Efficient Parallelization of Short-Range Molecular Dynamics Simulations on Many-Core Systems

R. Meyer

Department of Mathematics and Computer Science and Department of Physics, Laurentian University, 935 Ramsey Lake Road, Sudbury (Ontario) P3E 2C6, Canada

(Dated: May 15, 2013)

This article describes an algorithm for the parallelization of molecular-dynamics simulations with short-range forces on many-core systems with shared-memory. The algorithm is designed to achieve high parallel speedups for strongly inhomogeneous systems like nanodevices or nanostructured materials. In the proposed scheme the calculation of the forces and the generation of neighbor lists is divided into small tasks. The tasks are then executed by a thread pool according to a dependent task schedule. This schedule is constructed in such a way that a particle is never accessed by two threads at the same time. Results from benchmark simulations show that the described algorithm achieves excellent parallel speedups above 80% per processor core for different kinds of systems and all numbers of cores. For inhomogeneous systems the speedups are strongly superior to those obtained with spatial decomposition.

PACS numbers: 02.70.Ns,02.70.-c

I. INTRODUCTION

Molecular dynamics (MD) simulation is one of the most important numerical methods in computational physics, chemistry, biology and materials science [1, 2]. It is a very versatile method that allows particle based simulations of a wide variety of systems provided that a suitable model for the interactions between the particles exists. The huge advances of computational power have dramatically increased the number of particles that can be handled by MD simulations. When used in conjunction with a short-range interaction model — i.e. a model where particle interactions become zero if the distance between the particles exceeds a cutoff distance $r_{\text{cut}}$ — simulations containing tenth or hundredths of millions of particles can be performed without too much difficulties.

A necessary ingredient for large-scale MD simulations is parallel computing. Two principal strategies for the parallelization of MD simulations are: space (or domain) decomposition and particle decomposition [3]. On shared memory architectures one can also employ thread based approaches using for example OpenMP [4, 5].

Particle decomposition assigns a subset of the particles statically to one processor. This methods is most effective if each particle interacts with a large fraction of all other particles. For large systems where this is not the case particle decomposition is less efficient since it tends to increase the amount of inter-processor communication.

In the spatial decomposition approach (see e.g. Ref. [6]), the simulation cell is divided into as many domains as there are processors. Each processor is then responsible for the calculations of forces on particles in one domain. When a particle crosses the border between two domains, it is reassigned to the processor of the new domain. Spatial decomposition achieves very good parallel speedups under two conditions: the domains must be large enough so that most of the interactions happen between particles on the same processor and the particle density system must be sufficiently homogeneous in order to achieve comparable computational loads on the processors.

Recently, a number of variations of the spatial decomposition approach have been developed [2, 10]. These neutral territory methods have the potential to outperform traditional spatial decomposition for high levels of parallelism. These methods are however not well suited to handle strongly inhomogeneous systems.

While it is fairly straightforward to parallelize MD programs using OpenMP, this approach rarely leads to satisfactory speedups. The reason for this are uncoordinated accesses to the particle data by the threads. In order to avoid race conditions particle updates must be protected by synchronization constructs or Newton’s third law cannot be exploited. The performance is further degraded since accesses to the same particle by different threads may result in frequent transfers of cache lines between the CPU cores.

In this article, an alternative, approach to the parallelization of MD simulations on shared-memory systems is described. In the cell task method, the force calculation and the construction of neighbor lists are divided into a large number of small tasks. These tasks are then executed by a thread pool according to a dependent task schedule that avoids access conflicts between the threads.

The motivation for the development of a new MD algorithm are changes in computing technology as well as application of MD simulations. The trend in computing technology goes in the direction of systems with a large number of cores operating on a shared memory. Systems with more than sixty CPU cores are already available and the trend to larger numbers of cores is likely to continue for some time. This change in computing hardware needs to be addressed by developers of software for
high-performance computing. Although message passing systems like MPI [11] can be employed on both distributed systems as well as shared memory architectures, one might doubt whether this is the best approach on systems with a large number of cores. Some experimentation with new algorithms is therefore required in order to find the best way to exploit modern computing hardware.

The second motivation for this work are shifts in the application of MD simulations. The increases of available computing power enable simulations of increasingly complex systems. It is now possible to simulate complete nanodevices or nanostructured materials that combine different materials on length scales of several nanometers. The inhomogeneity of such systems reduces the efficiency of the spatial decomposition approach. The proposed algorithm has been designed specifically to provide an efficient parallelization for such systems.

What this article does not address are simulations on graphics processing units (GPUs). The particularities GPU's require specialized algorithms that are outside the scope of this work. The implementation of MD simulations on GPUs has recently been discussed in Ref. [12].

II. DESCRIPTION OF THE CELL TASK ALGORITHM

A. Partitioning of the problem into tasks

The primary objective of this work was the design of an efficient parallel MD algorithm that achieves high parallel speedups for large and strongly inhomogeneous systems. Secondary goals were consistent speedups for all numbers of processors (or cores) and to make the method robust against external perturbations that might temporarily delay computations on one or more processors.

The problems of spatial decomposition and simple thread-based approaches can both be traced to the kind of geometric information used by the methods. Spatial decomposition fails to achieve a good load balance if the size of the processor domains exceeds the characteristic length scale of inhomogeneities in the system. In the OpenMP approach on the other hand the efficiency is reduced by the need for synchronization which is ultimately due to the fact that no geometric information is used at all. These observations lead to the idea to use geometric information on a small length scale for the parallelization.

In order to avoid synchronization constructs during the calculation of forces on the particles, one must ensure that two threads will never update the same particle at the same time. This means that threads must keep a distance of $2r_{\text{cut}}$ between the particles on which they work. Fortunately, the necessary information to ensure this condition is readily available in many cases or it can be obtained easily. Most general purpose MD codes use the so-called linked-cell technique [1, 13] to facilitate the construction of neighbor lists. In the linked-cell method, the simulation box is subdivided into a grid of small cells whose width is larger than $r_{\text{cut}}$ and for each cell a linked list of the particles in the cell is constructed. A particle can then interact only with particles in the same cell or one of its neighbors.

The proposed parallelization method reuses the data structures from the linked cell technique to partition the problem of the force calculation into tasks. A task in this method consists of the calculation of the forces on all particles in one grid cell. Since particles in one cell do only interact with particles in neighboring cells, two tasks can be executed concurrently if there are at least two grid cells between them in all directions. Tasks that fulfill this condition are called non-overlapping otherwise they are said to be overlapping. Fig. 1 shows a two-dimensional example of a cell grid and a set of tasks that can be executed in parallel since they are non-overlapping.

The essence of the proposed algorithm is:

- The particles are binned into grid cells (as part of the linked cell technique).
- The force calculation is broken into cell tasks. A cell task consists of the calculation of forces on all particles in one grid cell.
- The tasks are executed in parallel by a thread pool according to a dependent schedule that ensures that only non-overlapping tasks are executed simultaneously.

The algorithm does not assume that the execution of tasks require similar amounts of computing time. Even in a homogeneous system this might not be the case and in inhomogeneous systems a substantial amount of grid cells might in fact be empty. Notwithstanding, a good load balance is achieved if the tasks are dynamically scheduled rather than statically. Dynamic scheduling assigns the tasks to the threads as these finish previous tasks. This keeps all threads busy and averages the imbalances. In addition to this, dynamic scheduling limits the impact of external disturbances such as unrelated processes running on one of the cores or differences in memory access.
times. If a processor is slowed down by external factors, other processors will take on a larger number of tasks thereby minimizing the delay. With static scheduling a delay of one processor leaves the rest of the system idling.

Another positive side effect of the cell task algorithm is an improved memory access pattern. Particles in the same grid cell have similar sets of neighbors. The calculation of the forces on particle in the same cell within the same tasks therefore leads to high cache reuse without a special ordering of the particles.

After the force calculation, the second most time consuming part of an MD simulation is usually the generation of the neighbor lists. In a program that uses the linked cell method the neighbor lists are naturally generated for all particles in a grid cell at once. For this reason it is straightforward to apply the cell task method to the generation of the neighbor lists. The only difference to the force calculation is the kind of work performed by a cell task. During the neighbor list construction a cell task constructs the neighbor lists of all particles in one grid cell.

Other typical parts of MD simulations like the integration of the equation of motion are usually implemented as a simple loop over the particles that access only one particle at a time. Such loops do not require special synchronization considerations and can be parallelized using simple threading techniques.

B. Task scheduling: The wave method

A central problem of the proposed algorithm is the generation and execution of a dependent task schedule that guarantees that only non-overlapping tasks run concurrently. This section presents two algorithms for this problem named the forward and backward wave method. Both methods result in the same schedule but they differ in the order in which tasks are added to the schedule.

The wave methods group the cell tasks into an ordered set of so-called waves. A wave is a set of non-overlapping tasks that can be scheduled independently. For a rectangular cell grid, a wave can be generated as the cartesian product of a set of indices for each direction. Since non-overlapping tasks require that the difference between indices is at least two there are at least three different sets of indices for each direction. However, if periodic boundary conditions are used in a direction where the number of cells is not a multiple of three, a fourth configuration is required. The total number of waves required for a three dimensional system is therefore 27, 36, 48 or 64.

The grouping of the tasks into waves established a partial ordering of the tasks. A simple scheduling method is then to allow tasks in the same wave to run concurrently but to require that a wave must be completely finished before the next wave can start. The problem of this algorithm is that it imposes a barrier after each wave which reduces the efficiency. Note however that this scheme can easily be implemented in an environment that does not allow for dependent task scheduling.

The wave methods use the ordering imposed by the waves to order the access to the grid cells by the tasks. Each grid cell is accessed by at most 27 tasks (less for cells at the simulation box boundary). The wave methods allow a task T to execute as soon as all tasks from previous waves that access one of T’s cells are completed. A further simplification is possible since the preceding tasks depend on each other. For each of its cells T only has to wait for the last task that accesses this cell before T (since the last task has already waited for all previous tasks accessing this cell). The set of last tasks accessing any of T’s grid cells are called the direct predecessors of T. Similarly the direct successors of T are those tasks which have T as a direct predecessor.

The scheduling algorithm of both wave methods uses an acyclic graph algorithm similar to the one described in the documentation of Intel’s Threading Building Blocks Library [14]. Each task has a data structure that stores its grid cell, the number of its direct predecessors, a list of its direct successors and a reference counter. The reference counter is initially set to the number of direct predecessors and all tasks that have no direct predecessors are added to the thread pools list of tasks that are ready for execution. The execution of a task involves the following steps:

1. The work associated with the task is carried out (force calculation, neighbor list generation, . . .)
2. The reference counter of all direct successors is decremented atomically.
3. Successor tasks whose reference counter becomes zero are added to the ready list.

The purpose of the wave algorithms is to generate for each task its number of direct predecessors and the list of its successors. The forward wave method maintains a three-dimensional array $P[nx][ny][nz]$ that stores for each grid cell the index of the last task that has accessed the cell. Initially this array is initialized with a special marker (-1) that indicates that the cell has not been accessed. The algorithm loops over the waves in ascending order starting with the first wave. For each task in a wave the following steps are performed:

1. For all cells accessed by the task, copy the corresponding element of P into a list.
2. Eliminate duplicates and the special marker from the list. The result is the task’s set of direct predecessors.
3. Store the number of predecessors in the task structure.
4. For each direct predecessor, add the task to its successor list.
5. Initialize an empty successor list
6. Overwrite elements of \( P \) used in step 1 with the task’s index.

A minor technical drawback of the forward wave algorithm is the fact that when a task is created by the algorithm, only the predecessors are known. Since the tasks needs to store its list of successors rather then its predecessors it would simplify the task generation if the list of successors were known at the time a task is created. This is achieved by the backward wave method. The backward wave method works like the forward method but it replaces the array \( P \) with an array \( S[nx][ny][nz] \) which stores for each grid cell the index of the next task that accesses the cell. The array is again initialized with the special marker which now means that no further task accesses the cell. The backward algorithm then creates the waves in reverse order from the last wave to the first. For each task in a wave the method performs the following steps:

1. For all cells accessed by the task, copy the corresponding element of \( S \) into a list.

2. Eliminate duplicates and the special marker from the list. Store the result as the task’s direct successor list.

3. For all direct successors: Increment the number of direct predecessors by one.

4. Initialize the task’s number of direct predecessors to zero

5. Overwrite elements of \( S \) used in step 1 with the task’s index.

C. Implementation

In order to be able to test the cell task algorithm in practice, the method has been implemented in a general purpose parallel MD code named \texttt{mdntp} [13]. This code has been developed for large-scale simulations using many-body potentials of the embedded-atom method type [16] or the similar tight-binding second moment potentials [17]. The code already supported parallelization through spatial decomposition so that parallel speedups can be compared.

The implementation of the task approach makes use of Intel’s Threading Building Blocks library (TBB) for the management of the thread pool and the task scheduling. TBB is a C++ library that is available under an open-source license for many platforms [13]. It was chosen since it provides the necessary features for the implementation of a dependent task schedule.

Since \texttt{mdntp} already uses the linked-cell technique, the implementation of the wave method required relatively little changes to the code since most data structures were already in place and ready to use. The majority of the code to be developed concerns the task scheduling. This code is concentrated in a C++ class that creates and executes the dependent task schedule. In order to execute the schedule, a functor object is passed to the scheduler. When a task is executed it invokes the functor which in turn performs the work for this cell task. The usage of functors makes it possible to use a single scheduler object for all kinds of cell tasks.

Other changes concern mainly the force-calculation and neighbor-list generation. In the force calculation the loops over all particles had to be changed so that they run only over the particles in one grid cell. In addition to this the original code written in C was moved to a C++ class so that the scheduler could invoke it. Some care was required to create thread local storage accumulators for quantities like the potential energy. Finally, the code for the integration of the equations of motion was modified to support thread based parallelization.

D. Optimizations and Refinements

1. Empty Task Skipping

For large systems with a substantial amount of empty volumes a large number of grid cells may be empty. Instead of scheduling a task for these cells, it makes sense to check for empty tasks during the generation of the task schedule and skip empty cells altogether. This may lead to a substantial reduction of the data structures and the scheduling overhead. The drawback of this optimization is that the task schedule must be regenerated every time that the linked-cell algorithm has run since particles might have moved into a previously empty cell. Without empty-task skipping, the task schedule needs only to be regenerated if the number of grid cells changes. For systems that have a low number of empty cells it is therefore more efficient not to skip the empty cells and to save the overhead of the repeated schedule generation instead.

2. Cell Task Blocking

Another way to reduce the number of empty tasks and the number of tasks to be scheduled is to group small cell blocks of \( b_x \times b_y \times b_z \) grid cells into a single cell task. This corresponds to an enlargement of the grid cells by an integer factor for the purpose of task scheduling (the grid cells are still generated in the usual way so that the generation of neighbor lists is unaffected by this optimization).

Cell task blocking is a double edged sword. On the one hand, the reduction of the number of tasks through the blocking can reduce the scheduling overheads. In addition to this cache performance is improves as long as all particles affected by a task fit into the cache memory.

On the other hand, both advantages turn into disadvantages for larger cell blocks. The thread scheduler requires a large number of short tasks in order to obtain
a good load balance. And the cache performance will decrease if the amount of memory used by the blocked task exceeds the size of the cache. For these reasons task blocking should be used with care. Benchmark test are recommended in order to find the optimal block size for a system.

### III. RESULTS

In order to test the efficiency of the proposed algorithm a series of benchmark tests involving three different configurations were carried out. The configurations are a cubic block of fcc bulk copper (1,000,188 atoms), a spherical copper nanoparticle with a diameter of 30 nm (1,117,151 atoms) and a porous system of partially sintered copper nanoparticles (1,992,220 atoms). These configurations are very different in terms of there homogeneity and the challenge they pose for the spatial decomposition method.

All simulations used the tight-binding second moment potential by Cleri and Rosato [17]. The simulations were run over a period of 100 simulation steps while regenerating the neighbor lists (and task schedule) at every 10th step. Executions times were measured excluding the time for initialization of the simulation and loading or saving of the configuration. All simulations were repeated five times and the execution times were averaged over these runs.

The simulations were run on a dual hex-core Intel Xeon X5650 processors with 1333 MHz DRAM. Simulations were run using the original serial version of mdntp, the MPI based spatial decomposition version using 2 - 12 MPI ranks and the task-based version using 1 - 12 threads. Task blocking with a block size of $2 \times 2 \times 2$ was employed in the task-based method in all cases. Empty tasks skipping was used except for the simulations of bulk copper.

Parallel speedups of the MPI version were calculated with respect to the original serial version. Speedups of the cell task method were calculated with respect to runs of the cell task program using one thread. The reason for this difference is that the C++ implementation of the force calculations in the cell task program enabled some additional optimizations that improve even the serial performance of the program. Backporting of all changes into the original version would be difficult and seems pointless since the aim here is not to compare absolute performances but parallel speedups.

#### A. Bulk copper

The crystalline bulk copper configuration was chosen as an example for a perfectly homogeneous system for which the spatial decomposition method works well. Fig. 2 shows the parallel speedups obtained by the task-based method and spatial decomposition for this system. As expected, spatial decomposition yields excellent speedups above 80% per processor for almost all numbers of processors. There are however dips in the curve at 5, 10, and 11 processors. The reason for this is that for these numbers the regular crystal lattice of the system cannot be divided evenly among the processors. This emphasizes another disadvantage of spatial decomposition: The efficiency of the method depends on details of the system and not all numbers of processors work equally well.

The task-based method on the other hand delivers much more consistent speedup factors that increase monotonously with the number of threads. With the exception of the cases of 5, 10, and 11 processors which have already been discussed, both methods obtain similar speedups. According to the figure the cell task method might have a slight advantage but this should not be overemphasized as this might be a result of the differences in the force calculation code.

#### B. Copper nanoparticle

The spherical nanoparticle system is inhomogeneous in the sense that the particle fills only a part of the volume of the simulation box leaving the rest empty. For the spatial decomposition method this configuration is challenging since there is no simple way to divide the simulation box into an arbitrary number of domains with equal shapes so that each domain contains a similar number of particles. Exceptions are the cases of 2, 4 and 8 pro-

![FIG. 2. (Color online) Parallel speedup factors in simulations of bulk copper as a function of the number of threads or processors for the task-based method (black circles) and spatial decomposition (green squares). The dashed blue line represents the ideal speedup of 1 per processor.](image-url)
FIG. 3. (Color online) Parallel speedup factors in simulations of a spherical copper nanoparticle as a function of the number of threads or processors for the cell task method (black circles) and spatial decomposition (green squares). The dashed blue line represents the ideal speedup of 1 per processor.

The cell task method on the other hand has no particular problems with the porous nature of this system. The speedups shown in Fig. 5 for the cell task method are very similar to those obtained for the other two benchmark systems.

IV. SUMMARY AND CONCLUSIONS

This article describes the design of a parallel algorithm for MD simulations with short-ranged forces on many-core systems is described. The aim of the cell task algorithm is to provide an efficient parallelization method for systems where spatial decomposition is not effective. Examples for such problematic cases are large inhomogeneous systems like nanostructured materials or nanodevices.

The cell task algorithm makes use of the linked-cell technique to subdivide the force calculation into small tasks. The tasks are then executed by a team of threads according to a dependent task schedule. This schedule is an important part of the algorithm. It avoids the situation that a particle is accessed by two threads simultaneously. This effectively eliminates the need for synchronization constructs in the force calculation which makes the algorithm very efficient.

Benchmark calculations reveal two advantages of the task-based parallelization of MD simulations: First, for inhomogeneous systems like nanoparticles or porous systems the cell task algorithm yields considerably better speedup factors than spatial decomposition. For homogeneous systems both parallelization approaches achieve comparable speedups.

The second advantage of the cell task method is a remarkable consistency of its performance. A comparison
FIG. 5. (Color online) Parallel speedup factors in simulations of a system of partially sintered copper nanoparticles as a function of the number of threads or processors for the cell task method (black circles) and spatial decomposition (green squares). The dashed blue line represents the ideal speedup of 1 per processor.

of Fig. 2, 3, and 5 shows that the speedups obtained by the task-based method are nearly identical for the three systems. In addition to this there are no preferential numbers of threads. The speedups grow monotonously with the number of threads. Certainly, the speedup per processor decreases for larger parallelism. This however is to be expected since for larger numbers of thread the performance is limited by memory access speed.

The consistent performance of the cell task method relieves users of MD programs from technical performance considerations. With this method there is no need to tailor the size or shape of the simulated system in order to achieve the highest speedup factor for the available number of processors.

The benchmark systems used in this article contain only one type of atoms. One can imagine that a system that contains different materials in different parts of the simulation box will face similar problems as a system that has areas with no material at all. It can however be expected that the cell task method delivers similar speedups for this type of system. The reason for this is the dynamic assignment of the tasks to the threads and the fact that the threads work on tasks from all areas of the system.

In the future the algorithm will be extended into a hybrid method that uses spatial decomposition for the communications among a distributed system of multicore nodes and the cell task algorithm for parallelization on the nodes. In addition to this the algorithm will be tested and optimized for Intel’s Many-Integrated-Core architecture.

ACKNOWLEDGMENTS

This work has been supported financially by Laurentian University and the Natural Sciences and Engineering Research Council of Canada (NSERC). Generous allocation of computer time by the Shared Hierarchical Academic Research Network (SHARCNET; http://www.sharcnet.ca/) is gratefully acknowledged.

[1] M. P. Allen and D. J. Tildesley, Computer Simulations of Liquids (Clarendon, Oxford, 1987).
[2] D. Frenkel and B. Smit, Understanding Molecular Simulation (Academic Press, San Diego, CA, 2002).
[3] H. J. C. Berendsen, D. van der Spoel, and R. van Drunen, Comp. Phys. Comm. 91, 43 (1995).
[4] http://www.openmp.org/.
[5] B. Chapman, G. Jost, and R. Van der Pas, Using OpenMP (MIT Press, Cambridge, MA, 2008).
[6] S. Plimpton, J. Comp. Phys. 117, 1 (1995).
[7] M. Snir, Theor. Comp. Sys. 37, 295 (2004).
[8] D. E. Shaw, J. Comput. Chem. 26, 1318 (2005).
[9] K. J. Bowers, R. O. Dror, and D. E. Shaw, J. Phys.: Conference Series 16, 300 (2005).
[10] K. J. Bowers, R. O. Dror, and D. E. Shaw, J. Chem. Phys. 124, 184109 (2006).
[11] http://www.mpiforum.org/.
[12] J. A. Anderson, C. D. Lorenz, and A. Travesset, J. Comp. Phys. 227, 5342 (2008).
[13] D. C. Rapaport, Comp. Phys. Comm. 62, 198 (1991).
[14] http://threadingbuildingblocks.org/.
[15] R. Meyer, Ph.D. thesis (in German), Gerhard-Mercator-Universität Duisburg, Germany (1998).
[16] M. S. Daw and M. I. Baskes, Phys. Rev. B 29, 6443 (1984).
[17] F. Cleri and V. Rosato, Phys. Rev. B. 48, 22 (1993).