Event-Driven Molecular Dynamics in Parallel

S. Miller[1] and S. Luding[1, 2]

[1] *Institut für Computeranwendnungen 1, Universität Stuttgart, Pfaffenwaldring 27, D-70569 Stuttgart, Germany*
[2] *Particle Technology, DelftChemTech, TU Delft, Julianalaan 136, 2628 BL Delft, The Netherlands*

(Dated: June 12, 2003)

Although event-driven algorithms have been shown to be far more efficient than time-driven methods such as conventional molecular dynamics, they have not become as popular. The main obstacle seems to be the difficulty of parallelizing event-driven molecular dynamics. Several basic ideas have been discussed in recent years, but to our knowledge no complete implementation has been published yet. In this paper we present a parallel event-driven algorithm including dynamic load-balancing, which can be easily implemented on any computer architecture. To simplify matters our explanations refer to a basic multi-particle system of hard spheres, but can be extended easily to a wide variety of possible models.

I. INTRODUCTION

Event-driven molecular dynamics is an effective algorithm for the simulation of many-component systems, which evolve independently, except for discrete asynchronous instantaneous interactions. As an example we will discusse a system consisting of N hard spheres in a box with periodic boundary conditions, but the algorithm can be extended to particles with different shapes and interacting via any piecewise constant potential or to completely different problems as well [1].

Event-driven molecular dynamics processes a series of events asynchronously one after another. A straightforward but simplistic approach [2] updates all particles at each event. A more sophisticated algorithm has been presented [3], which updates only those particles involved in an event. It has been succesfully applied to many different problems, among them granular gases [4, 5], tethered membranes [5, 6], and battlefield models.

In many physical systems the duration of the interaction of the components, e. g. the collision of two particles, is negligible compared to the time between these interactions. The simulation of such systems with traditional time-driven molecular dynamics is highly inefficient. Instead, it is straight-forward to consider the interactions as instanteneous events and solve the problem with an event-driven algorithm.

One reason why event-driven molecular dynamics has not become as popular as conventional time-driven molecular dynamics is the fact that parallelization is extremely complicated. The paradoxial task is to algorithmically parallelize physically non-parallel dynamics. Nevertheless some ideas and basic considerations about the parallelization have been proposed in [7]. They are especially suited for shared memory computers, but can be transferred to distributed memory architectures as well [8]. Apart from those ideas no full and general implementation of a parallelized algorithm including load-balancing has been published yet, to our knowledge. In this paper we present an algorithm, which is based on those ideas, but is enhanced and completed at several points. It can be implemented with generic tools such as MPI, and therefore it is suited for shared and distributed memory architectures alike.

In section II we explain the details of the implementation of event-driven molecular dynamics. The parallelization of the algorithm is presented in section III and a summary is given in section IV.

II. EVENT-DRIVEN MOLECULAR DYNAMICS

Section II A gives an outline of the main routine of the algorithm, which is composed of 4 steps (see Fig. 1). Then the most important data structures are introduced in section II B and hereby step 1 and 4 are treated. Step 2 and 3 are discussed in sections II C and II D, respectively. Finally, section II E deals with performance issues.

A. Outline

Event-driven molecular dynamics processes a series of discrete events. In a system of hard spheres events typically refer to instantaneous collisions, involving exactly two particles. Only these two particles are processed; the state of the other particles remains unchanged. So the state information of most particles refers not to the current time, but to a different point of time in the past.

Event processing includes a state update for the concerned particles (see section II C) and the calculation of the next future events for those particles (see section II D). An outline of the algorithm can be seen in Fig. 1.

The *serial* algorithm only adds to this event processing loop an initialization step at the beginning and periodic data output. The *parallel* algorithm needs several additional routines, which are described in section III.

event processing loop:

1. get next event from priority queue → current time
2. update states of both particles to current time
3. calculate new future events for both particles
4. schedule next event for both particles in priority queue
5. goto 1

FIG. 1: Outline of the main routine of the algorithm

particle states	: $N \times \{t_0, \mathbf{r}(t_0), \mathbf{v}(t_0), \text{counter}, \text{cell}, \text{id}\}$
event list	: $N \times \{t_{ev}, \text{type}, \text{partner}, \text{counter}\}$

FIG. 2: Data structures for N particles

B. Data Organization

The algorithm maintains two data fields, *particle states* and *event list*, which contain exactly one entry per particle (see Fig. 2). The former refers to the past of a particle, the latter to its future ($\rightsquigarrow \forall i$: state time $t_0(i) \leq$ current time $\leq$ event time $t_{ev}(i)$).

A *particle state* consists of the physical state of a particle, such as position, velocity, etc. immediately after the most recently processed event of that particle, the point of time of that event, an event counter (see section II C), a cell number (see section II D), and a global particle number, which is needed to identify particles on different processes (see section III).

The *event list* associates with every particle an event in the future. The data units of the *event list* consist of event time, event type, event partner, and a copy of the event counter of the partner.

All the events are scheduled in a priority queue which is usually implemented as an implicit heap tree [3, 7, 9], but other data structures with similar characteristics are possible, too [9, 10]. A heap tree is a binary tree with the following ordering rule: Each parental node has higher priority than its children; the children themselves are not ordered. So the root node always has highest priority, i. e. the earliest event time. To get the next event from the priority queue takes computational costs of $\mathcal{O}(1)$. Insertion of a new event in the priority queue is done with costs of $\mathcal{O}(\log N)$.

This data organization is more efficient compared to the algorithm in [3, 7], since no double buffering is needed here, however, the basic ideas of [3, 7] are still valid.

C. State Update

When a collision between two particles is processed, the states of these particles are updated from a point of time in the past to the current simulation time. First, the positions and velocities of the particles immediately before the collision can be derived by inserting the old state in the equation of motion. As a result of this first step the particles are touching each other. Then the interaction between the particles takes place and yields new particle velocities. Now, the event counter is increased, the state time is updated to the current simulation time, and the values of the positions and velocities immediately after the collision are stored in the *state* array.

A typical collision rule for hard spheres [11, 12] looks like

$$\mathbf{v}'_{1/2} = \mathbf{v}_{1/2} \mp \frac{1+r}{2} \left(\hat{\mathbf{k}} \cdot (\mathbf{v}_1 - \mathbf{v}_2) \right) \hat{\mathbf{k}} \, ,$$

where primes indicate the velocities $\mathbf{v}$ after the collision, $\hat{\mathbf{k}}$ is a unit vector pointing along the line of centers from particle 1 to particle 2, and r denotes the restitution coefficient. For the performance tests in the subsequent chapters we have used the simplest case without dissipation ($r = 1$). [15]

If the event partner has undergone collisions with other particles between the scheduling and the processing of the event, it becomes invalid. Validity of event partners can be checked by comparing the event counter of the partner to the copy in the event list. If they do not match, the partner has collided with other particles in the meantime, and the scheduled collision is no longer valid. Then event processing only refers to one particle and the state update described above only consists of the particle motion; no collision is performed. After that the algorithm continues in the normal way, i. e. the next event for the particle will be calculated and scheduled in the priority queue.

Note that the algorithm in [3, 7] uses a different strategy to detect invalid event partners: The algorithm checks at each collision if an event partner becomes invalid and if so marks this partner. This strategy is less efficient, because sometimes the same partner is invalidated several times. Besides, in the parallel algorithm additional communication between different processes might be necessary.

D. Event Calculation and Linked Cell Structure

When an event has been processed, new events for the particles involved in that event have to be calculated. In simulations of hard spheres this means the calculation of possible future collisions. If the particles move on ballistic trajectories

$$\mathbf{r}_i(t) = \mathbf{r}_i(t_0) + \mathbf{v}_i(t_0)\,(t - t_0) + \frac{1}{2}\mathbf{g}\,(t - t_0)^2 \, ,$$

two particles 1 and 2 will collide at time t_{12}:

$$t_{12} = t_0 + \Big(- \mathbf{r}_{12} \cdot \mathbf{v}_{12} - \sqrt{(\mathbf{r}_{12} \cdot \mathbf{v}_{12})^2 - (r_{12}^2 - (R_1 + R_2)^2)\, v_{12}^2} \Big) / v_{12}^2 \,,$$

where $\mathbf{v}_{12} = \mathbf{v}_2(t_0) - \mathbf{v}_1(t_0)$ and $\mathbf{r}_{12} = \mathbf{r}_2(t_0) - \mathbf{r}_1(t_0)$ are the relative velocities and positions of the particles at time t_0, and R_i are the radii of the particles. If t_{12} is imaginary or smaller than t_0, the particles will not collide.

If the algorithm would have to check for collisions with all other particles, performance would be very poor for large numbers N of particles. So we divide simulation space in C cells with equal sides. Each particle belongs to the cell in, which its center lies. If the cell size is larger than the maximal interaction length of the particles, i. e. the particle diameter in the case of hard spheres, event calculation has to check for possible collisions of two particles only if they belong to the same cell or if their cells are neighbors. (One square cell has 8 neighbors in 2D and a cube has 26 neighbors in 3D.)

However, the algorithm has to keep track of cell changes. So additional events, namely cell changes, come into play. They are treated just in the same way as the collision events; only the collision partner is the boundary of a cell. The difference to a collision event is that only one particle is involved in a cell change, and the velocity of the particle does not change at event time. Cell changes at the boundary of the simulation space require that the position vector jumps to the opposite side of the simulation volume due to the periodic boundary conditions.

E. Optimal Cell Numbers

On average there are $3^D N/C - 1$ particles in the neighborhood of each particle. So in the limit $C \ll N$ this number becomes very large and many possible events have to be calculated after each collision. On the other hand, if $C \gg N$, then each particle has to cross many cell boundaries between a collision of two particles, and thus more events have to be treated to complete the simulation. These two contributions compete with each other, the first one is proportional to $(C/N)^{-1}$ and the second one is proportional to $(C/N)^{1/D}$. Fig. 3 (left) shows that there is a broad minimum of the simulation time for low densities at the optimal cell number $C/N \approx 1.5$ in 2D and $C/N \approx 8$ in 3D. So the program can choose an optimal cell number before the calculation starts. Note that in the high density limit, especially in 3D, the optimal cell number cannot be reached, because the size of the cells has to be larger than the particle diameter (see Fig. 3 (right)). So, in this case C should simply be chosen as big as possible.

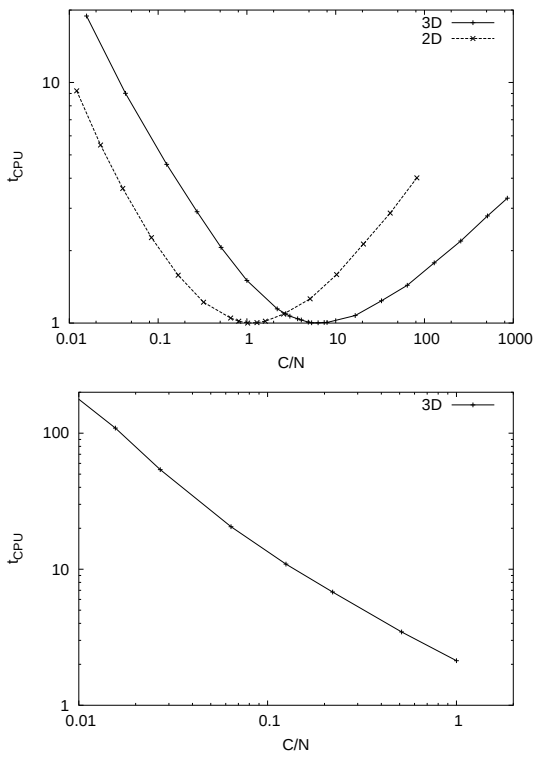

FIG. 3: Simulation time t_{CPU} (arbitrary units) plotted against the number C of cells in 2D and 3D with the serial algorithm. Used are $N = 10000$ particles (2D) and $N = 8000$ particles (3D) with the volume fractions $\nu = 0.008$ (2D), $\nu = 0.0005$ (3D, top fig.), and $\nu = 0.5$ (3D, bottom fig.).

III. THE PARALLEL ALGORITHM

In section III A we demonstrate how parallelization can be achieved via domain decomposition and dynamic load-balancing. Then, in section III B we point out the difficulties, which arise in parallelizing this algorithm and the necessity of state saving and error recovery (see section III C). In section III D the parallel algorithm is explained in detail. Finally, some performance issues are discussed in section III E.

A. Domain Decomposition and Dynamic Load-Balancing

Parallelization is achieved via domain decomposition. Each cell is affiliated with a process, but this affiliation can change during the simulation if the load of the processes has to be rebalanced.

In order to realize maximal performance, the computational load should be distributed equally among the processes. If inhomogenities exist from the very beginning or emerge during the simulation, dynamic load-balancing becomes important. Cluster formation in granular gases is a typical example for this case. [4, 13]

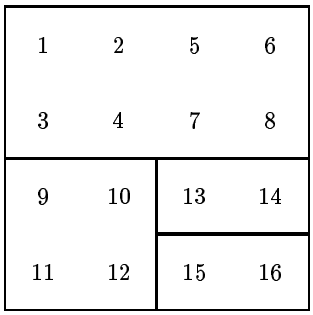

FIG. 4: Cell numbering layout in 2D with an exemplary domain decomposition.
Inhomogeneous load distribution leads to unequal domain sizes.

Domain decomposition should thus take into account the following points: Firstly, the load of the processes should be distributed homogeneously. Secondly, in order to minimize the process communication, the border area of each process domain should be as small as possible, which implies that the ideal process domains are squares resp. cubes. Thirdly, a simple and fast function, which assigns a cell to a process and vice versa is required.

We meet these demands in the following way: Cell numbers are assigned to cell coordinates in a tree-like structure, see a small 2D-example in Fig. 4. (Note that a realistic example has thousands or millions of cells.) These cell numbers are obtained by interleaving all the bits 1 to k of the cell coordinates: $z[k], y[k], x[k], \ldots, z[1], y[1], x[1]$. The result is a cell number in the range 0 to $2^{kD} - 1$. (For convenience these numbers are increased by one in Fig. 4.) [16]

Then a block of 2^n consecutive cell numbers is affiliated with each process, where n can be different for each process. Suppose the cells 1-16 in Fig. 4 should be distributed over 4 processes. If a lot of particles are aggregated in the lower right corner, load-balancing could result in the following layout: Process I is affiliated with cells 1-8, process II with cells 9-12, process III with cells 13-14, and process IV with cells 15-16.

Every now and then the load of the processes has to be checked. A reasonable and simple measure of the load is the number of particles or the number of collisions in the process domain. If a restructuring of the domain decomposition would result in a significantly better load-balance, three processes, respectively, exchange information about their particle and cell data, so that two light-weight neighboring processes can merge their domains and a heavy-weight process can split its domain in two halves. In the example above, if the initially inhomogeneous system becomes homogeneous, this procedure could result in the merging of the domains of processes III and IV and a splitting of the domain of process I. Process II does not participate in this rebalancing, but it should be informed that its neighbor processes have changed, of course. In the end the layout would look like that: Process I is affiliated with cells 1-4, process II with cells 9-12, process III with cells 5-8, and process IV with cells 13-16.

B. Causal Order

A parallel approach to simulate asynchronous events has to make use of the concept of local times. Each process handles the events in its domain and thereby increases the local simulation time. When particles cross domain boundaries the affected processes communicate with each other and events are inserted into or removed from the event lists of the processes. If the event time of a newly inserted particle is less than the local simulation time, causality is violated, because a collision of that particle with another one, which has already been processed, could have been missed.

In general, there are two strategies to circumvent this problem: In a *conservative* approach only those processes that are guaranteed not to violate causality are allowed to handle their events, the rest of the processes has to idle. In an *optimistic* approach the processes do not have to idle. If causality is violated, a complex rollback protocol which corrects the erroneous computations has to be executed. Whether a conservative approach is efficient or not depends highly on the maximal event propagation speed.

Unfortunately, in event-driven molecular dynamics there is no upper limit for the speed of events propagating along a chain of particles [7], even if the particles themselves are moving very slowly. In other words, in the conservative case one process is operating and all others are idling and we are back to the serial algorithm. So we are left with the optimistic strategy and have to undertake the task of implementing a rollback protocol.

C. State Saving and Error Recovery

When a causality error is detected, the simulation is restarted from the latest saved state. So the algorithm has to make a backup copy of the simulation data periodically and has to ensure that there is no latent causality error in this backup. The latter is guaranteed if all processes are synchronized at saving time. This means that only those processes are allowed to continue their computations whose local simulation times have not yet reached synchronization time. Note that there are other operations, like e. g. data output or load-balancing, which require periodic synchronization anyway. Besides, without synchronization the local simulation times tend to drift apart, which makes causality errors very likely [14].

To find the optimal backup interval, we make use of an adaptive strategy. If no causality error turns up between two successive save operations, the interval increases, otherwise it decreases. If other operations trigger synchronization, this occasion is used for state saving, too, of course.

parallel loop:

1. communication about border zone events → timestep
2. timestep ← min(timestep,time(synchronous tasks))
3. for(timestep) **event processing loop** (see Fig. 1), send border zone particle information
4. receive and process particle informations
5. error detection
6. if(error) rollback
7. if(global min(current time) = time(synchronous tasks)) state saving, load-balancing, data output
8. goto 1

FIG. 5: Outline of the parallel algorithm

If a causal error occurs, all processes perform a rollback to the saved state. Furthermore a synchronization barrier is scheduled for the time, when the error occured. This prevents the same causality error from happening again. Of course, another error could occur at an earlier point of time. Then the simulation would perform another rollback and an earlier resynchronization would be scheduled.

For comparison the algorithm described in [7] needs two backup copies of the simulation data. So our strategy reduces memory consumption further by a factor 3/2.

D. Border Zone and Process Communication

The border zone (in [7] it is called *insulation layer*) consists of the cells C_{border} whose neighbors belong to a different process. With the domain decomposition described in section III A, there is always a "monolayer" of border zone cells at the boundary of a process area.

Each process thus maintains a list of virtual border zone cells which actually belong to its neighboring processes and a list of virtual particles residing in the virtual border zone cells. Those virtual particles can act as partners for real particles. But no events are calculated for them directly and they are not represented in the event list, since they are real particles on another process. The events are already calculated there and will be communicated to the adjacent processes.

However, it is highly inefficient for a process to communicate after every event with its neighbors. So the parallel algorithm is designed in a stepwise manner (see Fig. 5): Each computing step lasts until the next event in the border zone of a process, which is obtained in the following way: An event is associated with each particle. Each event belongs to a cell (or two adjacent cells, if two colliding particles reside in different cells or if a particle changes from one cell to another). The smallest event time in a cell is called the cell time t_{cell}. These cell times are stored in an additional priority queue similar to the event list in section II B. This list contains the scheduled cell times for all local border zone cells and returns the minimum time $t_{\text{step}} = \min_c \left(t_{\text{cell}}(c)\right) = t_{\text{cell}}(c^\star)$ which belongs to cell $c^\star$.

Now, this time is used as the preliminary length of the calculation step. But firstly, the neighboring cells of $c^\star$, if located on other processes are checked. If they have scheduled an even earlier event, t_{step} will be shortened to that event. The communication is done in the following way: Each process sends $c^\star$ and $t_{\text{cell}}(c^\star)$ as a query to the neighboring processes, which check the adjacent cells for their event times and reply with the minimum thereof. If this answer is smaller, then it is used as t_{step} instead (see Fig. 5, step 1). Periodic tasks like data output, load-balancing or state saving, which require synchronization of the processes can shorten t_{step} even more (see Fig. 5, step 2).

One could think of more rigid policies, which determine the length of a step, see e. g. [7]. We have also tried other strategies, which reduce the number of rollbacks. But on the other hand, they reduce parallelism, too. A large part of the processes are idling, the communication overhead increases, and the overall performance goes down.

After the communication phase, parallel event processing starts and proceeds until the calculated point of time t_{step} (see Fig. 5, step 3), i. e. on average $\mathcal{O}\left(C/C_{\text{border}}\right)$ iterations are processed. If the algorithm encounters an earlier border zone event which has not been anticipated, the event processing step is stopped immediately to prevent the occurence of a causality error. But normally, the last processed event is a regular border zone event. Then, after the last particle state update, the state information has to be communicated to the neighboring processes.

When a process has finished its computing step, it reads the particle state messages, which it has received during this step and adapts its data structures accordingly (see Fig. 5, step 4). Real particles can only be affected as collision partners of virtual particles. For a virtual particle there are several possibilities: A virtual particle can become a real particle by changing from one process to another, it can remain a virtual particle, but with a different position, velocity or cell number, or finally it can emerge on or disappear from the virtual particle list, because it enters or leaves the border zone, respectively. If a virtual particle becomes real and newly calculated events for this particle violate causality, i. e. they are happening before the local simulation time, an error is signaled. Moreover, the processes check the border zone cell times and compare them with the replies to their neighboring processes. If the actual cell time is smaller than the reply (which corresponds to the current time of the neighbor process), a causality error has already happened with high probability, because a collision has been missed on the neighbor process, and an

error is signaled, too (see Fig. 5, step 5).

If no error is detected, the simulation continues with the next step. Otherwise a global rollback is performed (see Fig. 5, step 6) and the erroneous simulation step is restarted from the latest saved state (see section III C). To prevent the reappearance of the same error, a synchronization barrier is issued at the time when the error occured.

Other synchronization barriers can stem from periodic tasks such as state saving, load-balancing or data output (see Fig. 5, step 7).

However, most of all parallel loops do not comprise synchronization of the processes, but only communication between them.

E. Parallel Performance

First of all, parallelization imposes several performance penalties on the algorithm, among them communication overhead, idle time, state saving, and rollbacks after erroneous computation steps. On the other hand, the computational costs are shared among several processes. If the latter outweighs the former, parallelization can be considered succesful. The same holds true for memory requirements. State saving makes a copy of the whole system state and therefore doubles the memory usage. But here as well parallelization combines the limited resources of single processes.

Figs. 6-7 show that the speed-up of the parallelization, i. e. the reciprocal of the simulation time, in 2D and 3D, for fixed system size, is approximately proportional to $P^{1/2}$, where P is the number of processes. Furthermore, as shown in Fig. 8, the parallelization is also scalable if the number of processes is chosen proportional to the system size. There are deviations for small P when the effect of the parallelization overhead is large. In addition, it has been shown in [7] that the error recovery method presented in section III C is not scalable for $P \to \infty$. But for practical purposes, at least until $P = 128$, the algorithm remains scalable.

We have tested the parallel algorithm on a computer cluster with Pentium III 650 MHz processors and a 1.28 GBit/s switched LAN. On a supercomputer with optimized communication hardware, scalability should be even better.

For real physical applications with more than 10^2 collisions per particle, the maximal number of particles typically is limited, by both, available memory, see Fig. 9, and computing time, to about 10^6 particles per process, i. e. a total number of particles of about 10^8.

IV. SUMMARY

We have demonstrated how to parallelize event-driven molecular dynamics successfully. Our algorithm is based on some ideas from [7], i. e. we have used an optimistic

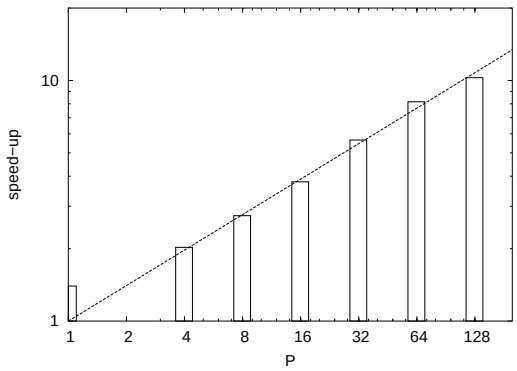

FIG. 6: Speed-up for different numbers P of processes in 2D. $N = 5 \cdot 10^5$ particles, volume fraction $\nu = 0.3$, $C = 1024^2 \approx 10^6$ cells.
The dashed curve has a slope of 0.49. The data point for 1 process deviates from the dashed curve, because the serial algorithm has no communication overhead. Note the logarithmic axes.

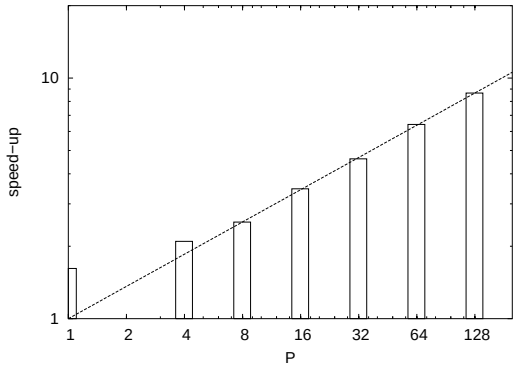

FIG. 7: Speed-up for different numbers P of processes in 3D. $N = 2 \cdot 10^6$ particles, volume fraction $\nu = 0.25$, $C = 128^3 \approx 2 \cdot 10^6$ cells.
The dashed curve has a slope of 0.45.

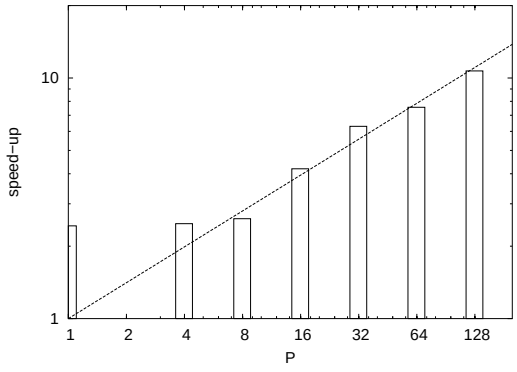

FIG. 8: Speed-up for different numbers P of processes in 3D. $N/P = 5 \cdot 10^4$ particles per process, volume fraction $\nu = 0.2$. The dashed curve has a slope of 0.50.

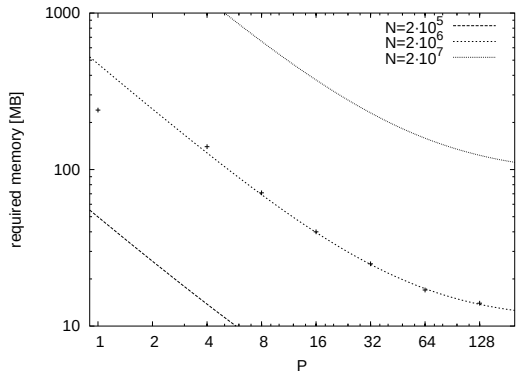

FIG. 9: Memory requirements per process for different numbers P of processes in 3D.
The curves are obtained by a simple estimate, the points represent the measured values for $N = 2 \cdot 10^6$. The curves are obtained by a simple estimate $N(c_1 P^{-1} + c_2 P^{-1/3} + c_3)$, where the first term refers to real particles, the second term to virtual particles, and the third term to global data. The data point for 1 process deviates from the estimated value, because the serial algorithm has no need for state saving.

parallelization approach which performs a rollback protocol if a causality error occurs. But the algorithm is enhanced in several ways:

Firstly, we have implemented dynamic load-balancing, which makes simulation of inhomogeneous systems possible. Computing time is further reduced by an adaptive linked-cell structure which determines the optimal cell sizes.

Secondly, we have transferred the shared memory approach of [7] to distributed memory architectures as well. The parallelization has been realized with MPI. In order to minimize idle time, we have made use of asynchronous communication, i. e. send and receive actions are decoupled from each other. In addition, the amount of communication is limited to the absolute minimum: The event processing can continue steadily until a border zone event takes place on the local process or is expected to take place on a neighboring process. Only then the calculation has to be interrupted in order to communicate with the neigboring processes. So, even on a cluster with rather poor communication hardware, parallelization yields a speed-up proportional to $P^{1/2}$ — at least up to $P = 128$ parallel processes.

Thirdly, we have optimized the data structure of the algorithm. With event-driven molecular dynamics insufficient memory is often a more serious problem than computing time. In total our optimizations reduce memory requirements to one third as compared to the method proposed in [7].

This enabled us to perform simulations of real physical problems with up to 10^8 particles.

[1] B. D. Lubachevsky, Bell Labs Tech. J. **5**, 134 (2000).
[2] B. J. Alder and T. E. Wainwright, J. Chem. Phys. **31**, 459 (1959).
[3] B. D. Lubachevsky, J. Comput. Phys. **94**, 255 (1991).
[4] S. Luding and H. J. Herrmann, Chaos **9**, 673 (1999).
[5] S. Luding, Computer Physics Communications **147**, 134 (2002).
[6] S. Miller, S. Luding, and G. Gompper (2003), in preparation.
[7] B. D. Lubachevsky, Int. J. Comput. Simul. **2**, 372 (1992).
[8] M. Marín, Comput. Phys. Commun. **102**, 81 (1997).
[9] D. E. Knuth, *The Art of Computer Programming* (Addison-Wesley, Reading, MA, 1968).
[10] M. Marín and P. Cordero, Comput. Phys. Commun. **92**, 214 (1995).
[11] S. Luding, Phys. Rev. E **63**, 042201 (2001).
[12] S. Luding, Advances in Complex Systems **4**, 379 (2002).
[13] S. Luding, Comptes Rendus Academie des Science **3**, 153 (2002).
[14] G. Korniss, M. A. Novotny, P. A. Rikvold, H. Guclu, and Z. Toroczkai, Materials Research Society Symposium Proceedings Series **700**, 297 (2001).
[15] Soft spheres can be approximated via particles consisting of concentric shells with piecewise constant interaction potentials.
[16] An example: The bottom left cell in Fig. 4 with the coordinates 0/3 (bit pattern 000/011) has the number corresponding to the bit pattern 001010, i. e. 10. Incrementation by one yields the cell number 11.