On-the-fly memory compression for multibody algorithms

Wolfgang ECKHARDT [a] , Robert GLAS [b] , Denys KORZH [a] , Stefan WALLNER [b] and Tobias WEINZIERL [c,1]

[a] *Department of Informatics, Technische Universität München, Germany*
[b] *Physics Department, Technische Universität München, Germany*
[c] *School of Engineering and Computing Sciences, Durham University, Great Britain*

Abstract. Memory and bandwidth demands challenge developers of particle-based codes that have to scale on new architectures, as the growth of concurrency outperforms improvements in memory access facilities, as the memory per core tends to stagnate, and as communication networks cannot increase bandwidth arbitrary. We propose to analyse each particle of such a code to find out whether a hierarchical data representation storing data with reduced precision caps the memory demands without exceeding given error bounds. For admissible candidates, we perform this compression and thus reduce the pressure on the memory subsystem, lower the total memory footprint and reduce the data to be exchanged via MPI. Notably, our analysis and transformation changes the data compression dynamically, i.e. the choice of data format follows the solution characteristics, and it does not require us to alter the core simulation code.

Keywords. n-body simulation, data compression, communication-reducing algorithms

Introduction

Widening memory gaps between compute units and the main memory, stagnating main memory per core as well as network bandwidth restrictions [1] lead into a dilemma in supercomputing: scientific interest and weak scaling laws require codes to increase the problem size, while strong scaling tells us that scaling is limited; but upscaling that keeps pace with the growth of concurrency misfits the aforementioned architectural trends. This problem can be studied at hands of multibody problems such as smoothed particle hydrodynamics (SPH) where upscaling translates into an increase of particle counts.

Facing memory and bandwidth constraints, it is convenient to switch from double to single precision. This allows to run twice as many computations for the same memory access characteristics, twice as many particles can be studied with the same memory footprint, and the bandwidth requirements for a given setup are halved. Where the rigorous switch from the C datatype `double` to `float` is not feasible for accuracy and stability constraints, some codes switch from one representation into the other in differ-

[1]Corresponding Author: Tobias Weinzierl, School of Engineering and Computing Sciences, Durham University, Lower Mountjoy South Road, DH1 3LE Durham, United Kingdom; E-mail: tobias.weinzierl@durham.ac.uk.

ent application phases. Notably in linear algebra algorithms such techniques have been applied successfully—though mainly due to speed reasons rather than concerns about the memory footprint [2]. Yet it remains a problem-specific, sometimes tricky and often even experiment-dependent decision whether to work with reduced accuracy. Furthermore, mixed precision algorithms require the modification of core compute functions (kernels). They are not minimally invasive in terms of coding, while the best-case savings are limited to a factor of two.

We study SPH's memory footprint challenge at hands of the Sedov blast Sod shock benchmark (see [3,4], e.g.) realised with a C++ merger of the AMR framework Peano and its particle administration [5,6,7] with the SPH kernels from SWIFT [8] using double precision. To lower the memory demands without kernel modifications, we propose to generalise concepts from [9]. Control volumes cluster the computational domain. Cells of linked-cell or modified linked lists algorithms [10] act as such volumes. Within each volume, we analyse all particles located inside as soon as all computational work such as force computations or particle position and velocity updates for these particles have terminated. We determine the average value and deviation of the particles' attributes such as position, speed or density from the average, and we switch into a hierarchical attribute representation, i.e. to hold the derivations from the means. Basically, one reference particle is chosen per cell and all values are stored relative to this particle's properties. These hierarchical values are stored with only few bytes where global accuracy constraints allow us to do so. Before their next usage, all particle data is back-transformed into plain C++ data types. We compress and uncompress the particles on-the-fly. The same technique is applied to the MPI data exchange.

To the best of our knowledge, our realisation of this simple idea goes significantly beyond other work: First, we are able to offer precision formats with down to two bytes per floating point value. This allows us to introduce savings beyond the magic factor of two experienced for `double` vs. `float`. Second, our approach is completely dynamic, i.e. it anticipates the solution behaviour. It compresses data only where compression is beneficial and preserves a prescribed accuracy. In SPH, it anticipates the smoothness of the solution. Third, our compressing is deployed to separate threads and runs parallel to the original code. Fourth, it does not require any alterations of the original compute kernels. It is minimally invasive. Finally, the idea also can be applied straightforwardly to MPI data exchange.

We study the proposed ideas on a cluster equipped with Xeon Phi accelerators. Memory constraints here play an important role. While we focus on a benchmark setup, our methodological contributions apply to other application areas and real-world setups as well. The remainder is organised as follows: We start from a description of our benchmark code (Section 1) before we introduce our idea of on-the-fly compression in Section 2. Some remarks how this compression embeds into the simulation life cycle precede numerical results in Section 4. A brief outlook and remarks on future work (Section 5) close the discussion.

1. Case study

As the present paper studies data layout considerations at hands of SPH, it studies a continuous medium represented by particles (Figure 1). Each particle carries a unique

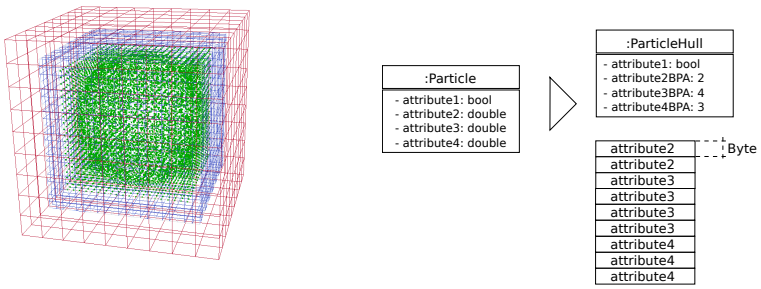


Figure 1. Left: A snapshot of a typical Sedov blast simulation. Right: `tearApart` decomposes a particle object into its hull and a byte stream.

smoothing length and interacts with any other particle closer than the smoothing length. We restrict to short-range interactions modelled with finite smoothing lengths. The extension of the present ideas to long-range interactions is technically straightforward.

Figure 2. Pseudo-code of overall case study algorithm.

```
1:  while time<terminal time do
2:      // 1st sweep: calculate density
3:      for all cells c in grid do
4:          for all cells c' that share at least one face with c do                ▷ Includes cell c
5:              Load c' if not loaded before throughout this sweep
6:              for all particles p in c do                                        ▷ Outer loop
7:                  for all particles p' in c' do                                  ▷ Inner loop
8:                      If p ≠ p', update attributes of p such as density dermining smoothing length
9:                  end for
10:                 Newton iteration: re-evaluate inner loop if smoothing length computation not converged
11:             end for
12:             Store away c' if not required anymore throughout traversal
13:             Adopt the AMR structure if smoothing lenghts permit/require
14:         end for
15:     end for
16:     // 2nd sweep: calculate forces                  ▷ Same loop structure as density calculation
17:     // 3rd sweep: half kick and drift, i.e. increase time by Δt/2          ▷ Same loop structure as density
18:                          ▷ calculation without particle-particle interaction; might change admissible Δt
19:     // 4th sweep: recalculate density               ▷ Same code as density calculation before
20:     // 5th sweep: calculate forces                  ▷ Same loop structure as density calculation
21:     // 6th sweep: half kick, i.e. increase time by Δt/2   ▷ Same loop structure as previous half kick
22: end while
```

Our simulation workflow comprises a sequence of nested loops (Figure 2). An outer loop steps through the simulation time. We rely on a global time stepping where each particle advances in time by the same delta. Local time stepping has no impact on the data flow. Yet, it changes the memory access characteristics. The time stepping itself is realised in leap frog form. It splits up the particles' position updates into two updates corresponding to half the time step size each. Each update comprises a force calculation, a position update (and an update of other quantities) as well as a recomputation of the particles' smoothing lengths. Such a scheme is in $\mathcal{O}(|\mathbb{P}|^2)$ for $|\mathbb{P}|$ particles. It would be inefficient to compare all particles with all particles because of the finite smoothing length. We thus rely on a grid and linked cell lists [11,12]:

We split up the computational domain into cells—in our case cubes due to a spacetree/octree formalism—track for each cell all adjacent cells, embed the particles into the cells, i.e. make each cell hold its particles, and check only particles from one cell vs. particles of its own or neighbouring cells. Adaptive mesh refinement (AMR) is directly introduced by nonuniform smoothing lengths. To keep the computational work as small as possible, our dynamic refinement criterion tries to introduce as small cells as possible. To allow us to realise the plain linked cell idea where only direct neighbour cells are checked, this minimum mesh size is constrained by the maximum of the smoothing lengths of all particles held within a cell. It may never underrun. Such an AMR-based linked cell strategy is popular in various other application areas such as molecular dynamics, too. Different to the latter codes, our smoothing length is a non-linear, time-dependent function of the particle properties.

The Sedov blast Sod shock setup acts as test bed for the introduced algorithmic ingredients (Figure 1). Cubic splines dominate the particle-particle interaction, and boundary treatment effects are neglected as we simulate only few time steps. For the realisation of the AMR, we rely on our meshing framework Peano [5,7]. All physics code fragments stem from SWIFT [8]. For the assignment of particles to the grid and vice versa, i.e. for gluing particles and grid together, we rely on the PIDT technique [6].

Several application characteristics guide our considerations: All reasonable and accurate simulations depend on the ability to handle as many particles as possible. If the total memory available is small, this memory has to be used carefully. All particle-particle interactions are computationally intense—one reason for the growing popularity of particle formalisms in supercomputing (see for example [13] for other application areas)—and thus natural vectorisation candidates given a proper data layout. All time steps move particles (twice) and the data structures thus have to be well-suited to reorder particle sets and to exchange particles between ranks and cores.

Such a melange of characteristics poses an interesting challenge for clusters with Xeon Phi accelerators. The arithmetic intensity and the localised operations make it promising with respect to the wide vector registers and high core counts. Its memory requirements, the Phi's strict alignment rules and the comparably small memory per core as well as the interconnect heterogeneity (core to core, accelerator to accelerator on same host, accelerator to accelerator on different hosts, ...) however render efficient coding challenging. Proper data structure choices play a major role. In this context, we furthermore note that SPH-type codes are complex. Changes of the data layout thus are problematic both economically and with respect to bugs. A minimally invasive approach to memory footprint tuning that keeps computational kernels unaltered is desirable. Our approach is minimally invasive and relies on a simple smoothness assumption similar to [9]: As the particles represent a continuum, spatially close particles often hold similar physical properties.

2. On-the-fly data compression

Each cell in the grid holds an set of particles. Since particles change their position only in two out of six sweeps, since only few particles travel from one cell into another cell per position update, and since we have to obtain high vectorisation efficiency, we hold them continuously rather than in linked lists or maps. For the majority of steps, the particle

sequences remain invariant. Each particle carries eleven scalar floating point quantities plus three vector quantities being the curl, the velocity and the position in space. Traditionally, two storage paradigms for such a setup do exist: array of structs (AoS) and struct of arrays (SoA). Hybrids are possible.

There are pros and cons coming along with each variant. SoA is advantageous for vectorisation. Its memory access characteristics are better than AoS if individual steps require subsets of the particles' attributes. The former property might loose importance due to gather and scatter instructions in future AVX versions as long as particle sets fit into the caches. AoS makes the particles' reassignment to cells and distributed memory parallelisation easier as particles are collocated in memory. Challenges however arise if particles are augmented with non-double attributes on hardware such as the Xeon Phi that require strict alignment. Compilers can reduce memory fill-ins (padding) due to attribute reordering—attributes are held in a struct with decreasing size—but some memory is 'lost'. As no scheme is always superior to the other, some codes change representations on-the-fly. For simplicity, our case study code is based upon AoS only, and we neglect tuning techniques transforming AoS into SoA temporarily to exploit vector units. AoS also integrates directly into our particle handling [6] triggering MPI calls.

We observe that the information density for a particular attribute within a cell is limited. Let $a(p)$ be a generic attribute of particle $a \in \mathbb{P}$. It is held in double precision.

$$A(c) = \frac{1}{|\mathbb{P}(c)|} \sum_{p \in \mathbb{P}(c)} a(p)$$

is the standard mean for all particle attributes within a cell. Then,

$$\hat{a}(p) = a(p) - A(c)$$

is a hierarchical attribute representation, i.e. the attribute value relative to the mean value. We use the term hierarchical as our idea is geometrically inspired [9]. It introduces a two-scale notion of attributes, as the nodal (read real) attribute content results from a coarse/generalised mean value plus a surplus. We assume that for many particles within one cell, the number of significant bits in $\hat{a}$ is small. Significant bits are those that are required to reconstruct the original value a up to sufficient/machine precision. Though there might be escapees, most particle attributes cluster around their respective mean value as the particles represent a continuum. If represented in hierarchical form, double precision of these attributes is luxury.

We hence introduce `tearApart` and `glueTogether`. `glueTogether` is the inverse of `tearApart` subject to precision considerations as detailed below. Both require a reference particle defining the mean values $A(p)$. `tearApart` furthermore is passed an error threshold ε. It removes all double precision arguments from the particle and returns a particle hull—an object with all the non-double attributes such as `bools` plus one number per double attribute with values from one to six—as well as a stream of bytes (Figure 1). The hull can be stored with techniques from [9] efficiently and does not require any alignment. Efficiently means that all the integer numbers are squeezed into one long integer. The byte stream is a linearisation of all the attributes. They first are converted into their hierarchical representation. Second, we rewrite them as $\hat{s} \cdot 2^{\hat{e}},\ \hat{s} \in \mathbb{N}$, i.e. we explicitly break up IEEE double precision. The exponent $\hat{e}$ third is stored as one byte on the stream. Finally, we store $\hat{s}$ as an integer value with a fixed number of bytes such that the whole attribute needs $bpa \geq 2$ bytes. bpa (bytes per attribute) is held within

the hull. Let $f_{bpa}(a(p))$ encode the storage of an attribute. `tearApart` chooses bpa minimal subject to $|f_{bpa}(a(p)) - a(p)| \leq \varepsilon$. For $bpa > 7$, no memory is saved. In this case, `tearApart` skips any particle transformation and returns the unaltered particle. For $bpa \leq 7$, `tearApart` reduces the memory requirements. A combination of the two operations and their byte stream idea with SoA is straightforward but not followed up here. If we apply `tearApart` on a sequence of particles, we note that we obtain a heterogeneous sequence of objects regarding their memory footprint. Depending on the particles' properties and the precision threshold, `tearApart` decides for each particle how many bytes are sufficient to encode the data or whether tearing them apart pays off at all. Obviously, the inverse `glueTogether` does not require a threshold.

3. Integration into simulation workflow and parallelisation

We do not use our decomposition into hierarchical, compressed attributes plus hull as one and only data representation. Instead, we apply `tearApart` after a cell's data has been used for the last time throughout a grid traversal. Its counterpart `glueTogether` acts as preamble to any computation after the first load of a cell. Particles are compressed in-between two grid traversals and are held with double values as long as they are required for computations. The two data conversations plug into all algorithm phases.

To facilitate this life-cycle, we rewrite $A(c)$ per cell into a tuple $(A,\tilde{A})(c)$. `tearApart` relies on $A(c)$ input to decompose the particles. In parallel, it determines the average value of attribute a in $\tilde{A}(c)$. `glueTogether` reconstructs all particle structs at hands of $A(c)$. Afterwards, it sets $A(c) \leftarrow \hat{A}(c)$. This tuple-based scheme allows for a single pass realisation. Each compression works with an average from the previous grid traversal. Though $A(c)$s' lagging behind by one traversal probably yields non-optimal compression factors, it does not harm the correctness.

As compression and reconstruction of the particles plug into the first usage or the last usage of a cell as a preamble or epilogue respectively, no alterations of the compute kernels are necessary. The approach is minimally invasive. As we stick to AoS throughout the computations, memory movements due to particle moves/reordering are minimised and the transfer of whole particles through MPI is straightforward. As we analyse the average attribute values on-the-fly on a per-cell basis, our approach is localised. It yields high compression rates where the particle attributes are homogeneous. Yet it is robust in regions where they differ significantly from each other. As we decide per particle whether compression pays off, iterations end up with data structures where some particles are compressed and others not. Since we hold the particles per cell in an array that obviously has to be able to grow due to `glueTogether`, `tearApart` and `glueTogether` implicitly sort the particles according to their derivation from the mean values: We make `tearApart` run through a cell's particle sequence reversely, and we make `glueTogether` append particles at the end of a sequence. A plain C++ vector suffices. The later the particle within a cell's particle sequence the higher the probability that it is compressed. This sorting that implicitly kicks in after the first iteration ensures that no frequent particle reordering due compression is necessary.

With the uncompress-compute-compress life-cycle, the effective memory demands of the code depend on the fact how long particles have to remain uncompressed; how long they are 'active', i.e. in-use. In-between grid sweeps, the memory footprint is smaller

than or equal to the original scheme besides the average value tuples per cell. As cells are by an order of magnitude fewer than particles, this impact can be neglected. For regular Cartesian grids, an estimate on the upper number of active cells is straightforward. Assuming homogeneous particle counts per cell then yields statements on the maximum memory footprint. Such bounds are, to the best of our knowledge, unknown for AMR in general. Empirically known however is the fact that a traversal of the adaptive grid by a space-filling curve (SFC) is advantageous. SFCs yield localised traversals due to their underlying Hölder continuity. Subdomains induced by a segment of an SFC have a small surface relative to their volume, i.e. their contained cells [14]. This property translates into the subdomain of active cells. The total memory footprint of uncompressed particles thus is relatively small. However, quantitative bound exist only for regular tessellations. We use the Peano SFC.

While the particle moving and, thus, the exchange of whole particles via MPI rely on plain structs, all other application phases in Algorithm 2 do not exchange all particle data such as particle positions. They exchange attribute subsets. Our code transfers these quantities in Jacobi-style, i.e. they are computed and sent out at the end of the traversal. Prior to the subsequent grid sweep we then merge them into the data on the receiver side. We therefore may either send out data prior to the compression epilogue or exchange compressed quantities. The latter reduces the bandwidth requirements.

Our grid decomposition is non-overlapping [7]: Cells are uniquely assigned to ranks while the vertices in-between are replicated along domain decomposition boundaries. We propose—in accordance with the PIDT scheme from [6]—to hold particles within the dual grid. Technically, the particle lists $\mathbb{P}(c)$ are split 2^d times, assigned to vertices, and all particles are stored within those lists whose vertices whose vertices are closest to their particle positions. As vertices along subdomain boundaries are replicated among all adjacent ranks, also the mean values are available on each rank; a payoff of the tuple storage. MPI data exchange thus can use the average tuples. No modifications become necessary. The compressed MPI exchange is minimally invasive.

`tearApart` is an operation that delays the program execution. We model it as task. Once all particles within a cell are not used anymore, their memory location remains invariant. We spawn a `tearApart` task compressing data while the original SPH algorithm continues. The compression runs parallel to computations. In return, we introduce a flag per particle list that is secured by a semaphore. It is set once `tearApart` finishes and checked by the load process triggering `glueTogether` prior to any uncompression.

`glueTogether` introduces overhead as well. As grid traversals are deterministic, this phase can be deployed to a prefetching task, too. While this is, in principle, straightforward, we do not follow-up it here. `tearApart`/`glueTogether` tasks are not visible to the original code and do not increase its complexity. However, estimates on the total memory footprint have to be validated carefully for the concurrent particle handling. Conversions might be delayed, and thus the total memory footprint might increase.

4. Results

All experiments were conducted on the Beacon system's Xeon Phi 5110P accelerators. Each accelerator hosts 8 GByte of memory, and four accelerators are plugged into one Xeon E5-2670 host. The Phis are programmed in native mode, and we do not use the

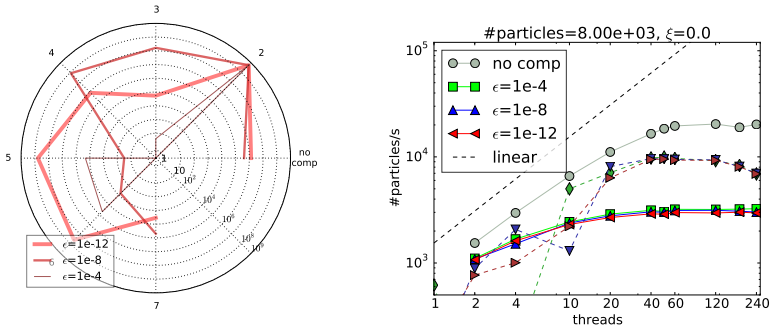
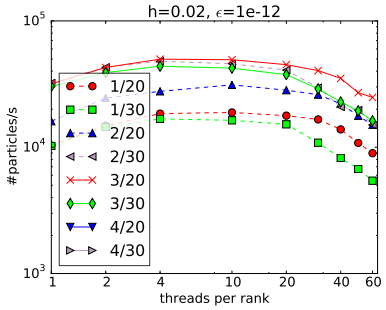


Figure 3. Left: Number of attribute compressions with different bpa for $\xi = 0.8, h = 0.008$. Right: TBB scaling on one Xeon Phi for different compression rates (solid lines) and scaling if `tearApart` is deployed to separate tasks (dotted lines).

host. All codes were translated with the Intel 2015 compiler, all results are given in particle updates per second, i.e. the timings are normalised by the total particle count. The shared memory parallelisation relies on Intel's Threading Building Blocks (TBB).

Our setup starts from particles aligned in a Cartesian grid with spacing h. Each particle experiences a slight random perturbation $\xi \cdot h$ of its position with $\xi \in (0,1)$, i.e. the setup is not perfectly symmetric. The AMR criterion is chosen initially such that the smoothing length corresponds to $1.1255h$. The particles' mass is set to h^3, and their internal energy equals $\frac{3}{2} \cdot 10^{-5}$ everywhere besides in a sphere of radius 0.1 around the centre of the cubic computational domain. Within the sphere, the particles' internal energy is increased by $\frac{10^3}{33} \cdot h^3$. This additional energy component triggers the blast (Figure 1). An increase in time is to some degree equivalent to increasing ξ.



N	ppn	rpp	tpr	no comp	-1e-8	-1e-12
1	1	30	1	49,067	64,848	46,814
1	1	30	2	73,741	83,746	67,565
1	1	30	10	82,425	79,295	78,741
1	1	40	1	48,326	59,436	45,467
1	1	40	2	63,909	66,385	59,204
1	1	40	10	65,597	61,513	62,313
1	2	30	1	57,434	67,161	54,411
1	2	30	2	75,873	92,684	72,077
1	2	30	10	86,267	92,287	82,055
1	3	30	1	59,817	73,298	57,112
1	3	30	2	80,106	91,154	55,218
1	3	30	10	87,389	76,706	76,852
2	1	30	1	59,064	71,281	55,811
2	1	30	2	77,964	87,222	77,982
2	1	30	10	91,752	98,843	88,700
4	1	30	1	61,875	70,723	59,901
4	1	30	2	82,149	84,785	60,920
4	1	30	10	93,154	87,544	71,332

Figure 4. Left: Throughput for different combinations of accelerator count/MPI ranks per Xeon Phi ($\xi = 0$). The compression is not applied on MPI messages. Right: Throughput of hybrid code where compression is always applied to the MPI message sizes. If compression inside the domain is switched off, the domain boundary applies $\varepsilon = 10^{-12}$. N=number of nodes, ppn=phis used per node, rpp=ranks per phi, tpr=threads per rank. Results are strong scaling measurements with $h = 0.01$.

While small ε increase the number of particles that are not or almost not compressed ($bpa = 6$), $bpa = 2$ is sufficient for the majority of particles for all ε (Figure 3). The original memory footprint stems from particles with 176 bytes each plus the memory required for the AMR grid. For both, bit optimisations from [9] have been applied. The hull of the particle including the `bpa` flags in contrast is 40 bytes. As such, we compress

the memory footprint to 0.25 ($\varepsilon = 10^{-12}$), 0.21 ($\varepsilon = 10^{-8}$) or 0.17 ($\varepsilon = 10^{-4}$). These measurements comprise overheads required for dynamic data structures.

We observe reasonable scaling of the code (Figure 3), but the code can not exploit more than one hardware thread per floating point unit. While the compression allows us to upscale the problem per node, it reduces the code's throughput. However, we note that the compression can be ran in the background of the actual solve. With the compression in the background, we flood the Phi with tasks [15] and thus start to make up for the compression's runtime penalty. It remains open whether task-based `glueTogether` could close the gap completely.

If we use more than one Xeon Phi with compression only applied within the domain, we obtain the best throughput for three Xeon Phis running 20 MPI ranks with 4 TBB threads each: with a hybrid code, slight overbooking of the floating point units pays off. If we use more than one node, the performance deteriorates. If we study the best-case throughputs and apply the compression on data exchanged via MPI (Figure 4), we observe that more signficiant overbooking (30 ranks with 10 threads, e.g.) starts to pay off. We also are faster than TBB-only codes on one accelerator. Up to three Xeon Phis per node yield a performance improvement—though far from linear—while the compression with background tasks allows the hybrid code now to offer the memory compression for free in terms of runtime. Multithreading has closed the compression's runtime gap. We furthermore observe that four accelerators distributed among four nodes yield higher throughput than four accelerators plugged into one node. Reasons for this have to be some kind of resource competition. All in all, the compression of the MPI messages speeds up the code by a factor of five compared to an uncompressed data exchange, while the basic performance characteristics remain preserved. See [6] for a discussion of the code's scaling behaviour—the present figures study soley strong scaling.

5. Outlook and conclusion

The present work introduces techniques that help us to squeeze more simulation into given memory and communication bandwidth allowance. This will become mandatory for the exascale era [1]. Picking up the seminal fourth recipe of [16], our approach might fall into a class of techniques that help us to deliver *more science per byte*. An important advantage of the present work compared to classic mixed precision is that the original compute kernels remain unaltered.

While our algorithmic setting is flexible, we find that basically either aggressive compression with two bytes per floating point numbers or (almost) no compression at all are used. Reasons for this might be a result of the chosen use case, but the effect deserves further studies. If only few *bpa* choices are sufficient, the particle hull footprint can be reduced further. In the context of data compression, we reiterate that our technique is well-suited to equip codes with higher than double precision without making the memory footprint explode. Also, we suggest that a reduction of memory footprint and, thus, data moves reduces the energy consumption of codes. This deserves further investigation.

Acknowledgements

We appreciate the support from Intel through Durham's Intel Parallel Computing Centre (IPCC) which gave us access to latest Intel software. Special thanks are due to Matthieu Schaller for his support and advise on the SWIFT code [8]. All underlying software is open source and available at [5]. This material is based upon experimental work supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee. The project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE).

References

[1] J. Dongarra, P. H. Beckman, et al. The International Exascale Software Project Roadmap. *IJHPCA*, 25(1):3–60, 2011.

[2] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180:2526–2533, 2009.

[3] P. Gonnet. Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures. *SISC*, 37(1):C95–C121, 2015.

[4] V. Springel. E pur si muove: Galiliean-invariant cosmological hydrodynamical simulations on a moving mesh. arxiv e-prints, 0901.4107, mnras. *Mon. Not. of the R. Astron. Soc.*, 2009.

[5] T. Weinzierl et al. Peano—a Framework for PDE Solvers on Spacetree Grids, 2015. www.peano-framework.org.

[6] T. Weinzierl, B. Verleye, P. Henri, and D. Roose. Two particle-in-grid realisations on spacetrees. *Parallel Computing*, 2015. (submitted, arXiv, 1508.02435).

[7] T. Weinzierl. The Peano software—parallel, automaton-based, dynamically adaptive grid traversals. Technical Report arXiv150604496W, eprint arXiv:1506.04496, Durham University, 2015.

[8] P. Gonnet, M. Schaller, et al. Swift—shared-memory parallel smoothed particle hydrodynamics (sph) code for large-scale cosmological simulations, 2015. http://www.swiftsim.com.

[9] H.-J. Bungartz, W. Eckhardt, T. Weinzierl, and C. Zenger. A precompiler to reduce the memory footprint of multiscale pde solvers in c++. *Future Generation Computer Systems*, 26(1):175–182, January 2010.

[10] W. Mattson and B. M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2-3):135–148, 1999.

[11] B. Quentrec and C. Brot. New method for searching for neighbors in molecular dynamics computations. *Journal of Computational Physics*, 13(3):430 – 432, 1973.

[12] R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. Academic Press, 1988.

[13] R. Yokota, G. Turkiyyah, and D. Keyes. Communication complexity of the fast multipole method and its algebraic variants. *Supercomputing frontiers and innovations*, 1(1), 2014.

[14] H.-J. Bungartz, M. Mehl, and T. Weinzierl. *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *LNCS*, chapter A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves, pages 1064–1074. Springer-Verlag, Berlin, Heidelberg, 2006.

[15] M. Schreiber, T. Weinzierl, and H.-J. Bungartz. Cluster optimization and parallelization of simulations with dynamically adaptive grids. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 484–496, Berlin Heidelberg, 2013. Springer-Verlag. preprint.

[16] D. E. Keyes. Four Horizons for Enhancing the Performance of Parallel Simulations Based on Partial Differential Equations. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2000.