

Parallelization of Molecular-Dynamics Simulations Using Tasks

Ralf Meyer^{1,2} and Chris M. Mangiardi¹

¹Department of Mathematics and Computer Science, Laurentian University,
Sudbury, ON P3E 2C6, Canada

²Department of Physics, Laurentian University, Sudbury, ON P3E 2C6, Canada

ABSTRACT

This article discusses novel algorithms for molecular-dynamics (MD) simulations with short-ranged forces on modern multi- and many-core processors like the Intel Xeon Phi. A task-based approach to the parallelization of MD on shared-memory computers and a tiling scheme to facilitate the SIMD vectorization of the force calculations is described. The algorithms have been tested with three different potentials and the resulting speed-ups on Intel Xeon Phi coprocessors are shown.

INTRODUCTION

Molecular dynamics (MD) is one of the most important computational methods in materials science. Although the size of problems that can be studied with MD has increased tremendously thanks to the progress in computing technology many problems remain that can only be tackled with MD if more computational power becomes available. It is therefore important that the computer programs for MD simulations are able to make full use of modern CPUs. Further problems are created since the systems that are studied with MD have not only become larger but also more complex and less homogeneous. This increased complexity leads to load balancing issues that can significantly reduce the computational efficiency.

In recent years, the power of computers has been improved mainly by two means: The number of independent compute cores integrated in modern multi-core processors has been increased and single instruction multiple data (SIMD) units with increasingly wide vector registers have been implemented. The latest step in this direction is the Intel Xeon Phi coprocessor, a many-core processor that integrates 60 compute cores with four hardware threads per core and 512 bit wide SIMD units.

This article describes our work on the development of new algorithms for MD simulations with short-range forces. The algorithms were specifically designed to be efficient on multi- and many-core processors with wide SIMD vector units like the Xeon Phi. In addition to this, the algorithms avoid certain types of load-balancing problems arising in simulations of inhomogeneous systems.

ALGORITHMS

Parallel algorithms for molecular dynamics

The exploitation of multiple processing cores and SIMD units both require parallel programming albeit in different forms. SIMD units achieve data level parallelism, whereas multi-core processors enable task parallelism. MD is inherently a parallel problem since the calculation

of the forces, which usually is the most time consuming operation, can be performed independently for each particle. In practice, however, things are not so simple.

Many simulations in materials science use short-ranged force models where the force between two particles is zero if the distance of the particles exceeds a cut-off radius r_{cut} . While short-ranged forces significantly reduce the computational weight of the force calculation, they make MD an irregular problem since the number of interaction partners, often called neighbors, is not necessarily the same for all particles. Moreover the neighbors of a particle cannot be stored consecutively causing problems for the vectorization of the calculation.

The standard approach for the parallelization of MD simulations with short-ranged forces is domain decomposition. In this method, the volume of the simulated system is decomposed into smaller domains of identical or similar shape. Each domain is assigned to one processing core that is responsible for the calculations of the forces of the particles inside its domain. Domain decomposition works on systems with distributed memory and it scales well with large numbers of processors if the system is sufficiently homogeneous. In inhomogeneous — e.g. porous — systems, however, load-balancing issues arise. In addition to this, domain decomposition is very sensitive to delays of individual processors.

Task-based parallelization: The cell method

Our parallel implementation of MD uses a task-based approach, named the cell method, to distribute the calculation of the forces (and neighbor-lists) over the available compute cores. While the domain decomposition approach divides the calculation into as many areas as there are cores, the cell method typically divides the work into a larger number of small tasks. The tasks are then scheduled dynamically for execution by the available cores. The dynamic scheduling results enables close to ideal load balancing and is robust against performance differences of the cores. Details of this method are described in [1].

Most MD programs for short-range forces use the so-called linked-cell method [2] for the construction of a list of neighbors for each particle. This technique divides the simulation cell into a grid of small cells and bins the particles into these cells. The cell method reuses the binning of the particles into cells to define its tasks. A task is the calculation of the force on all particles in a given cell.

Implementations of MD on computers with shared memory face a problem: In order to save time, MD programs typically calculate the force \mathbf{F}_{ij} between two particles i and j only once and update both particles at the same time (since according to Newton's third law $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$). On shared-memory computers this results in race conditions when two processing cores update the same particle simultaneously. Simple synchronization techniques that avoid the race conditions entail significant performance penalties.

The cell method avoids the problem of race conditions with the help of conditional task scheduling. Since the cell binning is originally used to construct the neighbor lists, it is known that the interaction partners of the particles in a given cell are located in the same cell or the adjacent cells. The cell method uses this information to schedule the tasks in such a way that two tasks that might update the same particle will never run at the same time. A similar idea is described in Ref. [3]. While the implementation in the latter article uses a locking/unlocking mechanism to restrict access to particles in the same cell, the cell method uses a conditional task schedule to achieve the same effect without explicit locking.

After the force calculation, the construction of neighbor lists is the second most time consuming operation in MD simulations with short-ranged forces. The cell method can be used for the construction of the neighbor lists as well. A task is then defined as the construction of the neighbor lists of all particles in a cell.

SIMD vectorization

Optimal performance of SIMD vector units requires that data items are stored consecutively in memory, which is not the case in MD simulations with short ranged forces. The Xeon Phi architecture has scatter/gather instructions that allow the loading and storing of vectors from non-consecutive addresses, but these instructions are much slower than consecutive stores and loads to aligned addresses. Another problem are conditional computations that occur frequently in force calculations. Conditional computations can be vectorized with the help of masked operations that use a bit mask to limit the effect of an instruction to a subset of the vector elements. While masking enables the vectorization of conditional computations, it lowers the performance compared to non-conditional operations since computations are done for the full vector even if only a part of the vector requires it.

Our MD implementation uses a tiling algorithm to take advantage of SIMD vector units. The calculation of the forces loops over all pairs of particles belonging to a task. This loop initially only calculates the components of the distance vector of the particle pairs and stores them into tiling buffers. When the buffers are full or there are no more pairs left, the forces are calculated. Since the data in the buffers is stored consecutively, this part of the force calculation can be vectorized. Once all pairs in the buffers have been calculated, the resulting forces are written back to the particles. For best performance, the tiling buffers should be small enough to fit into the L1 cache. On the Xeon Phi, we typically use a buffer size of 128.

Our tiling algorithm has two main advantages. First, once the distance components are stored in the buffers, the computation of the forces can be broken into several consecutive loops. Simpler loops facilitate the compiler's analysis of data dependencies and they reduce register pressure. Also, if one loop cannot be vectorized efficiently, this does not prohibit the vectorization of the other loops. The second advantage of the tiling scheme has nothing to do with SIMD vectorization. Many MD programs provide options to calculate some additional properties like local stresses. If this evaluation happens inside the force calculations, just to test whether the calculation is required can have a notable effect on the performance. The tiling scheme makes it possible to move this test out of the innermost loop and perform the test only once per buffer evaluation.

RESULTS

In order to test the performance of our algorithms, we performed benchmark simulations of four different systems using three different potentials. The first system uses Cleri and Rosato's tight-binding second moment potential [4] to simulate a cubic block of bulk copper containing 4,630,500 atoms. The second benchmark employs Mendeleev's potential 2 [5] in a simulation of bulk iron using 4,000,752 atoms. Our third test case simulates a liquid system of 4,000,752 atoms with the help of the Lennard-Jones (LJ) 12-6 potential [6,7] with a hard cut-off [8] at $r_{\text{cut}} = 2.33 \sigma$. The fourth benchmark system again uses the copper potential from Ref. [3] to simulate a porous system of sintered copper nanoparticles containing 1,999,220 atoms.

The main difference between the first three benchmarks is the computational complexity of the potential. The simplest potential of the three is the LJ potential employed in the third benchmark. It requires no special functions and includes no conditional computations except for the cut-off at distance r_{cut} . The copper potential used in the first and fourth benchmark is computationally more intensive than the LJ potential. It requires the evaluation of three exponential functions for each particle pair but still no conditional computations beyond the hard cut-off distance. The iron potential from Ref. [5] is far more complex than the other two potentials and it contains multiple conditional computations. The purpose of the fourth benchmark is to study the effect of the structure of the configuration on the performance of the simulations. The porous nature of this configuration has two consequences: First, it makes this system a bad candidate for domain decomposition as shown in Ref. [1]. Second, this configuration contains a large number of surface atoms. This reduces the average number of neighbors per particle, which in turn increases memory traffic and reduces cache efficiency.

In order to compare the performance of the four test systems, we measured the time necessary to perform 1000 simulation steps while rebuilding the neighbor lists at every 10^{th} step. Since the first construction of neighbor lists is considerable slower due to memory allocation issues, we actually ran the simulations over 1010 steps and started the measurement of the time at the beginning of the tenth step. Measurement of the time ended immediately after the 1010^{th} step so that the time to write results into files and other shutdown activities was not included. We ran the simulations on Intel Xeon Phi coprocessors (model 5110P) using 1 to 240 cores. We also ran the simulations on a typical host system containing two 2.3 GHz 6-core CPUs (Intel Xeon E5-2630) with AVX instruction set and hyper-threading enabled. All simulations were repeated five times and the average time calculated.

In order to study the effect of SIMD vectorization we did not use a different implementation of the potentials. Instead we used compiler options to turn vector operations on and off.

Figure 1 shows the speedups that we obtained for the four benchmark systems on Intel Xeon Phi coprocessors. All speedups were calculated relative to the average execution time with one compute core and without vectorization. The first observation is that in all cases, the algorithms scale very well to large numbers of cores. Up to 60 cores (which is the number of physical cores) all systems show excellent speedups close to the ideal behavior. Above sixty cores, when more than one thread runs on at least some physical cores, the efficiency is reduced which is expected since multiple threads compete for the resources of one core. Note that the design of the Xeon Phi usually requires two to four threads per core for optimal performance [9].

From the left panel of the figure it can be seen that without SIMD vectorization the bulk Fe and Cu systems achieve maximum speedups above 120, which is excellent for a 60 compute-core device. The other two benchmark systems achieve somewhat lower maximum speedups around 90. With vectorization, the picture changes: The two Cu systems then clearly have the largest speedups, followed by the Fe potential.

Table I summarizes and complements the data for the four test systems. In addition to the maximum speedup S_{max} obtained on a Xeon Phi, the table also shows the ratio of the maximum speedups with and without vectorization $S_{\text{vec}}/S_{\text{novec}}$, the speedup S_{serial} due to vectorization when using a single compute core and the speedup compared to the best time obtained on the host system S_{host} .

The comparatively low speedup in case of the LJ potential with and without vectorization is probably a consequence of the low computational weight of this potential. With less time spent

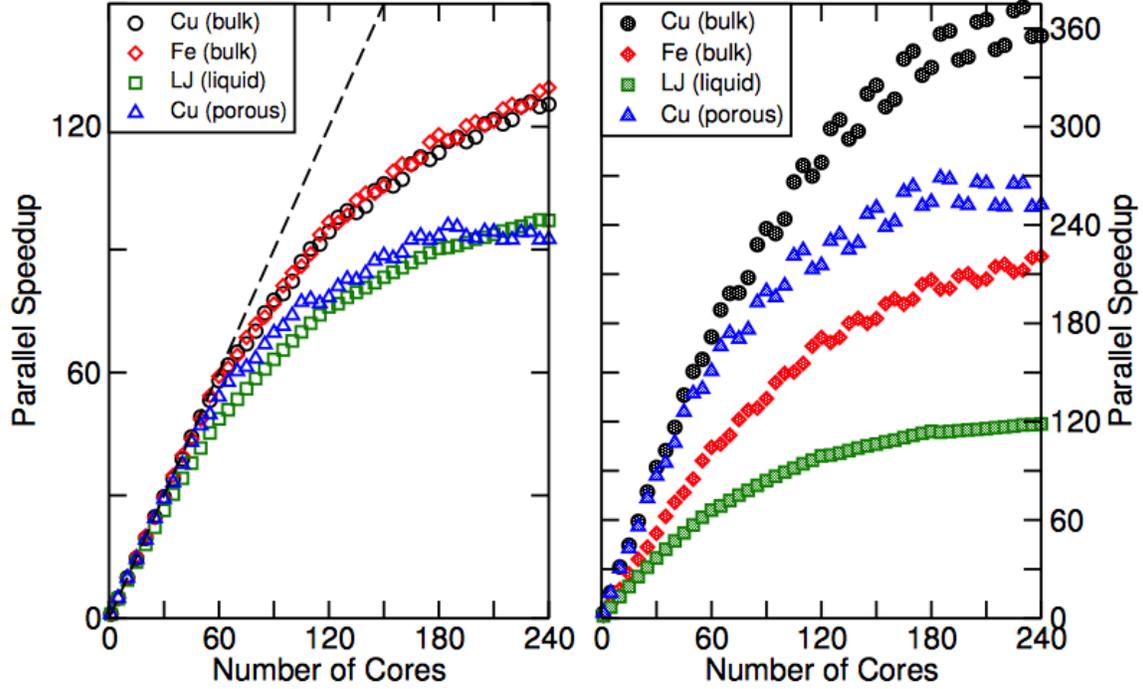


Figure 1: Speedups of the benchmark simulations on Intel Xeon Phi co-processors without SIMD vectorization (left) and with SIMD vectorization (right). The dashed line in the left panel indicates the ideal parallel efficiency.

Table I: Maximum speedups obtained for the four benchmark systems. See text for details.

	Cu (bulk)	Fe (bulk)	LJ (Liquid)	Cu (Porous)
S_{\max}	373.1	220.9	118.6	269.0
$S_{\text{vec}}/S_{\text{novec}}$	3.0	1.7	1.22	2.8
S_{serial}	3.2	1.8	1.5	3.1
S_{host}	1.8	1.3	0.9	1.4

in the force calculation, other factors become more important. This can be memory access time or other parts of the code that are not parallelized and/or not efficiently vectorized.

Without SIMD vectorization, the second test system using Mendeleev’s iron potential achieves the best speedup of all systems. This can be explained by the complexity of this potential, which reduces the impact of other code parts as well as memory access time. The use of the vector units, however, increases the performance of the iron simulation only by a factor of 1.7 – 1.8 whereas the gain factor is about 3 for the copper systems. This difference is due to the numerous conditional computations in case of the iron potential which reduce the efficiency of the vector instructions.

The tight-binding second moment potential used in the first and last test system has a higher computational complexity than the LJ potential but less conditional instructions than the iron potential. This explains why the copper test systems benefit more from SIMD instructions than the other two systems.

What remains to be explained is the difference of the parallel speedup of the copper systems. A possible reason for this is the size of the systems. In order to be fully efficient, the cell method

requires that more tasks can be run simultaneously than there are compute cores. For smaller systems, it can become difficult to fully utilize all of the cores of a Xeon Phi. Moreover, as pointed out above, the porous system contains more surface atoms, which increases memory traffic and adds to the overhead of the force computations.

CONCLUSIONS

We have implemented a novel parallel algorithm for MD simulations with short-ranged forces on shared-memory computers with SIMD vector units. Benchmark calculations on Xeon Phi coprocessors show very good speedups for three different potentials. Computationally heavy potentials show better speedups due to parallelization since the effect of memory access and other code parts is reduced. Conditional computations however hamper the efficiency of SIMD vectorization. Compared to a host system with conventional Xeon CPUs, the Xeon Phi coprocessor gives speedups between 0.9 and 1.8. It should be seen, however, that the current versions of the Xeon Phi represent the first generation of this many-core architecture. Future systems will most likely perform significantly better.

ACKNOWLEDGMENTS

This work has been supported financially by the Natural Science and Engineering Research Council of Canada (NSERC) and Laurentian University. The research was enabled by support provided by Compute/Calcul Canada, Compute/Calcul Québec and the Shared Hierarchical Academic Research Computing Network (SHARCNET).

REFERENCES

1. R. Meyer, Phys. Rev. E **88**, 053309 (2013); J. Phys.: Conf. Ser. **540**, 012006 (2014).
2. M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*, REP edition (Clarendon, Oxford, 1989) pp. 149–152.
3. G. J. Ackland, K. D’Mellow, S. L. Daraszewicz, D. J. Hepburn, M. Uhrin, and K. Stratford, Comp. Phys. Comm. **182**, 2587 (2011).
4. F. Cleri and V. Rosato, Phys. Rev. B **48**, 22 (1993).
5. M. I. Mendeleev, S. Han, D. J. Srolovitz, G. J. Ackland, D. Y. Sun, and M. Asta, Phil. Mag. **83**, 3977 (2003).
6. J. E. Lennard-Jones, Proc. Roy. Soc. Lond. A **106**, 463 (1924); Proc. Phys. Soc. **43**, 461 (1931).
7. M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*, REP edition (Clarendon, Oxford, 1989) p. 9.
8. M. P. Allen and D. J. Tildesley, *Computer Simulations of Liquids*, REP edition (Clarendon, Oxford, 1989) p. 29.
9. J. Jeffers and J. Reinders, *Intel[®] Xeon Phi[™] Coprocessor High-Performance Programming* (Morgan Kaufman, New York 2013) pp. 31–32; R. Rahman, *Intel[®] Xeon Phi[™] Coprocessor Architecture and Tools*, (APress, 2013) p. 55.