JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **35,** 76–90 (1996)
ARTICLE NO. 0070

Design and Analysis of a Parallel Molecular Dynamics Application

GREG EISENHAUER AND KARSTEN SCHWAN

College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332

This paper investigates the parallelization of molecular dynamics simulations. Topics addressed include the development of abstractions that support alternatives to traditional neighbor-list based approaches to molecular dynamics simulations, the evaluation of alternative methods for domain decomposition and program synchronization, and the design of a flexible framework for constructing realistic molecular dynamics simulations. Moreover, this framework is designed to be suitable for implementation of programs for both shared and distributed memory MIMD multiprocessors. One specific problem addressed by our research is that MD simulations often scale poorly with increases in system size and in numbers of processors, in part due to the commonly used neighbor list approach. Our research uses a domain decomposition approach that exploits molecular locality to reduce overall computation and to attain good scalability. Experimental evidence concerning scalability and performance is gathered on KSR supercomputers. © 1996 Academic Press, Inc.

1. INTRODUCTION

Despite the growing use of parallel computing by the scientific community, few nontrivial scientific applications specifically designed for modern multiprocessors are available for general study and experimentation. While scientific ''kernels'' are more common, their use can be problematic because they fail to take into account many aspects of the complex nature of real applications. Similarly, simpler parallel applications such as small scale sorting, matrix multiplication, and Gaussian elimination may be insufficiently interesting as objects for experimentation with techniques such as dynamic load balancing, program steering, and runtime program configuration.

This paper presents MD, a molecular dynamics simulation. MD is a complete high-performance parallel molecular dynamics system. It offers abstractions with which programmers can implement efficient strategies for managing the control and dataflow required for the basic molecular dynamics calculations as well as for global statistics and scientific visualization. In addition, MD is designed to support experimentation with alternative methods of domain decomposition and thread synchronization and with alternative approaches to exploiting molecular locality. Related work in this area includes [9, 8], both of which describe computational approaches similar to the one taken in MD. In particular, they describe the linked-cell method of distributing computation in molecular dynamics simulations. As in MD, the linked-cell method assigns particles to processors based on their location in three-dimensional cells (domains in MD) of size greater or equal to cutoff distance. In addition to the exploitation of molecular locality, the algorithm presented here differs from the linked-cell approach in that it supports domains of arbitrary shapes and has no restrictions on the sizes of those domains.

MD is written using Cthreads, a multiprocessor threads package that supports thread-based parallelism and provides portability between a variety of shared-memory multiprocessor and uniprocessor platforms. Although MD is currently implemented on shared-memory multiprocessors, its basic structure is easily adapted for and ported to distributed-memory machines. In addition to this portability, MD offers substantial flexibility in its support for experimentation with alternative domain decompositions or synchronization methods, or with alternative approaches to exploiting molecular locality. Accordingly, an important contribution of this paper is the presentation of various aspects of MD's design, the tradeoffs involved and their effects on performance. These discussions form the bulk of Sections 3 and 4 and serve to support the main conclusions of this work.

One important conclusion of our research is that molecular locality can be successfully exploited to reduce computational complexity. A second conclusion is that flexibility in terms of being able to support multiple elements, molecular systems, schemes for domain decomposition, and communication/synchronization methods need not increase computational costs. On the contrary, such flexibility is important in order to attain high performance for applications written with MD.

Examining the design and behavior of MD is useful for several reasons. First, the application itself is of interest to molecular dynamicists because many similar molecular dynamics programs can be built within the modular framework of MD. Second, applications designers may benefit from an understanding of the tradeoffs made in MD's implementation. Third, MD provides computer scientists with a portable, complex, and dynamic sample high performance application for use in the investigation of topics in operating or programming systems for high performance parallel machines.

0743-7315/96 $18.00

This paper first presents the structure of the general MD computation and our basic approach to its parallelization. Section 3 then examines some of the important design tradeoffs within the basic approach. To simplify the presentation this section only considers issues central to the molecular dynamics simulation itself, which is called Core MD. Next an analysis of the application's performance in various circumstances is presented, including studies of the effects of exploiting molecular locality and using different domain decompositions. Finally, the paper discusses aspects of the application that are not covered in the discussion of Core MD and presents our conclusions.

2. THE STRUCTURE OF MD COMPUTATION

The solution technique employed in the type of molecular dynamics simulations supported by MD is as follows: given the locations of all particles in the system, calculate the forces on each particle, calculate movement of the particles in some time interval Δt, update the locations and repeat.

Discounting global statistics, analysis, and visualization, the most computationally intensive aspect of this solution technique is the calculation of the long-range pair interactions between particles. This calculation is $O(n^2)$ in complexity, where n is the total number of particles in the worst case. In comparison, calculating the forces resulting from intramolecular bond interactions is an $O(n)$ computation. In order to reduce computational demand, it is common to employ a *cutoff radius*. Particles outside the cutoff radius are assumed to have a net zero contribution to the calculation.

2.1. Problem Description

MD specifically targets physical systems where most or all of the particles are bound into the larger association of a physical molecule. The sample application exercising MD is a physical simulation used by physicists at Georgia Institute of Technology [10]. The simulation is designed to investigate evolutionary trends of the structure and dynamics of n-hexadecane ($C_{16}H_{34}$) films on a crystalline substrate modeling $Au(001)$. In this simulation, the alkane system is described via intramolecular and intermolecular interactions between pseudoatoms (CH_2 and terminal CH_3 segments) and the substrate atoms. The calculational cell is a square cylinder which is periodically repeated in the x–y directions. Temperature is controlled via infrequent scaling of the particles' velocities. For this sample system, the pseudoatoms are represented as MD particles. The alkanes are represented as MD molecules. For purposes of implementation, the substrate is also organized into ''molecules,'' although it is in fact a particle lattice. This results in a more regular code structure and allows MD to take advantage of the spatial locality of these substrate ''molecules.'' The alkanes remain associated in a chain with very predictable bond lengths throughout the simulation. This provides an implicit locality that can be exploited. The next section provides a global view of our implementation.

2.2. Parallelization Scheme

MD employs a spatial-decomposition approach to parallelization. The physical system is divided into spatial domains, and each domain is allocated to a processor. Molecules are assigned to domains by their centers of mass, and the processor on which a domain is placed is responsible for calculating the forces on the domain's alkanes. In order to calculate the forces on its alkanes due to the presence of alkanes in other domains, each processor must obtain information from all spatial domains whose particles might affect its own, called *neighbor domains*. The number of neighboring spatial domains is determined by the geometry of the domains, the sizes of the domains, and the size of the cutoff radius. The information flow between domains represents the main communication requirement for this class of MD applications, and it must take place once for each simulation timestep. Other communication requirements include the occasional need to calculate a global temperature value to control the temperature of the simulation. This is done infrequently for stable systems and has little impact on overall communications.

Given this approach, the basic process of simulation involves the following steps from each processor:

1. obtain needed particle location information from neighbor domains,
2. calculate all the forces on local particles,
3. apply the calculated forces to yield new particle positions, and
4. publish the new particle locations.

In addition to particle location information that is exchanged as in steps 1 and 4 above, processors must exchange information about molecules that are moving from one domain to another. In practice, this is a small amount of information that is easily piggybacked onto the normal particle location exchange, and its impact on the computation is minimal. The force calculation of step 3 can be further broken down into three substeps for calculating the following forces:

1. intramolecular forces,
2. intermolecular forces due to other particles in the same spatial domain, and
3. intermolecular forces due to particles in neighbor domains.

Interestingly, only the last of these force contributions actually depends on information from other domains. Thus the first two calculations can proceed in parallel with the communication step. To facilitate latency hiding for communications, the MD framework not only permits association of execution threads with domains, but it also permits the use of a separate thread for the intramolecular calculations, so that these calculations can proceed while the proc-

essor's main thread performs the communication steps. The facilities offered by MD for synchronization, latency hiding in communications, and creation and deletion of execution threads rely on the Cthreads parallel programming platform described elsewhere [5].

2.3. Summary of Core MD

The sections above describe the computational core of MD, which would serve as a starting point for abstracting a computational "kernel" from MD. However, as with many complex applications, this computationally intensive core comprises less than one third of the actual code. Interesting noncore aspects of the application are presented in Section 5. For clarity of presentation, the main components of the core are summarized below.

The first abstraction offered in MD is that of a *particle*, which is the basic dynamic element of the simulation. Each particle is represented by a data structure containing the elements' type, location, derivatives of motion, and fields for the accumulation of forces on the element. All particles in MD are members of *molecules*, which are defined as sets of particles that exhibit some spatial locality throughout the simulation. Molecules are represented as data structures which contain an array of particles as well as additional information about the molecule itself, such as its center of mass, its spatial extent (or *bounding box*), and the number of particles it contains.

Molecules are located in a two- or three-dimensional space and this system simulation space is divided into one or more *domains*. The number and configuration of the domains is a *decomposition*. A decomposition is characterized by the number of domains it contains, the neighbor lists of those domains, and a function that maps coordinates in simulation space to domains. Entire molecules are assigned to the domain which contains their center of mass. The domain assignment of a molecule may change as the molecule moves in space during the course of the simulation. Domain are the units of allocation of data and of execution threads to processors. At least one thread is associated each domain and that thread is responsible for performing both the intramolecular computations for the domain's molecules and the intermolecular computations for its and its neighbor domains' molecules. However, additional execution threads may be created within each domain in order to overlap interdomain communications with computations internal to each domain.

3. SYSTEM DESIGN

The parallelization approach described above is the result of several design tradeoffs. For example, an early decision in MD implementation was to attempt to avoid the large memory requirements typical with traditional interaction list approaches. While interaction lists are an attempt to reuse previous interaction decisions to reduce simulation computation requirements, MD tries instead to reduce these requirements by exploiting the intrinsic spatial locality of a molecule. Given this decision, it is natural to choose the molecule, rather than the particle, as the basic unit of decomposition. This choice has the advantage of eliminating the reassembly of molecules for intramolecular force calculations, but it also introduces additional complexity in domain decomposition. Specifically, because a molecule can span a domain boundary and because that whole molecule must be assigned to one domain, domain boundaries are somewhat fuzzy. This fuzziness affects load balancing and domain geometries. Several important characteristics of the design of MD are described next.

3.1. Geometry and Neighbor Lists

Earlier discussion states that each processor must obtain information from its neighbor domains in order to compute the forces on its particles. What domains compose the neighbor domains depends upon the geometry of the spatial decomposition and the size of the cutoff radius being used. If two domains are sufficiently close for molecules assigned to one domain to affect molecules assigned to another, those domains are neighbors. All neighbor domains for each domain are recorded in the domain's neighbor list.

For simple systems, the neighbor list for each domain may be as small as 2. This is the case in certain slab-based decompositions where the neighbor list for each processor contains the domains that are physically adjacent to that processor's domain. For example, consider the simple domain decomposition depicted in Fig. 1. The physical system is represented as a box divided into eight domains along a single axis. For systems in which the cutoff radius is small compared to the width of a domain, as is the situation depicted in Fig. 2a, only molecules in domains which are physically adjacent to a particular domain can exert forces upon its particles. This situation, with a decomposition along a single dimension and a relatively small cutoff radius, yields the least communication overhead. The effects of a relatively large cutoff radius as compared to the size of the domain are depicted in Fig. 2b. In this case, molecules in from Domains B, C, and D can all potentially affect molecules in Domain A. Thus all those domains must be considered neighbors of A, although the domains are not adjacent.

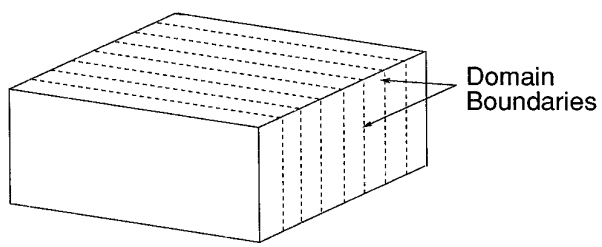

FIG. 1. A simple domain decomposition. The cube is the physical system, divided into domains by slices along a single dimension.

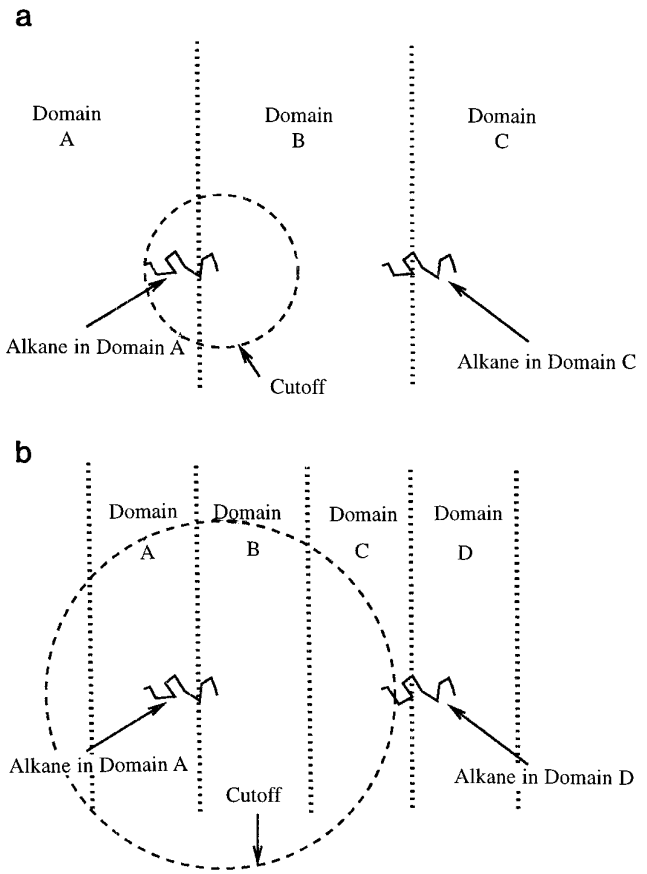

FIG. 2. Possible decompositions: (a) smaller cutoff relative to domain width; (b) larger cutoff relative to domain width.

While the simplest decompositions are slices over one or more dimensions, the MD framework is quite flexible regarding decomposition geometries. This is important because a spatial domain and its molecules comprise the unit of allocation to a processor, so that the spatial decomposition strongly affects the resulting communication patterns, computational load balancing, and subsequently, the performance of the parallel code. MD is written so that given the set of domains and their neighbor lists, any style of decomposition can be supported. This is also of considerable importance when simulating oddly shaped or irregular systems. Section 4 will discuss the performance effects using different spatial decompositions in the simulation.

To establish a new decomposition in MD, simply calculate a neighbor list for each domain and provide a function, *coords_to_domain*(), that returns the containing domain when evaluated on a coordinate in the simulation space. In the shared-memory implementation of MD, a domain's neighbor list is actually a list of pointers to data structures called *exports structures*. The exports structure contains, among other things, the list of particle locations as well as the mutexes, conditions, and state variables necessary to control access to that list. Each domain maintains its private copy of an exports structure and also has pointers to the exports structures of its neighbors. While communication between neighboring domains might take place via messages in a distributed memory implementation of MD, in its shared memory implementation, communication entails processors accessing information in, and performing appropriate synchronization operations on, items in their own and their neighbors' export structures.

Because the nature of the decomposition has a dramatic effect upon system performance and because the disposition of the particles in the system changes over time, it is also important to consider changing a decomposition dynamically, while the simulation is running. The mechanisms in place to handle molecules moving from one domain to another make this a relatively straightforward task. After updating each particle's location based on the latest results of the force calculations, each processor invokes the *coords_to_domain*() function on each of its molecules to determine if any molecules have crossed into another domain. Molecules which are found to have crossed are placed in the moving list in the processor's exports data structure. When neighboring domains access the exports data structure to acquire new particle location information, they evaluate the *coords_to_domain*() function for each molecule there and adopt any that now belong to their domain. If, rather than molecules moving, the domain boundary itself changes between iterations, then the transferred molecules are simply treated like molecules whose motion has carried them from one domain to another. Our experimentation with dynamically modifying domain boundaries through program monitoring and steering presented in [3] demonstrates the importance of this ease of change in domain boundaries.

3.2. Communication and Global Synchronization

One of the steps in the breakdown of local computation described in Section 2.2 above was "obtain needed particle location information from neighbor domains." This represents the dominant communication requirement for MD. This section discusses how this communication requirement affects the design of MD for both shared memory and distributed memory systems.

3.2.1. Shared Memory Systems

In a shared-memory implementation the communication requirement is more clearly understood as a synchronization requirement. Each processor can place a list of the locations of its particles in a memory location visible to its neighbors. A processor cannot perform all pair calculations for iteration $n + 1$ until it has its neighbors particle locations for iteration n. Once finished with iteration $n + 1$, a processor cannot update the particle locations in its own list until it is sure that its neighbors are finished with the data for iteration n. The simplest way to enforce these conditions is to use barrier synchronization to ensure that

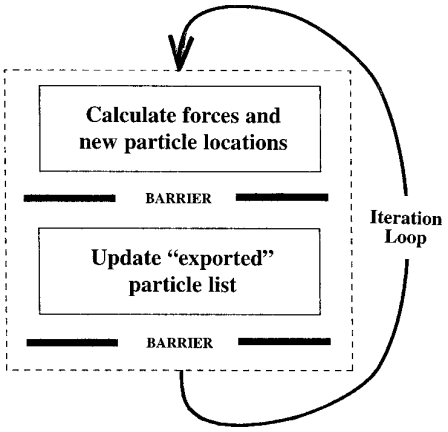

FIG. 3. Representation of local control flow for each processor using barrier synchronization.

no processor updates its locations too soon or starts the next iteration before its neighbors are ready. Figure 3 depicts the stages of MD computation when barrier synchronization is employed.

One disadvantage of the use of barrier synchronization is that it forces *all* processors to wait for a single tardy processor when strictly only that processor's neighbors must wait. Barrier synchronization also ignores the fact that some of the force computations for iteration $n + 1$, such as the intramolecular force calculations, can actually proceed before neighboring particle location information for iteration n is obtained. Using looser synchronization would potentially allow some processors to get ahead of their neighbors, within the constraints mentioned in the paragraph above. This additional flexibility can improve program performance by allowing useful computation to be performed in times when the barrier model would be waiting.

Figure 4 is a representation of each domain's computation when using neighbor synchronization. Specifically, Fig. 4 depicts a control-flow diagram for a complete iteration step performed by a single processor. The boxes with square corners represent fully local computation, and the rounded boxes represent stages that potentially require synchronization with other domains. The calculation of intramolecular and local forces is shown separately from the calculation of forces due to particles from neighboring domains. The fact that these calculations can proceed during the communication step creates the potential for hiding delays in the neighbor synchronization. The synchronization style depicted in Fig. 4, where the intramolecular and local forces are calculated in a separate thread, is called "neighbor synchronization with forking." A variant of this style that does not employ the additional thread is called "neighbor synchronization without forking." The performance impact of the structure and flexibility of the local computation and the nature of interprocessor synchronization is examined in Section 4.4.

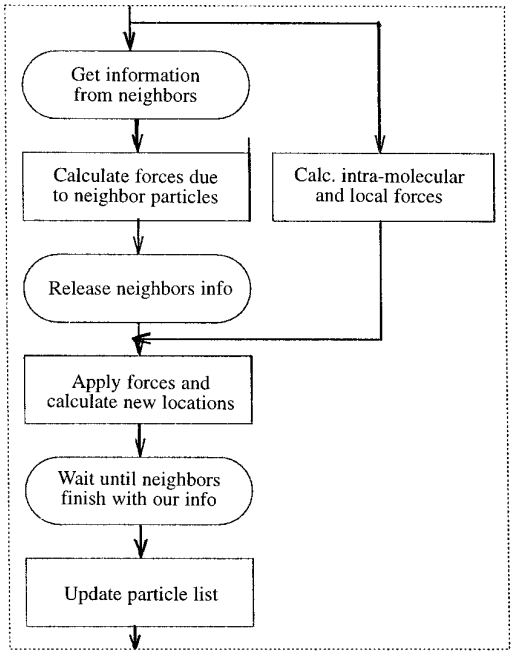

FIG. 4. Use of neighbor synchronization in local calculations.

3.2.2. Distributed Memory Systems

Although MD is only now being ported to a nonshared memory environment, its efficient operation in both shared and distributed memory architectures is a critical design goal of MD. Toward this end, the shared-memory version of MD isolates processor memory interactions to a few locations in the code. Specifically, the core MD calculation exploits shared memory only for exchanging particle location information and for calculating system temperature values. Of these two, information exchange for temperature calculations is less important because in a stable system, temperature control is not necessary on every iteration. Furthermore, a distributed memory implementation of this computation can be performed with a small number of global sum and broadcast operations, both of which are typically directly supported by the hardware of target high performance machines.

The exchange of particle location information is considerably more important to the performance of MD. Fortunately, because processors send identical information to each of their neighbors in each time step, such exchanges are easily implemented with broadcast or multicast operations. Some of the performance properties of distributed MD implementations are evaluated in research complementary to our own [7].

3.3. Use of Molecular Locality

In molecular dynamics simulations which employ a cutoff radius, a common implementation technique is to maintain, for each particle in the system, an *interaction list* of all other particles within this cutoff radius. Interaction lists

attempt to exploit *temporal* locality, the quality that decisions about particle interactions made in one iteration are much the same as the iterations before and after. The lists provide a means by which decisions about which particles interact can be saved and reused from iteration to iteration, thus reducing the overall computational requirements of the simulation. While such lists can often be used for several iterations, they must periodically be recomputed because particle motion will eventually render them invalid. However, the use of interaction lists has several disadvantages. In particular, the problem of computing and updating the interaction list grows with the square of the number of particles in the simulation, and the interaction list's memory requirements can be quite large.

One of the design goals of MD was to utilize *molecular* locality to help reduce the overall calculation. Molecular locality exploits the quality that the particles in a particular molecule are physically close to each other and that this can be utilized to reduce the computational demands of making interaction decisions. The use of this technique does not imply that interaction lists cannot be used as well. In fact, molecular locality could be exploited in order to reduce the cost of computing interaction lists. However, if molecular locality is effective enough, its exploitation may obviate the need for interaction lists. This would be particularly advantageous in situations where the memory requirements for interaction lists may be the limiting factor in how large a system can be simulated.

To examine the exploitation of molecular locality in MD, first consider the basic set of nested FOR loops required to calculate the pair forces as abstracted below:

```
FOR each mol1 in local alkanes
 FOR each mol2 in [local alkanes
        + imported alkanes + substrates]
  FOR each particle in mol1
   FOR each particles in mol2
    if (particle distance > cutoff)
      exit loop;
    compute pair forces;
   END FOR
  END FOR
 END FOR
END FOR
```

MD adds some conditionality to this basic structure to exploit molecular locality in several ways. This conditionality is expressed in Fig. 5. The first conditional exploits molecular locality to eliminate some set of particles from further consideration for pairwise computations. Specifically, associated with each molecule is a set of coordinates defining its *bounding box*, which is the maximum extent that the molecule has in space. If the distance between the bounding boxes of two molecules is greater than the cutoff radius, no particle from one such molecule can exert force on particles of the other. Therefore, by comparing the bounding boxes of two molecules at the first

```
FOR each mol1 in local alkanes
 FOR each mol2 in [local alkanes + imported alkanes + substrates]
  if (molecules cannot interact) exit loop;          # first conditional
  FOR each particle in mol1
   if (particle cannot interact with mol2) exit loop;   # second conditional
   FOR each particle in mol2
    if (particle distance > cutoff) exit loop;        # third conditional
    compute pair forces;
   END FOR
  END FOR
 END FOR
END FOR
```

FIG. 5. Expanded core computation structure.

conditional, it can quickly be determined whether they are too far apart to interact. This check can potentially eliminate hundreds of distance calculations in the inner loops. Similarly, at the second conditional, a brief comparison of the coordinates of the particle with those of the bounding box of *mol*2 can eliminate a whole molecule from further consideration in pairwise computations with respect to that particle.

Spatial locality is also exploited in another way in MD. Because of the loop structure, the second and third conditionals are evaluated first for one particle, then for the next particle in the alkane chain, etc. Because the bond length is fairly stable, adjacent particles are never far apart, and one can often use the results of one calculation to predict the results of the next several calculations. This *chain locality* can be exploited to eliminate expensive distance calculations. For example, if the first particle in a molecule is outside the cutoff radius by more than three bond lengths, then *at least* the next three particles in that molecule will also be outside that cutoff radius. Section 4.3 examines the efficacy of these techniques for exploiting molecular locality.

4. PERFORMANCE EVALUATION

The evaluation of MD presented in this section focuses largely on the performance of the abstractions presented in the previous section. Specifically, we first compare the performance of the molecule-based domain description in MD with the performance of an interaction list based molecular dynamics application running on a Cray YMP. We also explore the performance characteristics of MD abstractions in detail by examining different physical systems and by evaluating some of the design tradeoffs described above. In addition, we consider MD performance with different characteristics of the simulated system, including the decomposition geometries and the number of processors involved. One conclusion we draw from these results is that many of the important performance characteristics of MD vary considerably depending upon the nature of the particular physical system being simulated. This supports our hypothesis that the flexibility and configurability of the MD framework are critical for attaining high performance across a range of MD simulations.

The experimental results described in this section are attained on Kendall Square Research KSR-1 and KSR-2 machines. The KSR supercomputer is a NUMA (nonuniform memory access) shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Each node consists of a 64-bit processor, 32 Mbytes of main memory used as a local cache, a higher performance 0.5 Mbyte subcache, and a ring interface. CPU clock speed is 20 MHz, with peak performance of 20 Mflops per node (on the KSR-2, CPU clock speed and peak performance are doubled). Access to non-local memory results in the corresponding cache line being migrated to the local cache, so that future accesses to that memory element are relatively cheap. The parallel programming model implemented by the KSR's OSF Unix operating system is one of kernel-level threads which offer constructs for thread fork, thread synchronization and shared memory between threads. This kernel-level thread facility is called Pthreads. MD itself is implemented with the Cthreads parallel programming library [5], a user-level threads facility that is built on top of Pthreads. As Cthreads performs its functions outside the kernel (within a Pthread), it has performance advantages over Pthreads concerning thread creation, context switches, and synchronization. User-level thread implementations like Cthreads are limited in some ways, but for this application these limitations are not critical. Cthreads also has some advantage in portability across shared memory platforms and gives users the flexibility to design and use threads primitives customized to their applications.

4.1. Overall Results

For comparison purposes, this section presents the performance results for the base system mentioned in Section 2.1. This system consists of 300 *n*-hexadecane molecules (16 particles each) on a substrate of 1350 gold particles. The substrate is organized into 90 molecules.

Normative comparison. Our base point of comparison is a Fortran program that simulates this system using the interaction list technique. It has been used in previous experimentation by the physicists with whom we are working. Table I compares the amount of computation time required per iteration for the MD application and the interaction list based application. For the interaction list based approach, a Cray Y-MP requires an average of 0.65 s to compute a single timestep of the simulation. The majority of the CPU time is spent in updating the interaction list, which is done only every eighth timestep. The same Fortran code running on a single KSR-1 processor averages 15.4 s per timestep, which roughly compares with the 18.6 s required by MD using a single processor. However, an application implemented with MD on the 64-node[1] KSR-2 using just 49 processors can achieve better performance than a highly tuned interaction list based implementation on a Cray YMP. The KSR version of MD measured here has not been optimized heavily. Specifically, we have not added machine-specific optimizations such as KSR's prefetch or poststore instructions, we have not optimized the assembly code generated by the KSR's C compiler, and we have not customized the underlying parallel programming library (Cthreads) to remove any specific operating system inefficiencies. Given these results, it is fair to say that parallel MD on the KSR can attain supercomputer performance. This is especially satisfying given the relative slowness of KSR processors as compared to the Cray YMP machine.

TABLE I
Comparison of MD Performance on KSR-1 and KSR-2 vs Interaction Lists on Cray YMP

	Seconds per timestep
Cray YMP running FORTRAN interaction list algorithm	0.65
KSR-1 running FORTRAN interaction list algorithm	15.4
KSR-1 running MD using 1 processor	18.6
KSR-2 running MD using 49 processors	0.56
KSR-1 running MD using 60 processors	0.74
KSR-1 running MD using 49 processors	0.81

It would be misleading to use the numbers from this table for any direct comparison of interaction list techniques vs. molecular representations of particles. A practical reason that troubles a comparison is that the specific solution algorithms used in the Fortran code are slightly different from those used in MD, and the Fortran code has not been optimized or parallelized for the KSR. Furthermore, the performance of MD is affected by more factors than just the number of particles in the physical system.[2] More fundamentally, MD's approach and interaction list techniques are each seeking to exploit different aspects of physical simulations, and their performance will differ considerably depending on the nature of the system under simulation. As was discussed in Section 3.3, interaction lists make assumptions about the presence of temporal locality in interactions. One can remove these assumptions by recalculating interaction lists on every timestep, thereby obviating the need for keeping the lists but also significantly degrading performance. Similarly, MD assumes the presence of molecular locality and if that assumption were removed to put it on a par with the interaction list approach, its performance may be degraded to perhaps the same level.

Because these factors make it difficult to compare MD with interaction list based approaches, we instead present the results in Table I (and in Table II) to establish that the techniques employed in MD are of power similar to

[1] At most 60 of the 64 nodes were employed because several nodes are reserved for operating system functions and cannot be used by applications.

[2] Section 4.2 contains further discussion of how MD performance improves with larger or less dense systems.

that of the interaction list approach. This is a valuable conclusion because the MD approach is much more memory efficient than keeping interaction lists and would allow simulation of systems whose memory demands may be too excessive under the interaction list approach. Furthermore, the remainder of this section will show that MD's performance scales well with machine size and improves considerably for larger system sizes. This implies that MD has the potential to greatly outperform interaction lists for certain types of simulated systems.

Speedup results. While the performance achieved above is satisfying, it is more important in the long term to understand how MD will scale to larger physical systems and to larger parallel machines. Of particular importance is how computational costs change when using the MD framework. For an interaction list based application, we would expect the time required to compute a timestep to rise with with the square of the number of particles involved in the simulation. For MD, the function expressing computational cost is considerably more complex. In order to explain MD computational costs, let us first examine the sample system. Figure 6 compares the actual iteration times for MD and for the sample system with the iteration times one would expect assuming perfect speedup (i.e., the time for one processor divided by the number of processors). Note that while the iteration time decreases with increasing number of processors, speedup is not linear. This behavior can be explained by certain characteristics of the sample physical system being simulated. In this system, the cutoff radius amounts to 13.4% of the x-y extent of the system and a stretched-out alkane can span up to 43% of the same extent. One consequence of these numbers is that the system is very highly coupled. At least in the x and y dimensions, no domain can be so far from another domain that none of molecules could interact. This means that regardless of the way the system is decomposed, each domain must have all other domains as members of its neighbor list, or some potential interactions might be missed. Therefore, independent of system decomposition, every processor must communicate with every other processor on every timestep.

In general, global communication in every timestep is a major performance problem for physical simulations on parallel machines, but it cannot be avoided for the sample system because the integrity of the calculation must be maintained. Other parameters will also affect the speed at which a timestep can be calculated. These parameters include characteristics of the simulated system (such as density and cutoff radius) as well as other parameters easily varied with MD (such as the number of processors employed and the decomposition scheme). Because changing the physical simulation affects the behavior of MD, it is important to understand how the time required to calculate a timestep varies with these parameters in order to evaluate the utility of MD's flexibility. The next section will examine some of the consequences of changing the physical parameters for a single simulated system.

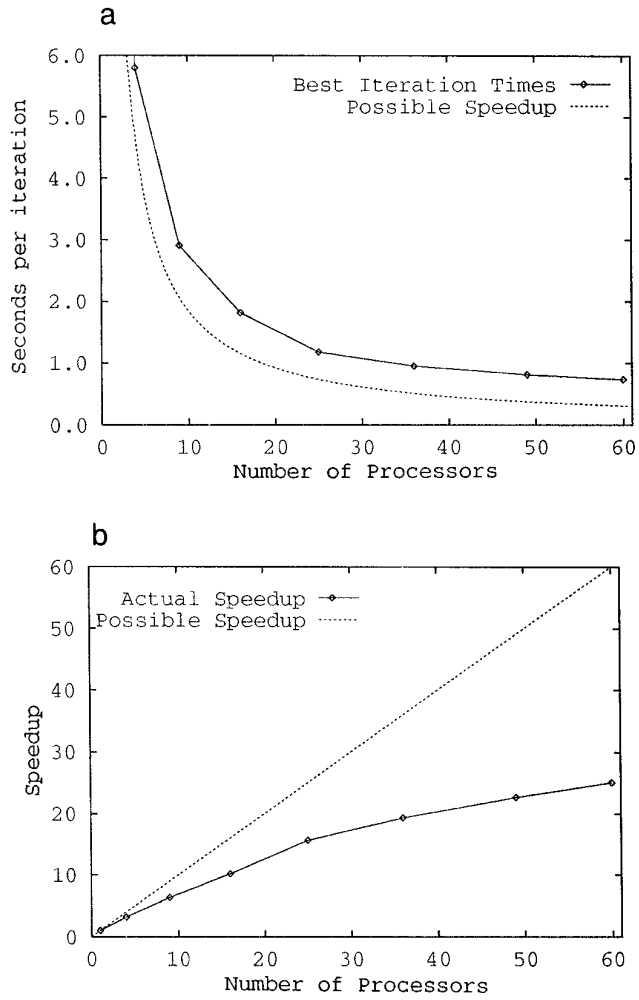

FIG. 6. Best KSR-1 MD performance for the sample system compared to linear speedup: (a) time; (b) speedup.

4.2. Different Physical Systems

In order to further examine the behavior of MD, we have created artificial physical systems that differ from the base sample system in various characteristics. These systems have been derived from the original base system by mechanical processes such as particle trimming and replication. While simulations so created do not necessarily have significance to the physics community, they should give an indication of the behaviors of actual simulations with similar characteristics. In the case of scalar performance, we compare MD to the FORTRAN interaction list implementation. These comparisons are not possible for parallel MD because of the absence of a comparable parallel FORTRAN code.

The Thin64 System. One artificial system which was examined is called *Thin64*. It was created by "thinning" the base system by a factor of 64 (removing all but every 64th molecule and substrate) and placing 64 copies of the trimmed system in an 8×8 grid. The resulting system has roughly the same number of alkanes and substrate

elements as the base system, but it is considerably less dense and should have fewer interactions than the base system. More important from the point of view of MD's performance, however, is that the ratio of cutoff radius and molecular extent to the dimension of the system have been diminished by 8. The result is a substantially less coupled system.

Scalar Performance. Table II is illustrative of the variations in performance that can be countered with changes in the physical system. In this table we see that despite the fact that Thin64 has 7% more alkanes and 40% more substrate particles,[3], MD is able to accomplish an iteration in only 8.8 s using a single processor on the KSR-1. This is less than half of MD's iteration time for the base system and less than one third of the time required by the interaction list implementation. This result effectively illustrates that for some physical systems, MD's molecular locality based approach can have significant performance advantages over interaction lists.

Parallel Performance. While the results shown in Table II are compelling, there are also interesting differences when it comes to the parallel performance of MD. Figure 7 compares the iteration time and speedups for the base system and the Thin64 system. The differences in speedup when running with the Thin64 system are indicative of how the different characteristics of physical systems and the number of processors employed affect the behavior of MD. The efficiency of MD can be affected by almost every characteristic of the sample system, including density, uniformity, typical bounding box size and even typical molecule orientation. However, we do not characterize MD performance by such high-level system characteristics because, while they do interact in complex ways to affect the efficiency of the simulation, they are not variables that one is typically free to change. Instead, we focus on the value of the novel aspects of MD's design, including its use of molecular locality to reduce computational load and its flexibility with respect to the nature of the domain decomposition and the synchronization scheme employed.

Because the characteristics of the simulated physical system do impact many aspects of system performance, we

TABLE II
Computation Time Required for a Single Iteration for MD and Interaction List Approaches on a Single KSR-1 Processor for Both Base and Thin64 Systems

	Base	Thin64
Interaction List	15.4	28.1
MD	18.6	8.8

[3] The fact that the Thin64 system has more elements than the base system is an artifact of the method used to produce it.

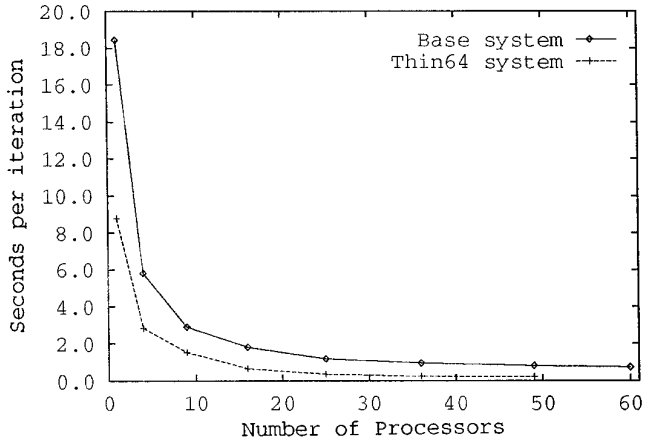

FIG. 7. MD performance results comparing the base system and the thin64 system.

will employ several other artificially generated physical systems below. Systems named Thin*xx* have been created much like Thin64, by thinning the original system by a factor of *xx* and replicating it *xx* times. This process reduces the real density while maintaining roughly the same number of particles as the original system. Systems named Flat*xx* have been created by chopping off the top 75% of the alkanes in the original system and replicating the result *xx* times. These systems have physical characteristics similar to the original system, but lower computational demands than simply replicating the original. The Flat and Thin systems respectively represent relatively tightly and loosely coupled systems of various scales. In the course of these discussions we have used a variety of these synthetic systems as appropriate to illustrate various points. These sample systems cannot span the full range of system characteristics, but they are indicative of how changes in these characteristics can affect the characteristic of MD performance.

4.3. Molecular Locality Exploitation

As discussed in Section 3.3, MD exploits molecular locality and avoids the use of interaction list techniques to reduce the overall computational load of the pairwise calculations. While it is possible to use molecular locality and interaction lists together, the results presented in Section 4.1 indicate that molecular locality exploitation alone can achieve performance similar to that of approaches involving interaction lists. However, the efficiency of molecular locality exploitation can vary depending upon a variety of factors specific to the system being simulated. In particular, the proportion of potential interactions that is rejected at each level of the nested loop performing pairwise computations varies with several characteristics of the simulated system, such as: the cutoff radius being used, the overall size of the physical system, the specific geometry of the domains, the number of particles in a molecule, and the average sizes of molecule bounding boxes. These characteristics are specific to the physical system being simulated

TABLE III
Efficacy Percentages for Locality Exploitation with a Variety of Simulated Systems and Domain Decompositions

			Level 1	Level 2		Level 3
System	Number of domains	Decomposition	Passed	Passed	Computed	Computed
Base	1		12.9%	34%	71.7%	70.1%
	8	slab	13.3%	31%	70.7%	70.5%
	27	slab	14.8%	33%	70.3%	70.2%
	64	slab	15.4%	33%	48.4%	46.8%
	64	cubic	15.2%	31%	47.8%	46.8%
Thin49	1		0.31%	87%	29.3%	65.8%
	8	slab	0.93%	78%	32.1%	71.7%
	27	slab	2.19%	75%	32.8%	70.5%
	64	slab	2.56%	75%	32.8%	70.5%
	64	cubic	1.57%	69%	36.2%	82.9%
Flat49	1		0.66%	42%	45.1%	49.7%
	8	slab	1.51%	41%	47.2%	60.6%
	27	slab	3.52%	38%	46.5%	57.9%
	64	slab	4.91%	39%	47.0%	56.3%
	64	cubic	1.65%	43%	50.0%	70.5%
Maximum beneficial value (see text)			92%	20%	60%	50%

and may even change over the course of a given simulation. This section explores issues affecting the efficiency of locality exploitation in the simulation of physical systems.

Of the conditional expressions in Fig. 5, only the third is required for correct execution of the code. The first two exist only to reduce overall computational load by eliminating some potential interactions early in the routine. The value of these conditionals depends on their cost as well as on the number of potential interactions they are able to eliminate. However, as mentioned above, the number of interactions being eliminated depends on a variety of factors. Analysis is further complicated by our attempt to exploit the chain locality discussed in Section 3.3. Exploiting chain locality means attempting to infer the results of several comparisons from a single distance calculation. Whether exploiting such chain locality is worthwhile depends both on the time required to make this inference and the quality of the predictions being made. Because this quality also varies across different physical systems, this decision must be made for each particular simulation. In summary, for each physical system the programmer must decide whether to exploit molecular locality at the first two conditionals and whether to exploit chain locality at the second and third conditionals.

Decisions concerning the efficacy of exploiting molecular and chain localities may be based on measurements evaluating (1) the computational costs of the conditional operations (compared to implementations not utilizing them), (2) the percentage of interactions eliminated at each level, and (3) for levels in which chain locality is exploited, the percentage of conditionals actually computed rather than inferred. Table III lists some of these measured values for several physical systems and for several decompositions of each of those systems.

In analyzing the benefits of locality, the interesting cases are not those in which particles actually interact, which implies that computations *must* pass through all conditionals and reach the ''compute pair forces'' stage. What impacts the efficiency of the conditional tests is at what level noninteractions are eliminated. If a first-level test can determine that two n-particle molecules can never interact, then n comparisons are saved at the second level or n^2 comparisons are eliminated at the third level (if there were no second-level test). At the same time, if the first level test takes longer than the average number of computations eliminated in the second and third level tests, then it is clearly of no benefit. To see how costs and probabilities interplay, one can examine the situation for the first conditional. In our KSR implementation, the first conditional requires 14.9 μs to compute. A simple implementation of the second conditional executes for 11.3 μs. If p is the fraction of potential interactions eliminated, then the first conditional is beneficial only if $(11.3 * n * p > 14.9)$. That is, the first conditional is beneficial if on average the cost of the checks it eliminates $(11.3 * n * p)$ is larger than its own cost (14.9).[4] For 16 particle systems ($n = 16$) then, the first conditional is beneficial if p, the fraction of molecules eliminated, is greater than 8%. That is, the first level test can pass 92% of the molecular pairs it evaluates and still be a computational improvement over doing no test and passing all pairs. A check of the ''Level 1'' column in Table III shows that all the systems studied pass many fewer

[4] This analysis is a slight simplification. It assumes that all potential interactions that would have been eliminated by the first level conditional are instead eliminated by the second level conditional, and that the execution time of the second level conditional is not affected by chain locality.

potential interactions at the first level than this 92% maximum. Thus, the use of the first level conditional is of clear benefit in all of the systems studied in our work.

The computational benefits of using the second level conditional are less apparent. An analysis using a cost of 1.75 μs for the third level conditional reveals that to be beneficial, the second conditional must eliminate at least 40% and pass less than 60% of the interactions that reach it. Table III shows that some systems pass more than 60% of the computations at level 2, so for those systems the second level conditional actually consumes more computation time than it eliminates. Unfortunately, we know of no simple property of a system that can predict whether this conditional will be costly or beneficial. The result for a particular system may even change as the system changes during simulation. A flexible implementation can choose to use the conditional only when it is effective.

An analysis of the value of exploiting chain locality yields somewhat inconclusive results for our sample physical systems. For example, the KSR implementation of the third conditional executes in 1.75 μs. If chain locality is utilized, the third conditional requires 7.1 μs when it must recompute from scratch, but only 0.5 μs to use a previously inferred decision. This implies that to be worthwhile, chain locality must infer 80% and compute at most 20% of the conditionals at this level. At the second level, the normal conditional requires 11.3 μs, while the inferring and inferred require 20.6 and 0.74 ms respectively. With these numbers, chain locality is of benefit as long as a maximum of 50% of the conditionals are computed and the rest inferred. An examination of Table III shows that for the physical systems and decompositions presented, chain locality reaps no benefit at the third level, but is sometimes beneficial at the second level. As is the case for the second level conditional, we know of no simple property of the system that can predict whether chain locality will be effective.

It is clear from the results above that the exploitation of molecular locality is an effective technique for reducing the overall computational load of the pairwise computations. However, its varying effectiveness indicates that flexibility and adaptability in the application of this technique are important to attain the highest performance. Though not directly substantiated by results here, it is intuitively clear that molecular locality exploitation will be most effective if the physical extent of the molecule (and thus of the bounding box) is small. If the molecules are flexible, like the polymers in the base system, locality efficacy may vary as the simulation proceeds and may even vary from domain to domain within a run. While static decisions about the levels at which to exploit molecular locality may provide satisfactory results, dynamic configuration techniques similar to those explored in [6] may be useful in obtaining the highest performance for a MD simulations.

4.4. Synchronization Costs

Section 3.2.1 presented three different approaches to synchronization and particle location exchange, one based on barriers and the other two based on synchronization performed between neighboring domains. It was postulated that the looser neighbor-based synchronization models would result in performance gains. Table IV presents the actual KSR-2 execution times when completing a simulation iteration for several different physical systems and decomposition strategies, with the three different synchronization styles discussed in Section 3.2.1:

Barrier synchronization–where barriers are used to synchronize the computation, shown in column *Barrier*.

Neighbor synchronization without forking–where synchronization takes place directly between processors holding neighboring domains, shown in column *Simple* under the heading *Neighbor*.

Neighbor synchronization with forking–where the intramolecular and local force calculations are performed in a separate thread to hide communication latency or to hide pauses for synchronization, shown in column *Fork* under the heading *Neighbor*.

One interesting conclusion from these measurements is that for many combinations of physical system, number of domains, and decomposition style, performance is not affected by the nature of the synchronization scheme. The use of neighbor synchronization shows a slight performance improvement over barrier synchronization only when smaller systems are spread over many processors and thus the iteration time is small. With fewer processors or with larger systems (i.e., longer iteration times), the ratio of computation to synchronization time is such that the effect of using different synchronization schemes is negligible. Similarly, there are virtually no differences between the Simple and Fork schemes. In fact, rather than reducing overall iteration time through hiding communication latency, the Fork times are uniformly slightly higher than simple neighbor synchronization.

TABLE IV
Computation Times for Several Systems with Different Synchronization Schemes

Physical system	Number of domains	Decomposition	Synchronization scheme		
				Neighbor	
			Barrier	Simple	Fork
Base	16	slab	0.908	0.908	0.909
Base	16	cubic	1.06	1.06	1.06
Thin16	16	slab	0.252	0.247	0.250
Thin16	16	cubic	0.347	0.344	0.346
Thin16	54	slab	0.134	0.120	0.122
Thin16	54	cubic	0.139	0.121	0.123
Flat16	54	cubic	1.21	1.21	1.20

One possible explanation for the relatively small differences in iteration times with varying synchronization schemes may lie in the stability of the simulation itself. The additional flexibility of neighbor synchronization is most useful if loads on individual processors vary slightly from iteration to iteration. In that circumstance, neighbor synchronization enables overlaps between iterations that that are not possible with barrier synchronization. However, if the computational load on the processors is stable, the particular processor that has the most load will tend to dominate the total execution time, and interleaving of iterations will not significantly benefit the calculation. While any implementation attribute that slows down this bottleneck processor will slow down the entire application, changes that do not cause slowdowns in this processor may not affect the overall iteration time.

A highly stable load may also negate the benefits of forking a separate thread to handle the intramolecular and local force calculations. Potentially, this thread would allow useful work to continue while a domain's main thread is waiting for synchronization with other processors. However, our experiments have shown that with a stable load the highest-load domain whose compute time dominates the iteration time never waits for synchronization. This situation results from the lower-load domains taking advantage of the flexibility in the neighbor synchronization to work ahead of their slower neighbors. Where a faster (less loaded) processor might have to wait for a slower (more loaded) one to complete some calculations, the slowest processor always finds its required data available and never pauses at synchronization points. Accordingly, there are no synchronization gaps in which the extra thread might be executed, and its creation only adds the overheads of the thread *fork*() and *join*() operations.

The experimental results presented in this section have not shown that flexibility in the synchronization mechanism is beneficial. We are conducting further experiments which should demonstrate the benefit of flexibility. As part of our work on program steering which is discussed in [2], we are developing an external system that will perform dynamic load balancing in MD. When this system is in place, the effects of changing load should demonstrate the value of the flexibility in MD synchronization schemes.

Given the computationally intensive nature of the MD simulation, the critical factor in the performance of MD on the KSR is the quality of the load balancing across different domains. Therefore, the flexibility of domain decomposition in MD is vitally important to the attainment of high performance for MD simulations. Specifically, while iteration times differ only by a few percent when using alternative synchronization methods, minor differences in load balance may cause performance degradation of as much as forty percent. This is discussed in more detail in the next section.

4.5. Load Balancing

The numbers reported for the Thin64 system in Fig. 7 were achieved using a cylindrical decomposition, where each domain is a square cylinder of simulation space. To this point, this paper has presented only the best decompositions of the systems which have been examined. The geometry of the decomposition is an important factor in MD behavior, but, in the shared-memory implementation of MD, the most important geometric factor in determining the speedups available in any particular decomposition is how well the resulting computational loads are balanced across domains. This section presents performance results that demonstrate the value of even subtle changes in decomposition geometry.

The domain decompositions used in the measurements presented in previous sections have not relied strictly on geometry to establish domain boundaries. Instead, the boundaries were allowed to shift slightly in either direction, so that roughly equal numbers of molecules are assigned to each. This balancing technique based on molecule counts is useful in the sample physical systems used in our evaluations because they can be decomposed into domains with nearly identical characteristics. Realistic MD simulations typically exhibit irregularities in system properties such as molecular distributions and densities. Such irregular systems require either a more complex model of MD computational costs, or the use of dynamic load balancing techniques to react to actual rather than predicted load.

Figure 8 depicts a view of a single dimension of a system in which molecules are somewhat evenly distributed. In this system, a strict geometric decomposition fails to take into account molecular boundaries and can, therefore, result in a poorly balanced set of domains. Allowing the domain boundaries to shift slightly generates a decomposition in which the molecules are more evenly distributed.

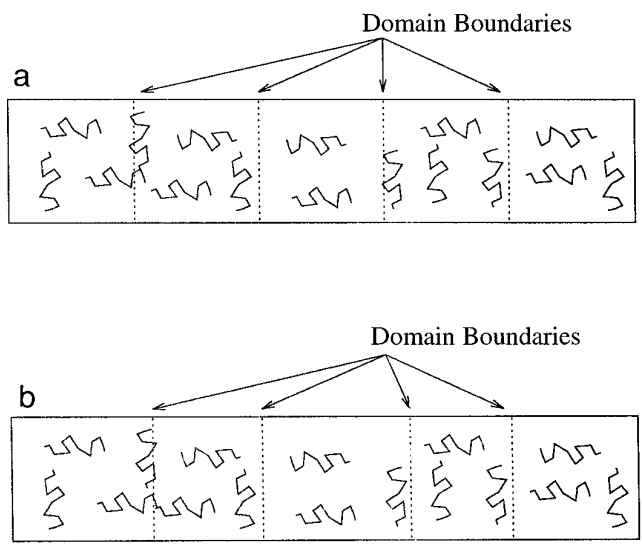

FIG. 8. Strict and balanced decompositions.

If all other characteristics are similar, the number of molecules in a domain greatly determines the time it takes to calculate an iteration for that domain. The overall time for an iteration will be lower if all the domains require about the same amount of time. So in a uniform system total iteration time is minimized if the molecules are equally distributed.

Figure 9 illustrates the beneficial effects of balancing the domains in a slab decomposition of our sample system. The balanced and geometric decompositions perform similarly if there are few processors. However, the probability of an unbalanced decomposition increases as the work is more thinly spread. As a result, the performance of the strict geometric decomposition is somewhat erratic and generally worse than the balanced decomposition by up to 40%. The results shown in Fig. 9 were achieved using neighbor synchronization with forking. The relative performances of the balanced and geometric decompositions do not vary with the synchronization scheme utilized, which is consistent with the results presented in Section 4.4.

The large differences in performance caused by minor changes in decomposition geometry demonstrate the value of MD's flexibility in domain geometry. While we have not yet experimented with systems that are much less uniform than the system described in Section 2.1, such uniformity is more likely to be the exception than the rule. A domain decomposition that was a poor match for the physical system being simulated would suffer even greater performance degradation than the example system here. These results also indicate the importance of dynamic load balancing as molecules move during the course of the simulation. The domain boundary shifts necessary to do dynamic rebalancing are among the class of simple decomposition changes that can be easily handled by mechanisms already in place in MD, as discussed in Section 3.1.

The previous section has examined the value of molecular locality exploitation and MD flexibility in synchronization mechanisms and decomposition geometry. These characteristics determine much of the behavior of the computational core of MD. The code associated with what might be considered the core amounts to little more than one third of the entire application, however. The next section discusses noncore aspects of MD.

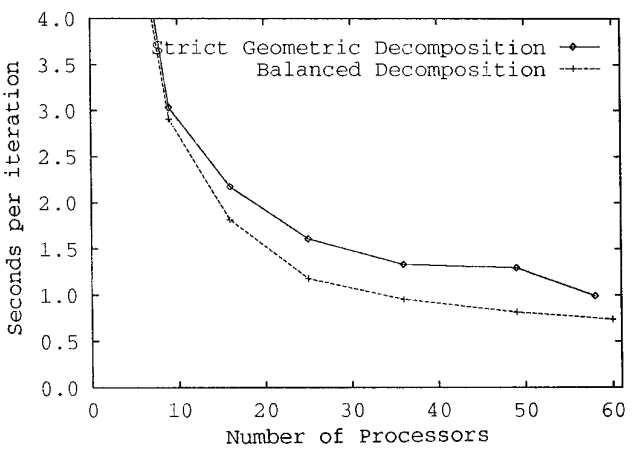

FIG. 9. Iteration times for the strict and balanced decompositions.

5. NONCORE MD

Much of the functionality of the MD simulation is outside of its core calculations. For example, the noncore code in MD performs system initialization and shutdown, calculates decomposition characteristics, initializes exports structures, performs scientific visualization, and provides for input, output, and checkpointing of molecule datasets. How important are the execution times of functions in the noncore portions of MD? The answer to this question depends on the user requirements that determine the frequencies of function execution. For example, if real-time scientific visualization is performed online as described in [2], the execution times of functions performing state collection and visualization processing can become quite important. Similarly, if the application requires frequent checkpoints for later study of particle trajectories or if rapid transfer of particle location information to another machine for online analysis is necessary, fast portable I/O becomes a major concern. Our experience with the MD simulation indicates that meeting any one of these requirements may result in significant additional computational requirements sometimes causing performance bottlenecks exceeding those presented by core computations.

Because the context of our work on program monitoring and steering requires the capabilities described in the previous paragraph, the MD system offers support for both scientific visualization and fast portable I/O. This section briefly describes MD's current approaches and capabilities and our plans for their future extension.

5.1. Binary I/O Library

One of the problems associated with large scientific applications is that they often have large datasets to read and write. Sometimes these datasets are produced on one machine and analyzed on yet another. For speed, one would like to use binary I/O formats, but writing data in binary usually makes it impossible to read them on another machine. Converting the data to human-readable ASCII allows portability, but it both increases the size of the resulting data file and greatly increases the amount of time required to read and write it. Binary files also suffer from the "forgotten format" problem. If datafiles have long lifetimes it is easy to lose track of exactly how the data file was written. If the file is in ASCII, there is some hope of examining the numbers and guessing the format, but if the file is binary there is little one can do to reclaim the data.

To address these problems, we have developed an I/O library for binary I/O that produces compact, self-describing files [1]. At the beginning of these files is information

that describes the format of the records in the file, including the names, sizes, and data types of the fields in the records. There is also information in the header about characteristics of the binary data, including the floating point format used by the machine, sizes of basic data types, and byte ordering employed. To support examination of these files, a simple utility program interprets the record format information in the header and converts the contents of the file to ASCII for easy human examination. The file format contains sufficient information to make it easy to write programs that make few assumptions about their input data. For example, MD makes no assumptions about the number of alkanes or substrate particles in its input file. It simply reads as many records as exist. All the different physical systems mentioned in this paper are created with simple programs that read and write files using these routines. In addition, binary datasets created with these routines can be compared between the KSR supercomputer and Sun SPARCstations. While these routines have already proven their utility, they require some additional development. In particular, support for converting between differing floating point formats is lacking and more support for user-defined types is necessary.

5.2. Visualization Library

Scientific visualization and data analysis are often thought of as after-the-simulation activities to be performed on stored simulation output. However, as analysis and visualization become more complex, they increasingly require parallelization themselves. This fact, coupled with the desire to allow greater scientific insight during long simulation runs, implies that the supercomputers employed for high-speed simulations will soon be called upon to perform these visualization calculations as well as core simulation computations. Accordingly, MD includes a basic visualization capability which can run concurrently with the core simulation.

Several interesting issues are raised when considering on-line MD visualization. One such issue addresses the basic tradeoffs between computational costs and visual accuracy when producing visualizations. Complex visualization techniques may provide an accurate rendition of a set of particles with appropriate lighting models, shadows, etc. However, such techniques are computationally costly, must be internally parallelized if they are to result in online visualizations, and are often too costly for their on-line use. For example, journal-quality color graphics are not suitable if the main purpose of online visualization is to get a general idea of the progress of the computation.

In response to this issue, MD initially provides online visualizations that avoid complex graphic rendering techniques. Specifically, MD's renderer first reduces three-dimensional particle locations to two dimensions via a viewing transformation and then sorts them in order of depth. Each particle is then drawn as a simple colored square with a black border. Such rendering is performed in back-to-front order so that shallower particles, which are visually closer, overwrite deeper particles (much like what is done in standard z-buffer algorithms). Rendering is parallelized only in that the production of frames is pipelined. When data becomes available for the next frame, its computation is initiated on a different processor. Therefore, if the computation of a single frame requires the same amount of time as three timesteps, then three frames will be computing concurrently. While the first frame will not appear until after the third timestep, subsequent frames will appear on every timestep, lagging the computation by three frames.

Because of the success of this approach, we are now constructing higher quality visualization support based on the Silicon Graphics GL library, and we plan to evaluate the viability of such quality for online visualization.

A second issue addressed by our research concerns the available network bandwidth, which currently limits the speed of online visualization. A variety of improvements are underway. First, we are experimenting with data compression protocols and other means to overcome the bandwidth limitations. Second, we would like to enhance the visualization capabilities. Allowing interactive control over the visualization (rotate, zoom, remove layers, etc.) and allowing different characteristics of the molecule (such as temperature, stress, etc.) to affect how it is visualized are two promising possibilities. Third, we would like to exploit the capabilities of the parallel machine to produce a more visually pleasing image. The current rendering technique can provide only primitive depth cues. Ray tracing or some other technique would produce a more comprehensible image.

6. CONCLUSIONS

This paper has presented a summary of the design of MD and an analysis of its basic performance characteristics. It has examined MD's scalability for moderate numbers of processors, presented in detail the performance of several aspects of MD, and discussed the features of MD that make it a non-trivial subject for additional studies.

The detailed performance analysis shows that the exploitation of molecular locality is a useful and viable approach to either supplement or supplant interaction-list based techniques. However, the varying efficacy of molecular locality exploitation indicates that flexibility and adaptability in the application of this technique is also important to maintain the highest performance. The importance of balancing system load shows the value of matching the geometry of system domain decomposition with the shape and structure of the physical system being simulated. This result strongly supports the conclusion that flexibility in decomposition geometry is vital for insuring high performance of MD simulations.

Our current implementation of MD for shared-memory machines was designed to fit distributed memory machines as well. We believe that the scalability of MD and the minimal impact that different synchronization strategies

have on MD performance indicate that ''design for distributed memory'' is a successful approach to creating efficient shared-memory applications. It has also allowed easy porting of MD to distributed memory platforms [4].

The source code for MD as well as the sample data used in this paper is available via WWW at http://www.cc.gatech.edu/systems/projects/MD. Cthreads, a multiprocessor threads library, is required for building MD and is available at http://www.cc.gatech.edu/systems/projects/Cthreads.

ACKNOWLEDGMENTS

We thank Bill Ribarsky and Uzi Landman for their help in getting this project started, Jian Ouyang for his work on the force calculations, and Tingkang Xia and Charles Cleveland for their help in debugging the physics of the sample system.

REFERENCES

1. Eisenhauer, G. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *Anonymous ftp from ftp.cc.gatech.edu*.
2. Eisenhauer, G., Gu, W., Kindler, T., Schwan, K., Silva, D., and Vetter, J. Opportunities and tools for highly interactive distributed and parallel computing. Technical Report GIT-CC-94-58, Georgia Institute of Technology, College of Computing, Atlanta, 1994.
3. Eisenhauer, G., Gu, W., Schwan, K., and Mallavarupu, N. Falcon—Toward interactive parallel programs: The on-line steering of a molecular dynamics application. *Proceedings of the Third International Symposium on High-Performance Distributed Computing (HPDC-3)*. IEEE Comput. Soc., San Francisco, 1994, pp. 26–34.
4. Prince Kohli, Ahamad, M., and Schwan, K. Indigo: User-level support for building distributed shared abstractions. Technical Report GIT-CC-95-36, College of Computing, Georgia Institute of Technology, Atlanta, 1995.
5. Mukherjee, B. A portable and reconfigurable threads package. *Proceedings of the Sun User Group Technical Conference*. 1991, pp. 101–112.
6. Mukherjee, B., and Schwan, K. Experiments with a configurable lock for multiprocessors. *Proc. of the Twenty Second International Conference on Parallel Processing*, Vol. 2, 1993, pp. 205–208.
7. Muller-Plathe, F., Scott, W., and van Gunsteren, W. F. PARALLACS: A benchmark for parallel molecular dynamics. *Comput. Phys. Commun.* **84** (1994), 102–114.
8. Rapaport, D. C. Multi-million particle molecular dynamics—Design considerations for distributed processing. *Comput. Phys. Commun.* **62** (1991), 217–228.
9. Smith, W. Molecular dynamics on hypercube parallel computers. *Comput. Phys. Commun.* **62** (1991), 229–248.
10. Xia, T. K., Ouyang, J., Ribarsky, M. W., and Landman, U. Interfacial alkane films. *Phys. Rev. Lett.* **69,** 13 (Sept. 28, 1992), 1967–1970.

Received February 9, 1995; revised September 27, 1995; accepted January 22, 1996

GREG EISENHAUER is a research scientist and Ph.D. student in the College of Computing at the Georgia Institute of Technology. He received the B.S. and M.S. degrees from the University of Illinois in Champaign–Urbana. He has worked at Honeywell's Systems and Research Center and at IBM's T. J. Watson Research Center. His research interests include parallel programming systems and environments and distributed object systems.

KARSTEN SCHWAN is a full professor in the College of Computing at the Georgia Institute of Technology. He received the M.S. and Ph.D. degrees from Carnegie–Mellon University in Pittsburgh, Pennsylvania. At Georgia Tech, he is also co-director of the HPPCEL laboratory for high performance computing, which is maintained jointly by computer scientists and end users of parallel machines. His current research in that context focusses on the creation of ''distributed laboratories'' in which domain experts and computer scientists can collaborate in the development, execution, and even interactive steering of complex distributed and parallel applications. His research is supported by various sponsors, including NSF, NASA, ARPA, and others.