Parallel Discrete Event Simulation of Molecular Dynamics Through Event-Based Decomposition

Martin C. Herbordt   Md. Ashfaquzzaman Khan   Tony Dean
Department of Electrical and Computer Engineering
Boston University; Boston, MA 02215

Abstract—**Molecular dynamics simulation based on discrete event simulation (DMD) is emerging as an alternative to time-step driven molecular dynamics (MD). DMD uses simplified discretized models, enabling simulations to be advanced by event, with a resulting performance increase of several orders of magnitude. Even so, DMD is compute bound. Moreover, unlike MD, causality issues make DMD difficult to scale. Here we present a microarchitecture-inspired parallel algorithm for DMD: speculative execution enables multithreading, while in-order commitment ensures correctness. Our initial not-yet optimized implementation obtains scalability for a multicore processor when running realistic simulation models.**

I. Introduction

Computer simulation is a fundamental tool for exploring biomolecular conformations at scales inaccessible to conventional "wet-lab" experiments. Prevalent simulation techniques are based primarily on time-step driven molecular dynamics (MD). While MD has generated impressive results, it inherently falls short—by several orders of magnitude—of being able to model many important biological phenomena. These include, e.g., the protein association and aggregation subsequent to misfolding that is integral to many disease processes [3], [16]. Orders of magnitude larger still are the scales in space and time required to model nuclear- and cell-level systems.

An emerging alternative is molecular dynamics based on discrete event simulation, referred to as discrete molecular dynamics, or DMD [2], [3]. DMD uses simplified models: atoms as hard spheres, covalent bonds as infinite barriers, and van der Waals forces as square wells. This discretization enables simulations to be advanced by event, rather than time step. Events occur when two particles cross a discontinuity in interparticle potential. The result is simulations that are up to 10^8 to 10^9 times faster than traditional MD, with $10^6\times$ being regularly achieved [3]. The simplified model can be substantially compensated for by the capability of researchers to interactively refine simulation models [17].

Even so, current DMD simulations are also compute bound, sometimes taking a month or more (e.g., [14]), although with far less resources than are used for high-end MD simulations. In fact, a major problem with DMD is that, as with discrete event simulation (DES) in general [4], causality concerns make DMD difficult to scale to a significant number of processors [9]. While the parallelization of DMD has been well-studied (see, e.g., [9], [15] and much other work), we are aware of no existing production parallel DMD (PDMD) codes.

The difficulty in parallelizing DMD (as, in general, in parallel discrete event simulation or PDES) is that dependencies arise unpredictably and virtually instantaneously. In some PDES application domains, it is possible to circumvent this by predicting a window during which event execution is "safe" (conservative execution) or by making a similar assumption to ensure that the amount of work that may need to be undone is limited (optimistic execution) [4]. DMD, however, is chaotic: there is no safe window [7].

Our approach is motivated by the following observations.

- A recent advance in the DMD queueing structure has reduced the complexity of the most time-consuming operations from $O(\log N)$ to $O(1)$ [11]. This has mitigated the previously substantial advantage of reducing the size of the event queue, either by distribution or any other means.
- Previous PDMD work has been based on spatial decomposition. While workable in one and two dimensions, 3D simulation is far more complex. This requires, for cubic decomposition, that each thread exchange information with a large number of neighbors (27) for potential conflicts. Or, if decomposition is done by slices, then it must handle a drastic increase in the ratio of surface area to volume and so the number of interactions per thread-pair.
- Many of the successful DMD implementations were reported more than a decade ago (and for 2D). Since then, event processing speeds have increased dramatically (through advances in both processors and algorithms), especially when contrasted with interprocessor communication latency. Event processing times are now on the order of a single message or twenty cache misses.

We therefore parallelize using event-based-, rather than spatial-decomposition. Overall, our method is based on an approach that has proved successful in hardware [5], [10]: *parallelize through deeply pipelined execution, but maintain serial commitment.* In software, this translates to there being a single centralized queue. Multiple threads get events from the queue and process them speculatively and in parallel.

This work was supported in part by the NIH through award #R01-RR023168-01 and by IBM through a Faculty Award; and facilitated by donations from Altera Corporation. Email: {herbordt|azkhan}@bu.edu, willam.dean2@gdc4s.com. Web: http://www.bu.edu/caadlab.
Currently with General Dynamics C4 Systems

Various types of hazards are checked by using shared data structures, and event processing is delayed or cancelled as necessary. But as with the hardware implementation (and with CPUs with speculative execution), in-order commitment assures correctness.

Although we have only begun exploring the ways in which this method can be optimized, we have already demonstrated speed-up for realistic simulation models with up to four processors on a multicore CPU. Overall, our contributions are as follows.

- A working parallel DMD system with a novel design that both achieves speed-up and can be extended for use in production applications.
- Enumeration of the issues in pipelined DMD processing. This includes defining causality and coherence hazards and specifying how to deal with them.
- Experiments with known optimizations and findings, e.g., that the insertion policies of Rapaport and Lubachevsky have similar performance with current technology.
- Experiments with new optimizations. These involve modifications in the data structures, synchronization granularity, and decomposition. They will be reported upon in the final version of this manuscript.

The rest of this paper is organized as follows. In the next section we summarize discrete event simulation and its application to molecular dynamics. We concentrate on the event queue and present in some detail a version that has not previously been integrated into PDMD. We follow with our PDMD design, starting with a discussion of DMD hazards in general, how we deal with them conceptually, and with our system. Then comes a description of our experiments: set-up, validation, and results. We end with a discussion of likely optimizations and sketch of future work.

II. Discrete Molecular Dynamics

A. DES/DMD Overview

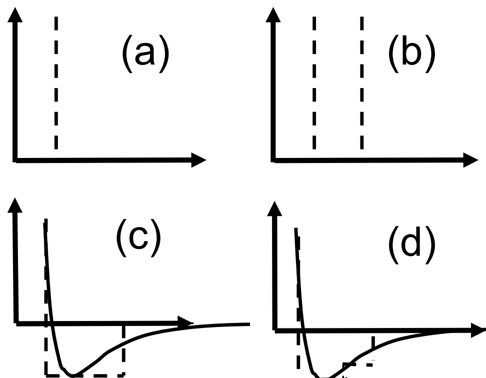

Fig. 1. DMD force models: Force versus interparticle distance for a) hard spheres, b) hard spheres and hard chain link, c) simple LJ square well, and d) LJ with multiple square wells.

MD is the iterative application of Newton's laws to ensembles of particles. It is transformed into DMD by simplifying the forces: all interactions are folded into step-wise potential models. Figure 1a shows the infinite barrier used to model a hard sphere; Figure 1b an infinite well for a covalent bond (hard chain); and Figures 1c and 1d the van der Waals model used in MD, together with square well and multi-square-well approximations, respectively. It is through this simplification of forces that the computation mode shifts from time-step-driven to event-driven.

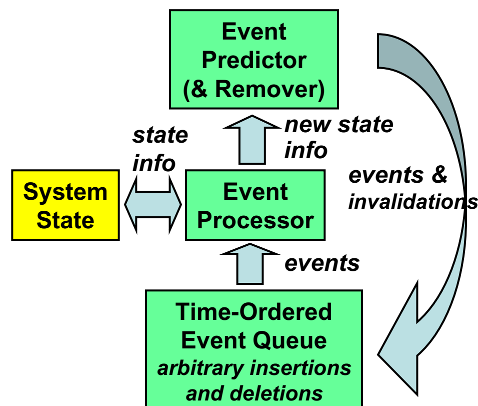

Fig. 2. Block diagram of a generic discrete event simulator.

Overviews of DMD can be found in many standard MD references, particularly Rapaport [13]. A DMD system follows the standard DES configuration (see Figure 2) and consists of the

- **System State**, which contains the particle characteristics: velocity, position, time of last update, and type;
- **Event Predictor**, which transforms the particle characteristics into pairwise interactions (events);
- **Event Processor**, which turns the events back into particle characteristics; and
- **Event Priority Queue**, which holds events waiting to be processed ordered by time-stamp.

Execution proceeds as follows. After initialization, the next event (involving, say, particles a and b) is popped off the queue and processed. Then, previously predicted events involving a and b, which are now no longer valid, are removed from the queue. Finally, new events involving a and b are predicted and inserted into the queue.

To bound the complexity of event prediction, the simulated space is subdivided into cells (as in MD). Since both the number of particles per cell (typically an average of one) and the number of cells in a neighborhood (27) are fixed, the number of predictions per event is also bounded and independent of the number of particles. One complication of using cells in DMD is that, since there is no system-wide clock advance during which cell lists can be updated, bookkeeping is facilitated by treating cell crossings as events and processing them explicitly.

One issue that has received much attention is how many of the newly predicted events to insert into the event queue. The original algorithm from Rapaport inserts all predicted events. Lubachevsky's method (see [7] and numerous variations) keeps only a single event per particle. The reduced queue size, however, comes at a cost: whenever the sole event involving a particle is invalidated, then events for that particle must be repredicted. There is thus a trade-off between the processing required to update the queue and that required for reprediction. While Lobachevsky's and related methods were long preferred [6], we show below that—with advances in processor technology and with new queue structures (i.e., by [11])—the balance appears to have shifted back to Rapaport.

B. Software Priority Queues

The basic operations for the priority queue are as follows: dequeue the event with the highest priority (smallest time-stamp), insert newly predicted events, and delete events in the queue that have been invalidated. A fourth operation can also be necessary: advancing, or otherwise maintaining, the queue to enable the efficient execution of the other three operations. The data structures typically are

- an array of bead records, indexed by bead ID;
- an event priority queue; and
- a series of linked lists, at least one per bead, with the elements of each (unordered) list consisting of all the events in the queue associated with that particular bead (see, *e.g.,* [13]).

Implementation of priority queues for DMD is discussed by Paul [11]; they "have for the most part been based on various types of binary trees," and "all share the property that determining the event in the queue with the smallest value requires $O(\log N)$ time [8]." Using these structures, the basic operations are performed as follows.

1. Dequeue: The tree is often organized so that for any node the left-hand descendants are events scheduled to occur before the event at the current node, while the right-hand descendants are scheduled to occur after it [13]. The event with highest priority is then the left-most leaf node. This dequeue operation is therefore either $O(1)$ or $O(\log N)$ depending on bookkeeping.

2. Insert: Since the tree is ordered by tag, insertion is $O(\log N)$.

3. Delete: When an event involving particles a and b is processed, all other events in the queue involving a and b must be invalidated and their records removed. This is done by traversing the beads' linked lists and removing events both from those lists and the priority queue. Deleting all events invalidated by a particular event is $O(1)$ on average as each bead has an average of slightly less than two events queued, independent of simulation size.

4. Advance/Maintain: Binary trees are commonly adjusted to maintain their shape. This is to prevent their (possible) degeneration into a list and so a degradation of performance from $O(\log N)$ to $O(N)$. With DMD, however, it has been shown empirically by Rapaport [12] and verified by us elsewhere, that event insertions are nearly randomly (and uniformly) distributed with respect to the events already in the queue. The tree shape is therefore maintained without rebalancing, although the average access depth is slightly higher than the minimum.

C. Paul's Algorithm

In this subsection we summarize work by G. Paul [11] which leads to a reduction in asymptotic complexity of priority queue operations from $O(\log N)$ to $O(1)$, and a substantial benefit in realized performance.

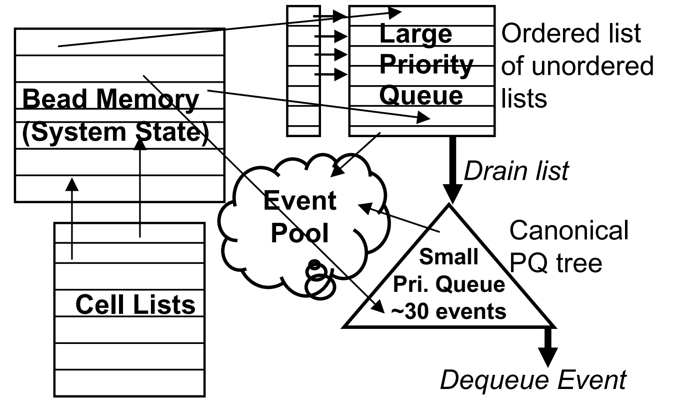

Fig. 3. DMD data structures with Paul queue.

The observation is that most of the $O(\log N)$ complexity of the priority queue operations is derived from the continual accesses of events that are predicted to occur far in the future. The idea is to partition the priority queue into two structures. This is shown in Figure 3, along with most of the other major data structures. A small number of events at the head of the queue, say 30, are stored in a fully ordered binary tree as before, while the rest of the events are stored in an ordered list of small unordered lists. Also retained are the bead (particle) memory and the per-particle linked lists of events that are used for invalidates.

To facilitate further explanation, let T_{last} be the time of the last event removed from the queue and T be the time of the event to be added to the queue. Each of these unordered lists contains exactly those events predicted to occur within its own interval of $T_i \ldots T_i + \Delta t$ where Δt is fixed for all lists. That is, the ith list contains the events predicted to occur between $(T - T_{last}) = i * \Delta t$ and $(T - T_{last}) = (i + 1) * \Delta t$. The interval Δt is chosen so that the tree never contains more than a small number of events.

Using these structures, the basic operations are performed as follows.

1. Dequeue: While the tree is not empty, operation is as before. If the tree is empty, a new ordered binary tree is created from the list at the head of the ordered array of lists.

2. Insert: For $(T - T_{last}) < \Delta t$, the event is inserted into the

tree as before. Otherwise, the event is appended to the ith list, where $\lfloor i = (T - T_{last})/\Delta t \rfloor$.

3. Delete: Analogous to before.

4. Advance/Maintain: The array of lists is constructed as a circular array. Steady state is maintained by continuously "draining" the next list in the ordered array of lists whenever a tree is depleted.

For the number of lists to be finite there must exist a constant T_{max} such that for all T, $(T - T_{last}) < T_{max}$. In practice, most of the lists are small until they are about to be used.

Performance depends on tuning Δt. The smaller Δt, the smaller the tree at the head of the queue, but the more frequent the draining and the larger the number of lists. For serial implementations, Paul finds that choosing Δt so that the lists are generally < 20 maximizes performance. The number of lists can be bounded through methods described elsewhere.

III. Parallel DMD Design

A. PDMD Hazards

Parallelizing DMD presents certain difficulties. Given events E_{ex}, E_{pre}, and E_{can} where:

- E_{ex} is the event at the head of the queue being processed at time t,
- E_{pre} is an event predicted due to E_{ex},
- E_{can} is an event cancelled due to E_{ex}.

Then

- E_{pre} can be inserted at any position in the event queue, including the head,
- E_{can} can be at any position in the event queue, including the head, and
- another event E caused by E_{ex} (perhaps indirectly through a cascade of intermediate events) can occur at time $t + \epsilon$ after E_{ex} where ϵ is arbitrarily small and *at any distance from E_{ex} in the simulation space.*

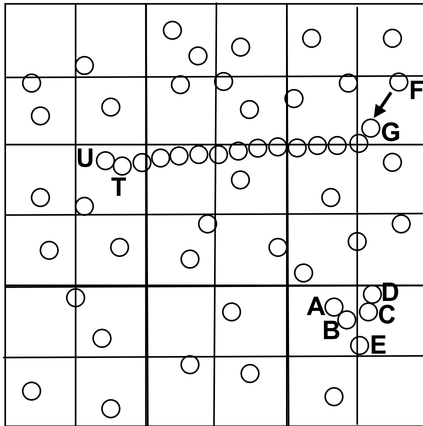

Fig. 4. Events AB and CD cause BC and cancel BE. Event FG causes TU almost instantly and at long distance.

Examples of these occurrences are shown in Figure 4. In the lower part, events AB and CD occur at times t_0 and $t_{0+\epsilon}$. Previously predicted BE gets cancelled, even though it is the event currently at the head of the queue. Newly predicted event BC will happen almost immediately and so gets inserted at the head of the queue. The upper part of Figure 4 shows how causality can propagate over a long distance. After FG, a cascade of events causes TU to happen almost instantly and on the other side of the simulation space. Although "long-distance" events such as in Figure 4 may appear to be rare, they are actually fundamental to polymer simulations: the polymer forms a chain with rigid links. A force applied to one end—say, by an atomic force microscope that is unravelling a protein—creates exactly such a scenario.

These conditions introduce hazards into potential concurrent processing of events. In each of the following cases, let E_1 and E_2 be the events in the processing queue with the lowest time-stamps.

Causality. If E_1 and E_2 are processed concurrently, there is potential for a causality violation. E_1 can cause E_2 to be invalid, either directly, or through a cascade of new events inserted into the event queue with time-stamps between those of E_1 and E_2.

Coherence. Again let E_1 and E_2 be processed concurrently. Even if both events execute without causality hazard, there may be a coherence hazard during event prediction. For example, a particle taking part in E_2 may be predicted to collide with a particle taking part in E_1, but only in the now stale system state prior to update due to E_1.

Causality and Coherence. A new event E_{new} caused by E_1 may be inserted into the queue ahead of E_2 and not invalidate E_2, but still result in a coherence hazard. That is, E_{new} could change the state used in E_2's prediction phase, or *vice versa*.

B. A Pipelined Event Processor

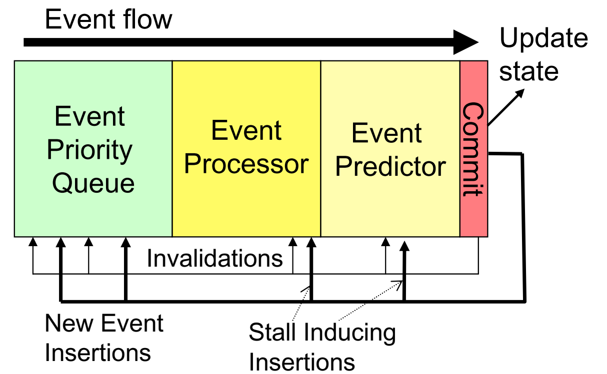

Fig. 5. DMD with a dedicated pipelined event processor. Not to scale – the event queue is several orders of magnitude larger than the processing stages even for modest simulations.

The main idea in our design is to process DMD in a single pipeline (as shown in Figure 5). That is, while a large number of events can be processed simultaneously, at most one event at a time is *committed.* Viewed another way, this

design is of a microarchitecture that processes events rather than instructions: the logic is analogous to that used in modern high-end CPUs for speculative instruction execution.

Commitment consists of the following steps: (i) update the system state, (ii) process all causal event cancellations and (iii) new event insertions, and (iv) advance the event priority queue. As in a CPU, dependences (this time among events rather than instructions) combined with overlapped executions cause hazards. And as in a CPU, these are compounded by speculation.

Causality hazards. A new event can be inserted anywhere in the pipeline, including the "processing" stages. But this cannot be allowed because then it will have skipped some of its required computation. Insertion at the beginning of the processing stages, however, results in out-of-order execution which allows causality hazards. A solution is to insert the event at the beginning of the processing stages, but to pause the rest of the pipeline until the event finds the correct slot. This results in little performance loss for simulations of more then a few hundred particles.

Coherence hazards. After an event E completes its processing, it begins prediction. There will be several events ahead of E, however, none of which has yet committed, but which will change the state when they do. This has the potential to make E's prediction incorrect because it may be made with respect to stale data (coherence hazard). One solution begins with the observation that E is predicting events only in its 27 cell neighborhood. It checks the positions of the events ahead of it in the predictor stages, an operation we call a neighborhood check, or *hood-check* for short. If the neighborhood is clear (*hood-safe*), then E proceeds, otherwise it waits. This results in substantially more performance loss than that due to causality hazards, but it is still not large for simulation spaces of 32^3 and greater.

Combined causality and coherence hazards. An event E can be inserted ahead of events that have already begun prediction assuming they were hood-safe. As before, E must be inserted at the beginning of the processing stages. The added complication is that events in the predictor stages with time-stamps greater than E must restart their predictions. Since the probability of such insertions is small, this causes little additional overhead.

C. PDMD Design

The conceptual hardware pipeline from Figure 5 is implemented in software as shown in Figure 6 (the queue is taken from Figure 3). It shows:

- Each event in the processing FIFO is handled by its own thread — the more threads, the longer (potentially) the FIFO.
- The FIFO is ordered by time-stamp to facilitate handling of hazards, but processing is not otherwise constrained (as it is in the microarchitecture shown in Figure 5).
- Events are committed serially and in order.

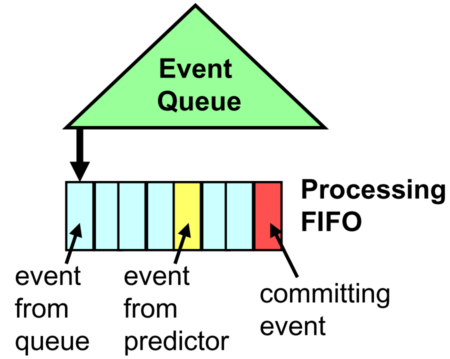

Fig. 6. Parallel DMD implemented with a processing FIFO. Events can be inserted from the event queue or directly from an event prediction.

- Events can be added to the FIFO in two ways. They can be dequeued from the event queue and appended to the back of the FIFO. Or they can be inserted directly from the predictor.
- Events appended to the back of the FIFO have the potential for coherence hazards. Events inserted anywhere else in the FIFO have the potential for both coherence and causality hazards.

Recall the basic data structures in DMD (shown for the most part in Figure 3):

- **System state** is an array of particle records each containing motion parameters and the time-stamp from last update;
- **Cell record array** with each record containing the IDs of its beads;
- **Event queue** is a priority queue of event records each containing the time-stamp, event type, and IDs of the particles involved; and
- **Array of per particle event lists** is a series of linked lists, one for each particle, connecting all events involving that particle (and used for invalidations).

There are three types of actions whose execution overlaps:

1) **Event execution,** including processing the event, predicting new events, and updating the system state and cell lists;
2) **Priority queue operations,** including dequeue/advance, insertions, and deletions; and
3) **Structure updates,** including the explicit handling of preallocated structures such as event records.

With the hierarchical priority queue, dequeue sometimes removes the last event from the fully ordered tree. This causes the next list (in the ordered list of unordered lists) to be "drained" and a new tree to be constructed. For PDMD these action-types interact in subtle ways making careful design essential.

Basic processing (no hazards) occurs as follows.

- A thread T which is looking for work dequeues the next event E from the queue, appends E to the FIFO, processes E, and predicts the new events resulting from E.
- T waits until E reaches the head of the FIFO at which time it commits E.
- Let E involve particles a and b. Commitment consists of updating the system state for a and b, invalidating events in the queue involving a and b, returning the event structure to the event pool, and inserting the newly predicted events into the queue.

Hazards are handled as follows.

- **Coherence Hazards.** When T dequeues E from the event queue, T must first perform a hood-check with respect to the events already in the queue. If E is not hood-safe, then E must wait.
- **Causality Hazards.** A causality hazard occurs when T inserts a newly predicted event E into the Processing FIFO, that is, ahead of events that have already begun processing. Unlike in pipelined DMD, there is no need to necessarily restart processing of any other events. It *is* necessary, however, for T to ensure that E is hood-safe by performing a hood-check with respect to events ahead in the FIFO.
- **Combined Causality/Coherence Hazards.** These occur when E's insertion into the FIFO causes an event prediction that has already begun to now be operating on stale data. This is handled as follows. When E is inserted, a "reverse" hood-check must be executed. That is, events behind E in the FIFO must be confirmed to be hood-safe. If, say, E' is determined not to be hood-safe, then the `HoodCheck` flag is set in E''s record indicating that prediction (and hood-check) must be repeated.

Several special cases and optimizations emerge.

- **Deleting events from the FIFO.** Let event E' in the FIFO be processed by thread T', and let T need to delete E' (as a consequence of committing E). One option is to kill T'; preferred, however, is to simply set a `Zombie` flag. T' continues processing, but before commitment checks the `Zombie` flag in E''s record. If it is set, then T' ceases execution on E' and looks for more work.
- **Restarting events that are no longer hood-safe.** A similar action occurs here with respect to the `HoodCheck` flag. At commitment, rather than removing E', hood-check and prediction are repeated.
- **Get new work while waiting for an event to become hood-safe.** If T is looking for work and the next event E is not hood-safe, then T finds and begins work on another event that is. This is an obvious optimization, but has an additional consequence. Threads looking for work now first check the Processing FIFO for events waiting for changes in their hood-safe status.

The complete procedure is as follows.

Step 1. Get Work (Lock required)
Thread T looks for work until it finds a hood-safe event E with highest possible priority in the FIFO or a simulation-end-condition is reached. T appends events to the FIFO from the event pool if it has to. In this way, one of the threads processes the highest priority event and the rest of the threads process some lower priority hood-safe events.

Step 2. Do Event Processing (No Lock required)
If E has not been invalidated by any previous event, then T processes E and new events are predicted and saved as temporary data to be inserted during commitment.

Step 3. Commit Event (Lock required)

- T waits until it is at the head of the FIFO or a simulation-end-condition is reached.
- If E was invalidated in the mean time (by an earlier event processed by a different thread), its computation is discarded and the event is deleted from the FIFO.
- If E is not invalidated, but became hood-unsafe in the mean time (by an earlier event processed by a different thread), its computation is discarded but the event is NOT deleted from the FIFO. It will be re-computed.
- If E neither was invalidated nor became hood-unsafe, its computation result is committed.
- Commitment invalidates events as needed, removing them from the queue and flagging those in the FIFO. It also inserts newly predicted events. If these go into the FIFO, then hood-checks must be performed as described.
- T goes back to Step 1.

IV. RESULTS

The base code is from Rapaport [13] which have we modified to support the Paul data structures (see [5] for details). Simulations were run on a Dell workstation-class PC with a 64-bit quadcore Xeon 2GHz CPU. Programs were compiled using GCC with standard optimization settings and run under Linux. All variations in the code were validated against the original. In all cases we simulated a single particle type as a hard-sphere. Also, the cell size was chosen so that the average occupancy was one particle per cell. This generally maximized performance. For particle densities, we use .1 to roughly approximate a liquid. We also ran experiments for other densities, e.g., to approximate an ideal gas, but this has little effect on performance. A note on the efficiency of the code – while we did not explicitly optimize it, say, for memory hierarchy, both the original and our extensions have been written for performance. For example, all data structures are fixed size arrays, and all pointers are actually array indexes.

Our first experiment is with respect to a single thread with the results shown in Figure 7. We compare two insertion protocols: "Rapaport" signifies that all predicted events are always enqueued, "Lubachevsky" that only the highest priority event per particle is ever enqueued. As is typical in presenting DMD results, we show event throughput. We count only payload events (collisions in this case), as cell crossings are implementation overhead and vary with simulation parameters.

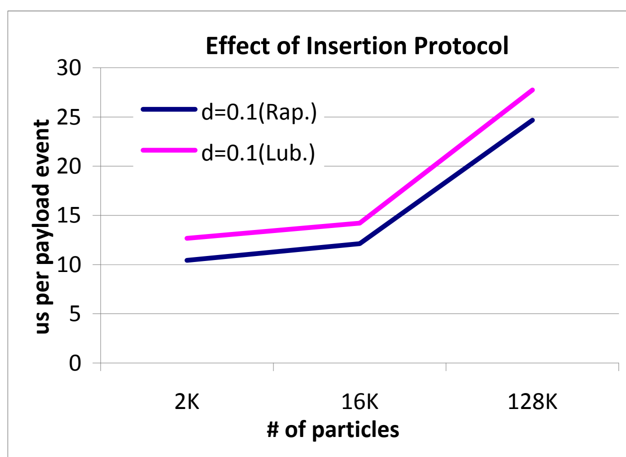

Fig. 7. Graph compares performance versus simulation size for two insertion protocols. With "Rap." all predicted events are queued. With "Lub." at most one event per particle is queued. Density is .1.

Depending on the simulation, there are from 3-6 cell crossings per payload event.

The first observation is that the event latency varies from 10 to 15 us/event for small simulations, increases to 25-28 us/event for medium sized simulations. This shows a factor of 3 improvement in just a few years (see, e.g., [10]) and a several order-of-magnitude improvement over older studies. The second is that for these cases the Lubachevsky protocol improves performance from 12% to 22% with the fraction diminishing as the simulation size increases.

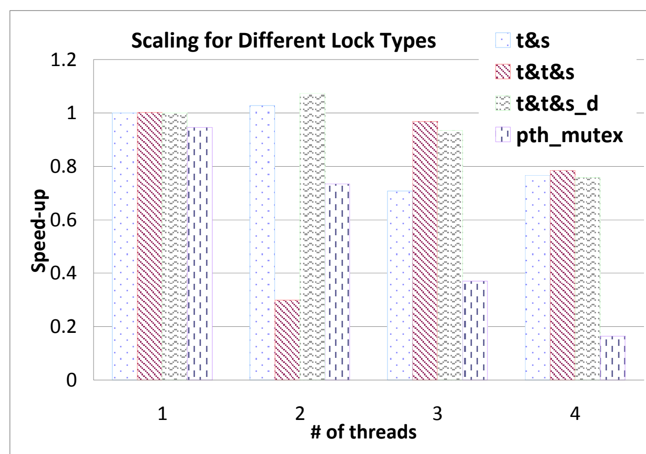

Fig. 8. Graph compares performance of various locks in terms of speed-up versus number of threads. 1M particles are simulated. Density is .1. There is no delay.

We now describe results from experiments with respect to our parallel DMD implementation. We immediately found that with such an efficient baseline implementation, the parallelization must be equally efficient.

In particular, the original mutex from the pthreads library appeared to have insufficient performance do to system calls within the lock. To replace this we tried three hand-written variations, based, respectively on test-and-set, test-and-test-and-set, and test-and-test-and-set with backoff (see, e.g., [1]). Results from all four locks are shown in Figure 8. We observe that test-and-test-and-set with backoff delivers the most consistently good performance and currently use that exclusively.

Figure 9 shows scaling of the parallel DMD code with respect to particle density and model complexity. The 0-delay model corresponds directly to the hard spheres that were simulated. The 18us-delay corresponds to the simulator used by the Dokholyan group at the University of North Carolina for modeling biomolecules. The extra time comes from modeling of more complex forces which in turn requires many more predictions per particle per event. Since these prediction computations are local to each thread, we emulate them with a simple delay loop. The final delay model corresponds to a simulator that performs substantial processing per event, such as might be needed for reactions. All simulation are for 1M particles and the densities shown.

The first observations are from the simple ideal gas model. The parallel implementation, including locks, adds about 20% overhead to the serial code. This overhead, plus the exclusion in the critical sections, results in no speed-up being obtained for that model. Clearly further optimizations are needed. The second observation is for the 18-us delay simulator. Here good speed-up is obtained for all four cores, although it is far from linear. Finally, the complex simulation shows excellent scalability through four threads.

V. Discussion and Future Work

We have presented a new algorithm for parallelizing DMD that is based on an event-based composition as would be appropriate for a pipelined event processor. The solution requires handling complex interactions due to different types of hazards (causality, coherence) as well as synchronization at multiple levels. These issues are exacerbated by the combination of chaos and order in DMD simulations: events are typically predicted both within infinitesmal time intervals and with long distance effects.

Our results show that this method is beneficial for biomolecular (and other) simulators with at least moderately complex force models. We believe that this is currently the only such DMD code.

Improvements in microprocessor technology and DMD algorithms have reduced event processing times to the point where they are equivalent to a single message transfer or a dozen cache misses. This makes further speed-up challenging, but not impossible. We have just begun exploring the numerous possible optimizations with respect to workload balance, synchronization granularity, and tuning of data structures.

Acknowledgments We thank the anonymous reviewers for their helpful comments.

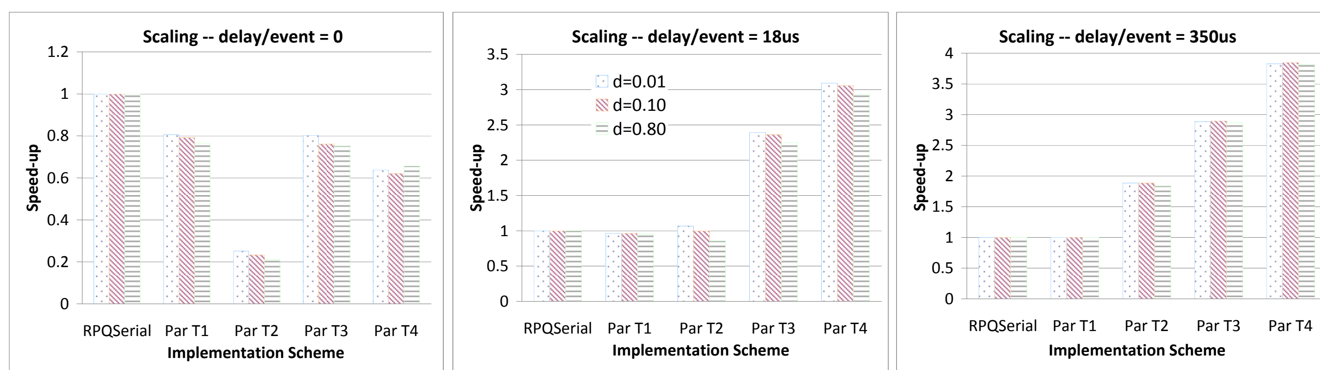

Fig. 9. Graphs show scaling of parallel DMD for various numbers of threads, simulation densities, and model complexities. Performance is normalized to serial time. All simulations are for 1M particles. Bars show densities of 0.01, 0.1, and 0.8 respectively.

REFERENCES

[1] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan-Kaufmann, 1999.

[2] F. Ding and N. Dokholyan, "Simple but predictive protein models," *Trends in Biotechnology*, vol. 3, no. 9, pp. 450–455, 2005.

[3] N. Dokholyan, "Studies of folding and misfolding using simplified models," *Current Opinion in Structural Biology*, vol. 16, pp. 79–85, 2006.

[4] R. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[5] M. Herbordt, F. Kosie, and J. Model, "An efficient $O(1)$ priority queue for large FPGA-based discrete event simulations of molecular dynamics," in *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, 2008, pp. 248–257.

[6] A. Krantz, "Analysis of an efficient algorithm for the hard-sphere problem," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 3, pp. 185–209, 1996.

[7] B. Lubachevsky, "Simulating billiards: Serially and in parallel," *International Journal in Computer Simulation*, vol. 2, pp. 373–411, 1992.

[8] M. Marin and P. Cordero, "An empirical assessment of priority queues in event-driven molecular dynamics simulation," *Computer Physics Communications*, vol. 92, pp. 214–224, 1995.

[9] S. Miller and S. Luding, "Event-driven molecular dynamics in parallel," *Journal of Computational Physics*, vol. 193, no. 1, pp. 306–316, 2003.

[10] J. Model and M. Herbordt, "Discrete event simulation of molecular dynamics with configurable logic," in *Proc. IEEE Conference on Field Programmable Logic and Applications*, 2007, pp. 151–158.

[11] G. Paul, "A complexity $O(1)$ priority queue for event driven molecular dynamics simulations," *J. Computational Physics*, vol. 221, pp. 615–625, 2007.

[12] D. Rapaport, "The event scheduling problem in molecular dynamics simulation," *Journal of Computational Physics*, vol. 34, pp. 184–201, 1980.

[13] ——, *The Art of Molecular Dynamics Simulation*. Cambridge University Press, 2004.

[14] S. Sharma, F. Ding, and N. Dokholyan, "Multiscale modeling of nucleosome dynamics," *Biophysical Journal*, vol. 92, pp. 1457–1470, 2007.

[15] H. Sigurgeirsson, A. Stuart, and W.-L. Wan, "Algorithms for particle-field simulations with collisions," *Journal of Computational Physics*, vol. 172, pp. 766–807, 2001.

[16] C. Snow, E. Sorin, Y. Rhee, and V. Pande, "How well can simulation predict protein folding kinetics and thermodynamics?" *Annual Review of Biophysics and Biomolecular Structure*, vol. 34, pp. 43–69, 2005.

[17] B. Urbanc, J. Borreguero, L. Cruz, and H. Stanley, "*Ab initio* discrete molecular dynamics approach to protein folding and aggregation," *Methods in Enzymology*, vol. 412, pp. 314–338, 2006.