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

Dr. Luigi Iapichino
Leibniz Supercomputing Centre

KNL Workshop, CINECA
March 22nd - 23rd, 2017
Work contributors

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

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ
- Expert in performance optimization and HPC systems

Dr. Luigi Iapichino  
Scientific Computing Expert  
Leibniz Supercomputing Centre

- Member of the Intel Parallel Computing Center (IPCC) @ LRZ
- Expert in computational astrophysics and simulations

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

Preprint of this work: https://arxiv.org/abs/1612.06090
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.

- First developed in the late 90s as **serial** code, later evolved as an MPI and a hybrid 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:

\[
\rho_i = \rho(\mathbf{r}_i) = \sum_j m_j W(|\mathbf{r}_i - \mathbf{r}_j|, h_j)
\]
Target architectures for our project

**Intel® Xeon processor**
- E5-2650v2 Ivy-Bridge (IVB) @ 2.6 GHz, 8-cores / socket. TDP: 95W, RCP (03/2017): $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. 
- AVX2, FMA.
- E5-2699v4 Broadwell (BDW) @ 2.2 GHz, 22-cores / socket. 
- AVX2, FMA.

Intel® Xeon Phi™ processor 2nd generation

- Knights Landing (KNL) Processor 7250 @ 1.4 GHz, 68 cores. 
- 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 on node-level performance, through minimally invasive changes.

- We use tools from the Intel® Parallel Studio XE (VTune Amplifier and Advisor).

Simulation details: www.magneticum.org
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%

Multi-threading parallelism
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);
    }
}
Improved performance

- Lockless scheme: lock contention removed through "todo" particle list and OpeMP dynamic scheduling.
- Time spent in spinning only 3%

Multi-threading parallelism
Improved speed-up

- On **IVB** @ 8 threads
  - speed-up: **1.8x**
  - parallel efficiency: **92%**

- On **KNC** @ 60 threads
  - speed-up: **5.2x**
  - parallel efficiency: **57%**

Multi-threading parallelism
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;
    }
}
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.

```c
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: ~60 bytes per particle.

- Software **gather / scatter** routines.

- Minimally invasive code changes:
  - **SoA** in the kernel.
  - **AoS** exposed to other parts of the code.
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.

Performance results
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.
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 granting access to their computing facilities.
Back-up: SoA implementation details

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

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;
...
```

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

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