JOURNAL OF COMPUTATIONAL PHYSICS 34, 184–201 (1980)

The Event Scheduling Problem in Molecular Dynamic Simulation

D. C. RAPAPORT*

Physics Department, Bar-Ilan University, Ramat-Gan, Israel

Received July 10, 1978; revised March 27, 1979

Molecular dynamics simulation studies of hard sphere and related many-body systems tend to be heavy consumers of computer time. In order to perform event-driven simulations of this kind in an efficient manner both the organization of the event list data structure and the procedures for modifying its contents must be carefully designed. A highly efficient event list structure based on the binary tree is proposed, and the algorithms for performing the event scheduling and related operations are described. Numerical analysis of the performance of these algorithms reveals that they behave as if the event list were constructed from randomly occurring events. Other considerations involved in improving the simulation performance are discussed.

1. INTRODUCTION

The techniques of molecular dynamics computer simulation have, in recent years, come to be regarded as essential tools for investigating the behavior of classical many-body systems. The applications of these techniques cover a wide variety of problems in solid- and liquid-state physics; recent reviews of the subject are to be found in Refs. [1, 2].

Two distinct approaches to the simulation problem have been adopted. The earlier method, due to Alder and Wainwright [3], is designed for use with model systems whose interparticle interactions involve only step potentials. This kind of simulation is "event-driven," in the sense that the particles move with constant velocities except for the moment when any pair of particles experience an impulse interaction due to their having reached a potential step, for example, when two hard spheres collide, or when two spheres enter or leave their mutual square well. The time increments by which the simulation proceeds are the intervals between collisions, and these are determined by the system itself.

The alternative method, introduced by Rahman [4], is suitable for systems with smooth (differentiable) potentials, and requires solving the coupled differential equations of motion using numerical finite difference techniques. In this approach the simulation normally advances in fixed time steps and, consequently, the presence of truncation errors must be taken into account. The two simulation methods yield

* Present address: Baker Laboratory, Department of Chemistry, Cornell University, Ithaca, N. Y. 14853.

0021-9991/80/020184–18$02.00/0

results of comparable accuracy when applied to related systems such as the square well and Lennard-Jones fluids, but no comparison of their relative computational efficiencies—defined as the ratio simulated time/computation time—has been undertaken to date. In this article we will be concerned only with event-driven simulations.

In a recent study of polymers constructed from chains of linked hard spheres [5] it became apparent that at the core of the computer program performing the simulation lie the computations necessary for maintaining the list of scheduled events. Even when handled in an efficient manner, these computations consume a good 30 % of the total processing time; thus the search for an optimal method is clearly justified, especially in view of the fact that the computational requirements for typical molecular dynamics calculations tend to be measured in hours, even on the fastest of computers.

The purpose of this article is to describe a method for molecular dynamics simulation, with particular emphasis on handling the event scheduling problem. In Section 2 we discuss the important aspects of a hard sphere fluid simulation; in practice the method is often applied to more complicated systems, the sole proviso being that the potential be steplike in nature. Section 3 reviews several event list data structures and then proceeds to a detailed discussion of the structure eventually adopted. The algorithms employed in manipulating the event list are presented in Section 4. Numerical analysis of the algorithm performance and the factors affecting the overall efficiency of the simulation are discussed in Section 5.

2. Hard Sphere Fluid Simulation

In this section we review the principal features of the molecular dynamics simulation of a hard sphere fluid, with the exception of the problem of managing the event list data structure, which will be discussed in subsequent sections. The manner in which the results of the simulation are used to estimate the thermodynamic and transport properties is not described here; an extensive review of this subject is to be found in [2]. Only the simplest hard sphere system is considered; the extension to more complex systems (e.g., square well fluids [3]) is straightforward.

We consider a cubic of edge R containing N_A hard sphere atoms of diameter σ. In order to reduce the effect of finite system size periodic boundary conditions are employed. Since there are no long-range interatomic forces present the atoms follow linear trajectories, but when two atoms i and j approach to a distance σ they collide elastically, with velocity changes

$$\Delta \mathbf{v}_i = -\Delta \mathbf{v}_j = -(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})/|\, \mathbf{r}_{ij} \,|^2,$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ and $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$ are the relative coordinates and velocity just prior to the collision. As a consequence of the periodic boundary conditions, if atoms i and j are close to opposite boundaries of the region, collisions between i and the appropriate periodic image of j (or vice versa) can also occur.

At every moment in time the details of the next collision of each atom must be

known in order for the simulation to proceed correctly. The atoms move with constant speed between collisions and, if at time t atoms i and j have coordinates $\mathbf{r}_i$ and $\mathbf{r}_j$, they will either never collide, or a mutual collision will ocur at time $t + \tau_{ij}$, where τ_{ij} is the solution of $| \mathbf{r}_{ij} + \mathbf{v}_{ij}\tau_{ij} | = \sigma$, namely,

$$\tau_{ij} = -\{b_{ij} + [b_{ij}^2 - v_{ij}^2(r_{ij}^2 - \sigma^2)]^{1/2}\}/v_{ij}^2,$$

where $b_{ij} = \mathbf{r}_{ij} \cdot \mathbf{v}_{ij}$. Clearly for a collision to be possible $b_{ij} < 0$ and $b_{ij}^2 \geqslant v_{ij}^2(r_{ij}^2 - \sigma^2)$. Here once again the periodic boundary conditions require the determination of whether it is the atoms themselves or their periodic images which are involved in each collision.

In order to determine the possible future collisions of a particular atom each of the $N_A - 1$ other atoms of the system, and their images, must be regarded as potential candidates. This means that the computation time required to follow the evolution over a given interval of simulated time is $O(N_A^2)$. Since N_A is typically 100–1000 and sometimes even larger, and the computation of each τ_{ij} is nontrivial, this simple approach is clearly unworkable; a method of reducing the number of candidate atoms to a value independent of N_A is required.

The answer is to divide the region into L_c^3 cells whose edge length R/L_c exceeds the diameter σ; collisions are then possible only between atoms in the same and immediately adjacent cells. The periodic boundary conditions are incorporated by extending the term "adjacent" to include appropriate pairs of cells abutting opposite boundaries. Since the mean free path between collisions is normally much less than R it is clear that the next collision of a particular atom will generally be with an occupant of the same or a neighboring cell, and only these atoms need be examined. Nevertheless, in the event that an atom travels a considerable distance before colliding, keeping track of which cell the atom currently occupies will ensure that at some stage prior to the collision the collision partner will be found in one of the adjacent cells. This requires that in addition to determining future collisions it is also necessary to know when the next cell crossing is to occur, and at each cell crossing to examine possible future collisions with atoms in the newly adjacent cells. Following a collision the next cell crossing of each of the atoms involved must of course be redetermined.

The computations which utilize the cells do this by referring to membership lists of the atoms belonging to each cell. These lists can be constructed as in [2] or, alternatively, using a linked list approach as is the case in these simulations: corresponding to each cell there is a pointer into a linked list connecting the atoms belonging to that cell (a linked list is one in which the data are not stored in sequential order, but with each datum is associated a pointer to its successor—see Section 3). A further substantial reduction in computing time derives from reorganizing the task of coordinate updating. When two atoms collide the need to update all of the N_A atomic coordinates can be avoided by assigning a local time variable to each cell, and only updating the coordinates of the atoms belonging to the same cell(s) as the colliding atoms. Of course the different local times must be taken into account when predicting future collisions, but the result is that the time to process a single collision is now indepen-

dent of N_A. An alternative scheme for economizing the coordinate updating is proposed in [2].

3. Event List Structure

The data structure employed to hold the event notices for scheduled collisions and cell crossings, etc.—the event list—should be designed to satisfy several criteria: It must be ordered with respect to the time of occurrence of the events; addition to and deletion of event notices from the list—corresponding to the scheduling and canceling of events—should be performed efficiently by simple algorithms; and there should be no storage wasted, in the sense that when an event notice is deleted the space freed must be reusable by a subsequently scheduled event (the familiar "garbage collection" problem).

The simplest structure is a linear list. However, if consecutive event notices are stored as physically adjacent records in a table, each addition or deletion will require shifting an average of half the table contents. A more useful structure is the linked list, in which the physical order of the event notices is of no significance, but a pointer included with each notice defines a time-ordered path through the structure. If the linked list contains entries for N scheduled events the scheduling of an additional event will require an average of $N/2$ search steps to locate the correct insertion position. (If the list were stored in sequential order, rather than in linked form, a binary search would supply the insertion position in $O(\log_2 N)$ steps, but subsequent shifting of the notices still requires $O(N)$ movements.)

Since the nature of the simulation is such that the number of scheduled events exceeds the number of atoms in the system, structures of the type just described, which require $O(N)$ operations to insert a new notice, are clearly not suitable for use in a large molecular dynamics simulation. An enhanced version of the linked list is available which can, in principle, perform insertion in $O(N^{1/2})$ steps. The improvement stems from the use of an ancillary pointer list which provides more rapid access to the required location in the main event list [6, 7]. An initial search of the pointer list, whose length is chosen to be $O(N^{1/2})$, provides the starting point for a sequential search through a segment of the event list whose length is also $O(N^{1/2})$. However, the ancillary pointers must be fairly uniformly distributed over the event list to ensure $O(N^{1/2})$ behavior and, because the event list has a high turnover rate, the redistribution of the pointers may result in considerable overhead. Neither of the above references analyzes this problem in detail; in fact the simulated results shown in [7] suggest a weak $O(N)$ dependence. An alternative approach which is also expected to show $O(N^{1/2})$ behavior is discussed in Section 5.

An entirely different, and not immediately obvious, structure on which to base the event list is the binary tree. This data structure consists of a set of nodes—one per event notice—each of which contains the details of an event and when it is due to occur, but in addition includes links to two successor nodes (Fig. 1). The tree representation of the event list may be formally defined in a recursive manner: If a particular

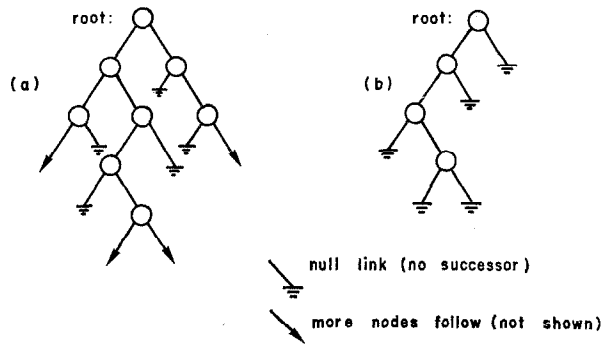

FIG. 1. Binary trees. The circles denote nodes and the line segments links; (a) shows part of a typical tree, (b) a tree that has degenerated into a linear list.

node represents an event due to occur at time t_e, then the left and right successor nodes correspond to events occurring at times $t_l \leqslant t_e$ and $t_r > t_e$. Note that one or even both of the pointers may be null if the corresponding successor node is absent. An extremely through discussion of trees is to be found in [8].

The tree representation of a particular set of event notices is by no means unique and depends on the order in which the notices were added to the tree. The formal definition given above does however provide the basis for defining the algorithms which are used to add and delete notices. If an N-node binary tree is balanced, in the sense that the distances from the root (the topmost node) to the extremities of all the branches are about the same, then the search for the correct insertion location will entail testing $\log_2 N (=1.39 \ln N) + O(1)$ nodes. In the unlikely case of a tree which has degenerated into a quasi-linear form, as many as N nodes may have to be tested. Justification for using a tree structure stems from the fact that it can be proved [9] that if the scheduled event times are random then the average number of tests during insertion is $2 \ln N + O(1)$, a value not too far from optimal. It turns out that in molecular dynamics simulations the scheduled collision times tend to resemble a uniform random number distribution—this is entirely reasonable since, over the entire system, there is very little correlation between collisions on a short time scale. Numerical analysis (Section 5) does indeed confirm that the tree retains the desired properties.

The simplest form of binary tree event list requires, in addition to the two pointers to successor nodes, *lnode* and *rnode*, that each node also include the scheduled occurrence time of the event, the quantity *eventtime*. There are also two data items containing details of the event denoted by *atype* and *btype*; if the event is a collision, these identify the atoms involved; if a cell crossing, then *atype* is the atom concerned and *btype* indicates which of six cell faces is crossed. Other information, such as the collision type in the case of a square well system, or the identification of an event in which some measurement is to be made, can readily be included.

The addition of further pointers to the nodes is required to adapt the basic tree structure for efficient molecular dynamics use. Node deletion (i.e., event cancellation) will prove to be a frequently performed operation; to facilitate alteration of the links

between nodes when a node is deleted, an additional pointer, *pnode*, to the node's predecessor is defined (Fig. 2). An additional quartet of pointers is added for the following reason. When atoms i and j collide all nodes involving i or j must be removed from the tree. The alternative to having to undertake an exhaustive search of the tree to locate these nodes is to link together all nodes which refer to events invol-

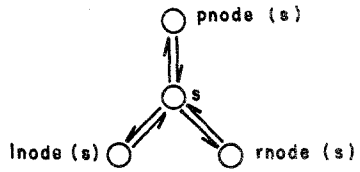

FIG. 2. The three link pointers connecting the nodes of the tree.

ving a particular atom. In practice, the collision event nodes involving atom i are divided into two linked lists according to whether i appears in the *atype* or *btype* position, and the connection between the nodes is implemented by means of a doubly linked circular list (Fig. 3) [10]. Each collision node thus belongs to two circular lists, labeled

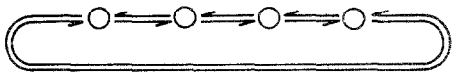

FIG. 3. Doubly linked circular list.

"*a*" and "*b*," and since the double linking requires a pair of pointers for each list, a total of four pointers per node are needed. The circular lists are not ordered in any way, and the manipulation of these pointers, referred to as *alnode*, *arnode*, *blnode* and *brnode*, is covered in Section 4. Figure 4 summarizes the complete structure of a single node.

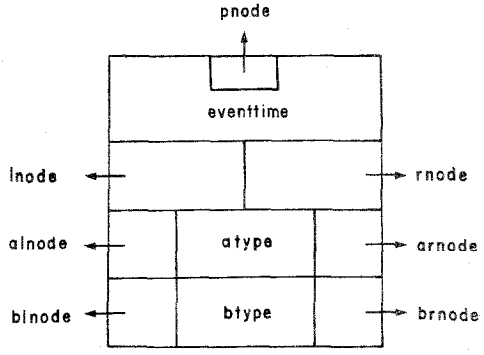

FIG. 4. Schematic form of the complete node used in the molecular dynamics simulations. The individual elements are described in the text. (This form is used in Fig. 5.)

The cell crossing event node is of similar form, but only a single atom is involved. Since there is always exactly one cell crossing event scheduled for each atom i, the four pointers *alnode*, etc., in the node are used to complete the two circular lists involving i.

At the start of the simulation the nodes are placed in a normal linked list—the pool—and are withdrawn as required and linked into the tree. When a node is no longer required it is returned to the pool. A typical event tree for a very small system ($N_A = 7$) is shown in Fig. 5. To simplify the algorithms, node zero serves as a fixed pointer into the remainder of the tree, and is not used to represent an event. Nodes 1 through N_A contain the cell crossing events for the corresponding atoms, and nodes

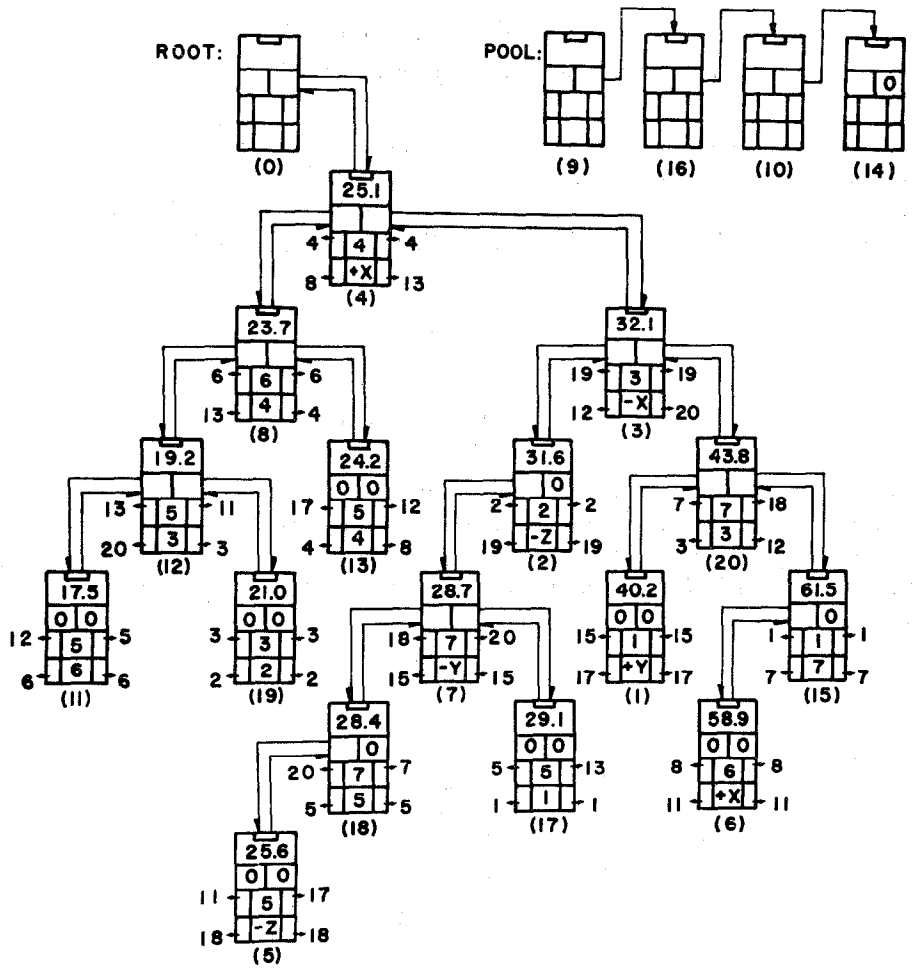

FIG. 5. The complete event list for a small hard sphere system (note that in the computer the tree is stored as a linear array). The numbers in parentheses are the node numbers. The circular list links are not drawn in full for reasons of clarity, but are represented by $\rightarrow n$ (indicating a link to node n). The next event to occur is at node 11.

$N_A + 1$ onward are used for collisions and other events. Note that if there are no collisions scheduled in which atom i appears in the *atype* position, then the "*a*" list is empty and the *alnode* and *arnode* pointers of node i both point to node i itself; likewise for an empty "*b*" list.

4. Event List Algorithms

During the molecular dynamics simulation the event list is being continually referenced and modified. In this section we describe the algorithms used to support the binary tree event list and the associated circular lists introduced in Section 3.

The two primary tasks involving the event list are the scheduling of a future event, and determining the next event due. Each of these tasks is described below as a procedure; each in turn calls on other procedures to perform operations such as canceling event notices, supplying vacant nodes, etc. Specifically, the event scheduling procedure SCHEDULE uses GETNODE to obtain a free node from the pool, and ADDTREE to insert the node at the correct location in the tree; SCHEDULE itself handles the circular list updating needed when a node is added. The next-event procedure NEXTEVENT first searches for the earliest event notice in the tree; if this turns out to be a collision CLEARCLIST is called to cancel future events involving the colliding atoms; and the now current event notice is deleted from the tree. CLEARCLIST scans a given circular list and deletes all the nodes through which it passes, using RETURNNODE to restore the nodes to the pool. DELETETREE relinks the tree pointers when a node is deleted.

The algorithms are presented using a simple pseudolanguage which employs the constructs "**begin** ⋯ **end**" to delimit procedures, "**if** ⋯ **then** ⋯ **else** ⋯" for conditional execution, and "**while** ⋯ **do** ⋯ **endwhile**" for iteration. Comments take the form /* ⋯ */. Procedure names are in capitals for clarity, the names of variables are in italics. Details of argument passing between procedures have been omitted since they are very much language dependent.

```
procedure SCHEDULE                  /* schedule event; arguments supplied include
                                       scheduled time, event type, and atom(s)
                                       involved; only collisions and cell crossings
                                       are treated here, but other event types are
                                       easily incorporated */
begin
if eventtype ≠ cellcrossing
then GETNODE (e)                    /* e denotes the new node */
else e ← bodynuma                   /* cell crossing event uses node with same
                                       serial number as atom (bodynuma) */
atype (e) ← facenum                 /* facenum is a code for the cell face to be
                                       crossed; facenum > nbody (the number of
                                       atoms) */
```

```
eventtime (e) ← scheduletime
ADDTREE (e)
if eventtype = collision
then atype (e) ← bodynuma          /* atoms are numbered bodynuma and
                                      bodynumb */
  btype (e) ← bodynumb
  /* update circular lists containing atoms */
  alnode (e) ← bodynuma, arnode (e) ← arnode (bodynuma)
  alnode (arnode (bodynuma)) ← e, arnode (bodynuma) ← e
  blnode (e) ← bodynumb, brnode (e) ← brnode (bodynumb)
  blnode (brnode (bodynumb)) ← e, brnode (bodynumb) ← e
end
procedure GETNODE (n)              /* get node from pool */
begin
if pool ≠ 0
then n ← pool, pool ← rnode (pool)
else poolempty ← 'true'            /* pool empty; terminate run */
end
procedure ADDTREE (n)              /* add node to tree */
begin
q ← 0                              /* search for insertion position */
if rnode (q) = 0
then rnode (q) ← n
else q ← rnode (q), found ← 0
  while found = 0 do               /* found = 1 indicates search completed */
    if scheduletime ⩽ eventtime (q)
    then if lnode (q) ≠ 0
      then q ← lnode (q)           /* branch left */
      else lnode (q) ← n, found ← 1
    else if rnode (q) ≠ 0
      then q ← rnode (q)           /* branch right */
      else rnode (q) ← n, found ← 1
  endwhile
lnode (n) ← 0, rnode (n) ← 0,      /* set tree links */
pnode (n) ← q
end
procedure NEXTEVENT                /* advance simulation to next event */
begin
e ← rnode (0)                      /* find earliest event; tree assumed nonempty */
while lnode (e) ≠ 0 do             /* descend leftmost branch of tree */
  e ← lnode (e), endwhile
timenow ← eventtime (e)            /* e denotes earliest event node */
bodynuma ← atype (e)
if btype (e) ⩽ nbody               /* what kind of event is it? */
```

```
then eventtype ← collision                /* collision */
  bodynumb ← btype (e)
  CLEARCLIST (bodynuma)                   /* cancel all events involving these atoms */
  CLEARCLIST (bodynumb)
else eventtype ← cellcrossing             /* cell crossing */
  facenum ← btype (e)
  DELETETREE (e)
end
procedure RETURNNODE (d)                  /* return node to pool */
begin
rnode (d) ← pool, pool ← d
end
procedure CLEARCLIST (i)                  /* remove all collision event nodes
                                             involving atom i by traversing both
                                             circular lists belonging to i */
begin
d ← i                                     /* start at cell crossing node and
                                             traverse 'a' circular list */
while alnode (d) ≠ i do
  d ← alnode (d)
  brnode (blnode (d)) ← brnode (d)        /* relink 'b' pointers */
  blnode (brnode (d)) ← blnode (d)
  DELETETREE (d), RETURNNODE (d), endwhile
alnode (i) ← i, arnode (i) ← i            /* 'a' list now empty */
d ← i                                     /* start at cell crossing node and traverse
                                             'b' circular list */
while blnode (d) ≠ i do
  d ← blnode (d)
  arnode (alnode (d)) ← arnode (d)        /* relink 'a' pointers */
  alnode (arnode (d)) ← alnode (d)
  DELETETREE (d), RETURNNODE (d), endwhile
blnode (i) ← i, brnode (i) ← i            /* 'b' list now empty */
DELETETREE (i)                            /* cancel cell crossing */
end
procedure DELETETREE (d)                  /* delete node from tree */
begin                                     /* search for node following d in time
                                             order; this node is to be linked
                                             directly to pnode (d) when d is deleted;
                                             the various cases which can arise are
                                             shown in Fig. 6 */
if rnode (d) = 0
then s ← lnode (d)                        /* case (i); lnode (d) may
                                             also be null */
```

```
else if lnode (d) = 0
  then s ← rnode (d)                    /* case (ii) */
    else if lnode (rnode (d)) = 0
      then s ← rnode (d)                /* case (iii) */
      else s ← lnode (rnode (d))        /* case (iv) */
        while lnode (s) ≠ 0 do
          s ← lnode (s), endwhile
        pnode (rnode (s)) ← pnode (s)   /* relink */
        lnode (pnode (s)) ← rnode (s)
        pnode (rnode (d)) ← s
        rnode (s) ← rnode (d)
      pnode (lnode (d)) ← s, lnode (s) ← lnode (d)
p ← pnode (d), pnode (s) ← p
if lnode (p) = d
then lnode (p) ← s
else rnode (p) ← s
end
```

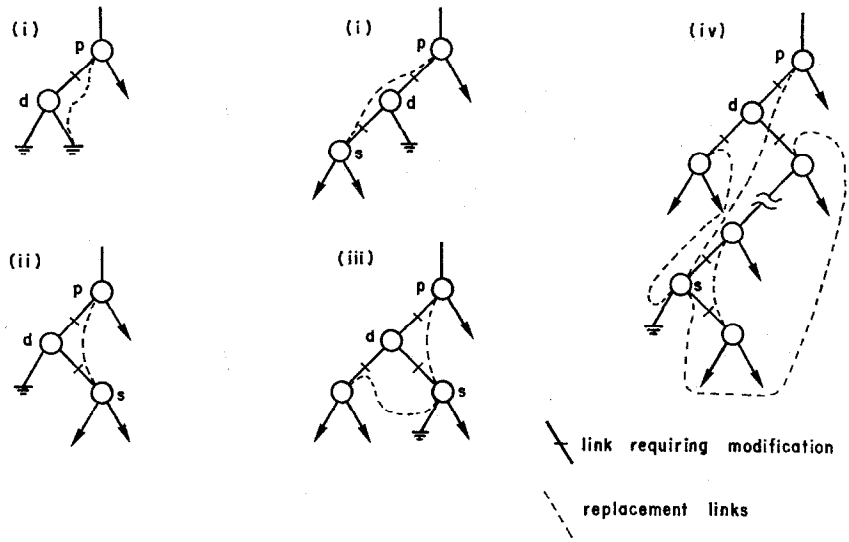

FIG. 6. Node deletion—the various cases arising in DELETETREE are shown. Nodes labeled d, p, s are the deleted node, its predecessor, and the node immediately succeeding it in the time order.

5. NUMERICAL RESULTS

In order to assess the overall efficiency of the molecular dynamics technique a series of numerical studies have been undertaken. While the main purpose of the analysis is to examine the performance of the algorithms for manipulating the tree

structured event list, results have also been obtained relating the computational efficiency to the number of atoms, the reduced volume, and the number of cells into which the region is partitioned.

The systems studied consisted of $N_A = 108$, 256, and 500 hard sphere atoms, at several reduced volumes. Each time a collision occurred during the simulation a record was made of the number of atoms tested (per atom) while searching for possible future collisions with each of the two colliding atoms (ν^{T}_{coll}) and, of these, the number of collisions which could actually occur and which had to be scheduled (ν^{S}_{coll}). Likewise, following a cell crossing, the numbers of atoms in the newly adjacent cells tested for future collisions (ν^{T}_{cross}) and collisions scheduled (ν^{S}_{cross}) were also recorded. In addition, the numbers of search steps required for insertion and deletion of event notices, the actual number of nodes in the tree, etc., were noted. These results were averaged at regular intervals; for intervals spanning several thousand collisions the averaged results showed no significant variation between intervals.

Results typical of those obtained prior to the averaging are shown in Figs. 7 and 8. The histograms in Fig. 7 show the distributions of the quantities ν^{T}_{coll}, ν^{S}_{coll}, ν^{T}_{cross}, and ν^{S}_{cross} obtained during one of the simulations. Figure 8 shows the distributions of the numbers of search steps required for insertion of a node (S_i) when scheduling an event, for determining the next event to occur (S_n), and for the relinking of pointers following the deletion of a node (S_d) corresponding to a canceled event. The definition of a search step in each of the three instances is the following: For insertion it is a single performance of the test "**if** *scheduletime* $\leqslant$ *eventtime* (q)" (ADDTREE). When searching for the next event it is a single iteration of the loop "**while** *lnode* $(e) \neq 0 \cdots$

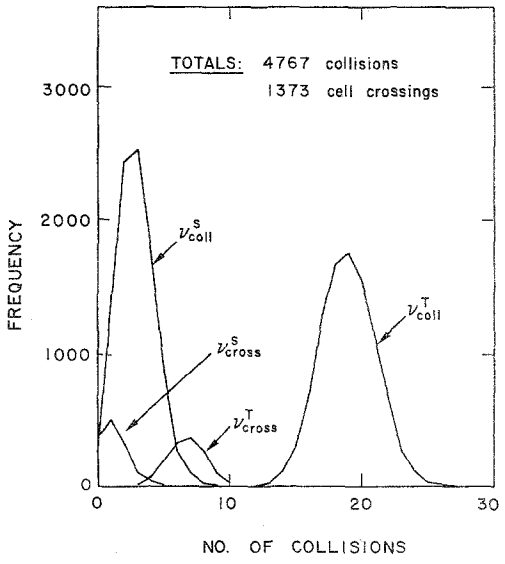

FIG. 7. Histograms showing the number of possible collisions tested and scheduled ($N_A = 256$, reduced volume $v = 2.0$).

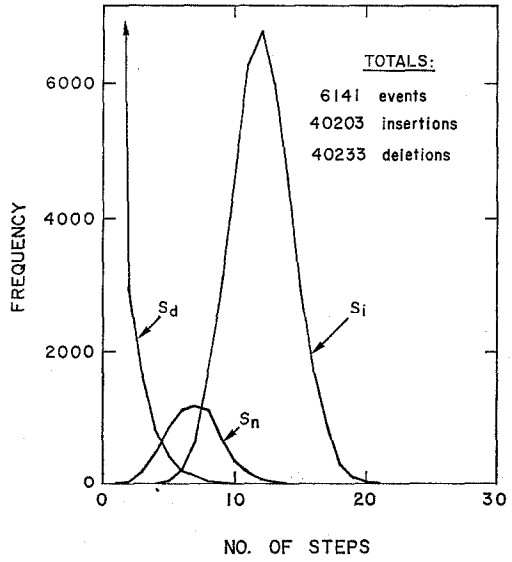

FIG. 8. Histograms showing the number of search steps (defined in the text) for the various event list operations ($N_A = 256$, $v = 2.0$).

endwhile" (NEXTEVENT). In the case of node deletion (DELETETREE), a count $S_d = 1$ is used to signify that one of the tests "**if** *rnode* $(d) = 0$" or "**if** *lnode* $(d) = 0$" was true; $S_d = 2$ indicates that the test "**if** *lnode* (*rnode* $(d)) = 0$" was true; and finally $S_d > 2$ means that the test "**while** *lnode* $(s) \neq 0$" in the search loop was repeated $S_d - 2$ times.

A graph showing the average number of search steps required for node insertion

solid line represents the result of a least-squares fit to the data points; it has the equation $\bar{S}_i = 1.56 \ln \bar{N} + 2.00$. The theory of random binary trees [9] predicts that the average number of search steps is $\bar{S}_i = 2 H_{N+1} - 2$, where H_n denotes the nth harmonic number. For $n \gg 1$, $H_n \sim \ln n + \gamma + O(n^{-1})$, where $\gamma = 0.577...$ is the Euler constant. Thus, for a random tree, the asymptotic behavior is given by $\bar{S}_i \sim 2 \ln N - 0.85$, and this is represented by the dashed line in Fig. 9. It is clear from the graph that, over the range of tree sizes encountered here, there is little significant difference between the observed N-dependence of $\bar{S}_i$ and that predicted theoretically for random trees. There is obviously no danger of the tree structured event list degenerating into a quasi-linear structure for which insertion becomes an $O(N)$ step process.

A second quantity for which a theoretical result is available is the expectation value of $\bar{S}_d - 2$, the number of times the search loop of the node deletion algorithm must be executed. In principle this value could show an $O(\ln N)$ dependence, in which case the relinking of nodes following a deletion would require considerable computation. However, the theory predicts $\bar{S}_d - 2 < 0.5$ independent of N [9], and the measured

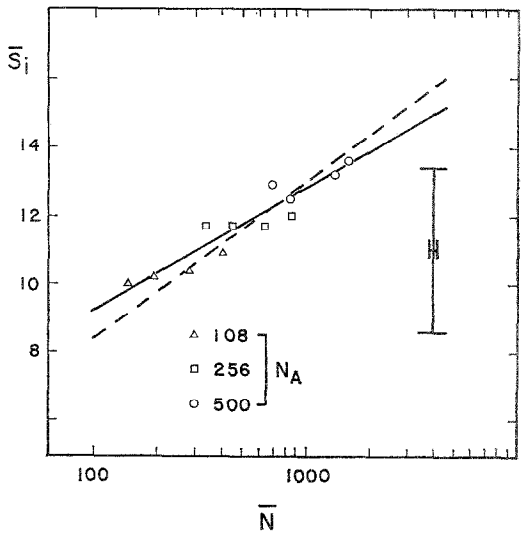

FIG. 9. Mean number of insertion search steps $\bar{S}_i$ vs mean tree size $\bar{N}$. The solid and dashed lines are the least-squares and theoretical results, respectively. Typical errors are indicated. The data correspond to the systems in Table I, and for given N_A, v increases to the left.

value during the simulation was close to 0.3, so node deletion is not a problem. A typical distribution of S_d is included in Fig. 8.

It is clearly of interest to ask how the tree structured event list compares with other possible organizational schemes. Unfortunately there is an almost total lack of information concerning the techniques used in other molecular dynamics studies and their performance. In a recent review of hard sphere molecular dynamics [2] the techniques described are based on those used in the original computations [3] together with certain enhancements, namely, the division of the region into cells, and a means for reducing the computational effort by postponing the position updating (unlike the method described in Section 2, this does not assign a local time variable to each cell). No performance figures are given, however.

An outline of the method employed by Alder and co-workers (unpublished), in which the computations related to the event list have an $O(N^{1/2})$ complexity, was kindly provided by an anonymous referee. The event notices are stored in order of insertion rather than in chronological order; the effort of determining which event is to occur next is reduced from an $O(N)$ step process to one of $O(N^{1/2})$ steps by maintaining an auxiliary list of $N^{1/2}$ soon-to-occur events which are extracted from the main event list at intervals of $N^{1/2}$ events. Furthermore, a canceled event notice is not removed from the list, but merely discarded when it becomes due. In comparing the relative performance of different molecular dynamics algorithms it is difficult to isolate the effect of the event list structure without an extensive analysis of program behavior. But, insofar as the overall method described in this paper is concerned, it will be shown below that it produces an approximately threefold increase in computation speed over that reported by Alder *et al.* [11].

The alternative $O(N^{1/2})$ algorithm [7] mentioned in Section 3 does not behave as expected, the probable reason for this being the overhead of maintaining the ancillary pointer list; the complexity of this maintenance has not been studied either theoretically or empirically. However, with both this structure and the binary tree the most frequently performed individual operation is the stepping from one node to the next, either in the event list itself or in the ancillary pointer list; this operation must be performed $O(N^{1/2})$ and $O(\ln N)$ times, respectively, for each node insertion. Thus, even though no detailed comparison of the two event list structures has been made, it would appear that for a list of $N \approx 1000$ event notices the binary tree approach is significantly more efficient; and this is without taking into account the additional computations required to support the pointer list, a task not required for the tree.

The most meaningful quantity for describing the efficiency of the simulation, or the ratio of simulated time to computation time, is the number of collisions simulated per unit (e.g. hour) of computer time. The overall efficiency of the simulation is dependent not only upon the structure of the event list but also on the processing time required for a single event. The division of the region into cells eliminates any N_A-dependence of the collision computations, but introduces additional cell crossing events which themselves require a certain amount of computation. For a given system (i.e., N_A and v fixed), varying L_c can be used to attain optimal efficiency. If L_c is small (i.e., $L_c^3 < N_A$) the number of atoms which must be examined when scheduling the future collisions of a particular atom can easily exceed 30, irrespective of v (see Tables I and II). On the other hand, if L_c is large—its upper limit is set by the requirement that the cell edge length R/L_c exceed the diameter σ—the mean cell occupancy will be low and hence fewer atoms need be examined, but the additional cell crossings which occur will lower the efficiency. At high density (e.g., $v = 1.5$) the atoms tend to be spatially localized, so this effect can be ignored and the largest allowed L_c used; but at lower densities (e.g., $v = 32.0$) it will be demonstrated below that the efficiency is not monotonically dependent on L_c, and some intermediate value of L_c should be employed.

Table I summarizes some of the statistics of the simulation runs. The collision rates given are for the IBM 370/168 computer. The tabulated quantity $\tilde{\nu}_{\text{coll}}^{\text{T}}$—the average of $\nu_{\text{coll}}^{\text{T}}$ defined earlier—is seen to be approximately proportional to the mean cell occupancy N_A/L_c^3. Both $\tilde{\nu}_{\text{coll}}^{\text{T}}$ and $\tilde{\nu}_{\text{coll}}^{\text{S}}$ (the mean of $\nu_{\text{coll}}^{\text{S}}$) are independent of N_A as expected. The average values of the quantities $\nu_{\text{cross}}^{\text{T}}$ and $\nu_{\text{cross}}^{\text{S}}$ defined earlier are not shown but are given by $\tilde{\nu}_{\text{coll}}^{\text{T}}/\tilde{\nu}_{\text{cross}}^{\text{T}} \approx 2.5$–$3$, $\tilde{\nu}_{\text{coll}}^{\text{S}}/\tilde{\nu}_{\text{cross}}^{\text{S}} \approx 2$–$3.5$ for the systems shown in the table. Typical distributions of these quantities were shown in Fig. 7.

The effect of varying L_c for given N_A and v is shown in Table II. While $\tilde{\nu}_{\text{coll}}^{\text{T}}$ decreases approximately as L_c^{-3}, the number of cell crossings per collision ($n_{\text{cross}}/n_{\text{coll}}$) increases linearly with L_c, and the collision rate exhibits a maximum at $L_c \approx 8$, corresponding to a mean occupancy of 0.2 atoms per cell. This behavior is a consequence of the fact that the total computation time depends on both n_{coll} and n_{cross}. Since the computation time associated with a single cell crossing is roughly one-sixth of that required for a collision, the collision rate initially increases with L_c as the mean cell occupancy drops, but later starts to decrease because the cell crossing events begin to

TABLE I

Summary of the Mean Numbers of Collisions Tested and Scheduled ($\bar{\nu}_{\text{coll}}^{\text{T}}$, $\bar{\nu}_{\text{coll}}^{\text{S}}$) per Atom Following a Collision; the Average Number of Cell Crossings per Collision ($n_{\text{cross}}/n_{\text{coll}}$), and the Collision Rate[a]

	v	L_c	$\bar{\nu}_{\text{coll}}^{\text{T}}$	$\bar{\nu}_{\text{coll}}^{\text{S}}$	$n_{\text{cross}}/n_{\text{coll}}$	Rate[b]
$N_A = 108$						
	1.5	4	44.1	5.2	0.24	530
	2.0	5	22.0	3.1	0.27	840
	4.0	6	12.5	1.5	1.03	980
	8.0	8	5.3	0.6	2.89	930
$N_A = 256$						
	1.5	6	30.5	4.5	0.15	690
	2.0	7	18.9	2.9	0.29	920
	4.0	8	12.5	1.5	1.01	1060
	8.0	11	4.8	0.6	2.99	1050
$N_A = 500$						
	1.5	8	25.7	4.2	0.17	710
	2.0	8	25.1	3.3	0.26	740
	4.0	11	9.3	1.3	1.11	1010
	8.0	13	5.8	0.7	2.76	880

[a] Typical standard errors are in the range 2–3 for $\bar{\nu}_{\text{coll}}^{\text{T}}$ and 0.7–2 for $\bar{\nu}_{\text{coll}}^{\text{S}}$.
[b] In units of 10^3 collisions per hour of computation time.

TABLE II

The Effect of Varying L_c for a System with $N_A = 108$, $v = 32.0$[a]

4	44.8	0.7	4.2	390
6	13.5	0.5	6.4	566
8	5.8	0.3	8.6	590
10	3.1	0.2	10.8	547
12	1.9	0.1	12.9	510

[a] The rate is highest at $L_c \approx 8$; the largest permitted L_c is 13.

dominate the computations. Before undertaking extensive computations it is well worth determining the optimal L_c in order to maximize the collision rate; it seems reasonable to conclude that ideally $L_c^3 > N_A$, a fact which has not been stressed in the literature.

The average of the collision rates in Table I is 0.86×10^6 collisions/hr. The simula-

tion program used to obtain these results was written in Fortran; the version used for extensive production runs had the tree algorithms coded in assembler language and runs 20% faster, i.e., at an average rate of 1.03×10^6 collisions/hr. This can be compared with a reported rate of approximately 0.25×10^6 collisions/hr on a CDC 6600 computer [11]. Measurements of the relative performance of the all-Fortran version of the program on the two computers (in each case using the standard optimizing compiler) indicate that the IBM collision rate is 50% greater than the CDC, thus the tree-based simulation is intrinsically a factor of 3 faster than that of [11]. Insufficient information is available to point to a specific reason for this improved performance, but presumably a good deal of it is due to the use of the tree structured event list; especially since even the assembler coded version of the tree algorithms consume some 30% of the total computational effort.

6. Concluding Remarks

In this paper we have described a scheme for performing hard sphere molecular dynamic simulations. Particular emphasis has been placed on the data structure and algorithms for handling the event list computations. The importance of a carefully designed data structure is due to the fact that, even when the event list management is performed efficiently, the processing requirements represent a considerable portion of the overall computation time.

The algorithms have been designed with a view to minimizing computation time, but some mention of the memory requirements is in order. Each event notice requires 40 bytes of storage to hold the necessary data and the pointers linking the node into both the tree and the circular lists (Section 3). This requirement is in addition to the memory needed for storing the atomic coordinates and velocities, and the tables required to support the use of cells. The total storage required for the 500 sphere system, including program, is 180 kbytes on the IBM 370. While this is by no means excessive and larger systems could easily be handled by merely expanding the various arrays, a limit is eventually reached where computation speed must be traded off against memory requirements. The possible ways of reducing storage include replacing the bidirectional links in the circular lists by unidirectional ones, use of a more compact tree representation [8], and a reduction in the number of cells into which the region is divided.

The system employed in this paper as a means for describing the molecular dynamics method was the hard sphere fluid. The techniques are readily extended to handle square well potentials, boundary walls which act as heat sources and sinks, and coupled sphere systems such as polymer chains. Molecular dynamics differs from other kinds of event-driven simulations, such as the transportation studies familiar from operations research, in that the majority of events scheduled are canceled before they are able to occur and consequently there is a high turnover rate in the event list, but there is no reason why the efficient event list organization described here should not prove useful in other applications.

REFERENCES

1. B. J. ALDER, *Annu. Rev. Phys. Chem.* **24** (1973), 325.
2. J. J. ERPENBECK AND W. W. WOOD, *in* "Modern Theoretical Chemistry" (B. J. Berne, Ed.), Vol. 6B, p. 1, Plenum, New York, 1977.
3. B. J. ALDER AND T. E. WAINWRIGHT, *J. Chem. Phys.* **31** (1959), 459.
4. A. RAHMAN, *Phys. Rev.* **136** (1964), A405.
5. D. C. RAPAPORT, *J. Phys. A* **11** (1978), L213; see also *J. Chem. Phys.* **71** (1979), 3299.
6. F. P. WYMAN, *Comm. ACM* **18** (1975), 350.
7. W. R. FRANTA AND K. MALY, *Comm. ACM* **20** (1977), 596.
8. D. E. KNUTH, "Fundamental Algorithms, The Art of Computer Programming," Vol. 1, Sect. 2.3, Addison–Wesley, Reading, Mass., 1968.
9. D. E. KNUTH, "Sorting and Searching, The Art of Computer Programming," Vol. 3, Sect. 6.2.2, Addison–Wesley, Reading, Mass., 1973.
10. D. E. KNUTH, "Fundamental Algorithms, The Art of Computer Programming," Vol. 1, Sect. 2.2.5, Addison–Wesley, Reading, Mass., 1968.
11. B. J. ALDER, D. M. GASS, AND T. E. WAINWRIGHT, *J. Chem. Phys.* **53** (1970), 3813.