Towards modernisation of the Gadget code on many-core architectures

Fabio Baruffa, Luigi Iapichino (LRZ)
Overview

- Modernising P-Gadget3 for the Intel® Xeon Phi™: code features, challenges and strategy for optimisation.
- Threading parallelism: minimising lock contention.
- Data layout: from AoS to SoA.
- Vectorisation: performance bottlenecks and proposed solution.
- First performance evaluation on KNL.

This work is done in the framework of the Intel® Parallel Computing Centre ExScaMIC (LRZ-TUM). Thanks to our collaborator N. Hammer (LRZ) and to our project partners K. Dolag and M. Petkova (USM München).
Gadget: numerical simulations of cosmological structure formation

- Leading application for the simulation of the build-up of the cosmic large-scale structure (galaxies and cluster of galaxies) and of processes at sub-resolution scales (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).
The code can be run at different levels of complexity:

- N-Body-only (a.k.a. Dark Matter) simulations
- N-Body + gas component
- Additional physics (sub-resolution) modules: Radiative cooling, star formation, chemical reaction network…
- The additional physics increases the memory requirement per particle up to ~ 1Kb (x10 wrt DM-only)
Features and complications 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.
- This puts significant constraints on the development:
  - Portability on all modern architectures (Intel® Xeon/MIC, Power, GPU,…);
  - Readibility for non-experts in HPC;
  - Do not break existing functionalities.
- The code consists of ~200 files, ~400k code lines, and makes extensive use of #IFDEF.
- External library dependencies: FFTW, GSL, HDF5.
Best approach for optimising the code

- Choice of a reasonable test case to benchmark (small/large, type of workload…).
- MPI Profiling: ITAC, Scalasca, …
- Vector utilisation: Advisor, coarse-grained timing.
- Mini-App approach in complex codes.
- In our example: isolation of the target kernel through serialisation.
Performance characteristics and optimisation strategy

- Initial analysis: most of the code components consist of two sub-phases of nearly equal execution time (40 to 45% for each of them):

  **Neighbour-finding phase**
  - Low floating-point rate
  - Lots of branches (~20%)
  - High L2 miss ratio (~36%)
  - Typical „pointer-chasing“ problem
  - Not easily amenable to be ported on Intel® Xeon Phi™

  **Physics computations**
  - High floating-point rate
  - 25% of the peak scalar fp performance
  - Low or sustainable cache and memory b/w requirements
  - Accesses (usually irregular) to array of huge data structures, **data cache misses**

- „Physics computations“ are more suitable for the optimization on Intel® Xeon Phi™.

- **Isolation of a typical kernel (subfind_density):**
  - Run as a stand-alone separate kernel (same input as original: sandbox model!).
  - Avoid the overhead of the whole simulation → Quick prototyping, allows native mode on the KNC.
  - Later: port optimizations back to the original code.
Kernel: serialisation and verification

- Serialisation: the process of translating data structures or objects state into a format that can be stored and easily retrieve
- This allows to isolate the computational kernels using realistic input workload
- Dumping data for comparison
Initial profiling (Intel® VTune™ Amplifier XE)

The initial analysis shows a severe shared-memory parallelization overhead.
Algorithm restructuring:

- Minimisation of the lock contention issue.
- Non-intrusive changes in the shared-memory implementation.
- Iteration only on the particles that really need to be recomputed at every step.

Original particle interaction scheme (pseudocode) before lock contention fix.

```plaintext
more_particles = ... # all particles
while more_particles:
    p = <first particle>
    while p:
        do in parallel:
            p = get_next_particle_atomic(partlist)    # LOCKS!
            if not should_compute(p):
                continue
            ngblist = find_neighbors(p)
            foreach n in ngblist:
                compute_interactions(p, n)
            more_particles = mark_particles_for_recomputation(partlist)
```

29.06.2016
Leibniz Supercomputing Centre
Improved performance

Intel® Xeon host:
- 91% efficiency on a single socket;
- 3.4x faster node-level performance;

Intel® Xeon Phi™:
- 5.5x improvement @ 120 threads;
- Locking still a problem at high thread counts.

Even better solution: lockless implementation (OpenMP dynamic scheduling)
What’s wrong with data layout?

- Modern **SIMD** architecture allows to apply the same instruction to multiple data elements

```
Struct Particle {
    float px, py, pz;
    float vx, vy, vz;
    ...
    P = Particle[N];  // AoS
```

```
Struct Particle {
    float *px, *py, *pz;
    float *vx, *vy, *vz;
    ...
    P.px = malloc [N];  // SoA
```

**Example**: Shows the storage of particles in **Arrangement of Slices (AoS)** vs. **Stride of Arrays (SoA)**.

<table>
<thead>
<tr>
<th>AoS</th>
<th>SoA</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[i].px</td>
<td>P.px[i]</td>
</tr>
<tr>
<td>P[i].py</td>
<td>P.px[i+1]</td>
</tr>
<tr>
<td>P[i].pz</td>
<td>P.px[i+2]</td>
</tr>
<tr>
<td>P[i].vx</td>
<td>P.px[i+3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>vec_reg1</th>
<th>vec_reg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec_reg1</td>
<td>vec_reg2</td>
</tr>
</tbody>
</table>

29.06.2016
Leibniz Supercomputing Centre
Current data organisation: Array of Structures (AoS), 224 bytes per particle.

Motivation: highly optimized for performance at large MPI task numbers.

Outcome: data cache misses, code is memory bound.
- Average memory B/W consumed: 5.5 GB/s (peak ≈16.5 GB/s)

Data structure hinders vectorisation.

In the kernel: ~ 17 iterations, 1.5M particles to be processed.
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.
- Gather from old to new data structure, compute with it, scatter back to old. Example of change in the data structure approach:

```c
void gather_particle_data(struct new_particle_data *dst, const struct particle_data *src, size_t N) {
    int i;

    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        dst->Vel[1][i] = src[i].Vel[1];
        dst->Vel[2][i] = src[i].Vel[2];
        dst->Type[i] = src[i].Type;
        dst->ID[i] = src[i].ID;
    }
}
```

```c
v2 += P[j].Vel[0]*P[j].Vel[0] +
```

```c
v2 += NewPart.Vel[0][j]*NewPart.Vel[0][j]
    + NewPart.Vel[1][j]*NewPart.Vel[1][j] +
    NewPart.Vel[2][j]*NewPart.Vel[2][j];
```
Performance outcomes

- Gather+scatter overhead small when compared both to execution time and to performance gain.
- Node-level performance improvement: +22% on the Xeon, +41% on the Xeon Phi™ (KNC).
- Xeon/Xeon Phi™ performance ratio: from 0.15 to 0.28
- According to VTune analysis, the bottleneck on memory latency (caused by cache misses) is solved.
- Current B/W consumption decreased to ≈ 2.5 GB/s, because of much lower data cache misses.
- The data structure is now vectorization-ready.
Improving vectorisation in the Gadget kernel

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

- Analysis with Intel® Advisor 2016:
  - Most of the vectorisation potential (10 to 20% of the workload) in the kernel “compute” loop.
  - Prototype loop in the Gadget code: iteration on the neighbours of a given particle.

- Similarity with many other N-body codes.
Obstacles to vectorization efficiency - pseudocode

for (n = 0, n < neighbouring particles (selected)) {
    j = ngblist[n];       // getting the index from the particle data structure (SoA)
    if (particle n within smoothing length) {               // Problem 1: if statement
        inlined_function1(…..);
        inlined_function2(…..);
    }
    vx += NewPart.Vel[0][j]; // Problem 2: indirect (strided) access to the data
    ...
    v2 += NewPart.Vel[0][j] * NewPart.Vel[0][j] + …; // additional load
    // (unnecessary): why does the compiler not reuse it from the register?
}

29.06.2016 Leibniz Supercomputing Centre
for (n = 0, n < neighbouring particles (selected)) {
    j = ngblist[n]; // getting the index from the particle data structure (SoA)
    inlined_function1(…..); // the if condition is moved inside the function
    inlined_function2(…..);
    vel1 = NewPart.Vel[0][j]; // still strided data access: next exposed hotspot
    ...
    vx += vel1; // optimised data load
    ...
    v2 += vel1 * vel1 + … ;
}
LOOP BEGIN at kernels/subfind_stripped.c(293,13) inlined into kernels/subfind_stripped.c(72,13)

....
remark #15328: vectorization support: gather was emulated for the variable NewPart.Mass:
   indirect access [ kernels/subfind_stripped.c(308,30) ]
remark #15328: vectorization support: gather was emulated for the variable NewPart.Vel:
   indirect access [ kernels/subfind_stripped.c(312,25) ]

....
remark #15305: vectorization support: vector length 4

....
remark #15300: LOOP WAS VECTORIZED

....
remark #15478: estimated potential speedup: 3.670
remark #15487: type converts: 2

....
LOOP END
Vectorisation: improvements from HSW to KNL

- Vectorisation of the kernel main “compute” loop (red bar) through better localised masking.

- On KNL: measured loop speed-up 6.6x. A vector efficiency of 83% is reached without using intrinsics.

- Both on HSW and KNL, vectorisation provides some performance improvement also in other parts of the kernel.
Node-level performance comparison between HSW, KNC and KNL

Features of the KNL tests:
- native runs on Xeon Phi™ 7210 @ 1.30GHz (KNL), 64 cores
- Intel® compiler 2016, -xmic-avx512
- KMP Affinity: scatter; Memory mode: Flat; Cluster mode: Quadrant.

Results:
- Previous optimisations (data layout, vectorisation) improved the speedup on all systems, by different factors.
- KNL scalability slightly better than HSW and KNC up to 128 threads.
- Necessity of using hyperthreading can be different between KNC and KNL.
Performance comparison: first results including KNL

- Initial version vs. vectorised including all optimisations.

- IVB, HSW: 1 socket w/o hyperthreading. KNC: 1 MIC, 240 threads. KNL: 1 node, 128 threads.

- Performance gain for Xeon Phi™ larger than for Xeon.

- Single-core execution time on KNL: 3.3x faster than KNC.
Summary

- Code modernisation as the iterative process for improving the performance of an HPC application.
- Our IPCC example: Gadget3.
  - Threading parallelism
  - Data layout
  - Key points of our work, guided by analysis tools.
  - Vectorisation
- This effort is (mostly) portable! Good performance found on new architectures (KNL) basically out-of-the-box.
- Investment on the future of well-established community applications, and crucial for the effective use of forthcoming HPC facilities.