Leibniz-Rechenzentrum
der Bayerischen Akademie der Wissenschaften

Towards modernisation of the Gadget code on many-core architectures

Fabio Baruffa, Luigi Iapichino (LRZ)

Overview

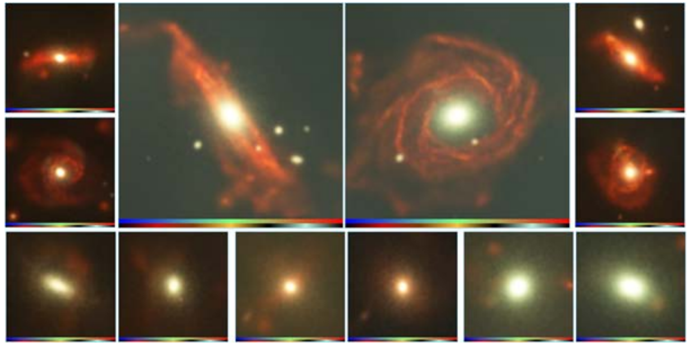
- Modernising P-Gadget3 for the Intel® Xeon Phi™: code features, challenges and strategy for optimisation.
- Threading parallelism: minimising lock contention.
- Data layout: from AoS to SoA.
- Vectorisation: performance bottlenecks and proposed solution.
- First performance evaluation on KNL.

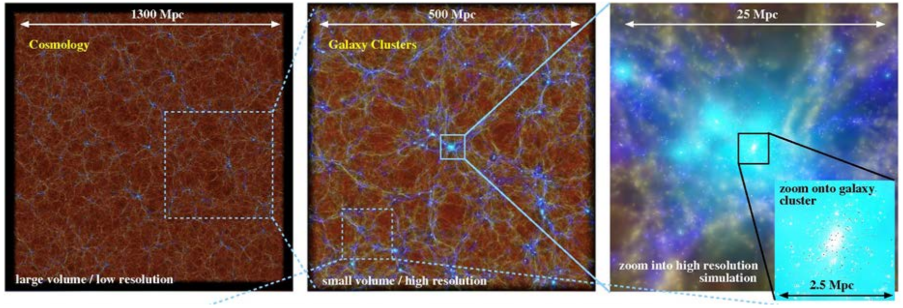
This work is done in the framework of the Intel® Parallel Computing Centre *ExScaMIC* (LRZ-TUM). Thanks to our collaborator N. Hammer (LRZ) and to our project partners K. Dolag and M. Petkova (USM München).

Gadget: numerical simulations of cosmological structure formation

nature
GENOME EDITING
Rewriting the rules for gene therapy
BCL-2 INHIBITORS
Potent new antitumour compounds
HUMAN BEHAVIOUR
Oxytocin — the 'trust hormone'
SURPRISING DINOSAURS
A sauropod, by a short neck
EVOLUTION OF THE UNIVERSE
Supercomputer simulation of the growth of 20 million galaxies

- 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).

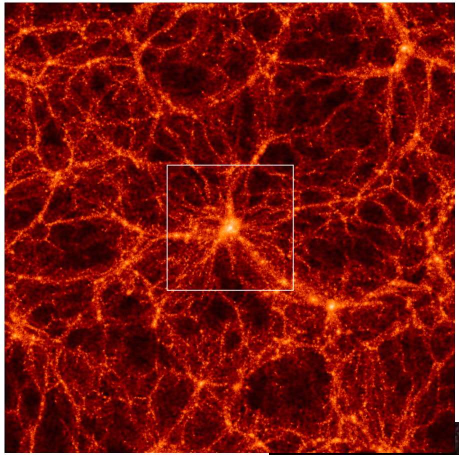



1300 Mpc
Cosmology
large volume / low resolution
500 Mpc
Galaxy Clusters
small volume / high resolution
25 Mpc
zoom into high resolution simulation
zoom onto galaxy cluster
2.5 Mpc

Challenges in GADGET simulations

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, …
- Node level performance metrics: VTune Amplifier, likwid, Allinea Map.
- 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™

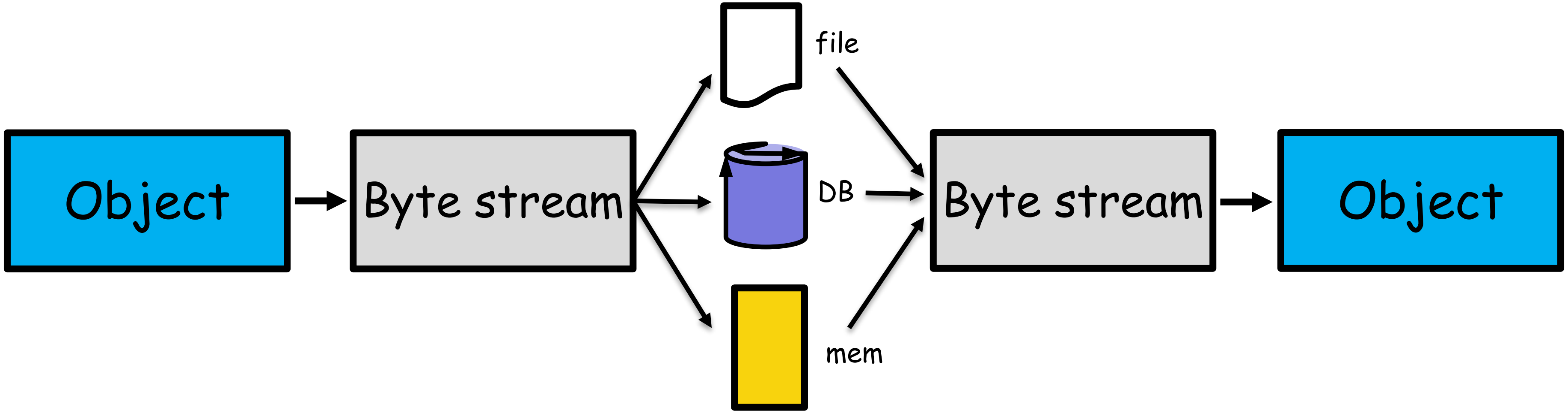
Physics computations

- 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**

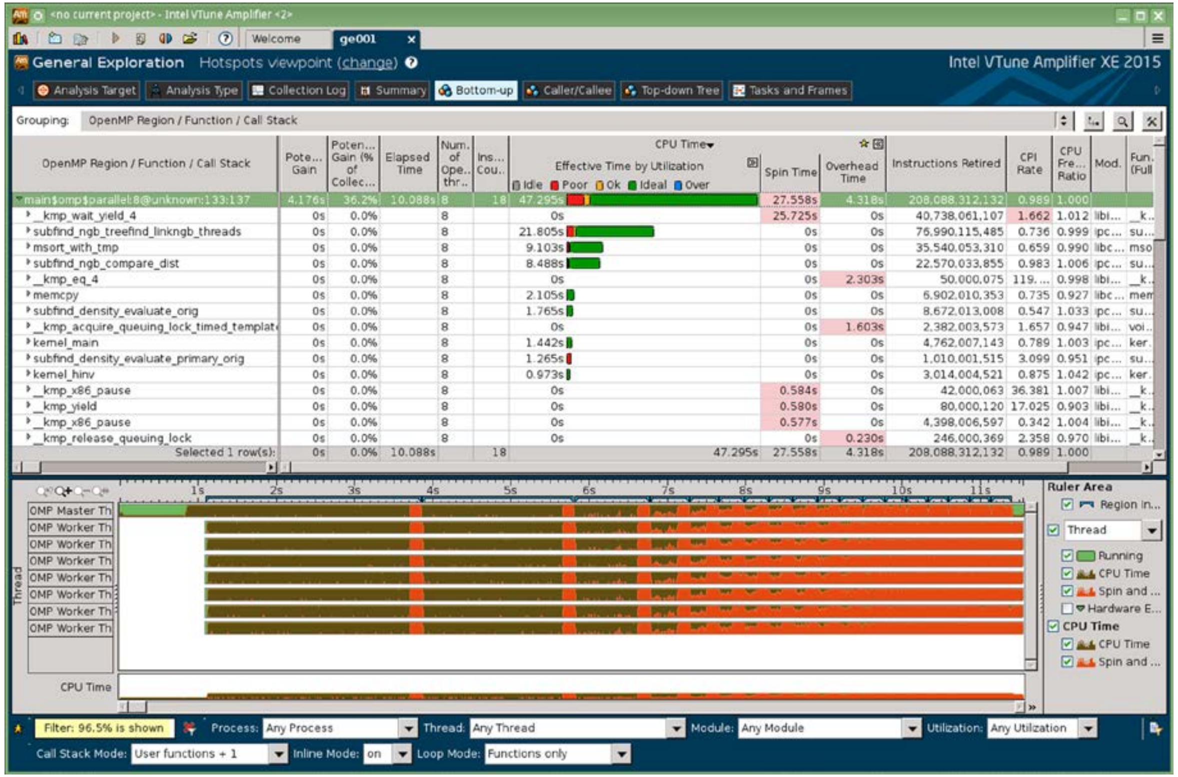
- „Physics computations“ are more suitable for the optimization on Intel® Xeon Phi™.
- Isolation of a typical kernel (subfind_density):
  - Run as a stand-alone separate kernel (same input as original: sandbox model!).
  - Avoid the overhead of the whole simulation → Quick prototyping, allows native 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)



<no current project> - Intel VTune Amplifier <2>

General Exploration Hotspots viewpoint (change)

Intel VTune Amplifier XE 2015

Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	Pote... Gain	Poten... Gain (% of Collec...	Elapsed Time	Num. of Ope.. thr..	Ins... Cou..	CPU Time: Effective Time by Utilization	CPU Time: Spin Time	CPU Time: Overhead Time	Instructions Retired	CPI Rate	CPU Fre... Ratio	Mod.	Fun. (Full
main$omp$parallel:8@unknown:133:137	4.176s	36.2%	10.088s	8	18	47.295s	27.558s	4.318s	208,088,312,132	0.989	1.000		
__kmp_wait_yield_4	0s	0.0%		8		0s	25.725s	0s	40,738,061,107	1.662	1.012	libi...	__k..
subfind_ngb_treefind_linkngb_threads	0s	0.0%		8		21.805s	0s	0s	76,990,115,485	0.736	0.999	pc...	su...
msort_with_tmp	0s	0.0%		8		9.103s	0s	0s	35,540,053,310	0.659	0.990	libc...	mso
subfind_ngb_compare_dist	0s	0.0%		8		8.488s	0s	0s	22,570,033,855	0.983	1.006	pc...	su...
__kmp_eq_4	0s	0.0%		8		0s	0s	2.303s	50,000,075	119. ...	0.998	libi...	__k..
memcpy	0s	0.0%		8		2.105s	0s	0s	6,902,010,353	0.735	0.927	libc...	mem
subfind_density_evaluate_orig	0s	0.0%		8		1.765s	0s	0s	8,672,013,008	0.547	1.033	pc...	su...
__kmp_acquire_queuing_lock_timed_templat	0s	0.0%		8		0s	0s	1.603s	2,382,003,573	1.657	0.947	libi...	voi..
kernel_main	0s	0.0%		8		1.442s	0s	0s	4,762,007,143	0.789	1.003	pc...	ker.
subfind_density_evaluate_primary_orig	0s	0.0%		8		1.265s	0s	0s	1,010,001,515	3.099	0.951	pc...	su...
kernel_hinv	0s	0.0%		8		0.973s	0s	0s	3,014,004,521	0.875	1.042	pc...	ker.
__kmp_x86_pause	0s	0.0%		8		0s	0.584s	0s	42,000,063	36.381	1.007	libi...	__k..
__kmp_yield	0s	0.0%		8		0s	0.580s	0s	80,000,120	17.025	0.903	libi...	__k..
__kmp_x86_pause	0s	0.0%		8		0s	0.577s	0s	4,398,006,597	0.342	1.004	libi...	__k..
__kmp_release_queuing_lock	0s	0.0%		8		0s	0s	0.230s	246,000,369	2.358	0.970	libi...	__k..
Selected 1 row(s):	0s	0.0%	10.088s		18	47.295s	27.558s	4.318s	208,088,312,132	0.989	1.000		

Filter: 96.5% is shown Process: Any Process Thread: Any Thread Module: Any Module Utilization: Any Utilization

Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

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

Pseudocode

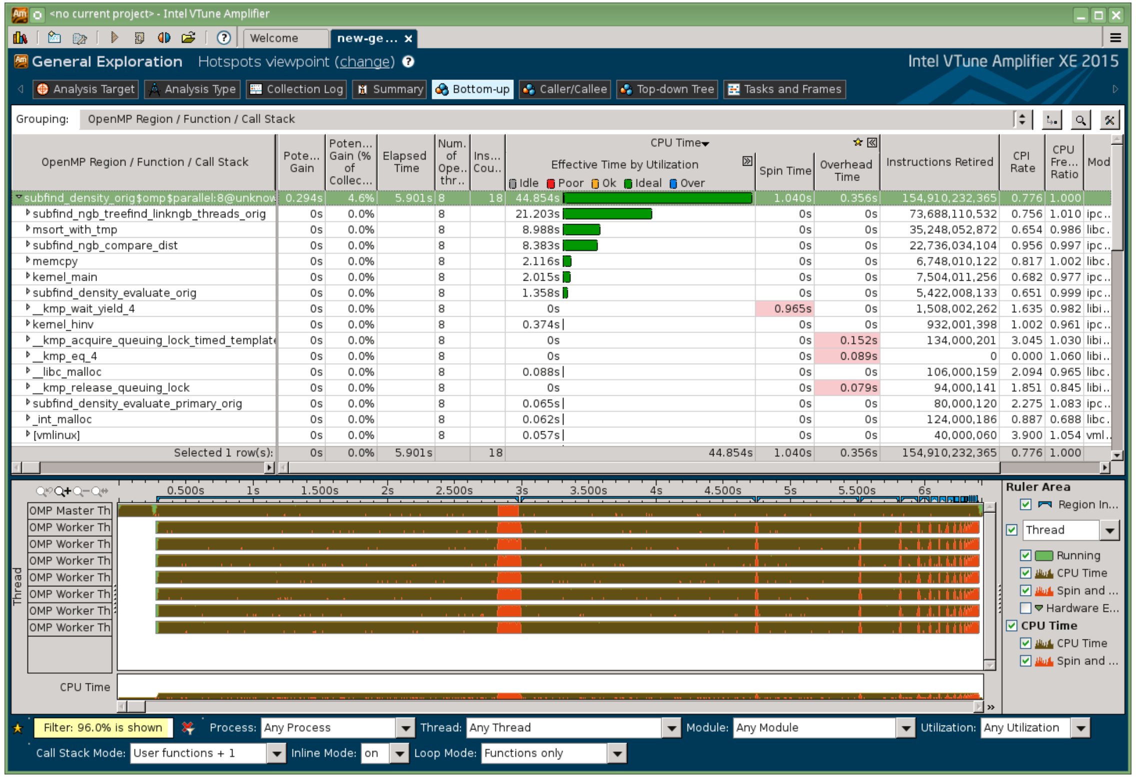
Original particle interaction scheme (pseudocode) before lock contention fix.

```
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)
```

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.

Improved performance

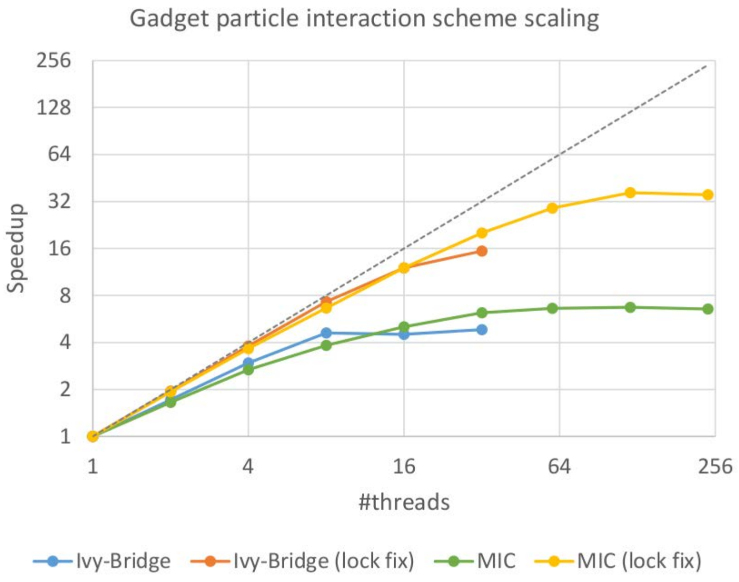


Grouping: OpenMP Region / Function / Call Stack

OpenMP Region / Function / Call Stack	Pote... Gain	Poten... Gain (% of Collec...	Elapsed Time	Num. of Ope.. thr..	Ins... Cou..	CPU Time: Effective Time by Utilization (▯ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	CPU Time: Spin Time	CPU Time: Overhead Time	Instructions Retired	CPI Rate	CPU Fre... Ratio	Mod
subfind_density_orig$omp$parallel:8@unknow	0.294s	4.6%	5.901s	8	18	44.854s	1.040s	0.356s	154,910,232,365	0.776	1.000	
subfind_ngb_treefind_linkngb_threads_orig	0s	0.0%		8		21.203s	0s	0s	73,688,110,532	0.756	1.010	ipc..
msort_with_tmp	0s	0.0%		8		8.988s	0s	0s	35,248,052,872	0.654	0.986	libc.
subfind_ngb_compare_dist	0s	0.0%		8		8.383s	0s	0s	22,736,034,104	0.956	0.997	ipc..
memcpy	0s	0.0%		8		2.116s	0s	0s	6,748,010,122	0.817	1.002	libc.
kernel_main	0s	0.0%		8		2.015s	0s	0s	7,504,011,256	0.682	0.977	ipc..
subfind_density_evaluate_orig	0s	0.0%		8		1.358s	0s	0s	5,422,008,133	0.651	0.999	ipc..
__kmp_wait_yield_4	0s	0.0%		8		0s	0.965s	0s	1,508,002,262	1.635	0.982	libi..
kernel_hinv	0s	0.0%		8		0.374s	0s	0s	932,001,398	1.002	0.961	ipc..
__kmp_acquire_queuing_lock_timed_template	0s	0.0%		8		0s	0s	0.152s	134,000,201	3.045	1.030	libi..
__kmp_eq_4	0s	0.0%		8		0s	0s	0.089s	0	0.000	1.060	libi..
__libc_malloc	0s	0.0%		8		0.088s	0s	0s	106,000,159	2.094	0.965	libc.
__kmp_release_queuing_lock	0s	0.0%		8		0s	0s	0.079s	94,000,141	1.851	0.845	libi..
subfind_density_evaluate_primary_orig	0s	0.0%		8		0.065s	0s	0s	80,000,120	2.275	1.083	ipc..
_int_malloc	0s	0.0%		8		0.062s	0s	0s	124,000,186	0.887	0.688	libc.
[vmlinux]	0s	0.0%		8		0.057s	0s	0s	40,000,060	3.900	1.054	vml..
Selected 1 row(s):	0s	0.0%	5.901s		18	44.854s	1.040s	0.356s	154,910,232,365	0.776	1.000	

Filter: 96.0% is shown | Process: Any Process | Thread: Any Thread | Module: Any Module | Utilization: Any Utilization

Call Stack Mode: User functions + 1 | Inline Mode: on | Loop Mode: Functions only



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.

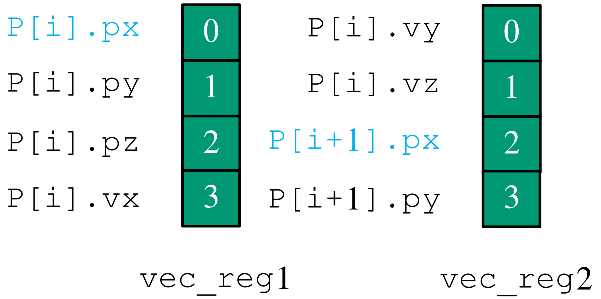
Even better solution: lockless implementation (OpenMP dynamic scheduling)

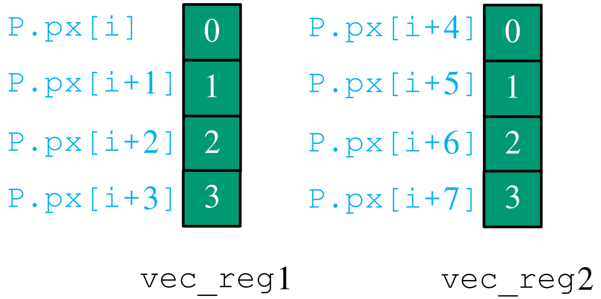
What‘s wrong with data layout?

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

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

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





Implementation in Gadget

- 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:

```
struct new_particle_data
{
MyDoublePos *Pos[3];
MyFloat *Vel[3];
short int *Type;
MyIDType *ID;
MyFloat *Mass;
int *DM_NumNgb;
MyFloat *DM_Hsml;
MyFloat *DM_Density;
MyFloat *DM_VelDisp;
};
```

```
v2 += P[j].Vel[0]*P[j].Vel[0] +
P[j].Vel[1]*P[j].Vel[1] + P[j].Vel[2]*P[j].Vel[2];
```

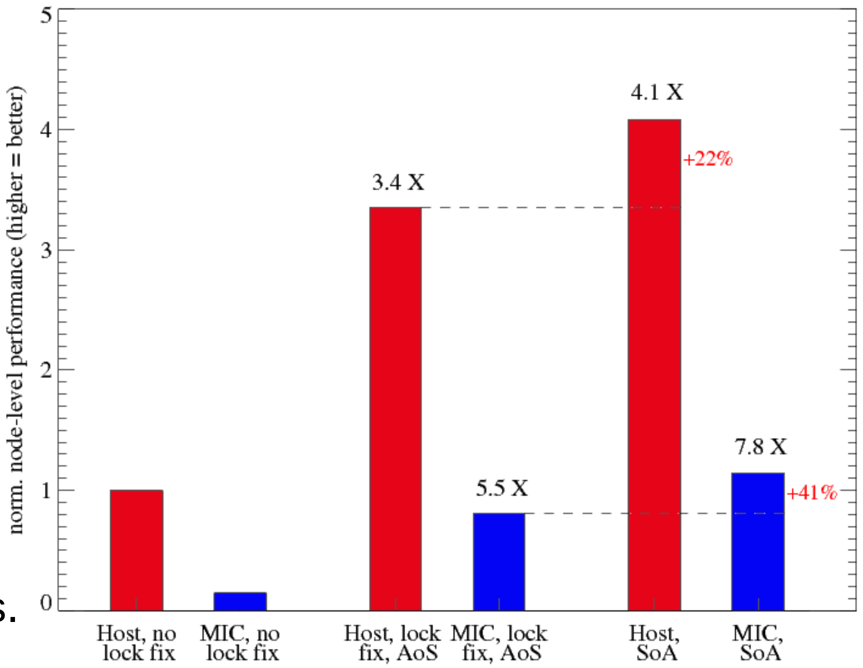
$\downarrow$

```
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];
```

```
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;
⋮
```

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 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?
}
```

Optimised pseudocode

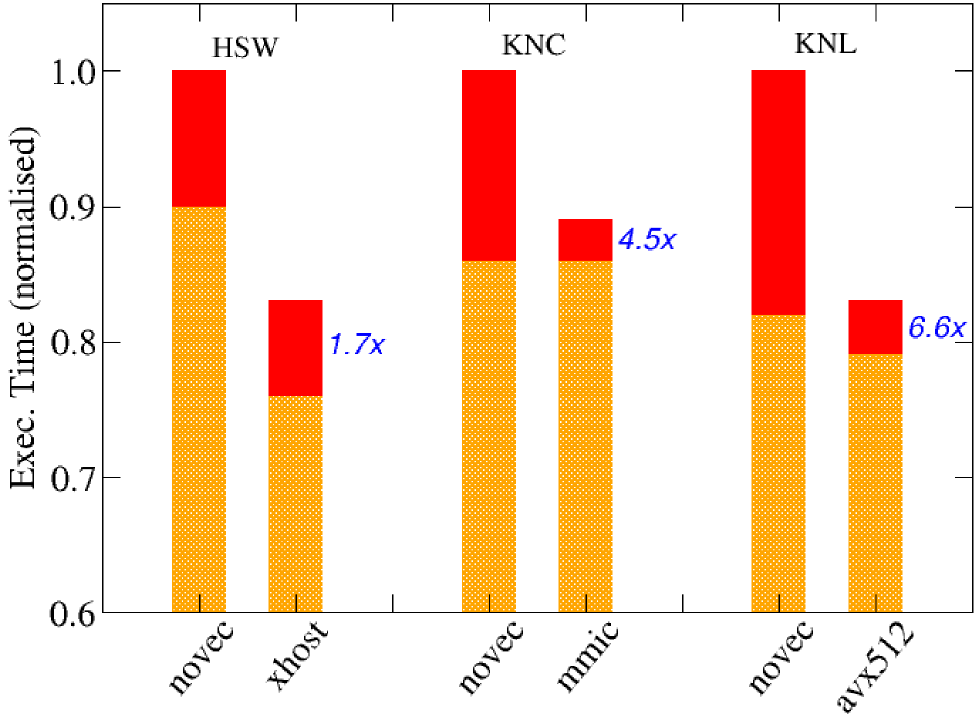
```
for (n = 0, n < neighbouring particles (selected)) {
        j = ngblist[n];          // getting the index from the particle data structure (SoA)

    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 + … ;
}
```

Compiler report on Intel Xeon Ivy-Bridge

```
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.



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

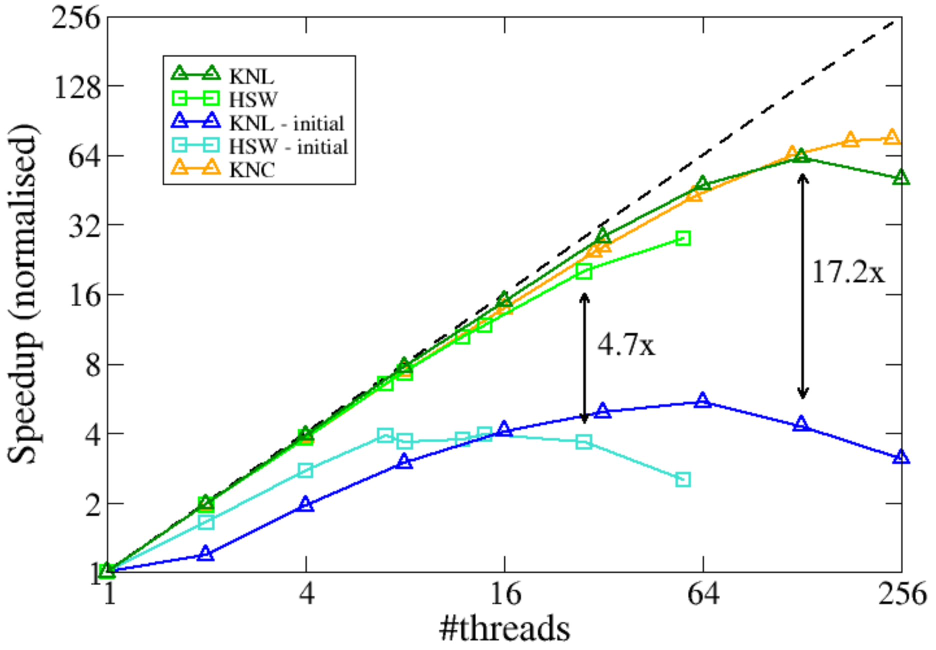
Node-level performance comparison between HSW, KNC *and* KNL

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:

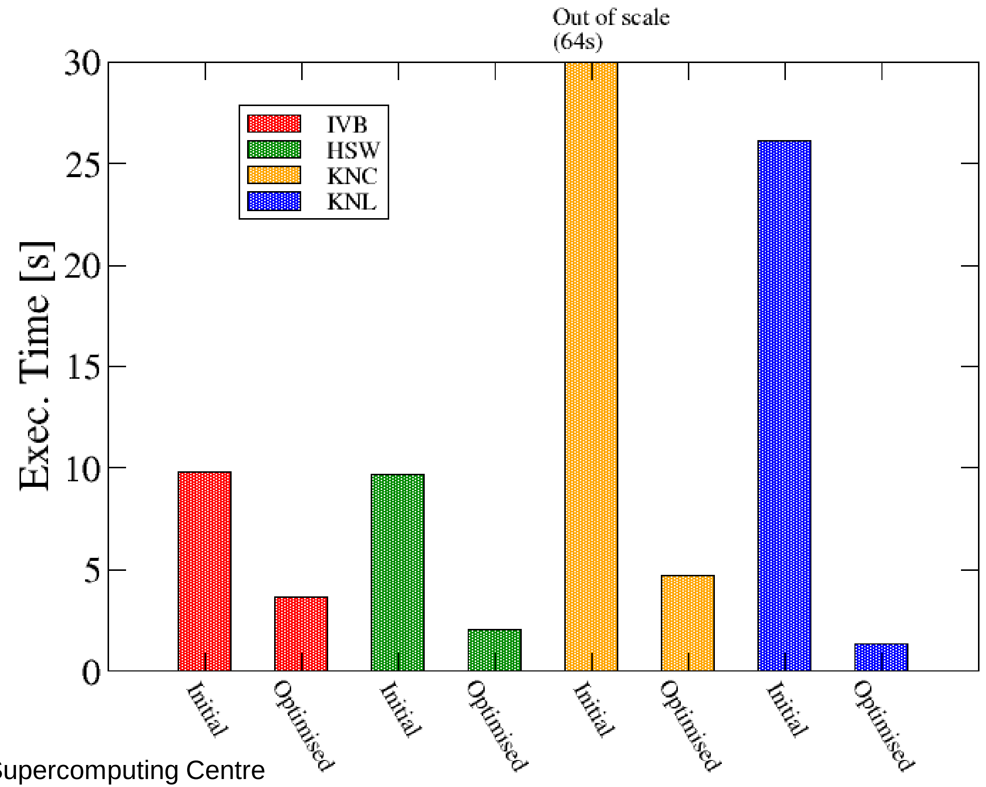
- Previous optimisations (data layout, vectorisation) improved the speedup on all systems, by different factors.
- KNL scalability slightly better than HSW and KNC up to 128 threads.



- 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.

$$\left.\begin{array}{l}\text{Threading parallelism}\\ \text{Data layout}\\ \text{Vectorisation}\end{array}\right\}\quad \text{Key points of our work, guided by analysis tools.}$$

- 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.