores / socket.
  TDP: 95W, RCP: $1116.
- AVX.

Intel® Xeon Phi™ coprocessor
1st generation

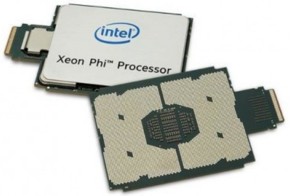
- Knights Corner (KNC) coprocessor 5110P @ 1.1GHz, 60 cores.
  TDP: 225W, RCP: N/D.
- Native / offload computing.
- Directly login via ssh.
- SIMD 512 bits.

Further tested architectures

Intel® Xeon processors

- E5-2697v3 Haswell (HSW) @ 2.3 GHz, 14-cores / socket.
  TDP: 145W, RCP: $2702.
- AVX2, FMA.

- E5-2699v4 Broadwell (BDW) @ 2.2 GHz, 22-cores / socket.
  TDP: 145W, RCP: $4115.
- AVX2, FMA.



Intel® Xeon Phi™ processor
2nd generation

- Knights Landing (KNL) Processor 7250 @ 1.4 GHz, 68 cores.
  TDP: 215W, RCP: $4876.

- Available as bootable processor.

- Binary-compatible with x86.

- High bandwidth memory.

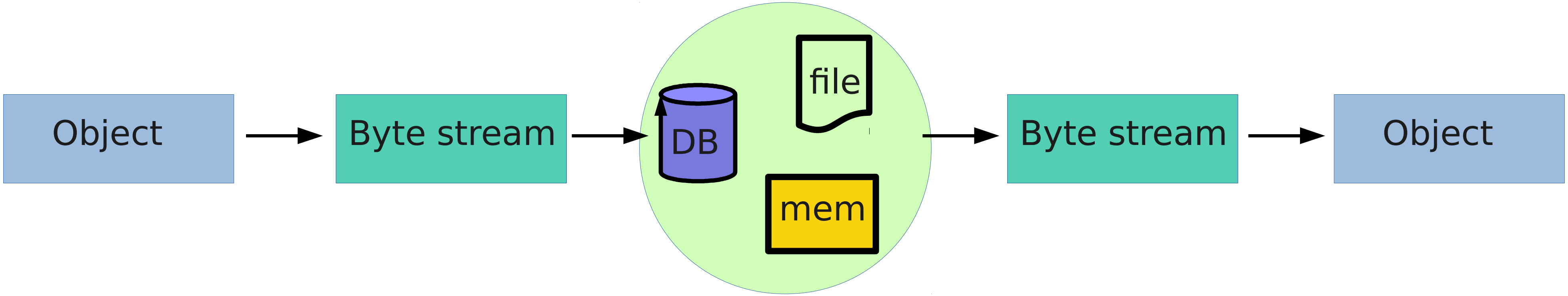
- New AVX512 instructions set.

Optimization strategy

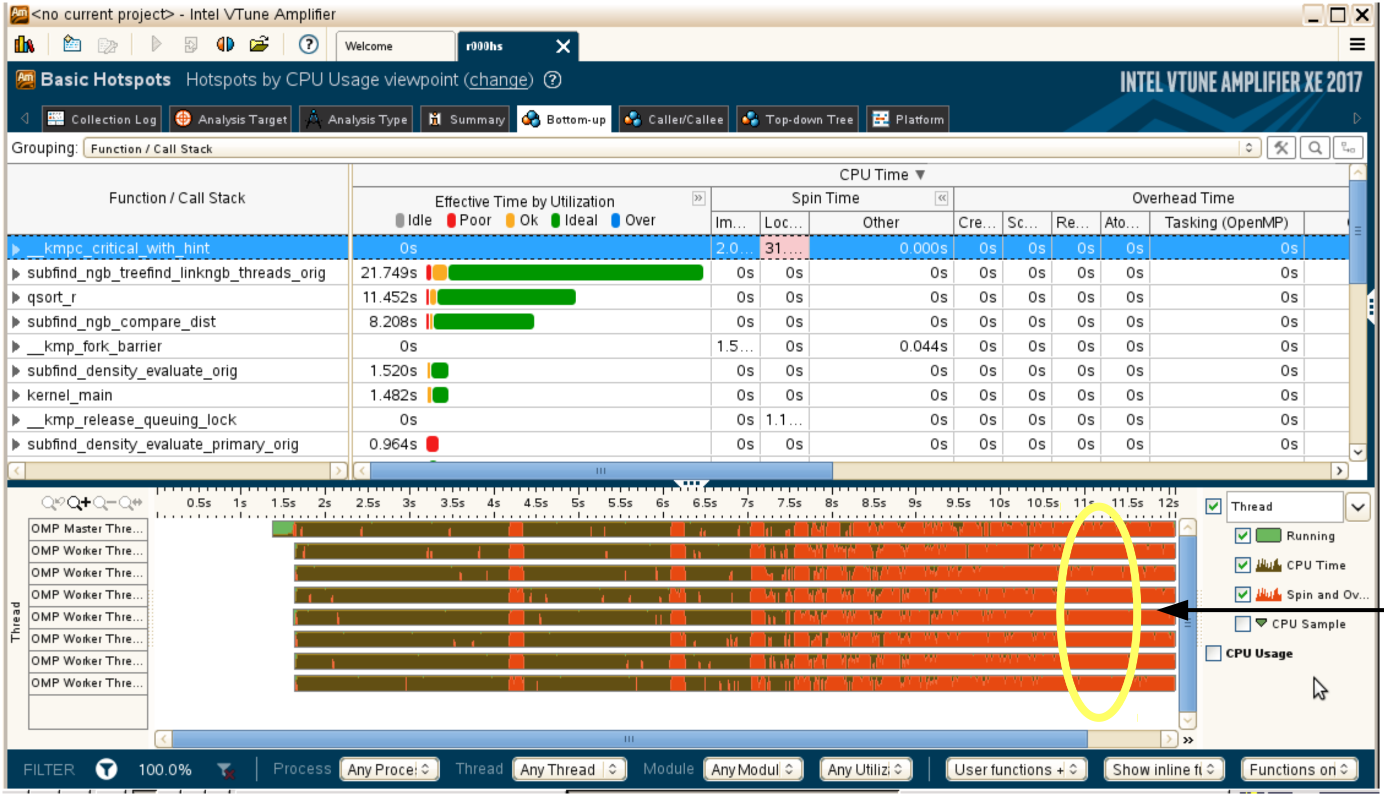
- We isolate the representative code kernel subfind_density and run it in as a stand-alone application, avoiding the overhead from the whole simulation.
- As most code components, it consists of two sub-phases of nearly equal execution time (40 to 45% for each of them), namely the neighbour-finding phase and the remaining physics computations.
- Our physics workload: ~ 500k particles. This is a typical workload per node of simulations with moderate resolution.
- We focus mainly on node-level performance.
- We use tools from the Intel® Parallel Studio XE (VTune Amplifier and Advisor).

Isolation of a kernel code

- Serialization: the process of converting data structures or objects into a format that can be stored and easily retrieved.
- This allows to isolate the computational kernel using realistic input workload (~ 551MB).
- Dumping data for compression.



Initial profiling



- Severe shared-memory parallelization overhead
- At later iterations, the particle list is locked and unlocked constantly due to the recomputation
- Spinning time 41%

thread spinning

Algorithm pseudocode

```
more_particles = partlist.length;
while(more_particles){
  int i=0;
  while(!error && i<partlist.length){
  #pragma omp parallel
  {
    #pragma omp critical
    {
       p = partlist[i++];
    }
    if(!must_compute(p)) continue;
    ngblist = find_neighbours(p);
    sort(ngblist);
    for(auto n:select(ngblist,K))
       compute_interaction(p,n);
  }
  more_particles = mark_for_recomputation(partlist);
}
```

while loop over the full particle list

each thread gets the next particle (*private p*) to process

check for computation

actual computation

Removing lock contention

```
todo_partlist = partlist;

while(partlist.length){
  error=0;
  #pragma omp parallel for schedule(dynamic)
  for(auto p:todo_partlist){
    if(something_is_wrog) error=1;
    ngblist = find_neighbours(p);
    sort(ngblist);
    for(auto n:select(ngblist,K))
        compute_interaction(p,n);
  }

//...check for any error
  todo_particles = mark_for_recomputation(partlist);
}
```

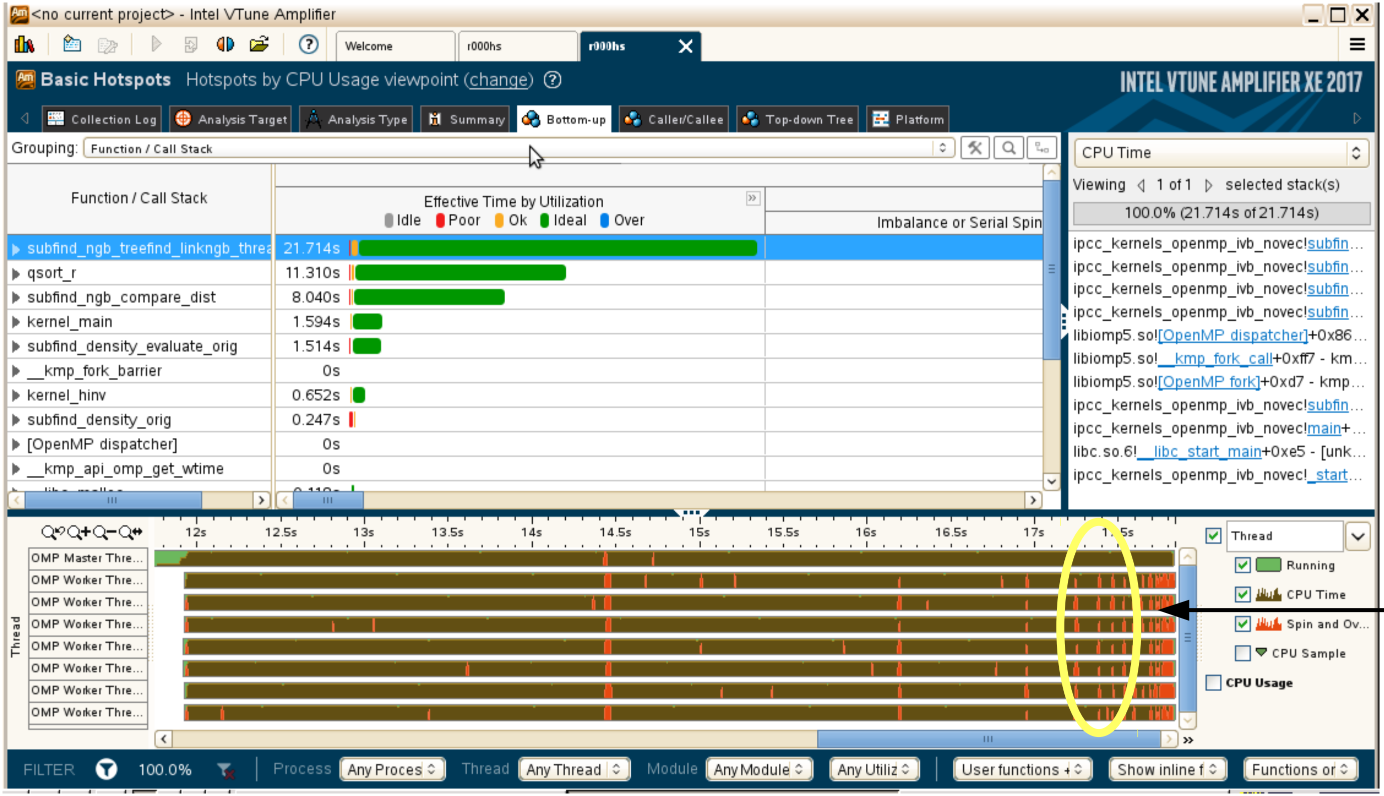
creating a ***todo*** particle list

iterations over the ***todo*** list (*private ngblist*)

actual computation

No-checks for computation

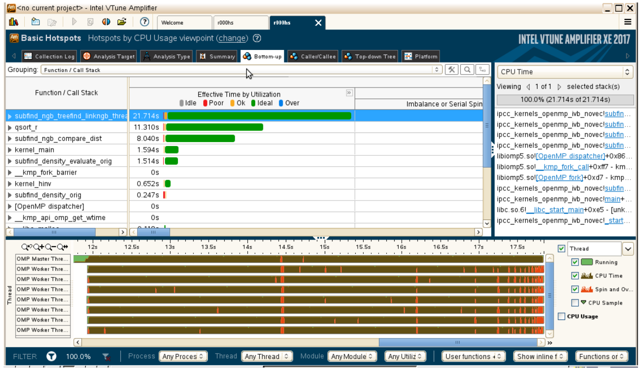
Improved performance

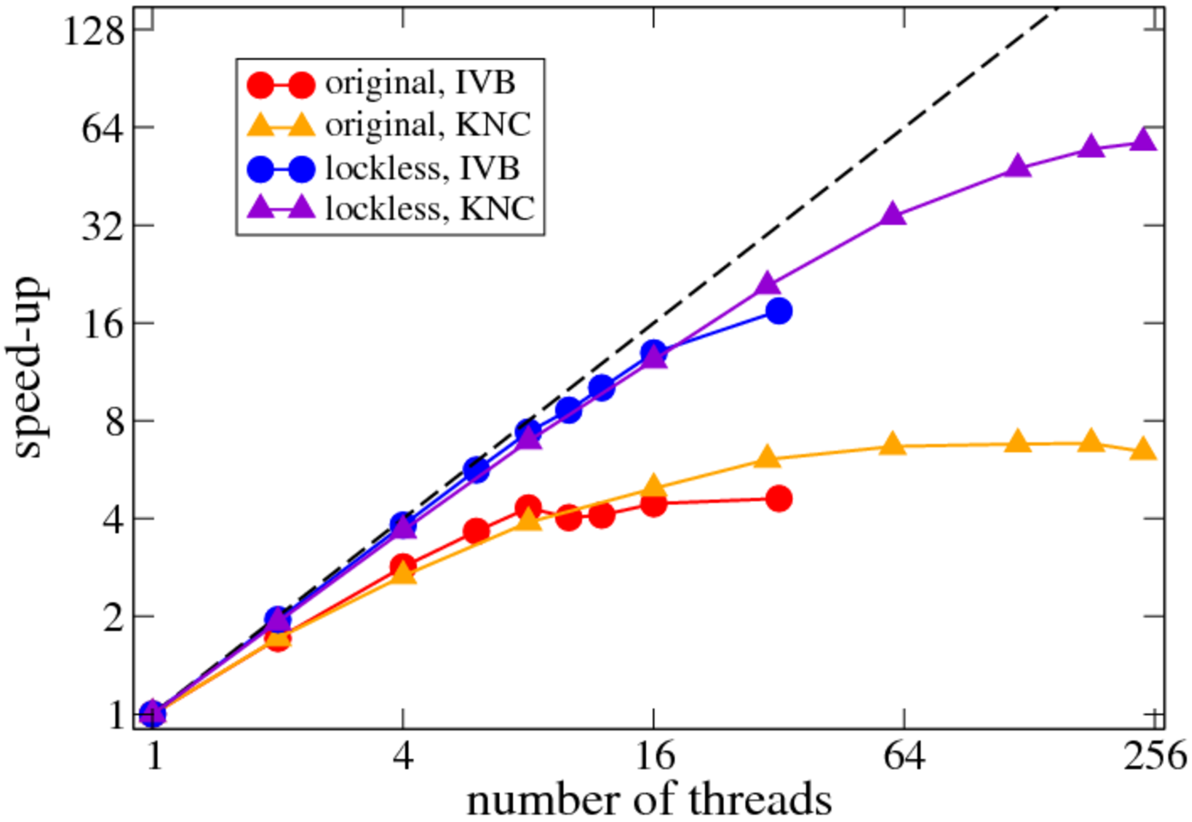


- Lockless scheme
- Time spent in spinning only 3%

no spinning

Improved speed-up





- On IVB
  - speed-up: 1.8x
  - parallel efficiency: 92%

- On KNC
  - speed-up: 5.2x
  - parallel efficiency: 57%

Obstacles to efficient auto-vectorization

```
for(n = 0, n < neighboring_particles, n++ ){
    j = ngblist[n];

    if (particle n within smoothing_length){

       inlined_function1(…, &w);
       inlined_function2(…, &w);

       rho   += P_AoS[j].mass*w;
       vel_x += P_AoS[j].vel_x;
       …
       v2 += vel_x*vel_x + … vel_z*vel_z;
    }
```

for loop over neighbors

check for computation

computing physics

Particles properties via **AoS**

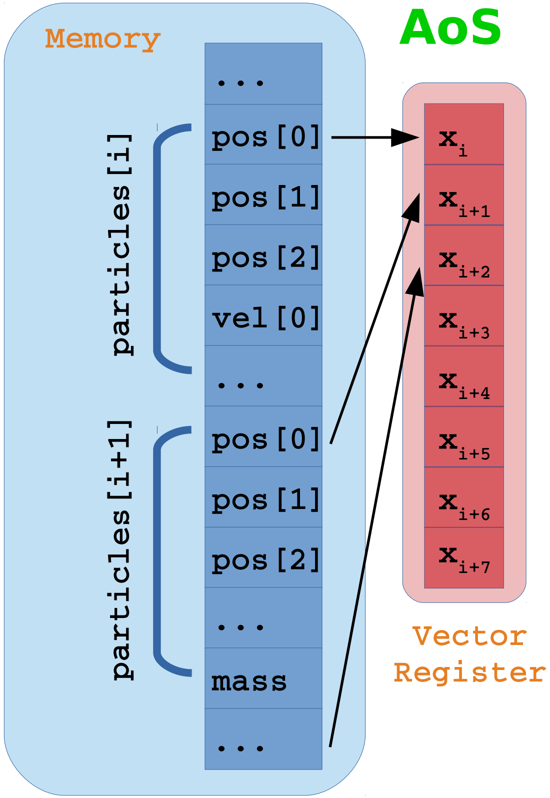
$$\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_j)$$

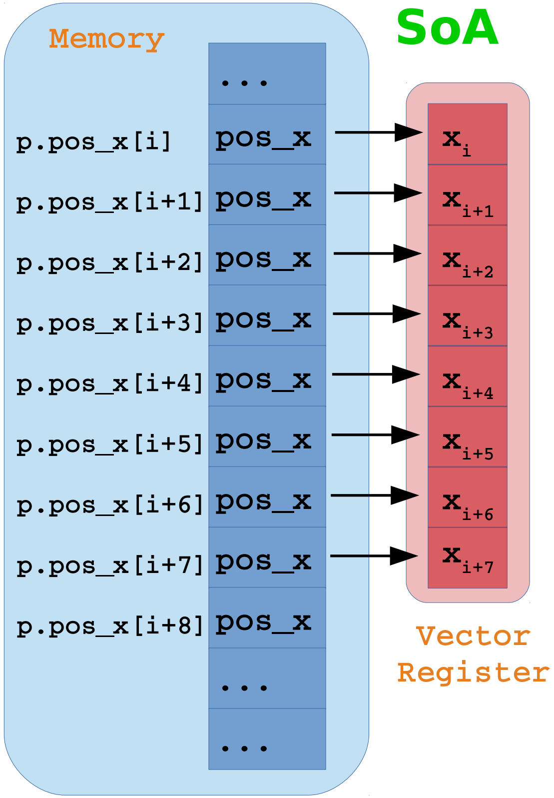
Data layout: AoS vs SoA

Automatically vectorized loops can contain loads from **not contiguous** memory locations → non-unit stride

- The compiler has issued hardware gather/scatter instructions.

```
struct ParticleAoS
{
  float pos[3];
  float vel[3];
  float mass;
}
struct ParticleSoA
{
  float *pos_x, *pos_y, *pos_z;
  float *vel_x, *vel_y, *vel_z;
  float mass;
}
```





Proposed solution: SoA

- New particle data structure: defined as Structure of Arrays (SoA).
- From the original set, only variables used in the kernel are included in the SoA: $\sim$ 60 bytes per particle.
- Software gather / scatter routines.
- Minimally invasive code changes:
  - SoA in the kernel.
  - AoS exposed to other parts of the code.

Implementation details

```
struct ParticleAoS
{
  float pos[3], vel[3], mass;
}
Particle_AoS *P_AoS;
P_AoS = malloc(N*sizeof(Particle_AoS);
```

```
struct ParticleSoA
{
  float *pos_x, … , *vel_x, …, mass;
}
Particle_SoA P_SoA;
P_SoA.pos_x = malloc(N*sizeof(float));
…
```

```
void gather_Pdata(struct Particle_SoA *dst, struct Particle_AoS *src, int N )
for(int i = 0, i < N, i++ ){
    dst -> pos_x[i] = src[i].pos[1]; dst -> pos_y[i] = src[i].pos[2]; …
}
```

```
…
rho   += P_AoS[j].mass*w;
vel_x += P_AoS[j].vel_x;
…
```

```
…
rho   += P_SoA.mass[j]*w;
vel_x += P_SoA.vel_x[j];
…
```

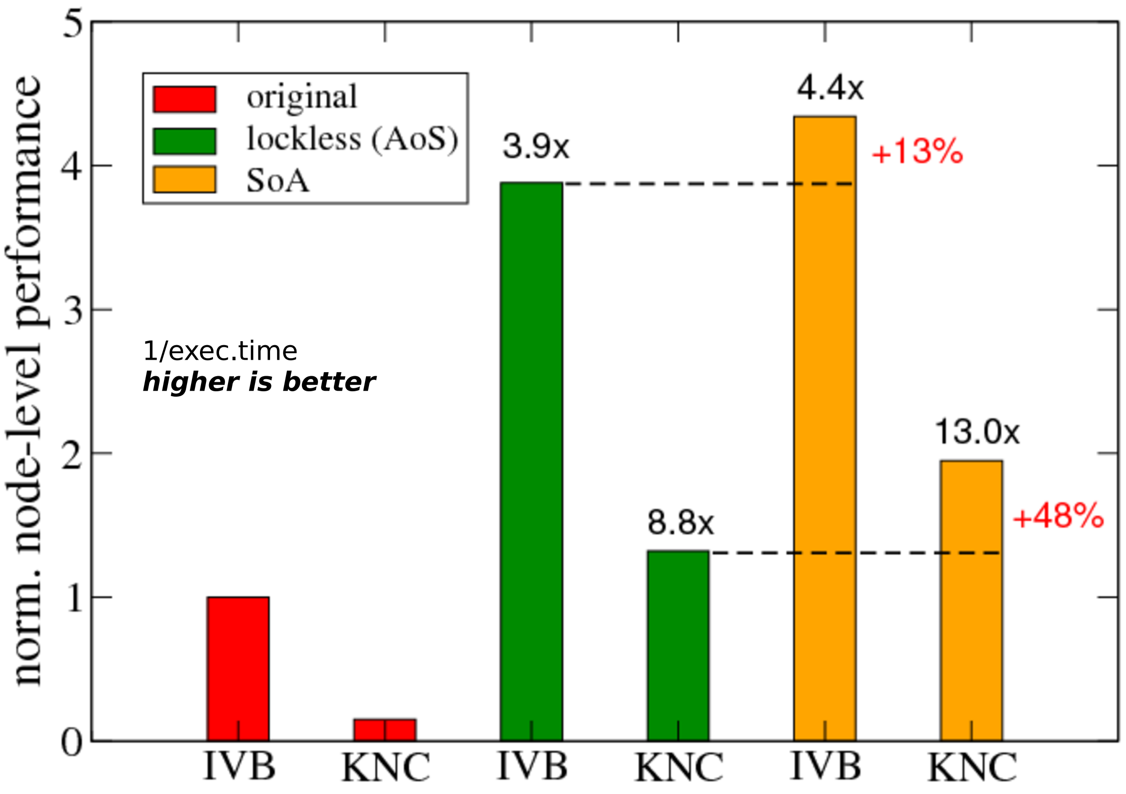
AoS to SoA: performance outcomes

- Gather+scatter overhead at most 1.8% of execution time.
→ intensive data-reuse

- Performance improvement:
- on IVB: 13%, on KNC: 48%

- Xeon/Xeon Phi performance ratio: from 0.15 to 0.45.

- The data structure is now vectorization-ready.



Optimizing for vectorization

- Modern multi/many-core architectures rely on vectorization as an additional layer of parallelism to deliver performance.
- Mind the constraint: keep Gadget readable and portable for the wide user community! Wherever possible, avoid programming in intrinsics.

- Analysis with Intel® Advisor 2016:
    - Most of the vectorization potential (10 to 20% of the workload) in the kernel "compute" loop.
    - Prototype loop in Gadget: iteration over the neighbors of a given particle.

- Similarity with many other N-body codes.

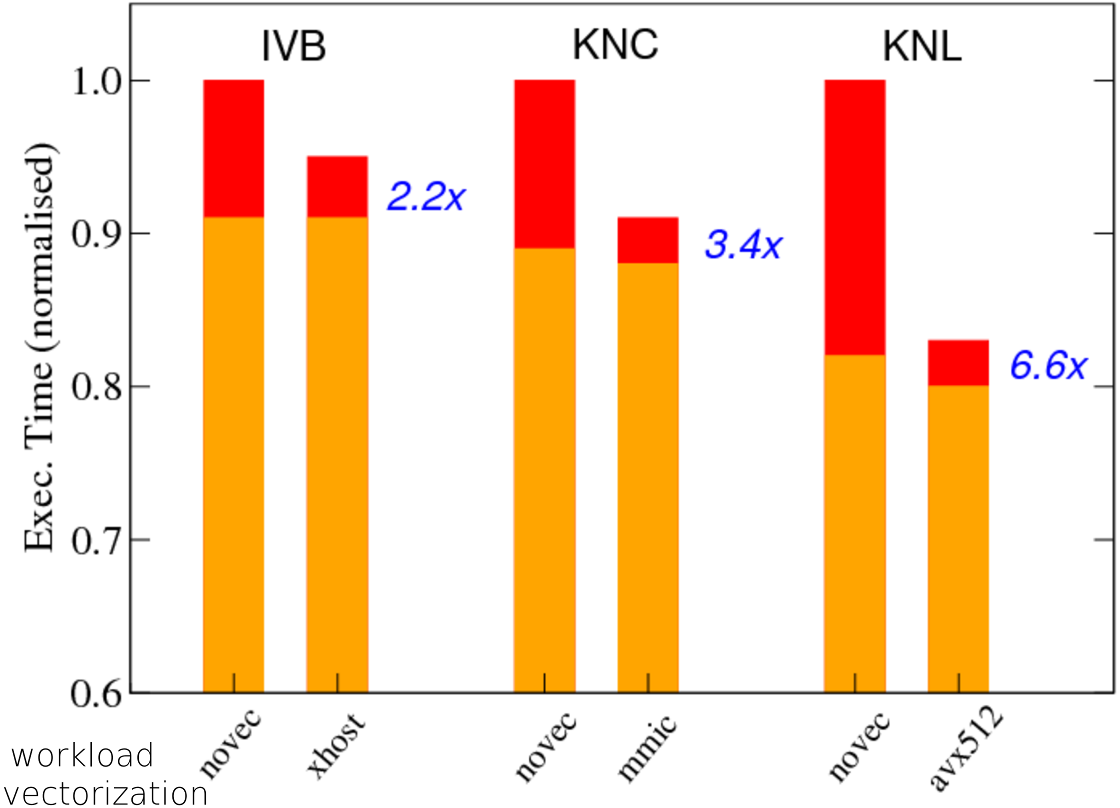
Vectorization: improvements from IVB to KNL

- Vectorization through localized masking (if-statement moved inside the inlined functions).

- Vector efficiency:

  perf. gain / vector length

  on IVB: 55%
  on KNC: 42%
  on KNL: 83%



- Yellow + red bar: kernel workload
- Red bar: target loop for vectorization

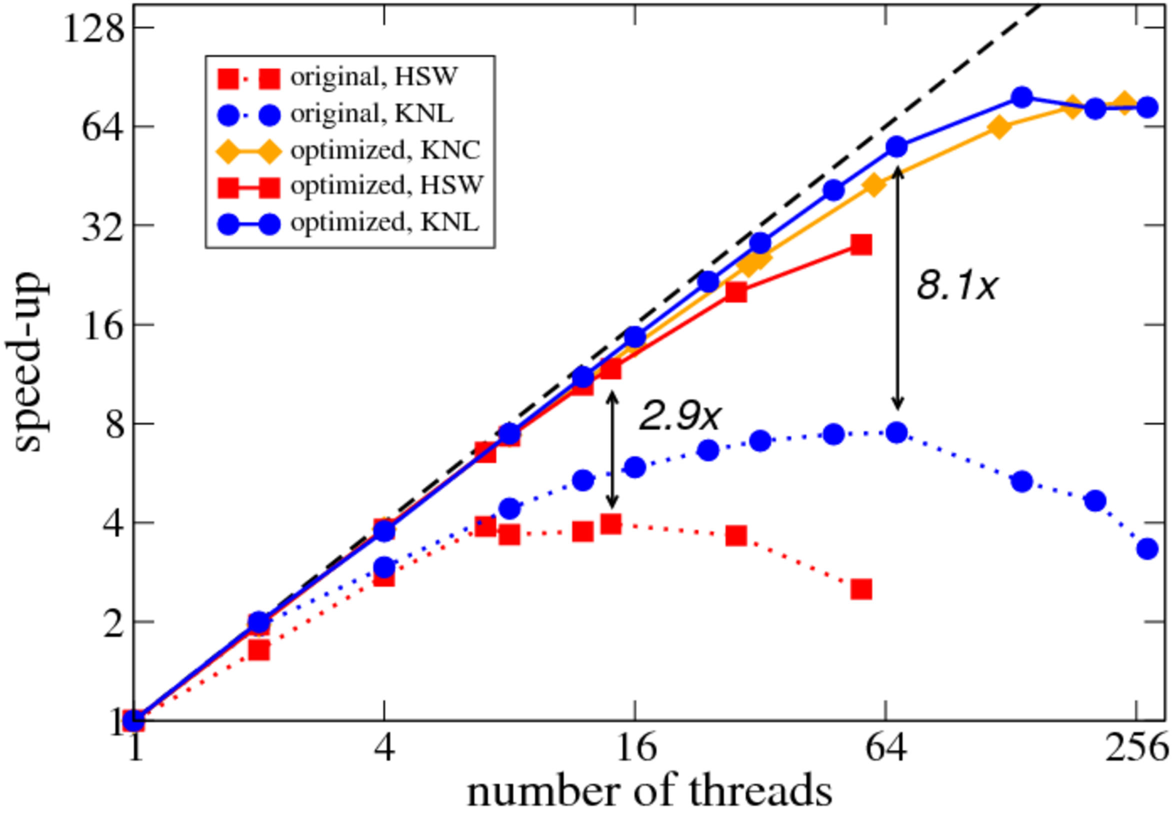
Node-level performance comparison between HSW, KNC *and* KNL

Features of the KNL tests:

- KMP Affinity: scatter;
  Memory mode: Flat;
  MCDRAM via numactl;
  Cluster mode: Quadrant.

Results:

- Our optimization improves the speed-up on all systems.
- Better threading scalability up to 136 threads on KNL.
- Hyperthreading performance is different between KNC and KNL

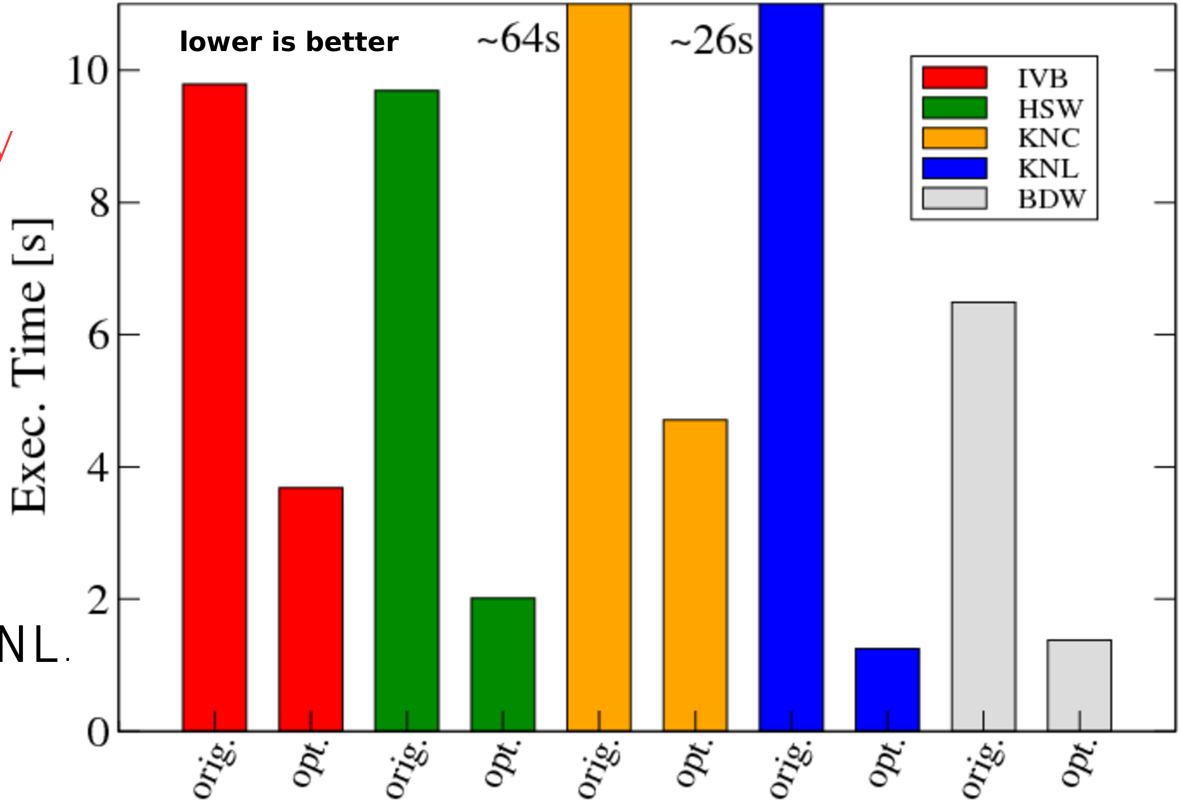


Performance comparison: first results including KNL and Broadwell

- Initial vs. optimized including all optimizations for subfind_density

- IVB, HSW, BDW: 1 socket w/o hyperthreading.
  KNC: 1 MIC, 240 threads.
  KNL: 1 node, 136 threads.

- Performance gain:
  - Xeon Phi: 13.7x KNC, 20.1x KNL.
  - Xeon: 2.6x IVB, 4.8x HSW, 4.7x BDW.



Summary and outlook

- Code modernization as the iterative process for improving the performance of an HPC application.
- Our IPCC example: P-Gadget3.

  Threading parallelism
  Data layout
  Vectorization

  Key points of our work, guided by analysis tools.
- This effort is (mostly) portable! Good performance found on new architectures (KNL and BDW) basically out-of-the-box.
- For KNL, architecture-specific features (MCDRAM, large vector registers and NUMA characteristics) are currently under investigation for different workloads.
- Investment on the future of well-established community applications, and crucial for the effective use of forthcoming HPC facilities.

https://arxiv.org/abs/1612.06090

Acknowledgements

- Research supported by the Intel® Parallel Computing Center program.
- Project coauthors: Nicolay J. Hammer (LRZ), Vasileios Karakasis (CSCS).
- P-Gadget3 developers: Klaus Dolag, Margarita Petkova, Antonio Ragagnin.
- Research collaborator at Technical University of Munich (TUM): Nikola Tchipev.
- TCEs at Intel: Georg Zitzlsberger, Heinrich Bockhorst.
- Thanks to the IXPUG community for useful discussion.
- Special thanks to Colfax Research for proposing this contribution to the MC² Series, and for granting access to their computing facilities.