

An Efficient SIMD Implementation of Pseudo-Verlet Lists for Neighbour Interactions in Particle-based Codes

Extended Abstract

James S. Willis
Institute for Computational
Cosmology (ICC)
Durham, UK
james.s.willis@durham.ac.uk

Matthieu Schaller
Leiden Observatory
Leiden, Netherlands
schaller@strw.leidenuniv.nl

Pedro Gonnet
School of Engineering and
Computing Sciences
Durham, UK
Google Switzerland GmbH
Zürich, Switzerland
gonnet@google.com

ACM Reference format:

James S. Willis, Matthieu Schaller, and Pedro Gonnet. 2018. An Efficient SIMD Implementation of Pseudo-Verlet Lists for Neighbour Interactions in Particle-based Codes. In *Proceedings of Super Computing Conference, Dallas, Texas USA, November 2018 (Super Computing 2018)*, 3 pages. https://doi.org/10.475/123_4

ABSTRACT

Particle-based simulations are widely used in many research fields, e.g. chemistry, physics, biology [2, 5, 7]. The under-lying algorithm evolves a system of particles via a set of pair-wise interactions. Short-range interactions are evaluated by computing pair-wise distances between particles and checking that they lie within a cut-off radius, h , of each other. A standard approach for efficient neighbour finding is to use a mesh of cells with edge lengths that match the particle cut-off radius. Particles are then interacted within pairs of neighbouring cells.

A pseudo-Verlet list [4] aims to reduce the number of spurious pair-wise distance calculations between neighbouring cells. The algorithm, due to its branching and inherently inefficient memory access patterns, does not lend itself to automatic SIMD-vectorisation, which is crucial for obtaining the best possible performance on modern CPU architectures. We therefore, on this poster, present a SIMD implementation of the pseudo-Verlet list that addresses these issues. We have applied our strategy to the SPH-based cosmological code SWIFT [6]. We report a speed-up of 2.21x, 2.41x and 3.65x over the scalar algorithm using AVX, AVX2 and AVX-512 instruction sets respectively.

It can be shown that for a uniform distribution of particles in two neighbouring cells (face-on), only $\leq 16\%$ of particle pairs will meet the criteria $r^2 \leq h^2$, where r is the separation between particles [3]. This leads to $\geq 84\%$ spurious pair-wise distance calculations between pairs of cells. This fraction can be reduced using the pseudo-Verlet list algorithm.

The pseudo-Verlet list algorithm first projects the particles onto the axis joining the centres of the two cells and then sorts them along the axis. By walking along the axis, the number of interactions that need to be tested is drastically reduced. The separation between particles in the left-hand cell (a), $p_i^{(a)}$, and particles in the right hand cell (b), $p_j^{(b)}$, are only computed for particles that lie within h_i , the cut-off radius of $p_i^{(a)}$, on the sorted axis. If any particles $p_j^{(b)}$ satisfy the criterion $\|p_i^{(a)} - p_j^{(b)}\|_2 < h_i$, where $\|\cdot\|_2$ denotes the Euclidean distance between two particles, an interaction is computed. The gain is maximal for cells only in contact by a vertex. This whole procedure (in 2D) is presented in Figure 1.

Most compilers (Intel, GNU and Clang) fail to auto-vectorise the algorithm efficiently due to multiple exit conditions and conditional statements. We therefore decided to perform explicit vectorisation of the code using vector intrinsics [1], which allows us to achieve the same level of performance across different compilers. We will outline a series of modifications that improve both the serial and vectorised versions of the code.

The overall vectorisation strategy involves picking one particle, $p_i^{(a)}$, from cell (a) and interacting it with a vector of particles, $p_j^{(b)}$, from cell (b) that are in range. The contributions from $p_j^{(b)}$ that are not within range are masked out.

To improve the memory access patterns we create a *particle cache* that contains the subset of particle properties needed for an interaction. The particles are cached in order along the cell-pair axis, using the sorted list of particle indices from the Verlet list. It is created using a structure-of-arrays data layout.

Once a neighbour has been found for a particle, its properties (e.g. density, density gradient...) are updated with contributions from the neighbour. Each update requires a memory write. This can be very time-consuming if a particle has a lot of neighbours and various properties to update. The number of writes can be reduced substantially if a running sum can be held for each updated property. In practice, this is implemented using a vector to accumulate the update from each neighbour. When a particle has found all of its neighbours a horizontal add is performed on the *accumulator* vector, the result of which is added to the particle. This results in only one write for each particle property modified, as the vector can be held in a register.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

Super Computing 2018, November 2018, Dallas, Texas USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 123-4567-24-567/08/06...\$15.00

https://doi.org/10.475/123_4

Computing the pairwise distances between particles is the most time-consuming part of the neighbour search algorithm. In order to reduce the number of distance calculations, we find the *leftmost* particle in cell (a) which is in range of any of the particles in cell (b), e.g.

```

1  int first_a = count_a;
2  while (first_a > 0 &&
3      dist_a[index_a[first_a-1]] + h_max_a > dist_b[0])
4      first_a--;

```

We then introduce the array `max_index_a` which contains, for each particle in cell (a), the index of the farthest possible interacting particle in cell (b). This array is constructed as follows:

```

1  int temp = 0;
2  while (dist_a[first_a] +
3      parts_a[index_a[first_a]].h > dist_b[temp]) temp++;
4  max_index_a[first_a] = temp;
5  for (i = first_a + 1; i < count_a; i++)
6      int temp = max_index_a[i - 1];
7      while (dist_a[i] +
8          parts_a[index_a[i]].h > dist_b[temp]) temp++;
9  max_index_a[i] = temp;

```

For each i th particle in cell (a) from `first_a` to `count_a`, we only need to inspect the particles in cell (b) from 0 to `max_index_a[i]` (see Figure 2). By consequence, only the particles from cell (b) from 0 to `max_index_a[count_a - 1]` need to be loaded into the particle cache. Note that we compute `max_index_a` as an approximate value using only a single pass over the particles; we could compute an exact value but it would require $O(N^2)$ operations.

Since we are computing interactions symmetrically, we compute the same indices for particles in cell (b) interacting with particles in cell (a), i.e. `last_b` and `min_index_b` respectively, and cache the particles accordingly. We therefore have to be careful to cache the union of the set of particles from both cells required for interactions in both directions.

REFERENCES

- [1] [n. d.]. . <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>.
- [2] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1-2 (2015), 19 – 25. <https://doi.org/10.1016/j.softx.2015.06.001>
- [3] Pedro Gonnet. 2007. A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations. *Journal of Computational Chemistry* 28, 2 (2007), 570–573. <https://doi.org/10.1002/jcc.20563>
- [4] Pedro Gonnet. 2013. Pseudo-Verlet lists: a new, compact neighbour list representation. *Molecular Simulation* 39, 9 (2013), 721–727. <https://doi.org/10.1080/08927022.2012.762097> arXiv: <http://dx.doi.org/10.1080/08927022.2012.762097>
- [5] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1 (1995), 1 – 19. <https://doi.org/10.1006/jcph.1995.1039>
- [6] Matthieu Schaller, Pedro Gonnet, Aidan B. G. Chalk, and Peter W. Draper. 2016. SWIFT: Using Task-Based Parallelism, Fully Asynchronous Communication, and Graph Partition-Based Domain Decomposition for Strong Scaling on More Than 100,000 Cores. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '16)*. ACM, New York, NY, USA, Article 2, 10 pages. <https://doi.org/10.1145/2929908.2929916>
- [7] Ilian T. Todorov, William Smith, Kostya Trachenko, and Martin T. Dove. 2006. DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism. *J. Mater. Chem.* 16 (2006), 1911–1918. Issue 20. <https://doi.org/10.1039/B517931A>

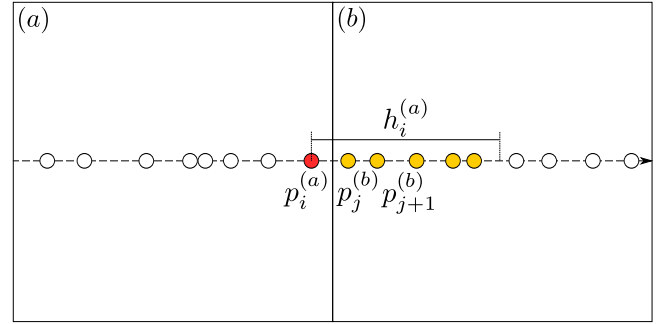
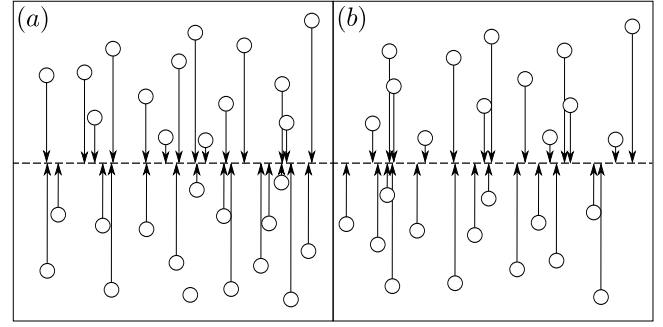


Figure 1: Performing the cell-pair interactions using a sorted pseudo-Verlet list. Top: Particles are projected onto the cell-pair axis and sorted according to their distance along the axis. The particles $p_i^{(a)}$ on the left (cell (a)) receive contributions from the particles $p_j^{(b)}$ on the right (cell (b)). **Bottom:** Each particle is traversed and a pair-wise distance calculation is only performed for particles in (b) that are within h_i on the axis as this provides an upper-bound on the 3D distance. Interactions are later computed for the particles $p_i^{(a)}$ that obey $\|p_i^{(a)} - p_j^{(b)}\|_2 < h_i^{(a)}$.

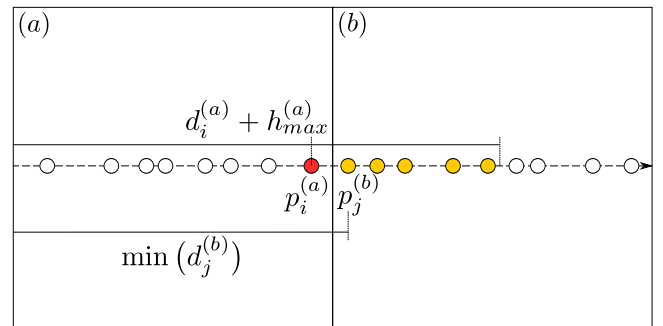


Figure 2: For a given particle $p_i^{(a)}$ (red circles), only the particles within a distance $d_i^{(a)} + h_{max}^{(a)}$ (yellow circles) are candidates for interactions in cell (b). These particles are located between index 0 and `max_index_a[i]` in the sorted array of particles of cell (b).

A ARTIFACT DESCRIPTION APPENDIX: AN EFFICIENT SIMD IMPLEMENTATION OF PSEUDO-VERLET LISTS FOR NEIGHBOUR INTERACTIONS IN PARTICLE-BASED CODES

A.1 Abstract

We provide a link to the source code that implements the optimisations described in the poster and outline how the code can be configured and compiled. We then go on to describe how the benchmark can be run in different configurations.

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** Pseudo-Verlet List
- **Program:** SWIFT
- **Compilation:** ICC, GCC
- **Data set:** SWIFT provided benchmark
- **Hardware:** x86 CPUs
- **Output:** Timing results reported by benchmark
- **Publicly available?:** Yes

A.2.2 HOW SOFTWARE CAN BE OBTAINED: SWIFT is an open source project and can be downloaded from GitLab at <https://gitlab.cosma.dur.ac.uk/swift/swiftsim>. The Git revision used to produce the benchmark results was ab123564.

A.2.3 Hardware dependencies: The code should run on many x86 platforms. It has been tested on the following Intel Xeon processors: Sandybridge, Broadwell, Skylake and Intel Xeon Phi Knights Landing processor.

A.2.4 Software dependencies: To configure and build the code the following tools are needed: libtool, HDF5, ICC/GCC compiler. A more detailed list can be found at <http://icc.dur.ac.uk/swift/onboarding.pdf>.

A.2.5 Datasets: We used a standardized benchmark of the core compute kernels of SWIFT. The benchmark, named *test27cells*, computes the interactions between one cell containing 216 randomly placed particles and its 26 neighbours which are similarly populated. This allows us to probe the speed-up from vectorisation in all the relevant geometrical configurations, and is representative of the workload in actual production runs using SWIFT. *test27cells* is found in the *tests* directory relative to the root directory of SWIFT.

A.3 Installation

Once the code is downloaded from the GitLab repository, it can be configured and built for the host architecture using:

```
$ ./autogen.sh
$ ./configure
$ make -j 6
```

The configure script should auto-detect which platform the code needs to be compiled for. The benchmark is then built with:

```
$ cd tests
$ make test27cells
```

This will compile the code with vectorisation enabled by default, to obtain timing results for a serial run the code can be configured via:

```
$ ./autogen.sh
$ ./configure --disable-vec
$ make -j 6
$ cd tests
$ make test27cells
```

A.4 Experiment workflow

The benchmark can be run using these commands:

```
$ cd tests
$ ./test27cells -n 6 -r 10000 -h 1.2348 -d 0.1 -v 1
```

Each command line argument, *-n -r -h -d -v*, changes the number of particles in each dimension of the box; the number of runs; sets the smoothing length of the particles; how much to perturb the particle positions and which velocity field to set.

A.5 Evaluation and expected result

After running the benchmark the timing results should be output to the terminal. The line containing "SWIFT calculation took" displays the total time taken, measured in clock ticks, averaged over the number of runs, which is specified by *-r*. The time can be converted to seconds by dividing it by the clock frequency of the processor that the benchmark was ran on. The benchmark can be run with vectorisation disabled to obtain a relative speed up due to the optimisations described in the poster.