JOURNAL OF COMPUTATIONAL PHYSICS 109, 306–317 (1993)

Efficient Algorithms for Many-Body Hard Particle Molecular Dynamics*

MAURICIO MARÍN

Departamento de Computación, Facultad de Ingeniería, Universidad de Magallanes, Casilla 113-D, Punta Arenas, Chile

DINO RISSO† AND PATRICIO CORDERO

Departamento de Física, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Casilla 487-3, Santiago, Chile

Received March 2, 1992; revised January 11, 1993

Many-body simulations are very CPU-time consuming, making the problem of having efficient algorithms specially relevant. In this paper we propose a strategy—for the simulation of hard particle systems—that is efficient, memory saving, and easy to understand and to program. The time intervals by which the simulation proceeds are the increments between collisions (events), and these are dictated by the system itself. Hence these are event-driven simulations. Our strategy is devised to (a) minimize the number of coordinate updates per event, (b) predict new events only between nearby particles, and (c) efficiently manage the events predicted during the simulation. Empirical results are given to show the performance of our strategy in different computers as well as to compare with other approaches. It is seen that our proposed algorithm is efficient for a wide density range. We also include an analysis of the performance of the strategy proposed.
© 1993 Academic Press, Inc.

1. INTRODUCTION AND SUMMARY

A well-established tool for computational studies of solids and fluids is N-body molecular dynamics. Event-driven molecular dynamics [1, 2] is the optimal choice with piecewise constant potential interactions.

When the interaction potential is piecewise constant, particles move free of interparticle forces except for isolated instants where they suffer impulsive forces or *events*. The movement of each particle between events obeys Newton's equation with whatever external forces (e.g., gravity) may exist. The time intervals by which the simulation proceeds are the increments between events, and these are dictated by the system itself. Such characteristics make these types of simulations particularly efficient in the sense that the simulation can be run a long time, measured in units of a system time like, e.g., the diffusion time. Applications of these types of simulations have been illustrative and inspiring (see, for example, [3–9].)

There are several other types of molecular dynamics simulations that require totally different computational techniques. In general those are time-driven simulations, meaning that a time step has to be fixed by the algorithm itself and not by the dynamics of the system [10–12].

Algorithms appropriate to make hard particle computer simulations have been developing since the late fifties, starting with the classical articles of Alder and Wainwright [1, 13]. In [13] a general algorithm and the idea of dividing the system in a grid of small cubes—*cells*—to eliminate calculations for particles that are far apart was introduced. In [14] a scheme for economizing coordinate updates and for keeping a list of possible future collisions as efficiently as possible was proposed.

One of the basic concepts used in this type of simulations is that of *future events*. Possible future events need to be calculated to decide which one will actually happen next. The simulation then jumps to that instant. The other events are kept in a *future events list* since some of them will take place several steps later and there is no need to recalculate them. Still some others will never take place because a collision *invalidates* them, namely, one of the partners suffers a previous collision.

The first strategies proposed to manage efficiently the future events list (FEL) had a cost $O(\sqrt{N})$ [15, 16]. A substantial improvement for hard particle molecular dynamics was given by Rapaport [2]. He created an efficient algorithm $O(\log_2 N)$ to maintain an *event list scheduling*. Optimizing the administration of the FEL is important because a significant fraction of the time of the simulation is spent on this job. Recently, in [17] a different approach was presented.

In this paper we propose a new strategy—based on a future events list and a basic cycle—to make simulations of systems consisting of N hard particles. The proposed

* Partially supported by FONDECYT Grant 90-1240.

† Partially supported by a fellowship from Fundación Andes and a FONDECYT Research Grant 90-0005.

0021-9991/93 $5.00

Copyright © 1993 by Academic Press, Inc.
All rights of reproduction in any form reserved.

strategy has the advantage that it is efficient (over a wide density range), memory saving, easy to understand, and easy to program.

In the basic cycle a time variable is associated to every particle in order to postpone particle state updates until it is absolutely necessary. Hence this is a *Delayed States Algorithm* (DSA). Minimizing the number of coordinate updates improves efficiency and reduces the rounding errors. The basic cycle can schematically be condensed to four operations: (a) picking the next event from the FEL, (b) updating—if necessary—the states of the particles involved in the current event, (c) calculating the new events for each particle involved in the current event, and (d) inserting these new events into the FEL.

The administration of the FEL has to be optimized, taking into consideration that the operations acting on the FEL have to pick up the next event, insert new events (*scheduling*) and erase (invalid) events that cannot happen. To be able to efficiently pick up the next event it is certainly convenient to make use of a binary tree [18] and this efficiency is further increased if one can reduce the number of accesses to the tree [19].

Our answer to these problems is the local minima algorithm (LMA) that we sketch in the following. For each particle i we build a singly linked list L_i [20] containing all the future events—that we denote by $\mathscr{E}_i(x)$, x being the other object involved in the event—that have been predicted for i. An event $\mathscr{E}_i(j)$ is in list L_i and not in list L_j. These $\mathscr{E}_i(x)$ are structured variables containing the necessary information associated to the specific event.

Each one of these N local lists has a *local minimum event*: the event with lesser time in L_i. These are the only events that enter the binary tree. The determination of the local minimum for disk i and its insertion in the binary tree is performed regardless of the existence of other possible events $\mathscr{E}_j(i)$ within the FEL.

Some local minima lose validity during the simulation because the partner particle suffers another collision first. This makes it necessary to carry a correcting operation to reestablish the condition that in the binary tree there be only valid local minima. We have been able to determine that such corrections occur infrequently during the simulation.

A drawback in some binary trees is the cost of keeping them balanced and of eliminating nodes [18]. We do not have these problems because we use a *complete binary tree* (CBT) [20]. Such a tree is balanced by definition and in our case it is never necessary to eliminate a node of the tree. Its structure is fixed on initialization. The base of the tree has N leaf nodes rigidly associated to each one of the N particles of the system.

At this point we have to clarify that only the labels i—that identify particles associated to the local minimal events $\mathscr{E}_i(x)$—enter the CBT and not the full structured variables. In this way, inserting and deleting is efficient because it is an operation over the lists L_i.

In this article we provide theoretical and experimental evidence supporting our strategy based on this mixed structure (local lists and a binary tree). We emphasize that the basic observations that justify our solution are that (a) the FEL necessarily contains many events that will never take place and (b) the most probable event associated to particle i that will happen next is the one with shorter time.

We will see that the efficiency of the LMA is close to optimum. In fact, the running time costs of picking the next event and scheduling n new events for each particle are $O(1)$ and $O(n-1+\log_2 N)$, respectively. That is, we have been able to reduce the number of accesses to the binary tree to roughly one access for every disk involved in each collision. We know of no other algorithm which can attain this performance.

It is important to emphasize that significant improvements have been made in the development of data structures and algorithms [18, 21] which are appropriate for general purpose FELs. These algorithms, however, are not suitable for the class of simulations that we are interested in because of their own sophistication, which introduces an unnecessary overhead.

The main point that must be considered to make efficient simulations of a general event-driven system are already present in the development of algorithms for the two-dimensional system of hard disks. For this reason and for the sake of simplicity our strategy is given in the context of a hard disk system. We have also chosen this kind of system due to the interest in event-driven molecular dynamics simulations for large systems (e.g., [4, 5, 7]). The system used in the simulations reported in this paper (with programs written in C-language) consisted of 2500 identical disks colliding elastically between each other and with the walls of a square box. There was no external field present. The initial conditions were set at random.

This paper can be thought of a divided into three parts. The proposed algorithms themselves are found in Sections 2 and 3. Empirical results are given in Section 4 to see how our full program works in different computers and also to make comparisons with other algorithms. In Section 5 we give our final comments. Appendix A is dedicated to analyze the performance of the local minima algorithm and it is shown that our strategy to postpone erasing invalid local minima is the best choice. Appendix B gives a notation and glossary table.

2. THE DELAYED STATES ALGORITHM (DSA)

In what follows we consider a two-dimensional system of N hard disks of diameter D in a rectangular box of $L_X \times L_Y$. The events by which the simulation proceeds—denoted as

$\mathscr{E}$—are either binary disk–disk collision (DDC events denoted $\mathscr{E}_i(j)$) or disk–wall collisions (DWC events denoted $\mathscr{E}_i(w)$). Letters i, j, k denote disks, and w denotes a wall. The reason to have an asymmetric notation $\mathscr{E}_i(j)$ is to emphasize that the events are calculated after a specific disk i has had a collision.

It is known [2, 13] that to improve the efficiency of the simulation it is convenient to divide the $L_X \times L_Y$ box into cells. Let us assume that it is possible to tile the box with $K_X \times K_Y$ identical square cells of size $\sigma \times \sigma$. The program keeps up a $K_X \times K_Y$ matrix $\mathscr{M}$ called the *cell–matrix*. This cell–matrix contains in its $\mathscr{M}_{ab}$ element the list of disks whose center is in the cell ab of the box. The *neighborhood* of a disk i currently in cell ab is the set of (usually nine) cells formed by the cell ab itself and its adjacent cells. In this way, choosing

$$\sigma > D, \tag{2.1}$$

a disk can only hit another disk in its current neighborhood. Since the disks are moving it is necessary to detect the instant when a disk crosses to an adjacent cell to include its possible collisions with objects (disks or walls) belonging to the (usually three) new cells that become part of the new neighborhood of that disk. Therefore a new type of event—called *virtual wall collision*—has to be incorporated: the crossing of a disk from one cell to another, denoted as VWC from now on.

Sometimes we will refer to *hard collisions* meaning events involving a disk and another real object (DDC or DWC events) to distinguish them from virtual wall collision events (VWC) which are artifacts of the algorithm.

The *state* s_i of every disk i can be defined by the position and velocity vectors,

$$s_i = (\mathbf{r}_i, \mathbf{v}_i). \tag{2.2}$$

To reduce the cost of every cycle of the simulation, however, it is possible to proceed as follows. At every cycle of the algorithm—when only one event occurs—only the disk(s) involved in that event is (are) *processed* (updating its (their) state(s) if the event is a hard collision and predicting and scheduling new events for the disk(s)), postponing any action on the states of the other disks. This makes it necessary to introduce a new variable to the state of every disk: the time τ_i when disk i was last updated. We use the symbol S_i to denote the complete state of disk i:

$$S_i \equiv (s_i, \tau_i). \tag{2.3}$$

To simulate chronologically the events it is necessary to know which is the next event. With this aim a *future events list* (FEL) is used. After getting and processing the next event from the FEL it is necessary to insert (schedule) new events in it.

The DSA makes use of the following functions:

- $T_W\ (S_i)$ returns the collision time predicted for disk i with a wall and the type of event (DWC or VWC).
- $T_D\ (S_i, S_j)$ returns the collision time predicted for the (i, j) disk–disk collision (DDC event).
- $F(s_i, t - t_0)$ the *free-evolution function* updates the state S_i of disk i from an instant t_0 to a later time t, knowing that i does not suffer any type of collision during the interval (t_0, t).
- $G_1(s_i, w)$ makes the instantaneous changes suffered by the state s_i of the disk i involved in a DWC.
- $G_2(s_i, s_j)$ makes the instantaneous changes suffered by the states s_i, s_j of the disks i, j involved in a DDC.
- Update $\mathscr{M}(i, w)$ moves the disk identifier i from one cell of the cell-matrix $\mathscr{M}$ to the new cell, after every VWC event.
- Neighborhood(event, i) gives the set of disks present in the neighborhood of disk i, taking into account the type of event (DDC, DWC, or VWC), the current cell of i and the position of the cell itself within the box. The reason why this function needs *event* as an argument is that when it is called after a VWC only the disks in the three new cells of the neighborhood need to be considered.
- NEXT_EVENT returns the next event to occur namely the event which has the lesser time in the FEL. This operation gives the type of event (a flag called event_type), the disk involved, the *partner* object x (disk or wall) and the time at which this event is scheduled to happen.
- SCHEDULE inserts a new event in the FEL. It takes four arguments: the type of the event (event_type flag) being inserted; the disk i involved; the partner object x (disk or wall); and the time at which this event will be scheduled to happen.

In this way, an efficient strategy to simulate the system consists of cycles formed by: (1) one NEXT_EVENT operation, followed by (2a) an updating of the state(s) of the disk(s) involved if the event was a hard collision (DDC or DWC) and (2b) updating the cell-matrix $\mathscr{M}$ if the event was a virtual wall collision (VWC), and (3) one or more SCHEDULE operations as in the Delayed States Algorithm in Fig. 2.1.

It is important to realize that the strategy so far defined yields the correct behavior of the system—provided the events are processed chronologically—in spite of the fact that at any given time the states of the individual disks are updated to different times. To obtain a snapshot of the system it suffices to apply the function F to bring all states to the current simulation time.

The floating point errors are reduced if the disk state

```
begin Initialization
   initialize all τ_i to zero
   initialize all states s_i
   build the cell-matrix M
   build the Future Events List
end Initialization.

MAINLOOP
   (event_type,i*,x*,t) ← NEXT_EVENT   /* x* labels a disk or a specific wall */
   Case event_type Of
      DDC:                             /* a disk-disk collision */
         for (k = i*,x*)  S_k ← F(s_k, t − τ_k)
         (s_i*, s_x*) ← G_2(s_i*, s_x*)
         for (k = i*,x*) PREDICTIONS(k, DDC)
      DWC:                             /* a hard wall collision */
         S_i* ← F(s_i*, t − τ_i*)
         s_i* ← G_1(s_i*)
         PREDICTIONS(i*, DWC)
      VWC:                             /* a virtual wall crossing */
         UpdateM(i*, x*)
         PREDICTIONS(i*, VWC)
   endCase
endMAINLOOP

PREDICTIONS(i, event)
begin
   (event_w, w, t) ← T_W (S_i)
   SCHEDULE( event_w, i, w, t)
   for (j ∈ Neighborhood(event, i) )
      t ← T_D (S_i, S_j)
      if (t < ∞) then SCHEDULE(DDC, i, j, t)
   endfor
end PREDICTIONS
```

FIG. 2.1. The Delayed States Algorithm.

variables are not actually updated until it is absolutely necessary. This happens whenever a disk undergoes a hard collision. In other words, if the next event is a VWC then the new events associated to the new cells of the neighborhood are calculated and added to the list of future events but the state of that disk is kept in its pre-VWC condition.

3. THE LOCAL MINIMA ALGORITHM (LMA)

In this section we describe our algorithm to administer the FEL. Let us define the *local minimum* associated to disk i as the event with lesser time among all the events $\mathscr{E}_i(x)$ scheduled for i. The Local Minima Algorithm is based on the intuitive idea that of all the events scheduled for a disk, the one with minimal time is the most probable one to occur. Therefore, *it suffices to compare the local minima* associated to every disk to be able to get the next event.

The operation described in the previous paragraph must be corrected if the next event is a DDC—$\mathscr{E}_i(j)$ say—which happens to be invalid. It will be invalid if the *partner* disk j has suffered a hard collision after $\mathscr{E}_i(j)$ was SCHEDULEd. Under such circumstances $\mathscr{E}_i(j)$ is dropped and the next local minimum for i is selected and compared with the other local minima. This corrective operation will be called RESCHEDULE and it is carried out as many times as necessary until a valid next event is selected. In Appendix A it is argued that such corrections do not occur very often.

The algorithm works with singly linked lists—L_i from now on—associated to every disk i, where the events $\mathscr{E}_i(j)$ scheduled for i are kept. A disk i may also appear in a $\mathscr{E}_k(i)$ which belongs only to L_k. Whenever i suffers a hard collision, L_i is cleared and new events are SCHEDULEd into it. On the contrary if i crosses a cell wall (virtual wall collision) new events are added to L_i. These linked lists L_i are useful to find the next local minimum for disk i when RESCHEDULing or after a virtual wall collision.

To detect the invalidated events, the algorithm uses an integer array—with components C_i—where the current value of the number of hard collisions for every disk i is kept. When scheduling new events, each DDC event $\mathscr{E}_i(j)$ stores a variable c that records the value of C_j associated to the partner disk j at the moment of its scheduling. The definition of a DDC event is

$$\mathscr{E}_i(j) \equiv (t, \text{event}, j, c), \tag{3.1}$$

where t is the time when the event is scheduled to occur.

To decide whether an event is valid, the current value C_j is compared with the value c kept in $\mathscr{E}_i(j)$. This comparison is used when deciding which is the next event and also when the local minimum of a list is being searched (it is convenient to drop the invalid events in L_i at this point).

The definition of a VWC or a DWC event has the form

$$\mathscr{E}_i(w) \equiv (t, \text{event}, w). \tag{3.2}$$

To compare efficiently the local minima we use a complete binary tree CBT [20], associating a disk identifier to each leaf (see Fig. 3.1a). The logic of the CBT is the same one often used in sports tournaments: to each competitor there is associated a fixed node at the base of the tree (a *leaf node*), the name of the winner of each individual match is transcribed to the parent node. The same rule is recursively applied going upwards in the tree. The absolute winner arrives at the root node which—in the case of the LMA—is the next event. For each disk the local minimal time is known. The only nodes that change their values dynamically are the internal nodes. The form of the CBT and the labels of the leaves remain unaltered.

Based on the local minimal times a tournament ($N-1$ matches) to reach the root decides the first event that starts the simulation. For each internal node a *match* (comparison of the local minimal times) is made between its children and the winner's identifier is transcribed to it (see Fig. 3.1b). The local minimal event associated to the disk i that reaches the root of the CBT is the first event.

After processing this event we have a new local minimum for i, forcing us to update the CBT. This is done performing new matches at every internal node in the path from leaf i to the root (as in Fig. 3.1c). If the event was a DDC, similar matches are made for the partner. This procedure determines the new event to be processed. From then on the CBT is updated after each new event.

Precise implementations for the operations NEXT_EVENT and SCHEDULE than make use of two new functions are given in Fig. 3.2:

- Min(i) returns the current local minimal event in L_i
- UpdateCBT(i) makes a sequential search of the event with minimal time in L_i and updates the necessary nodes of the CBT. During this search the invalid event in L_i are eliminated.

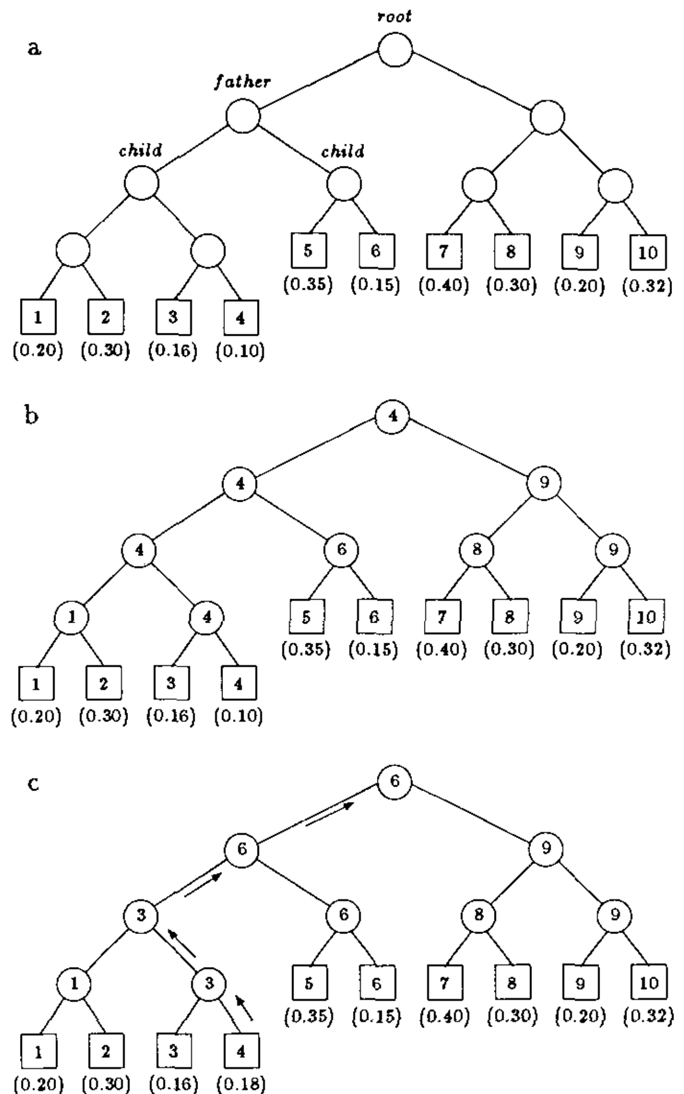

FIG. 3.1. (a) Structure of the CBT for 10 disks. The internal nodes are represented by circles and the leaves by squares. Each leaf is associated to a fixed disk. The local minimal times of every disk is written under each leaf. (b) Result of the tournament. Note that disk 4 has the lesser of the local minimal times. (c) The local minimal time of disk 4 has changed to 0.18 and new matches had to be played along the path from leaf 4 to the root as indicated by the arrows. Note that disk 6 has now the lesser local minimal time.

```
SCHEDULE(event, i, x, t )
begin
   if (event = CDD)
   then L_i ← L_i ∪ {(t, event, x, C_x) }
   else L_i ← L_i ∪ {(t, event, x) }
   endif
end SCHEDULE

NEXT_EVENT
begin
   UpdateCBT(i*)                   /* i*, x*, event_type describe the current event */
   if (event_type = CDD) then UpdateCBT(x*)

   repeat
      i ← RootCBT
      ℰ ← Min(i)
      if (ℰ.event = CDD) and (ℰ.c ≠ C_ℰ.x)  /* ℰ is invalid */
      then
         UpdateCBT(i)              /* Reschedule */
      else
         if (ℰ.event ≠ VWC)
         then
            L_i ← ∅
            C_i ← C_i + 1
            if (ℰ.event = CDD)
            then
               L_ℰ.x ← ∅
               C_ℰ.x ← C_ℰ.x + 1
            endif
         endif
      endif
   until ℰ is valid

   return (ℰ.event, i, ℰ.x, ℰ.t)       /*value returned by NEXT_EVENT */

end NEXT_EVENT
```

FIG. 3.2. The Local Minima Algorithm.

The CBT can be implemented using an array with $2N-1$ elements of type integer. The children of an event in position p are placed in the positions $2p$ and $2p+1$, so that the father of an event in the position q is at $\lfloor q/2 \rfloor$. The leaves of the CBT are between the positions N and $2N-1$ of this array. This fact allows for a quick access to start every CBT update.

Each element $\mathscr{E}_i(x)$ of a list L_i can be a *record* with one field for each one of the variables mentioned in (3.1). The fields j and c for the wall events $\mathscr{E}_i(w)$ can be used to keep the variable w and the current value C_i, respectively. Furthermore, each one of these structured variables $\mathscr{E}$ has to have a field with a pointer to the next member of L_i. Since every list L_i has one and only one wall event $\mathscr{E}_i(w)$ it is convenient to use these events as heads of the respective lists and—to have quick access to these heads—we place the wall events in an array of size N. The search for the local minimum in L_i starts at the head of the list.

The memory space to keep the DDC events is dynamically created and removed when necessary. It is convenient to define a subroutine to administer these memory blocks since soon after a block is liberated it will be used by

another DDC event. To make the comparisons in the CBT it is also convenient to have an array of size N such that its ith element contains a pointer to the local minimum of L_i.

4. EMPIRICAL RESULTS

In this section we give some empirical results that show the efficiency of the DSA–LMA strategy to perform event-driven molecular dynamics simulations. In all these simulations a system of $N = 2500$ disks was used as described in Section 2.

It should be noted that the size of the cells used in the simulation affects in an important way the efficiency of the algorithm. In Fig. 4.1 it can be seen that there is a cell size for which the running time of the simulation is optimum. In that figure m is the average number of disks per cell and ρ is the area density. The figure also shows, for example, that for $m = 3$ the running time almost doubles the optimum running time.

In Table I is the number of millions of collisions per CPU-hour that our DSA–LMA simulated in different computers. The computers used were: DG Aviion 200, DEC-5400 and SUN 690. All our code is in C language and the maximum speed optimization compiler option was used in every case. To measure the CPU time spent in each experiment the *clock* function from the standard C library was used. For each density and each machine the optimal value of m was used. The time taken by the initialization procedure prior to the basic cycle are not considered.

In the following we compare the CPU-time and memory requirements of the DSA–LMA and two programs—that we shall call A and B—that contain features of the strategies found in [2, 17], respectively. In every case the codification style has been the traditional one. Namely modular programs using functions having local parameters and variables. Our DSA–LMA will be called M.

TABLE I

Millions of Collisions per Hour of CPU Time in Three Different Computers with Different Densities

ρ	SUN	DEC	DG
0.05	07.30	3.58	3.13
0.10	09.56	4.56	3.94
0.20	12.54	5.80	5.22
0.30	15.10	6.94	5.81
0.40	15.86	7.57	5.86
0.50	16.07	7.41	5.60
0.60	15.86	7.08	5.30
0.70	14.49	6.30	4.81

It would be unfair to imply that we are directly comparing these author's strategies since that would necessarily require using their own programs. Still we are interested in providing a feeling as to how other strategies perform and consequently we compare the performance of A, B, and M in different machines.

In [2] the FEL is based on a binary search tree (BST) [20] and a basic cycle quite similar to the DSA, except that instead of using one simulation time for every disk—as we do—the author used one time for every cell. All the scheduled events are kept in the BST. To be able to erase the invalidated events associated to a disk i two doubly-linked lists are used. The first contains the $\mathscr{E}_i(x)$ and the other one contains the $\mathscr{E}_j(i)$. Program A has our DSA and [2]'s FEL.

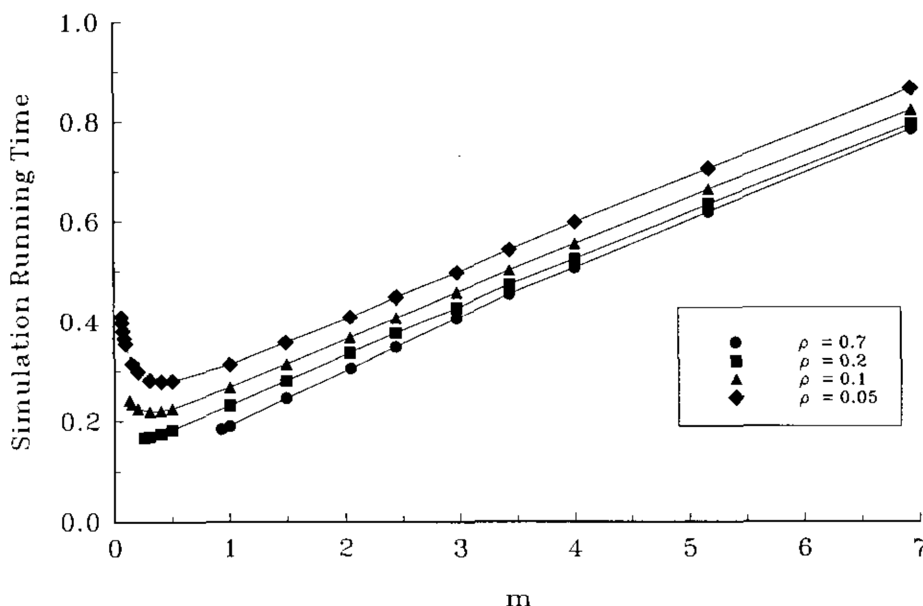

FIG. 4.1. Simulation running time—in arbitrary units—against the number m of disks per cell for different area densities.

In [17] only one event per disk is kept in the FEL. The FEL uses the binary structure of a heap [20]. The event $\mathscr{E}_i(j)$ inserted in the heap is the one with lesser time, provided that there is no $\mathscr{E}_j(x)$ event with a still smaller time in the FEL. It could be said that only *absolute minima* enter the FEL, while the rest of the predicted events are dropped altogether. This implies—among other things—that right after i crosses a virtual wall or its absolute minimum becomes invalid that it is necessary to calculate possible events with the disks in the whole neighborhood of i. Program B implements the strategy in [17].

The parameters to make the comparisons are: (a) the total CPU execution time ratios and (b) the ratio between the CPU times spent by the different FELs. For each program and each density the optimal value of m was used. The experimental points were obtained in runs of 10^5 collisions. Every measurement was performed five times, observing standard deviations of the execution times of less than 1%. There were no other important processes competing for the CPU.

For the DSA–LMA the time spent by the FEL was determined for each experiment measuring first—in separate runs—the cost associated to every function called by the LMA, using one million calls for different L_i sizes. During each experiment the total cost of the LMA was determined by counting the number of calls made by the LMA to the different functions and weighing each counter with the cost of the respective function.

For program A we measured the total simulation time and since the basic cycle time consumption is that of our DSA we derived the time spent by the FEL F_A by means of a simple subtraction. In program B the procedure was similar. Here the cost associated to recalculations was added to the FEL time F_B.

In Table II some experimental results are shown, comparing the performance of the different FELs. Column 2 (3) has the ratio between the times used by the FEL in A (B) over the time spent by the FEL in M.

In columns 4, 5, and 6 there are the amounts of memory (measured in Kbytes) required by the FELs used in M, A, and B, respectively. We associated eight bytes to the type *double* of the C language and four bytes for the types *int* and *pointer*. In B we took as part of the FEL's memory requirement the amount of memory needed to keep an additional copy of the states of the particles [17].

In Table III we compare the performance of programs A and B with M in different computers: DG Aviion 200, DEC-5400, and SUN 690. In columns 2, 4, and 6 there are the ratios between the total execution times of programs A and M. In columns 3, 5, and 7 there are the same for program B.

As we can see, program A loses efficiency as the density becomes greater. This happens because at higher densities the number of scheduled events is larger, increasing the number of accesses to the binary tree. The average number of scheduled events per disk after each event is shown in Table IV, column 2 (Sch). Column 5 (CBT) gives the relative weight of the CBT in the LMA. In this column is seen that the binary tree is the most costly component of the FEL despite the fact that the number of accesses to the CBT has been reduced to about unity per disk for every event processed in the basic cycle, as shown in Appendix A.

Program B, on the contrary, is more efficient at high densities. This is fundamentally due to the decrease of CPV events at higher densities. At low densities the large amount of CPV events downgrades the performance because too many recalculations are required. Column 3 (VWC) shows the fraction of virtual collisions over the total number of processes events.

These experiments show that program M keeps a good performance in all the density range studied. With the LMA we have reduced the relative cost of the FEL to below 23% of the overall simulation time (see column 4 (LMA),

TABLE II

The Ratio of the Time Spent by the Future Events Lists of Programs A and B over the Time Spent by the FEL in M

ρ	F_A/F_M	F_B/F_M	m_M	m_A	m_B
0.05	1.74	3.37	104.8	135.3	156.2
0.10	1.77	2.80	105.7	137.4	156.2
0.20	1.94	2.40	106.1	136.8	156.2
0.30	2.13	2.21	118.8	160.1	156.2
0.40	2.37	1.99	132.5	176.8	156.2
0.50	2.67	1.82	150.4	193.5	156.2
0.60	3.00	1.65	171.1	211.3	156.2
0.70	3.37	1.54	200.2	237.9	156.2

Note. Also the amount of memory (in Kbytes) required by the FELs used in M, A, and B. This table was obtained from simulations in a DG Aviion 200 workstation.

TABLE III

Ratio of the Total CPU Time of Programs A and B over the CPU Time of Program M in Different Computers

	SUN		DG		DEC	
ρ	A/M	B/M	A/M	B/M	A/M	B/M
0.01	1.05	1.58	1.12	1.68	1.09	1.69
0.05	1.00	1.44	1.16	1.53	1.07	1.62
0.10	1.08	1.39	1.18	1.41	1.08	1.46
0.20	1.09	1.28	1.21	1.31	1.14	1.28
0.30	1.11	1.25	1.24	1.25	1.18	1.16
0.40	1.14	1.18	1.26	1.19	1.21	1.15
0.50	1.21	1.15	1.29	1.14	1.28	1.13
0.60	1.35	1.13	1.33	1.11	1.37	1.11
0.70	1.32	1.08	1.37	1.08	1.41	1.06
0.89	1.59	1.09	1.44	1.05	1.63	1.02

TABLE IV

ρ	Sch	VWC	LMA	CBT
0.05	1.37	0.83	0.22	0.77
0.10	1.40	0.79	0.22	0.76
0.20	1.45	0.71	0.21	0.74
0.30	1.70	0.57	0.20	0.72
0.40	1.96	0.44	0.19	0.69
0.50	2.29	0.31	0.17	0.65
0.60	2.63	0.21	0.16	0.62
0.70	3.05	0.13	0.15	0.59

Note. The results in this table are explained in the text. They were obtained in a DG Aviion 200 workstation.

Table IV). Therefore further betterments in the FEL's performance would not imply a significant improvement of the overall program. To devise a significantly better program one would have to focus on other components of it.

5. FINAL COMMENTS

In this article we have presented and analyzed an efficient strategy to make event-driven simulations for systems of hard particles in one-processor machines. Our strategy compares positively with other algorithms and it can be used in simulations varying from dilute systems to near close-packing ones without any appreciable loss of efficiency in such extreme situations.

The algorithms and data structures were devised to (a) minimize the number of coordinate updates to increase efficiency and reduce the rounding errors, (b) conveniently adjust the size of the neighborhoods (by means of the cell matrix $\mathcal{M}$) to obtain the optimal running time, (c) keep the valid events to avoid recalculation, (d) lower the number of accesses to the binary tree to obtain the next event, and (e) reduce the cost of inserting and deleting events from the FEL, thanks to the use of the singly linked lists L_i.

A virtue of our strategy—that makes it a good alternative for interested physicists—is that it is straightforward. In particular, the singly linked lists and the CBT—implemented as an array of integers—are easy to program.

One-processor architectures allow to make event-driven simulations of relatively large systems in reasonable times e.g., in [3, 6]. Since this kind of simulations are strongly sequential, it is still not clear how to substantially improve the efficiency using multi-processor and/or vectorized architectures as discussed in [17, 22].

In multi-processor machines one cannot avoid the fact that the next-event has to be determined considering all the scheduled future events. In [17] the author gives details of a comparison with a parallel simulation which does not favor the parallel alternative. Furthermore, predicting new events for a particle with the objects in its neighborhood could be done in parallel. This gain would decreases the cost of the predictions in the DSA making the need of having an efficient FEL—as the one we have defined—still more important.

APPENDIX A: THE LMA PERFORMANCE

To measure the efficiency for the LMA we use the average number of event-time comparisons—that is, comparisons between the times of the scheduled events needed to organize the FEL.

Define L_{iV} as the set of DDC events $\mathscr{E}_k(i)$ for which i is the partner disk. In every list L_k there is at most one valid event involving disk i as the partner disk. When a disk i has a hard collision all the events in L_i and in L_{iV} are invalidated. Some of the invalidated events in L_{iV} are local minima and therefore they have already participated in the matches that take place to organize the CBT. Since they may produce a rescheduling sometime in the future, we say that *hard collisions generate possible future rescheduling*.

Let us consider the interval between two consecutive DDC events involving disk i. We want to know the number of comparisons that the LMA demands during that interval. Immediately after a hard collision, at a time we call t_0, a new list L_i of $L_i(t_0)$ events is associated with disk i. Afterwards, L_i starts to shorten (in the sense that some events in it are invalidated) until there is a VWC event, at time t_1, that causes new events to be scheduled into L_i. We call Δ the average net growth of a list L_i after each VWC event of i. After k virtual wall collisions the list length becomes $L_i(t_k) = L_i(t_0) + k\,\Delta$.

Leaving the cost of rescheduling aside, for every event that happens to i the LMA has to pick the event with minimum time in L_i at a cost of $L_i(t_k) - 1$ comparisons. Next, the CBT has to be updated at a cost of C_{CBT} comparisons. Hence if there are v VWC events between two consecutive hard collisions of disk i the average number of comparisons is given by

$$\sum_{k=0}^{v} (L_i(t_k) - 1 + C_{\mathrm{CBT}}) = (v+1)(L_D + C_{\mathrm{CBT}}), \quad \text{(A.1)}$$

where

$$L_D = L_i(t_0) + \frac{v}{2}\Delta - 1 \quad \text{(A.2)}$$

is the average length of the lists L_i, excluding the wall collision contained in every list.

To take into account the cost of rescheduling we have to consider that several reschedules may occur at any moment in the interval between two successive hard collisions of disk i. For each reschedule it is necessary to obtain the

next local minimum and update the CBT. Namely, the cost of rescheduling at a given time t ($t_{k-1} < t < t_k$) is $L_i(t) - 1 + C_{CBT}$. We assume that $L_i(t)$ can be taken to be the average value,

$$L_i(t) \approx \frac{1}{v+1} \sum_{k=0}^{v} L_i(t_k) = L_D + 1. \qquad (A.3)$$

Therefore the cost of the LMA, considering an average of r reschedules, is

$$C_{\text{LMA}} = (1 + v + r)(L_D + C_{\text{CBT}}). \qquad (A.4)$$

It is seen then that the efficiency of the LMA is conditioned by the rescheduling rate r. It will be shown that r is well below unity.

The worst case would be to have to update the CBT for *all* the events that come into L_i until i suffers a hard collision (this would be the event with maximal time!). Even if this worst case were to happen systematically we must bear in mind that still we are not updating the CBT for the events that become invalidated between two events of disk i. This worst case is, of course, rather unlikely to happen.

In the following we will show a relation for the expected value r of reschedules that take place between two successive hard collisions of a disk i.

Expected value for r. We are going to show that the number r of RESCHEDULEs that have a disk between two successive hard collisions is approximately equal to the number of future reschedules (η) generated on the average by any disk having a hard collision.

We assume that between two disk–disk collisions of a disk i each one of the $M-2$ disks that are in its neighborhood have a disk–disk collision as well. When a disk collides, an average of $\eta/(M-2)$ reschedules are generated for each one of the $M-2$ disks in the neighborhood. Hence a disk will have a total of $\eta(M-2)/(M-2)$ reschedules between two successive hard collisions, that is, $r \approx \eta$.

Expected value for η. When a disk i has a hard collision, all the events $\mathscr{E}_i(j)$ in L_i and all the events $\mathscr{E}_k(i)$ in L_{iV} become invalidated.

In any list L_k that we consider, the DDC events have the same probability P_d—and the VWC event has the probability P_V—of being local minimum, then $P_V + L_d P_d = 1$, where L_d is the number of DDC events that exist in a list L_k, including the event $\mathscr{E}_k(i)$ (i.e., $L_d \geq 1$). Note that L_d is not necessarily equal to L_D since each one of these numbers measure the number of DDC events in L_k and L_i in particular (and different) instants of the simulation.

Calling L_V the number of events invalidated in L_{iV}, we conclude that the expected number of times when $\mathscr{E}_k(i)$ is a local minimum in L_{iV} is given by

$$\eta = P_d L_V = (1 - P_V)(L_V/L_d). \qquad (A.5)$$

From the last expression it is seen that the value that controls the number of reschedules is the ratio L_V/L_d.

Experimentally we have observed that L_V/L_d is between 0.13 and 0.37 as seen in column 3 of Table A.I. This result shows that, in fact, the rescheduling rate r is well below unity. In this table we also show an experimental validation for expression (A.5). Column 4 gives the experimental values of the average number of reschedules per disk generated after a hard collision. Column 5 has the values of expression (A.5), obtained using the experimental values of P_V and L_V/L_d. Columns 6 and 7 are explained below.

Effective number of reschedules. Let us see that it is not convenient to process a reschedule as soon as it is generated. In fact, delaying this job as much as possible has the advantage that the number of reschedules is less than the value given by (A.5).

In our algorithm the total number of reschedules is dramatically reduced because we process the reschedules only when the invalidated $\mathscr{E}_j(k)$ reaches the root of the CBT and not as soon as it is invalidated. The reason is that during the lapse while the invalidated event $\mathscr{E}_j(k)$ migrates towards the root—caused by CBT updates—it is quite probable that another disk i will have a hard collision with j; namely, an event $\mathscr{E}_i(j)$ reaches the root, causing erasure of the invalidated event $\mathscr{E}_j(k)$ without having to reschedule it. We must remember that after each hard collision $\mathscr{E}_i(j)$, new lists L_i and L_j are built and matches for i and j take place in the CBT.

Column 6 (R_1) of Table A.I contains the fraction of the number of reschedules processed over the number of reschedules that are generated by hard collisions. It is seen that it is always important to postpone processing the

TABLE A.I

ρ	P_V	L_V/L_d	η[a]	η[b]	R_1	R_2
0.05	0.471	0.136	0.073	0.072	0.489	0.011
0.10	0.416	0.130	0.076	0.075	0.446	0.013
0.20	0.306	0.119	0.082	0.082	0.394	0.018
0.30	0.279	0.180	0.133	0.129	0.374	0.040
0.40	0.239	0.230	0.181	0.175	0.358	0.065
0.50	0.194	0.284	0.233	0.228	0.339	0.094
0.60	0.152	0.326	0.278	0.276	0.337	0.123
0.70	0.117	0.370	0.320	0.327	0.243	0.141

Note. The results in this table are explained in the text. They were obtained in a DG Aviion 200 workstation.

[a] Experimental value for η.

[b] η obtained through expression (A.5) using experimental values for P_V and L_V/L_d.

invalidated events. The efficiency gain is more significant as the density increases. Column 7 (R_2) shows the fraction of the number of reschedules processed over the total number of events that reach the root of the CBT during the simulation. The relative weight in the worst cases reaches about 14%.

The cost of updating the CBT. The cost of the CBT, C_{CBT}, can be better understood looking at Fig. 3.1c. In this figure we see that there are some leaf nodes at the lowest level, level K say, and the rest of the leaf nodes are one level up (level $K-1$). The cost (a) from level K is $\lfloor \log_2 N \rfloor + 1$ matches and (b) from level $K-1$ it is $\lfloor \log_2 N \rfloor$ matches. Furthermore, the number of leaves at level K is $2N - 2^{\lfloor \log_2 N \rfloor + 1}$ and at level $K-1$ it is $2^{\lfloor \log_2 N \rfloor + 1} - N$. Taking the average, the cost of the CBT is

$$C_{CBT} = \lfloor \log_2 N \rfloor + 2 - \frac{1}{N} 2^{\lfloor \log_2 N \rfloor + 1}. \qquad (A.6)$$

If N is an integer power of two then the minimal asymptotic cost $C_{CBT} = \log_2 N$ is reached. Hence we conclude that the average cost of the LMA for each event processed is close to $L_D + \log_2 N$.

APPENDIX B: NOTATION AND GLOSSARY

In the following list we summarize the main symbols and terms used throughout the paper. Some symbols exclusive of one section are not included here.

BST	Binary search tree
C_i	Current number of hard collisions of i
CBT	Complete binary tree
Cells	Subdivisions of the box
Cell-matrix	See under $\mathscr{M}$
D	Disk diameter
DDC	Disk–disk collision event
DWC	Disk-(hard) wall collision event
DSA	Delayed States Algorithm (Section 2)
$\mathscr{E}_i(j)$	Structured variable associated to a disk–disk collision
$\mathscr{E}_i(x)$	Structured variable associated to a disk-i–object x collision
$\mathscr{E}_i(w)$	Structured variable associated to a disk-(virtual or hard) wall collision
FEL	Future events list
Hard collision	Collision between real objects, either DDC or DWC events
Heap	Data structure based on a binary tree
i, j, k	Denote disks
Invalid event	Event $\mathscr{E}_i(j)$ scheduled into the FEL before j suffered a hard collision
K_X, K_Y	Number of cells in the X and Y directions
Leaf node	Node at the base of the CBT
L_i	The list of future events $\mathscr{E}_i(j)$ scheduled for disk i
LMA	Local Minima Algorithm (Section 3)
L_X, L_Y	Width and height of the box
Local minimum	Event with lesser time in a list L_i
L_{iV}	Set of events $\mathscr{E}_k(i)$ with i fixed
L_V	Average number of events $\mathscr{E}_k(i)$ invalidated in L_{iV} right after a hard collision
L_d	Average number of DDC events in the lists L_k containing an event $\mathscr{E}_k(i)$ (right after a $\mathscr{E}_i(x)$ event)
L_D	Average number of DDC events in any L_i
$\mathscr{M}$	Cell matrix of $K_X \times K_Y$ elements that keeps up the name of the disks currently present in every cell
m	Average number of disks in a cell
Match	Event time comparison between two local minima
N	Number of disks in the box
Neighborhood	Set of cells that surround a disk
Partner	Object x in an event $\mathscr{E}_i(x)$
P_d	Probability of a DDC in L_i being the local minimum
P_V	Probability for the VWC of a typical list L_i to be the local minimum
Reschedule	Search for a local minimum after a $\mathscr{E}_i(x)$ was found to be invalid
ρ	Area density: $\pi D^2 N/(4 L_X L_Y)$
Schedule	Insertion of a new event in a list L_i
S_i	State of disk i
σ	Linear size of every cell
T_D	Functions that predicts DDC events
T_W	Functions that predicts VWC and DWC events
τ_i	The last time that disk i participated in a hard collision
x	Any object, a disk, a hard or virual wall
Virtual wall	Border between two neighboring cells
VWC	Virtual wall crossing event
w	Denotes a wall

ACKNOWLEDGMENTS

One of us (M.M.) thanks R. Baeza-Yates for his guidance in topics related with data structures and algorithms. We also express our gratitude to P. Lira.

REFERENCES

1. B. J. Alder and T. E. Wainright, *J. Chem. Phys.* **27**, 1208 (1957).
2. D. C. Rapaport, *J. Comput. Phys.* **34**, 184 (1980).
3. M. Mareschal and E. Kestemont, *Phys. Rev. A* **30**, 1158 (1984).

4. E. Meilburg, *Phys. Fluids* **29**, 3107 (1986).
5. M. Mareschal and E. Kestemont, *J. Stat. Phys.* **48**, 1187 (1987).
6. D. C. Rapaport, *Phys. Rev. Lett.* **60**, 2480 (1988).
7. D. C. Rapaport, *Phys. Rev. Lett.* **43**, 7046 (1991).
8. B. D. Lubachevsky and F. H. Stillinger, *J. Stat. Phys.* **60**, 561 (1991); B. D. Lubachevsky, F. H. Stillinger, and E. N. Pinson, *J. Stat. Phys.* **64**, 501 (1991).
9. D. E. Risso and P. Cordero, in *Condensed Matter Theories*, Vol. 7, Mar del Plata, 1991, edited by A. N. Proto and J. Aliaga (Plenum, New York, 1991).
10. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford Science, London, 1990).
11. W. G. Hoover, *Molecular Dynamics* (Springer-Verlag, New York, 1986).
12. G. Ciccotti and W. G. Hoover (Eds.), *Molecular-Dynamics Simulation of Statistical-Mechanical Systems* (North-Holland, Amsterdam, 1986); G. Ciccotti, D. Fremkel, and I. R. McDonald (Eds.), *Simulation of Liquids and Solids* (North-Holland, Amsterdam, 1987).
13. B. J. Alder and T. E. Wainright, *J. Chem. Phys.* **31**, 459 (1959).
14. J. J. Erpenbeck and W. W. Wood, in *Statistical Mechanics B. Modern Theoretical Chemistry*, Vol. 1, edited by B. J. Berne, (Plenum, New York, 1977).
15. F. P. Wayman, *Comm. ACM* **18**, 350 (1975).
16. W. R. Franta and K. Maly, *Comm. ACM* **20**, 596 (1977).
17. B. D. Lubachevsky, *J. Comput. Phys.* **94**, 255 (1991).
18. D. W. Jones, *Comm. ACM* **29**, 300 (1986).
19. S. Carlsson, J. I. Munro, and P. V. Poblete, in *SWAT 88, Halmstad, 1988*, Vol. 1.
20. D. E. Knuth, *The Art of Computer Programming*, Vol. 3 (Addison–Wesley, Reading, MA, 1973).
21. G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures* (Addison–Wesley, Reading, MA, 1991).
22. D. C. Rapaport, "Thought on Vectorized Algorithms for Molecular Dynamics," in *Computer Simulation Studies in Condensed Matter Physics II*, edited by D. P. Landau, K. K. Mon, and H.-B. Schüster (Springer-Verlag, New York/Berlin, 1990).