JOURNAL OF COMPUTATIONAL PHYSICS **134,** 16–30 (1997)
ARTICLE NO. CP965510

Molecular Dynamics for Polymeric Fluids Using Discontinuous Potentials

Steven W. Smith, Carol K. Hall,* and Benny D. Freeman

Department of Chemical Engineering, North Carolina State University, Raleigh, North Carolina 27695-7905

Received July 24, 1995; revised April 15, 1996

Molecular dynamics simulation techniques for systems interacting with discontinuous potentials are discussed. Optimization and efficiency techniques are summarized for performing discontinuous molecular dynamics on serial computers with direct application to polymer-like fluids. Comparisons are presented for two algorithms: (1) single-event scheduling, and (2) multiple-event scheduling. The single-event scheduling algorithm is approximately 75% faster than the multiple-event scheduling algorithm for molecular fluids but yields equivalent performance for atomic fluids. For the single-event scheduling method, a combination of link lists and neighbor lists are used when searching for possible particle interactions. The combination of efficiency techniques permits multibillion time step simulations for relatively large systems on desktop workstations. Both discontinuous molecular dynamics codes for single and multiple-event scheduling algorithms are available on the Internet. The utility of the method is demonstrated for entangled chains, tethered chains, and heteronuclear chain mixtures. © 1997 Academic Press

I. INTRODUCTION

One of the most popular techniques for simulating the properties of large systems of molecules is molecular dynamics (MD). The trajectories of systems with 10^2–10^6 atoms are followed on the computer by repeatedly solving Newton's equations of motion. Macroscopic properties of the system are computed by taking averages of instantaneous properties as the system evolves through time. There are essentially two types of molecular dynamics simulations: (1) continuous molecular dynamics, in which the force acting on a molecule is the spatial gradient of a continuous potential energy field, (e.g., the Lennard-Jones potential) and (2) discontinuous molecular dynamics (DMD), in which the force acting on a molecule is impulsive and the potential energy field is discontinuous (e.g., the hard-sphere potential). In continuous molecular dynamics, the equations of motion are solved numerically at regularly spaced time intervals using finite difference techniques. In DMD, the equations of motion are solved analytically by locating the time between events or collisions and advancing the system to the next event.

Due to the simplicity of its time integration, DMD offers the opportunity to explore phenomena that occur on long time scales using relatively inexpensive hardware. Even though long-time phenomena can be studied using continuous potentials (which are efficient on vector supercomputers) this is not always practical due to the small time steps involved and the unfavorable economics associated with large allocations of machine time. In contrast, the DMD method can cover large time domains on less expensive desktop workstations and workstation clusters. Recent advances in the speed and performance of serial computers makes simulation of large systems interacting with discontinuous potentials attractive for investigating phenomena operating on time scales that cover many orders in magnitude. Furthermore, the development of low-cost, high-speed memory makes treatment of large systems possible on most workstations. Nevertheless, efficient algorithms are required if results are to be achieved in a timely manner.

To illustrate the power of using DMD for investigating long-time phenomena, we consider its application to the dynamics of polymeric fluids which are characterized by a spectrum of relaxation times due to their large molecular size. The DMD method was first applied to polymeric fluids by Rapaport [1, 2] for the study of polymer chain dynamics in solution. Many of the fascinating properties of polymeric fluids are dictated by the topology and volume of the chain molecule which can be easily captured in a simple tangent hard-sphere or pearl necklace model. Using the tangent hard-sphere model as a basis, Rapaport introduced small links between adjacent beads along the chain to partially decouple the motion of bonded beads. This decoupling technique allows the chain system to be treated in a fashion similar to hard-sphere molecular dynamics which was originally developed by Alder and Wainwright [3]. The addition of sliding links to the tangent hard-sphere model creates a bond-extension collision which maintains the connectivity of the chain. The bond-extension collision is treated in

* Corresponding author.

0021-9991/97 $25.00
Copyright © 1997 by Academic Press
All rights of reproduction in any form reserved.

a manner which is fundamentally identical to a square-well interaction.

The main purpose of this paper is to provide a summary of techniques developed for DMD and to offer suggestions for improving the efficiency of codes designed to investigate slowly relaxing systems (e.g., polymeric fluids) on workstations. To this end, we compare two algorithms which differ in their treatment of neighbor interactions and in the method for scheduling events in the system. The multiple-event scheduling algorithm uses subcells for interaction consideration and a doubly linked binary tree for event-scheduling. The single-event scheduling algorithm uses a dual-structured neighbor list and a simple binary tree. The single-event method yields a 75% increase in performance over the multiple-event method for polymeric models while both methods yield similar performance for hard-sphere fluids. Increased performance is beneficial for studying transport phenomena, where correlations must be averaged over long times. Both codes are available on the Internet (`http://turbo.che.ncsu.edu/smithsw/`) for the benefit of research workers interested in using models with discontinuous potentials.

We have organized the paper as a tutorial, starting with the basics of DMD as applied to hard-sphere systems and culminating in the implementation of the Rapaport model for polymeric fluids. Section II provides a brief description of the basic DMD method for hard-sphere systems. Section III details DMD techniques for polymeric fluids using the Rapaport model. Section IV summarizes efficiency techniques from the literature as well as new modifications to the single-event scheduling approach. Section V discusses numerical accuracy and compares computational performance for the single-event and multiple-event scheduling algorithms. In Section VI we illustrate the versatility of the DMD method for systems containing entangled chains, heteronuclear chains, and tethered chains.

II. DISCONTINUOUS MOLECULAR DYNAMICS

The first molecular dynamics simulations were performed by Alder and Wainwright [3–5] who took advantage of the fact that the equations of motion for a hard-sphere potential can be solved analytically. In the absence of a potential field, a particle trajectory is linear, and the particle moves with constant velocity until a collision occurs. A collision, or event in broader terms, occurs whenever the separation between two particles becomes equal to a point of discontinuity in the potential, which, for the hard sphere case, is just the particle diameter σ. When such an event occurs, the particle velocity will change in accordance with the dynamics of the model under study. A DMD simulation evolves on an event-by-event basis by locating the next event in the system, advancing the system to that point in time, computing the event dynamics, and repeating the process. Instead of taking uniform time steps, the conventional practice for continuous potentials, DMD programs use variable time steps and, for this reason, are also referred to as event-driven molecular dynamics.

The key steps in a DMD simulation are the calculation of the collision times, t_{ij}, between particles i and j, and the calculation of the postcollisional velocities for the colliding pair [6, 7]. The collision time between particles i and j is given by

$$t_{ij} = \frac{-b_{ij} \pm \sqrt{b_{ij}^2 - v_{ij}^2(r_{ij}^2 - \sigma^2)}}{v_{ij}^2}, \tag{1}$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the relative position, $\mathbf{v}_{ij} = \mathbf{v}_i - \mathbf{v}_j$ is the relative velocity, and $b_{ij} = \mathbf{r}_{ij} \cdot \mathbf{v}_{ij}$. The minimum of all possible collision times is the next collision time t_c. Each particle is advanced by t_c to a new position given by

$$\mathbf{r}_i(t + t_c) = \mathbf{r}_i(t) + \mathbf{v}_i t_c, \tag{2}$$

where $\mathbf{r}_i$ and $\mathbf{v}_i$ are the particle position and velocity. At this point in the simulation a single pair of particles are in contact and about to undergo an elastic collision while all the other particles remain separated. Postcollisional velocities velocities are calculated by imposing the constraints that both kinetic energy and linear momentum are conserved for the smooth, spherical particles.

III. CHAIN DYNAMICS

Rapaport [1, 2] pioneered an algorithm to simulate hard chains by allowing the distance between bonded spheres to freely vary over a range between σ and $(1 + \delta)\sigma$, where δ is small compared to one. The chain is composed of n spheres or mers of diameter σ attached to each other in a necklace fashion by sliding links. Bonded spheres of a chain can also be thought of as being attracted by an infinitely deep square-well potential of width equal to $\delta\sigma$. As δ approaches zero, the Rapaport model becomes equivalent to the tangent hard-sphere model. Thus, adjacent spheres along the chain experience two types of intramolecular collisions; first, an elastic hard-core collision at a distance of σ, and second, a bond-extension collision at $(1 + \delta)\sigma$ when the sliding link is fully extended. These two events constitute the bond vibration for the chain fluid [8]. Bellemans *et al.* [9] proposed a natural extension of the Rapaport model in which the reduced bond length, l/σ, is allowed to vary between $(1 - \delta)$ and $(1 + \delta)$ with an average close to 1. This allows direct comparison between Monte Carlo (MC) results for the tangent hard-sphere chain and the MD results of the Rapaport model. Denlinger and Hall [8] found good agreement between MC and

MD results for compressibility factors using the Rapaport model. Chapela and Martinez [10, 11] observed quantitative agreement for fluid structure between the tangent hard-sphere model and the Rapaport model.

Since the bonded spheres are partially decoupled in the Rapaport algorithm, linear trajectories for the individual segments are restored. As a result, collision times and postcollisional velocities are the same as for the hard-sphere case. The only difference is that an additional computation is required for the bond-extension time. The bond stretch is treated as a bounce-type collision [1, 4, 5] in an infinitely deep square-well potential. A disadvantage of the Rapaport algorithm is the large effort spent monitoring the bond vibrations [9] whose computational expense increases as $1/\delta$.

Creating initial configurations for chain fluids at high density is always a challenging problem. Chain dimensions increase considerably as the volume fraction decreases due to excluded volume effects. The Rapaport model provides a natural way to achieve high volume fractions which are of interest for liquid-state simulations. Chains can be grown from a random walk at low densities and relaxed to insure proper statistics. From this relaxed state, the volume fraction is increased by slowly growing the segment diameters while the simulation progresses. At prescribed intervals, all segment diameters are increased by an amount equal to the minimum separation between any two particles in the system. A bond-extension event is the most obvious point in the simulation to increase segment diameters since no two particles are in direct contact as would be the case in a core-collision event. Since all chain segments change size in the growth step, collision times must be recalculated to avoid the possibility of missed events. The maximum bond length is increased along with the segment diameter until the desired volume fraction is achieved.

IV. EFFICIENCY TECHNIQUES

A. *Time Lists*

Efficient algorithm design eliminates unnecessary calculations to achieve the highest possible performance. The most computationally expensive part of the DMD method is the location of the next event because it involves $N(N-1)/2$ calculations of event times where N denotes the number of particles in the system. The inefficiency of this calculation stems from recomputing many of the same event times at each iteration. Alder and Wainwright [4] realized that efficiency could be improved by: (1) storing event times of *soon*-to-occur events for each particle in time lists for later use, and (2) subtracting the next event time t_c from these previously stored values to avoid recalculation. By using multiple-event time lists, only new event times involving the colliding pair, i and j with the other $N-1$ particles need to be recomputed at each iteration.

a

Times			Partners		
t_1	t_1'	…	j_1	j_1'	…
t_2	t_2'	…	j_2	j_2'	…
⋮	⋮	…	⋮	⋮	…
t_{N-1}	t_{N-1}'	…	j_{N-1}	j_{N-1}'	…
t_N	t_N'	…	j_N	j_N'	…

b

Time	Partner
t_1	j_1
t_2	j_2
⋮	⋮
t_{N-1}	j_{N-1}
t_N	j_N

FIG. 1. Time list structure for (a) multiple-event and (b) single-event scheduling, where t_k represents the collision time for particle k and j_k is the partner involved in that collision. Primes denote multiple events for the same particle.

The multiple-event time lists are structured in the form of two matrices as illustrated in Fig. 1a: the first contains event times and the other contains the corresponding partner for that event. The ith row of the time matrix contains the event times t_i for several possible events involving particle i. The ith row of the partner matrix contains the particle index, j, of the partner that may collide with particle i at time t_i. The primes denote multiple events for that particle. The stored event times in each row may or may not be ordered in accordance with their event time. Certainly, an ordered row will minimize the effort to find the next collision time in the system. The number of events N_e (i.e., number of columns) stored for each particle will vary with an average of approximately five for the hard-sphere fluid at liquid-like densities.

When updating a multiple event time-list, only new collision times need to be recomputed for the particles i and j that have just collided. Furthermore, due to the velocity change in i and j, all other stored events that involve either particle i or j must be discarded from the time list. This is done by searching through the time list for any events involving i or j and simply removing that event. Direct removal of these events is possible because multiple events have been stored for each particle. In other words, the loss of a single event from any row in the time matrix does not require renewal of the entire row since other stored events in the row are still valid. For i and j, time list renewal is accomplished by searching for possible interactions between particle i and the other $N-1$ particles, and between particle j and the other $N-1$ particles such that row i and row j of the time list are completely refreshed. As a result, the multiple-event time list reduces computational work for collision time determinations at each iteration from $N(N-1)/2$ to $2(N-1)$. The next time step t_c for the system would be obtained by searching the time matrix for the minimum value.

An alternative approach, proposed by Allen [6], is to only store the *soonest*-to-occur event with its partner in a "single-event" time list. Since only a single event is stored

for each particle, the size and overhead associated with the time list is minimized. The shortest of all possible event times t_i for particle i is stored in the ith row of the time-list vector while the corresponding partner for that collision is stored in the ith row of the partner vector as shown in Fig. 1b. The next time step t_c for the system would be obtained by searching the time vector for the minimum value.

Single-event time lists significantly reduce memory requirements and simplify the maintenance of the time list structure. While single-event time lists reduce storage requirements from $2NN_e$ to $2N$, computational requirements increase because particles that *were destined* to collide with i or j (had i and j not changed direction) must also be updated at each time step. These extra updates are necessary because only one event is stored per particle in the single-event time list. At each iteration, approximately two more particles (in addition to i and j) must be updated. However, by constructing the single-event time list so that the collision partner of i is always greater than i, only half (on the average) of the possible interactions need to be considered for the two extra updates at each time step. This upward construction naturally eliminates double checking of interactions and allows minimal effort for time-list updates. The single-event time list update is divided into two routines: the first checks for possible interactions with all particles greater than i(`uplist`), and the second for those less than i(`dnlist`). As a result, the computational work associated with the time list update in single-event scheduling is actually on the order of $3(N - 1)$ which is only 50% more than the multiple-event method.

When employed in simulations of polymers based on the Rapaport model, time list techniques can also be used to reduce the frequency of the bond or square-well interaction calculations. In this case, the time list will not only contain core-collision events but also bond-extension and square-well events. For each particle, only the *soonest*-to-occur event (i.e., core, bond, or square-well) is stored in the time vector. An additional vector of length N stores an integer associated with the collision type (i.e., core, bond, or square-well) and generalizes the single-event scheduling technique for any number of different event types. For example, Chapela and Martinez-Casas [10, 11] used a series of discontinuities in the form of a stair-step potential to mimic a Lennard-Jones potential with a DMD algorithm. In this case, many different event types must be scheduled as a particle moves into and out of the attractive well.

B. *Neighbor Lists*

Since only particles which are in close proximity are destined to collide, the effort to update time lists can be reduced by restricting the search for future collision partners to neighboring particles. By restricting this search to nearest neighbors, the computational effort required to update the time list is dependent only upon the number of neighbors, which in turn, depends on fluid density. In the subcell method, introduced by Erpenbeck and Wood [12], a cell table technique is employed to divide the primary simulation cell into smaller subcells. The search for possible events is restricted to particles in the 27 subcells surrounding the subcell containing a given particle.

The approach taken here is to construct a neighbor list [13] that contains the index number of all particles within a spherical shell of radius r_n surrounding any given particle. Thus, when particle i experiences a collision, only collision times between particle i and particles in the neighbor list of i need to be considered for possible interactions. The neighbor lists must be updated before any particle can diffuse further than $(r_n - \sigma)/2$. An optimum shell radius r_n exists because larger shells (i.e., more neighbors) require more effort to update the time lists and smaller shells require that the neighbor lists be updated more often. Our studies suggest that an optimal neighbor list contains between 30–35 neighbors for liquid-like densities (volume fraction >0.3) using a hard-sphere potential. Since the hard-sphere potential is a short-ranged potential and the integration of the equations of motion is exact, the trajectory accuracy is not affected by neighbor list radius unless collisions are overlooked due to limited interaction considerations from a very small neighbor list. The optimal value (30–35) is only slightly larger than the value determined by using the first shell of neighbors (26) in a simple cubic lattice arrangement of the particles. Frequency of neighbor list updates depends on the system dynamics, with slowly diffusing systems requiring fewer updates. Neighbor list expiration is checked by periodically calculating particle displacements. Neighbor lists are renewed if the measured displacement exceeds a *safe* limit (i.e., $(r_n - \sigma)/2.5$) which is less than the maximum allowable limit (i.e., $(r_n - \sigma)/2$). This permits checking for neighbor list expiration at user prescribed intervals (i.e., every 100 collisions) rather than at each time step, but it requires several trial runs in order to select the appropriate checking interval which maximizes speed but does not permit particle overlap.

As noted previously in the discussion on single-event time lists, the search for possible collision partners is divided in two routines (`uplist` and `dnlist`) in order to eliminate double checking of ij pairs. Our neighbor lists are organized in a similar fashion. A doubly constructed neighbor list is employed wherein the neighbors for a given particle are placed in two vectors: (1) neighbors with an index $>i$ (`upnab`) and (2) neighbors with an index $<i$ (`dnnab`). A doubly constructed neighbor list does not increase the number of calculations for list construction, but it does add a slight overhead when sorting neighbors into each list. The following FORTRAN code illustrates one

method for neighbor list construction. Neighbors are collected in two vectors named upnab and dnnab each of length NN_n, where N_n is the maximum number (nnab) of neighbors for the chosen cutoff radius. The variables nup and ndown dictate the location for the particle in the "up" and "down" neighbor lists, respectively, while the two vectors nnabup and nnabdn keep track of the total number of neighbors in the upnab and dnnab lists.

```
Initialize pointer into upnab vector
      nup=0
      do 300 i = 1, N - 1
Calculate separations between
particle i and particles i + 1, N
          do 199 j = i + 1, N
           rxij = rx(i) - rx(j)
           ryij = ry(i) - ry(j)
           rzij = rz(i) - rz(j)
Apply periodic boundary conditions
           rxij = rxij - dnint(rxij)
           ryij = ryij - dnint(ryij)
           rzij = rzij - dnint(rzij)
           rsq(j) = rxij**2 + ryij**2
     &                  + rzij**2
199        continue
Sort neighbors into two vectors (upnab, dnnab)
based on cutoff r_n
          nnabup(i) = nup + 1
          do 299 j = i + 1, N
           if (rsq(j) .lt. rcut)then
This is a neighbor so insert into lists
            nup         = nup + 1
            upnab(nup)  = j
            nnabdn(j)   = nnabdn(j) + 1
            ndown = (i-1)*nnab + nnabdn(j)
            dnnab(ndown)= i
           end if
299       continue
300   continue
      nnabup(n) = nup + 1
```

Sequential packing of neighbors into the upnab vector is made possible by the upward construction of the do loop (i.e., loop 299 runs from $i + 1$ to N). Since the number of "down" neighbors for i is unknown at the beginning, the insertion position (ndown) into the dnnab vector must be computed. Furthermore, if the maximum number of expected neighbors (nnab) for a given r_n is 50 then dnnab must be dimensioned to at least $50N$ to inhibit array boundary crossing for the last particle (N). With the optimum neighbor list size of 30–35 neighbors, a dimension of $50N$ for the dnnab should be adequate. The double loop construction (loops 199 and 299) allows for vectorization of the first loop and is generally faster for optimizing compilers.

Doubly constructed neighbor lists also enhance the efficiency of deciding which particles besides i and j need to be updated in the single-event scheduling method described earlier. After each step in the simulation, time lists must be refreshed for the colliding particles (i, j) and for any particles that *would* have collided with i or j had the trajectories of i and j not changed. Any particle destined to collide with i would necessarily appear in the "down" neighbor list for i due to the fact that the collision partner for any particle always has a higher index number. To locate such particles, each down neighbor is checked for a scheduled event with i and then added to a temporary vector for later updating if such an event is found. The process is then repeated for particle j. The following code illustrates this. The variables kstart and kend are pointers to the first and last position of the dnnab vector where the down neighbors of i (then j) reside. The ncnt particles which need updating are packed into a single vector nupdat.

```
Check the down neighbors of i
        kstart = (i-1)*nnab + 1
        kend = kstart + nnabdn(i) - 1
        do 449 kk=kstart, kend
           k = dnnab(kk)
           if(partnr(k) .eq. i)then
              ncnt = ncnt + 1
              nupdat(ncnt) = k
           end if
449     continue
Check the down neighbors of j
        kstart = (i-1)*nnab + 1
        kend = kstart + nnabdn(j) - 1
        do 450 kk=kstart, kend
           k = dnnab(kk)
           if(partnr(k) .eq. j)then
              ncnt = ncnt + 1
              nupdat(ncnt) = k
           end if
450     continue
```

The next step is to call the appropriate routine to update collision times (i.e, uplist) using values from the nupdat vector. It should be noted that particle i is included in nupdat by loop 450 because i is always less than j and will necessarily appear as a down neighbor of j.

C. *Link Lists*

Further efficiency can be gained when the system size exceeds several hundred particles by implementing link lists [14, 15] to reduce the $N^2/2$ computations for neighbor list construction/renewal. The link list method divides the primary simulation cell into $M \times M \times M$ smaller subcells

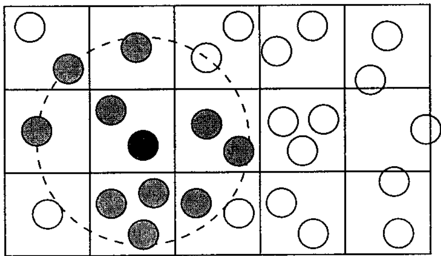

FIG. 2. Two-dimensional illustration of subcells for link lists used in neighbor list construction. The lightly shaded particles are neighbors for the darkly shaded particle since they lie within the cutoff radius (dashed circle). Neighbor lists are constructed by searching the nine subcells (2D) for possible neighbors.

and creates a list of the particles in each subcell. Neighbor list construction is restricted to searching the 27 subcells centered around the particle of interest. The subcell length l_s ($=L/M$) must be larger than the neighbor cutoff distance (r_n) to avoid missing possible neighbors. A two-dimensional analog of this procedure is shown in Fig. 2. Link-list construction is an order N computation involving minimal floating-point work. For N_s particles per subcell, neighbor list formation is reduced from $N^2/2$ to $27N_sN/2$ computations. Our results suggest that the optimal subcell size contains approximately eight particles for hard-chain simulations. The combination link list/neighbor list technique resembles the subcell method of Erpenbeck and Wood [12] without cell-crossing events to track particle displacements.

D. *Scheduling*

The search for the minimum value in the time lists poses significant computational work for large systems. A straightforward linear search through the single-event time list requires an order N operation to locate the next event in the system. An improved method for searching lists is available in the form of a binary tree [16, 17]. The binary tree method was first applied to DMD by Rapaport [18] who developed a very efficient and elegant binary tree for the multiple-event time lists. The binary tree algorithm for multiple-event scheduling is complex and includes two circular lists with two pointers each [18] in order to facilitate the cancellation of events when a particle collides. Because the velocity changes for the colliding pair, a collision invalidates all scheduled events associated with the colliding particles i and j. The circular lists facilitate location of all events associated with the colliding particles. The Rapaport binary tree is constructed in conjunction with a subcell division of the primary simulation cell so that a subcell crossing is the entry point into the tree.

Substantial simplification of the Rapaport implementation of the binary tree is possible in the single-event scheduling method. In this case the circular lists are eliminated and only one event per particle (the *soonest*-to-occur) is scheduled in the tree. Tree maintenance is also faster because the tree size is substantially smaller in the single-event algorithm (N vs NN_e). Also, fewer events must be deleted or added at each time step in the single-event method since only one event is scheduled per particle. Even though the search for the next event is reduced by application of a binary tree, the overhead associated with insertions and deletions from the tree does increase. Nevertheless, we obtain a speedup of 15–25% when applying a simple binary tree to the single-event scheduling algorithm.

A binary tree is structured to enable easy location of the next event in the system, facilitate scheduling of future events, and allow removal of cancelled events. A binary tree stores events in a set of nodes, where each node contains a link to its predecessor and two successors. The successor events are organized such that the left branch has a time less than its predecessor and the right branch has a time greater than its predecessor. Either one of the successor events may be a null event indicating termination for that branch of the tree. Each node contains an event time and pointers to the left, right, and previous nodes. The left-most node of the tree contains the minimum time for all events in the tree. In short, the binary tree is a link-list approach applied to the time list and provides a quick way to search the time list. For a single-event time list, the search for the minimum time involves N comparisons; the binary tree reduces this search to $\log_2 N$ comparisons if the events are sufficiently random to produce a well-balanced tree.

Figure 3 gives an example of a binary tree constructed from a particular time list. The nodes which form the tree are connected by three pointers: right node `noder`, left node `nodel`, and previous node `nodep`. Any particular tree structure is not unique, but rather, depends on the order in which events are added. In this case, tree nodes are inserted by moving sequentially down the time list. Since only one event per particle is scheduled, the node number corresponds to the particle index. The root node provides an entry point into the tree and is not used for event storage. The tree is constructed as follows. The event time for particle 1 is inserted as the first branch because particle 1 is the first to be scheduled. The second time from the list is inserted to the right of the first node (since $t_2 > t_1$) by setting `noder(1)=2` and `nodep(2)=1`. The third particle in the list (third node) is inserted by starting at the top of the list, moving right (since $t_3 > t_1$), then moving left (since $t_3 < t_2$). The tree pointers are then updated as `nodel(2)=3` and `nodep(3)=2`. Subsequent events are similarly scheduled by starting at the first node and comparing event times with each existing node and moving downward to the left or right depending on the relative times to find the appropriate insertion position. In this manner,

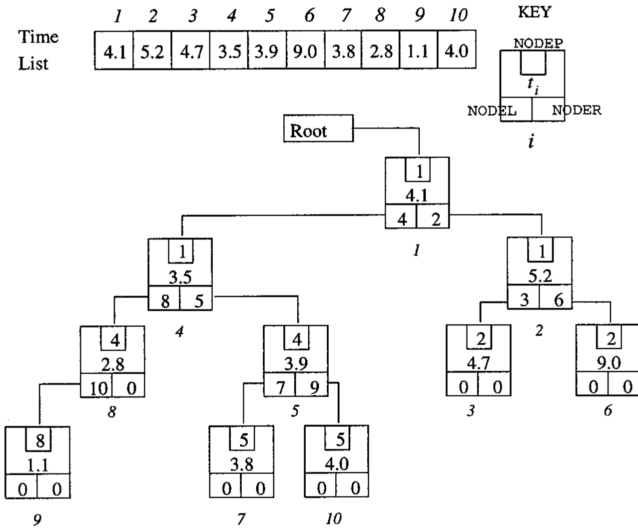

FIG. 3. Binary tree structure for event scheduling. The tree is constructed from the time list at the top. Italicized numbers represent the particle index as well as the node number. The next event involves particle 9. A zero in the pointer position represents termination of that branch.

any vertical plane through the tree will have smaller event times on the left than on the right. The minimum of all event times (t_9 in this case) is located at the leftmost position in the tree.

The following routines prepare a simplified binary tree for the single-event scheduling algorithm. A common block (`tree`) is used for storing the tree pointers while event times for each node are stored in the time list vector `coltim` which is also available through a common block. The routine `addtree` receives a particle index number and schedules the associated event by moving left and right through the tree to find the insertion position (`inpos`).

```
      subroutine addtree(newnod)
      dimension nodep(0:N),nodel(0:N)
      dimension noder(0:N)
      common /tree/ nodep,nodel,noder
Common block with event times here.
Put the newnod into the binary tree
based on its time tnew
      tnew = coltim(newnod)
If the tree is empty, put this node first
     inpos = 0
     inpos = noder(0)
     nfound = 0

Otherwise, search through all branches to find
the proper insertion position (inpos).
The right branch will always have a larger time.
      do while(nfound .eq. 0)
if TNEW < TEVENT(inpos) then go left.
         if(tnew .le. coltim(inpos))then
            if(nodel(inpos) .ne. 0)then
               inpos = nodel(inpos)
            else
               nodel(inpos) = newnod
               nfound = 1
            end if
if TNEW > TEVENT(inpos) then go right.
         else
            if(noder(inpos) .ne. 0)then
               inpos=noder(inpos)
            else
               noder(inpos) = newnod
               nfound = 1
            end if
         end if
      end do

Insertion position inpos now equals the
position preceding where the event was added,
so set the previous node to inpos.
      nodep(newnod) = inpos
And since the new node is a branch end
      nodel(newnod) = 0
      noder(newnod) = 0

      return
      end
```

Node deletion in the single-event scheduling algorithm is straightforward; since only one event is stored in the tree for each particle, only one deletion per particle is necessary for tree renewal. This is important since node deletion incurs substantial overhead in any binary tree algorithm. The primary objective of a node deletion routine is to identify which of the deleted node's successors should be linked to the deleted node's predecessor. The four separate situations which can emerge for a node deletion are represented in Fig. 4 (adapted from [18]). In this example, D represents the deleted node, P represents the predecessor node, and S represents the successor node. In case I, the deleted node is followed on the right branch by a null event and on the left branch by either another event or a null event. In this case, the predecessor node (P) should be linked to the left branch following the deleted node (D). In case II, the deleted node contains a null event on the left branch and a nonnull event on the right branch. In this case, the predecessor node should be linked to the right branch of the deleted event. In case III, the deleted node contains nonnull events on both the left and right branches, while the right branch contains a null left branch indicating that the right branch is the smallest event time for that particular branch. Since the event time on the right is larger than the event time on the left, the right branch is designated as the successor node. In this case,

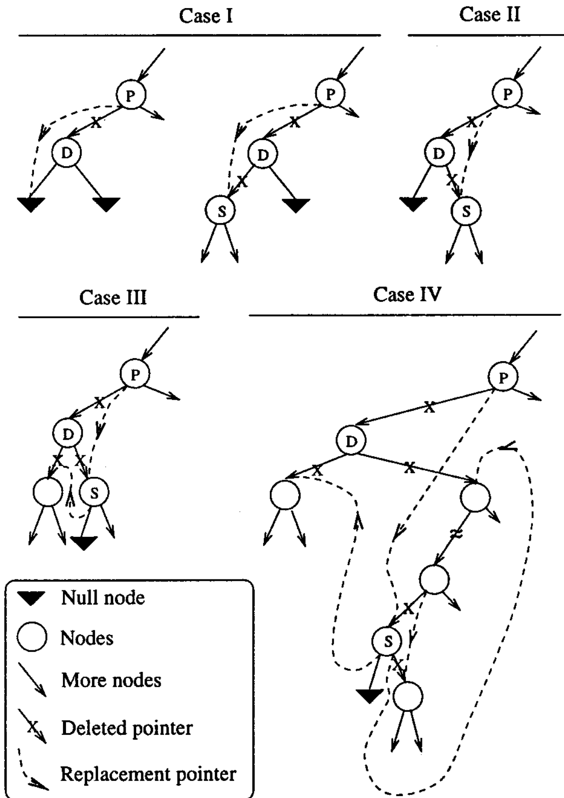

FIG. 4. Schematic of tree pointer manipulation for node deletion in four distinct cases (case I shows two subcases). The nodes labeled D, P, and S represent deleted, predecessor, and successor nodes, respectively. Lines marked by an X represent deleted pointers while dashed lines indicate replacement pointers. Adapted from Ref. [18].

the deleted node is simply replaced by the right branch. Case IV is the most complicated of the possible scenarios and also the most prevalent for any sizeable tree structure. In this case, the right branch of the deleted node has a nonnull left branch which is searched for the minimum event by moving to the leftmost node of that branch. This event is assigned as the successor and links are formed as shown in Fig. 4.

The routine `detree` receives a particle index number and deletes the associated event by connecting pointers as prescribed by Fig. 4. This routine is called after each update of the time list so that cancelled events can be removed. The routine `addtree` is called afterwards to reschedule the new time.

```
      subroutine detree(id)
      dimension nodep(0:N),nodel(0:N)
      dimension noder(0:N)
      common /tree/ nodep,nodel,noder
```
Delete a node from the tree and reconnect pointers.
```
      if(noder(id) .eq. 0)then
```
Case I
```
          is = nodel(id)
        else
```
Case II
```
          if(nodel(id) .eq. 0)then
            is = noder(id)
          else
            if(nodel(noder(id)) .eq. 0)then
```
Case III
```
              is = noder(id)
            else
```
Case IV
```
              is = nodel(noder(id))
              do while(nodel(is) .ne. 0)
                is = nodel(is)
              end do
```
Relink the pointers
```
              nodep(noder(is)) = nodep(is)
              nodel(nodep(is)) = noder(is)
              nodep(noder(id)) = is
              noder(is) = noder(id)
            end if
            nodep(nodel(id)) = is
            nodel(is) = nodel(id)
          end if
        end if
```
Set the previous node pointer
```
        ip = nodep(id)
        nodep(is) = ip
        if(nodel(ip) .eq. id)then
          nodel(ip) = is
        else
          noder(ip) = is
        end if
        return
        end
```

The last routine `nextev` locates the minimum time in the tree by searching down the left most branch and returning the particle index associated with the next event. This routine is called once at each iteration to determine the next event or colliding pair.

```
      subroutine nextev(i)
      dimension nodep(0:n),nodel(0:n)
      dimension noder(0:n)
      common /tree/ nodep,nodel,noder
```
Find the minimum event
```
      i=noder(0)
      do while(nodel(i) .ne. 0)
         i = nodel(i)
      end do
      return
      end
```

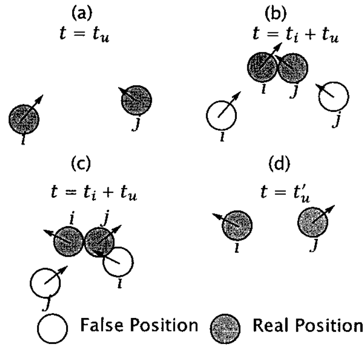

FIG. 5. Changes for particles i and j in false positioning before and after a collision. (a) False and real positions are the same at $t = t_u$. (b) Particles undergo a collision at time $t_i + t_u$ and change velocity. (c) Particles are rewound in the false frame based on new velocities. (d) Particles arrive at their correct locations at the next update $t = t'_u$.

E. *False Positioning*

Another efficiency technique introduced by Erpenbeck and Wood [12] consists of updating particle positions at prescribed intervals rather than at each time step. This procedure reduces computational work associated with advancing all particles in time after each event. Efficiency is improved because most particles move in discrete time steps along a linear trajectory as the simulation progresses. The method delays position updates for all particles, thereby, eliminating an order N calculation at each time step. This is accomplished by defining the false position $\mathbf{s}_i$ for particle i as

$$\mathbf{s}_i = \mathbf{r}_i - \mathbf{v}_i(t - t_u), \tag{3}$$

where t_u represents the time of the last position update, t is the current elapsed simulation time, and $\mathbf{r}_i$ is the true position of particle i at time t. For $t = t_u$ the false positions are identical to the true positions. The false position of a particle remains constant between updates except at a collision where a new false position is assigned such that the correct true position will be achieved at the next update. This is accomplished by rewinding the particle's false position based on its postcollisional velocity as illustrated in Figs. 5a–d. The two colliding particles (a) are placed temporarily in their true positions for the calculation of their new velocities (b). Based on their new velocities, the particles move backwards (relative to their colliding positions) by the amount of time since the last update $t - t_u$ (c). This movement will place the particles in a new false position such that the correct position will be achieved at the next update (d). After a collision, the new position in the false coordinate frame for particle i is

$$\mathbf{s}_i(\text{after}) = \mathbf{s}_i(\text{before}) - \Delta\mathbf{v}_{\text{coll}}(t - t_u). \tag{4}$$

Successive collisions by the same particle between positional updates are treated in the same manner since the rewound position of the particle will always give the correct location at the next update regardless of the number of events or collisions experienced.

While the use of false positioning reduces position updating to discrete time intervals (e.g., every $N/2$ events) it adds work to routines that search for possible interactions. Particles must be temporarily placed in their true positions in order to determine possible interactions correctly. The additional work depends on the number of neighbors, but it is independent of N. When searching for possible interactions, the true separation between particles can be determined in relative coordinates as

$$\mathbf{r}_{ij} = \mathbf{s}_{ij} + \mathbf{v}_{ij}(t - t_u). \tag{5}$$

This computation must be added to any routine which searches for possible interactions with the N_n neighbors of any particle. From our earlier discussion we noted that this occurs for approximately three particles at each time step, so false positioning eliminates N computations in favor of $3N_n$ computations, making the position update calculation independent of system size. The following section of code illustrates the temporary placement of particles in their true positions in order to determine possible interactions correctly. TNEXT is the time since the particle positions were updated and is equal to $t - t_u$.

```
vxij = vx(i) - vx(j)
vyij = vy(i) - vy(j)
vzij = vz(i) - vz(j)
rxij = rx(i) - rx(j)
ryij = ry(i) - ry(j)
rzij = rz(i) - rz(j)
```
Extra work for false positioning
```
rxij = rxij + vxij*tnext
ryij = ryij + vyij*tnext
rzij = rzij + vzij*tnext
```

In contrast to the use of local time variables for each particle [18], which would yield similar efficiencies, false positioning does not require extra storage. Position updates should be performed at the same time interval as checks for neighbor list expiration to minimize the amount of work associated with false positioning. Although longer delays in position updating improve algorithm efficiency, they increase roundoff error in collision time calculations and decrease momentum conservation.

V. NUMERICAL RESULTS

A. *Precision*

Computer simulation is only exact to the degree of precision afforded by the computer. A discriminating check on

precision is to run the simulation for a number of time steps, reverse the velocities, and run the simulation (backwards) for the same number of time steps. The initial and final configurations should be the same. The difference between the final and initial configurations is a measure of mathematical precision. The effect of loss of accuracy in the physical properties that one wishes to calculate is not easily measured. The good agreement found by Alder [19] between MD and MC results for the equation of state of hard-sphere systems indicates that trajectory inaccuracy introduces no bias into equilibrium properties.

One source of inaccuracy arises in the calculation of event times which involves the solution to a quadratic equation. For a general quadratic equation ($ax^2 + bx + c = 0$) the usual approach for determination of the two roots x_1 and x_2 is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \tag{6}$$

However, if a or c are small relative to b and b is positive, then the calculation of x_1 will involve the subtraction of two nearly equal quantities which could give an inaccurate root. For example, in Eq. (1) the value of $(r_{ij}^2 - \sigma^2)$ could be small, especially for bond-extension events in the Rapaport model. An alternative solution for the roots [20, 7] in Eq. (6) is

$$x_1 = d/a, \quad x_2 = c/d, \tag{7}$$

where

$$d \equiv -\tfrac{1}{2}[b + \mathrm{sgn}(b)\sqrt{b^2 - 4ac}], \tag{8}$$

where sgn(b) has a value 1 for positive b and -1 for negative b. For example, in Eq. (1) b_{ij} is less than zero for any colliding pair and the solution is given by

$$t_{ij} = \frac{(r_{ij}^2 - \sigma^2)}{\sqrt{b_{ij}^2 - v_{ij}^2(r_{ij}^2 - \sigma^2)} - b_{ij}}. \tag{9}$$

Since b_{ij} is less than zero, the denominator will involve an addition rather than subtraction. In the bond-extension case b_{ij} can be either positive or negative, and the algorithm must include extra logic to decide which of the two roots is appropriate, based on the sign of b_{ij}. This technique avoids the subtraction of two similar numbers and improves accuracy in the collision-time calculation.

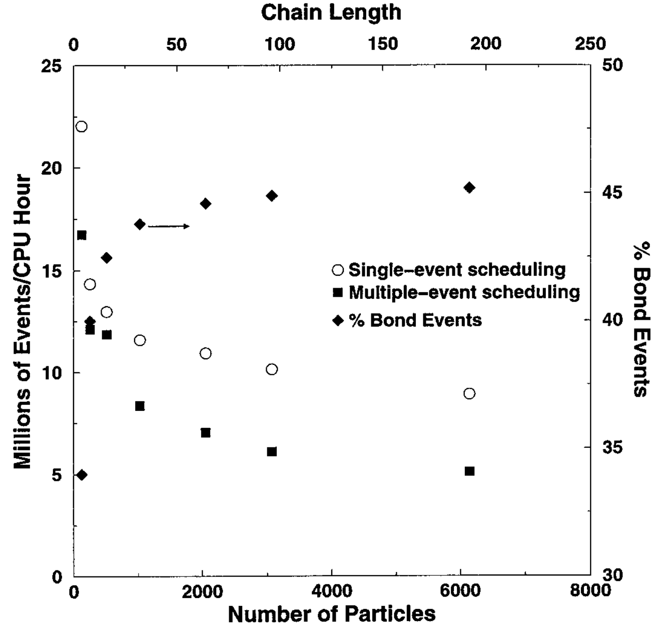

FIG. 6. Performance comparison between single-event and multiple-event scheduling algorithms for chain fluids at a volume fraction of 0.45. All runs were performed on a DEC 3000/400 workstation. In each case there are 32 chains while the chain length varies from 4 to 192. Percentage of events which correspond to bond-extensions for each chain length is shown on the second ordinate.

B. *Performance for Single- and Multiple-Event Methods*

We have coded both single-event and multiple-event scheduling algorithms in order to evaluate the advantages of each approach. Both algorithms are designed with hard or square-well chain simulations as the objective but are sufficiently modular to allow application to any model using discontinuous potentials. The single-event scheduling examples use a combination of neighbor lists and link lists for interaction consideration and a simple binary tree for event scheduling. The multiple-event scheduling examples use the subcell method for interaction calculation with approximately one particle per subcell and a binary tree with circular lists [18] for event scheduling and deletion. The single-event scheduling examples use a neighbor list of approximately 30 particles and link lists with 8–10 particles per subcell from which neighbor lists are constructed. Both algorithms use false positioning and the minimum image convention [21]. Periodic boundary conditions are only employed when calculating particle separations, so true particle positions may be outside the primary simulation cell.

Figure 6 compares the performance of the single- and multiple-event scheduling codes for hard-chain fluids, measured in millions of events per CPU hour, as a function of the number of chain segments N. These benchmarks were performed on a DEC 3000/400 workstation operating at

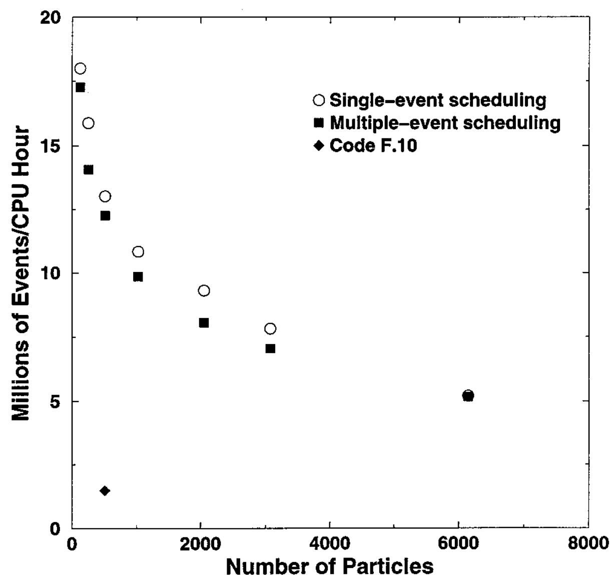

FIG. 7. Performance comparison between single-event and multiple-event scheduling algorithms for the hard-sphere fluid at a volume fraction of 0.45. Algorithm performance for code F.10 from Ref. [6] for 512 particles is also given.

133 MHz using 64-bit arithmetic. This machine is equipped with 16K of on-chip cache and 512K of secondary cache. All chain fluids use the Rapaport model with a bond factor of 0.1 at a volume fraction of 0.45. Memory requirements are approximately equivalent for the two algorithms. The system size ranges from 128 segments for the 4-mer fluid to 6144 segments for the 192-mer fluid. In order to make a direct comparison between the two algorithms, subcell-crossing events in the multiple-event algorithm are not included in performance ratings. In all cases the single-event scheduling is faster than multiple-event scheduling by as much as 75%. The percentage of events which constitute the bond vibration are also shown on the second ordinate of Fig. 6. For a chain-like fluid, or any molecular fluid, the bond vibration is often the limiting time step in molecular dynamics simulations. The short time-step for bond events causes rapid changes in particle velocities and, therefore, a higher probability that events will expire before they can be used in the multiple-event method. The resulting increase in overhead created by multiple deletions from the binary tree degrades the performance of the multiple-event algorithm for molecular fluids.

Figure 7 compares the performance of the single- and multiple-event scheduling codes for hard-sphere fluids measured in millions of events per CPU hour as a function of the number of particles N. System sizes for the hard-sphere fluids are the same as for the chain fluids in Fig. 6. The advantage that single-event scheduling offers for chain fluids is diminished for hard-sphere fluids. The two methods have similar performance because additional events scheduled in the multiple-event method provide enough savings in particle interaction calculations to offset the overhead associated with extra event deletions at each time step. For comparison, results for the code `F.10` from Allen and Tildesley [6] are also shown. The `F.10` algorithm implements a linear time list for single-event scheduling and no neighbor lists. The combination of neighbor lists, link lists, and a binary tree provides almost an order of magnitude increase in performance over the straightforward DMD implementation of `F.10`. In summary, the single-event algorithm is best suited for systems with low diffusivity where rapid velocity changes occur such as in the Rapaport chain model. The multiple-event method is best suited for systems with high diffusivity where considerable displacement may occur before particle velocities change.

Figure 8 compares the performance of the single- and multiple-event scheduling codes for chain fluids, measured in millions of events per CPU hour, as a function of the number of chain segments N. These benchmarks were performed on a SGI Challenge XL with a R4400 CPU operating at 200 MHz using 64-bit arithmetic. This machine is equipped with 32K of on-chip cache and 4MB of secondary cache. In all cases the single-event scheduling is faster than multiple-event scheduling with an increase from 16% to 25%. The DEC workstation has better performance for small systems ($N \leq 2048$) while the SGI computer has better performance for the larger systems. The cache and memory structures on the two machines may account for

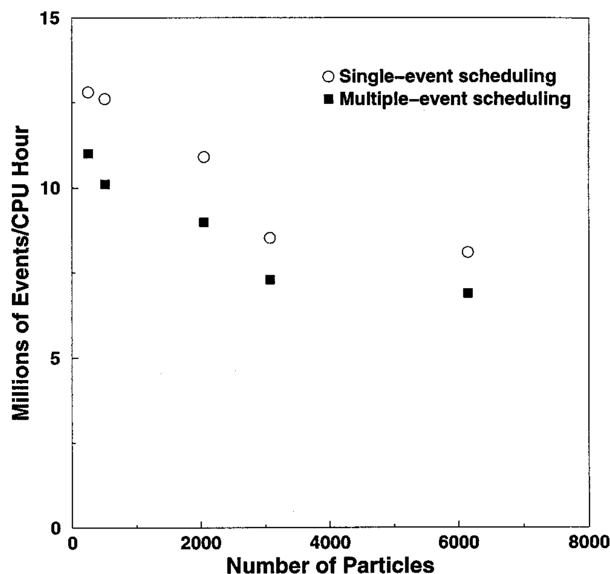

FIG. 8. Performance comparison between single-event and multiple-event scheduling algorithms for chain fluids at a volume fraction of 0.45. All runs were performed on a SGI Challenge XL computer. In each case there are 32 chains while the chain length varies from 4 to 192.

the difference in performance of the larger systems, noting that the SGI computer has a much larger secondary cache.

VI. EXAMPLES

Recent advances in the speed of serial processors coupled with low cost workstations makes the DMD method attractive for problems involving large time scales. Multi-billion time step simulations can be routinely carried out within 100 h of dedicated CPU time on a modest workstation. The following sections highlight the utility of the DMD method in general and the single-event algorithm in particular. The examples focus on current areas of interest in polymer physics concerning transport and thermodynamic properties. In many instances, the DMD algorithm is faster than continuous-space Monte Carlo techniques for a given degree of accuracy. An efficient DMD algorithm along with high speed serial computers permits multi-billion time step simulations which are about two orders of magnitude longer than the corresponding simulation of continuous potential models on vector computers.

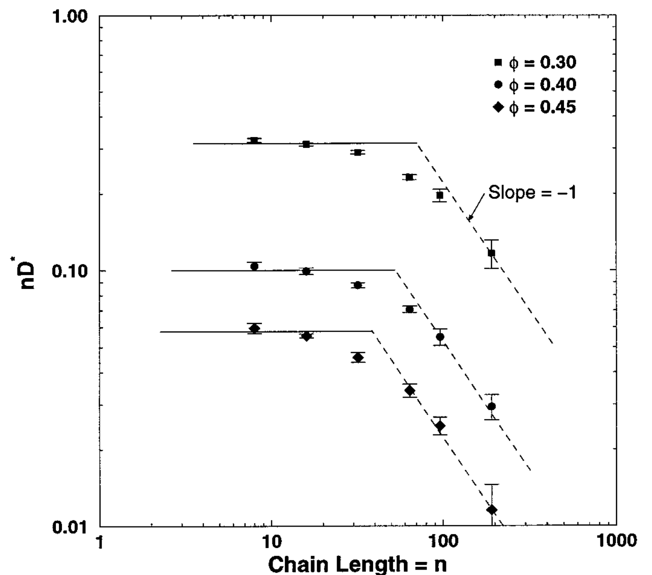

FIG. 9. Simulation results for the reduced self-diffusion coefficient scaled by chain length versus chain length for three volume fractions. Error bars represent one standard deviation.

A. *Dynamics of Entangled Chains*

Relative to simple Newtonian liquids, entangled polymer melts contain a rich variety of topological and excluded-volume interactions which create a spectrum of relaxation times. Polymer dynamics are most often discussed in terms of two models: (1) Rouse [22] dynamics for short, un-entangled chains, and (2) reptation [23] dynamics for highly entangled chains. In the Rouse model, the fluid surrounding a chain relaxes rapidly providing a stochastic background for a chain's motion. In the reptation model, the surrounding fluid relaxes on an infinite time scale, thus presenting fixed obstacles to a chain's motion.

The tube model of Doi and Edwards [24] is perhaps the most prominent implementation of the reptation concept. In this approach, topological interactions are treated as an effective field of obstacles forming a virtual tube that restricts chain motion. Atomic displacements larger than the characteristic dimension imposed by this field, the so-called tube diameter, occur predominantly along the chain contour. For displacements smaller than the tube diameter, segment motion is restricted only by chain connectivity and should be characterized by Rouse dynamics. Since the longest relaxation time in the Rouse model is proportional to the chain length squared, the tube model predicts that the self-diffusion, D, and shear viscosity, η, of short chains scale with molecular weight as $D \sim n^{-1}$ and $\eta \sim n$ in agreement with experiments on short chain melts [25]. When chain length exceeds a critical value, the so-called "entanglement" length, the tube model predicts that the molecular weight scaling for self-diffusion and shear viscosity changes to $D \sim n^{-2}$ and $\eta \sim n^3$. For entangled polymers, the experimentally observed molecular weight dependencies are $D \sim n^{-2}$ and $\eta \sim n^{3.4}$.

The single-event scheduling algorithm has been used to investigate the scaling of the diffusion coefficient for entangled chains [26, 27]. Simulations were performed on systems containing 32 chains of length $n = 8, 16, 32, 64, 96, 192$ at volume fractions ranging from 0.3 to 0.45. Self-diffusion coefficients were calculated in the molecular (center of mass) reference frame. Long simulation runs are essential to probe the slow relaxations which are imposed by topological constraints for entangled chain fluids. Simulation lengths ranged from 10^8 collisions for the 8-mer fluid to 2×10^{10} collisions for the 192-mer fluid; this is about two orders of magnitude longer in terms of collisions than any previously reported DMD simulation. These simulations were performed on a local cluster of low-cost DEC workstations in less than 3000 CPU hours for a single density. Furthermore, a workstation cluster allowed multiple chain lengths (six total) and densities (three total) to be studied simultaneously.

Figure 9 shows the scaling of the reduced self-diffusion coefficient with chain length. We observe a crossover from Rouse behavior ($D \sim n^{-1}$, solid line) to entangled behavior ($D \sim n^{-2}$, dashed line). The critical chain lengths at which crossover occurs decreases with increasing volume fraction. The data indicate an entanglement length which is less than 50 for both the 0.4 and 0.45 volume fractions in agreement with results for a Lennard-Jones based model [28]. These results suggest a scaling of $D \sim n^{-2}$ for the longer chains.

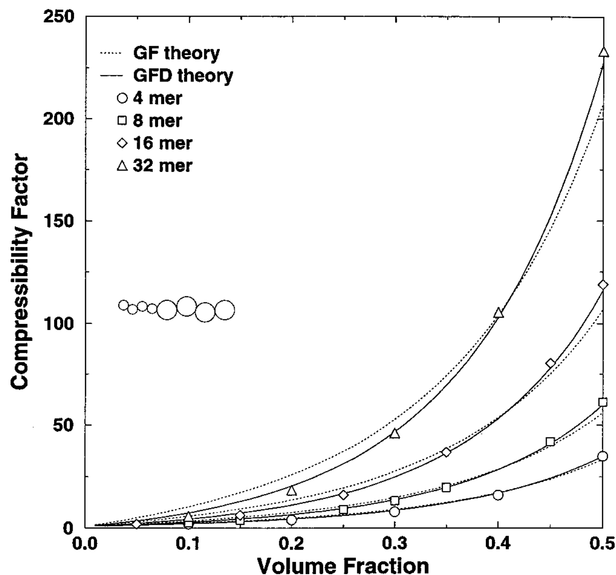

FIG. 10. Block copolymer compressibility factor vs packing fraction for 4, 8, 16, and 32 mers. Each block copolymer consists of a sequence of component A followed by component B with a diameter ratio (σ_A/σ_B) of 2 while the fraction of component A on the chains is 0.5. Points are the results of DMD simulations and the curves are the predictions of the GF and GFD equations of state. Error bars are smaller than the data point.

B. *Heteronuclear Chains*

Copolymers are heteronuclear chains composed of two or more distinct monomer types which differ in either size or strength of molecular interactions. In order to understand the molecular basis for the macroscopic behavior of such fluids, heteronuclear polymeric fluids are often modeled as fluids containing chains of tangent hard spheres of different sizes. Simulation studies can be used to test the accuracy of theoretical predictions based on this model.

Gulati *et al.* [29] have applied the DMD method to calculate the compressibility factors of fluids containing block, alternating, and random copolymers modeled as chains of tangent hard spheres of different sizes. The algorithm used is a modification of the DMD method described earlier in which segments of different sizes have masses proportional to their volume. The equations of motion are modified appropriately to incorporate the mass and size dependence. Compressibility factors are calculated from the collisional virial. This method is significantly faster and more accurate than its MC counterpart for calculating compressibility factors.

The model copolymer systems studied have block, alternating, and random geometries with segment diameter ratios ranging from 0.5–4.0 and varying chain lengths. Each of the systems is simulated over a range of volume fractions from 0.09–0.5. Figure 10 shows a comparison between the calculated compressibility factors Z and those predicted by the GF and GFD theories [29]. The compressibility factors calculated using the DMD method agree well with the compressibility factors calculated from MC simulation [31] and with the generalized Flory–Dimer theory prediction. The advantages of the DMD method over the MC method are: (1) a higher accuracy in Z is obtained for a given CPU time allocation and (2) the DMD method does not contain uncertainties in the system density.

C. *Double-Tethered Hard Chains at Interfaces*

A thin film can be formed on a surface by grafting the ends of polymer molecules to the surface. Such thin polymeric films are of interest in technological applications requiring precise control over adhesion and stabilization of surfaces. The conformational and dynamic properties of single-tethered chains (one end grafted to the surface) have been investigated in numerous theoretical [32–34] and computer simulation studies [35–37]. The conformational properties of double-tethered chains (both ends grafted to the surface) have been investigated by Monte Carlo simulations [38, 39] using chains confined to a cubic lattice. More recently, Gulati *et al.* [40] have used the single-event DMD algorithm to study an off-lattice model of double-tethered hard chain fluids in the presence of an impenetrable wall.

The algorithm used to simulate double-tethered chains is essentially the same as the DMD method described earlier, with slight modifications. The primary simulation box consists of an infinite parallelepiped in the z-direction with a hard surface at $z = 0$. Periodic boundaries are employed along the x and y directions but not along the z direction. For the double-tethered case, each chain end (N_a total) is anchored to the hard surface. Anchors are attached by a small string in accord with the Rapaport model. The effect of surface anchor density ($\rho_a = N_a/A$), anchor mobility, and chain length (n) on the structural, conformational, and dynamic properties of systems containing chains of 20, 40, and 80 segments at surface anchor densities ranging from 0.01–0.48 is studied. The algorithm generated approximately 18 million collisions per CPU hour on a DEC 3000/300LX for a system containing 500 segments.

The segment density profiles away from the wall are parabolic at intermediate densities and agree with the self-consistent field [41, 32] theories for polymer brushes. However, at high surface densities and short chain lengths, the segment density profiles oscillate indicating the presence of a layered structure. This phenomenon is not as prominent in lattice simulations where excluded volume interactions cannot be treated explicitly. From the density profiles, chains are stretched normal to the surface at high density and have an inverted mushroom shape at low density.

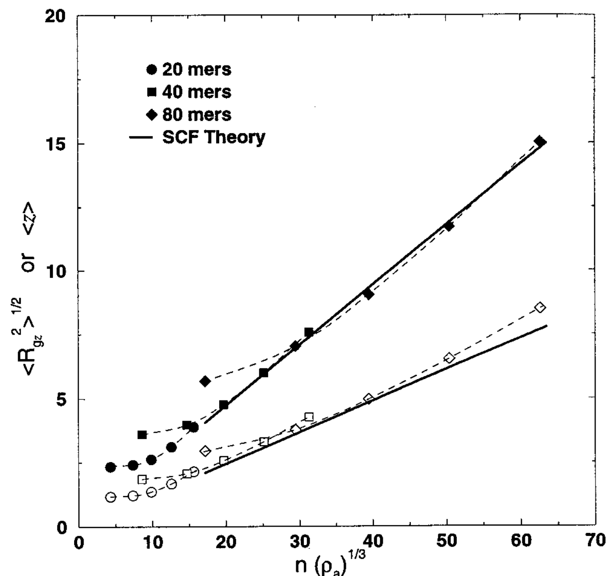

FIG. 11. Average thickness $\langle z \rangle$ (filled symbols) and z-component of the radius of gyration $\langle R_{Gz}^2 \rangle^{1/2}$ (open symbols) versus scaled chain length $n\rho_a^{1/3}$ for double-tethered chains. Prediction of self-consistent field (SCF) theory are also shown.

From self-consistent field theory, the average thickness of the surface layer (defined as the first moment of the density profile $\langle z \rangle$) should scale with the reduced chain length $n\rho_a^{1/3}$ in the limit of high surface density. Figure 11 illustrates this scaling by plotting $\langle z \rangle$ along with the z component of the radius of gyration $\langle R_{Gz}^2 \rangle^{1/2}$, as a function of $n\rho_a^{1/3}$. As $n\rho_a^{1/3}$ increases, each of the scaled quantities converge to a common line in agreement with the scaling predictions.

VII. CONCLUSIONS

The optimization and efficiency methods described in this paper should provide a basis for extending the DMD method to a variety of problems especially in the area of polymer physics. We have discussed and compared two approaches for handling the event-scheduling problem in DMD. The single-event scheduling technique is more efficient than the multiple-event scheduling technique especially when a short time event such as a bond stretch is present in the molecular model. We have also shown the feasibility of studying long-time dynamics on the latest generation of workstations. Through applications to a variety of chain models we have illustrated the utility of the DMD method for studying both thermodynamic and transport properties of polymeric models. Even after four decades of evolution and refinement in the techniques of molecular simulation, the simple discontinuous potential model is capable of providing a dependable foundation for theoretical treatment. Regardless of the application, the computational scientist will always desire to study larger systems and longer times, both of which are facilitated by discontinuous molecular dynamics.

ACKNOWLEDGMENTS

We thank R. Elliot of the University of Akron for helpful discussions on algorithm design and for providing a sample code for Rapaport's binary tree. We appreciate the contributions by H. Gulati of his results on heteronuclear and tethered chain models. This work was supported in part by the Computational Science Graduate Fellowship Program of the Office of Scientific Computing in the Department of Energy. This work was also supported by the Director, Office of Basic Sciences, Chemical Sciences Division of the U.S. Department of Energy under Contract DE-FG05-91ER14181. BDF gratefully acknowledges support through the Young Investigator Award of the National Science Foundation (CTS-9257911), and the Ford Motor Company. Acknowledgment is made to Cray Research Institute and the North Carolina Supercomputing Center for grants of computing time.

REFERENCES

1. D. C. Rapaport, *J. Phys. A: Math. Gen.* **11,** L213 (1978).
2. D. C. Rapaport, *J. Chem. Phys.* **71,** 3299 (1979).
3. B. J. Alder and T. E. Wainwright, in *Symposium on Transport Processes in Statistical Mechanics*, edited by I. Prigogine (Interscience, New York, 1956).
4. B. J. Alder and T. E. Wainwright, *J. Chem. Phys.* **31,** 459 (1959).
5. B. J. Alder and T. E. Wainwright, *J. Chem. Phys.* **33,** 1439 (1960).
6. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon Press, Oxford, 1987).
7. J. M. Haile, *Molecular Dynamics Simulation—Elementary Methods* (Wiley, New York, 1992).
8. M. A. Denlinger and C. K. Hall, *Mol. Phys.* **71,** 541 (1990).
9. A. Bellemans, J. Orban, and D. V. Belle, *Mol. Phys.* **39,** 781 (1980).
10. G. A. Chapela and S. E. Martinez-Casas, *Mol. Phys.* **53,** 139 (1984).
11. G. A. Chapela and S. E. Martinez-Casas, *Mol. Phys.* **59,** 1113 (1986).
12. J. J. Erpenbeck and W. W. Wood, in *Modern Theoretical Chemistry*, edited by B. J. Berne (Plenum, New York, 1977).
13. L. Verlet, *Phys. Rev.* **159,** 98 (1967).
14. B. Quentrec and C. Brot, *J. Comput. Phys.* **13,** 430 (1975).
15. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw–Hill, New York, 1981).
16. D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming* (Addison–Wesley, Reading, MA, 1968).
17. D. E. Knuth, *Sorting and Searching, The Art of Computer Programming* (Addison–Wesley, Reading, MA, 1973).
18. D. C. Rapaport, *J. Comput. Phys* **34,** 184 (1980).
19. W. G. Hoover and B. J. Alder, *J. Chem. Phys.* **46,** 686 (1967).
20. W. H. Press et al., *Numerical Recipes: The Art of Scientific Computing* (Cambridge Univ. Press, Cambridge, 1989).
21. M. P. Allen, *Mol. Simul.* **3,** 251 (1989).
22. P. E. Rouse, *J. Chem. Phys.* **21,** 1272 (1953).
23. P. G. de Gennes, *Scaling Concepts in Polymer Physics* (Cornell Univ. Press, Ithaca, NY, 1979).
24. M. Doi and S. F. Edwards, *J. Chem. Soc. Faraday Trans. II* **74,** 1789 (1978).
25. G. C. Berry and T. G. Fox, *Adv. Polym. Sci.* **5,** 261 (1968)

26. S. W. Smith, C. K. Hall, and B. D. Freeman, *Phys. Rev. Lett.* **75,** 1316 (1995).
27. S. W. Smith, C. K. Hall, and B. D. Freeman, *J. Chem. Phys.* **104,** (1996).
28. K. Kremer and G. S. Grest, *J. Chem. Phys.* **92,** 5057 (1990).
29. H. S. Gulati, J. M. Wichert, and C. K. Hall, *J. Chem. Phys.* **104,** 5220 (1996).
30. Deleted in proof.
31. J. M. Wichert and C. K. Hall, *Chem. Eng. Sci.* **17,** 2793 (1994).
32. S. T. Milner, T. A. Witten, and M. E. Cates, *Macromolecules* **21,** 2610 (1988).
33. S. T. Milner, T. A. Witten, and M. Cates, *Macromolecules* **22,** 853 (1989).
34. E. B. Zhulina, O. V. Borisov, and V. A. Priamitsyn, *J. Colloid Interface Sci.* **137,** 495 (1990).
35. M. Murat and G. S. Grest, *Macromolecules* **22,** 4054 (1989).
36. A. Chakrabarti and R. Toral, *Macromolecules* **23,** 2016 (1990).
37. P.-Y. Lai and K. Binder, *J. Chem. Phys.* **95,** 9288 (1991).
38. R. L. Jones and R. L. Spontak, *J. Chem. Phys.* **101,** 5179 (1994).
39. R. L. Jones and R. L. Spontak, *J. Chem. Phys.* **103,** 5137 (1995).
40. H. S. Gulati, R. L. Jones, R. J. Spontak, and C. K. Hall, *J. Chem. Phys.* **105,** 7712 (1996).
41. P. G. D. Gennes, *Macromolecules* **13,** 1070 (1980).