Parallel Simulation of Carbon Nanotube Based Composites*

Jyoti Kolhe[1], Usha Chandra[2], Sirish Namilae[3], Ashok Srinivasan[1], and Namas Chandra[3]

[1] Computer Science, Florida State University, Tallahassee FL 32306-4530, USA, asriniva@cs.fsu.edu

[2] Computer and Information Sciences, Florida A&M University, Tallahassee FL 32307-5100, uchandra@cis.famu.edu

[3] Mechanical Engineering, FAMU-FSU College of Engineering, Tallahassee FL 32312, chandra@eng.fsu.edu

Abstract. Computational simulation plays a vital role in nanotechnology. Molecular dynamics (MD) is an important computational method to understand the fundamental behavior of nanoscale systems, and to transform that understanding into useful products. MD computations, however, are severely restricted by the spatial and temporal scales of simulations. This paper describes the methods used to achieve effective spatial parallelization of a MD code that is based on a multi-body bond order potential. The material system studied here is a carbon nanotube (CNT). We discuss the scientific and computational issues in the development and implementation of parallel algorithms, when the domain needs to be discretized with fine granularity. Specific issues in terms of neighbor-list computation, communication reduction, and cache awareness are delineated, with corresponding benefit in terms of speed up. Important practical problems relevant to CNT based composites are studied, and the effectiveness of various strategies reported. Our implementation achieves efficient parallelization at a finer granularity compared with published works on CNTs with complex configurations.

1 Introduction

Molecular dynamics simulation involves numerical solution of Newton's equation of motion on a set of atoms interacting through an interatomic potential. The position and velocity vectors for N atoms (6 N variables) are computed for each time step, with time steps of the order of femto seconds (10^{-15} s). Sequential computing naturally consumes significant effort even with limited system sizes, for meaningful time periods of simulation. For example, to understand a system behavior for even a 100nm^3 of material for one nanosecond, it requires five

* We wish to acknowledge the School of Computational Science and Information Technology, Florida State University, for permitting use of their Teragold (IBM SP3) supercomputer. This work is partly funded by NSF grant # CMS-0403746.

million iterations (time step 0.2 femto seconds) for about a million atoms. The computational effort is further increased when chemical interactions are required to be modeled accurately using complex multi-body potentials. For example, a problem involving the pull-out of a CNT with 3000 atoms for 800,000 time steps takes about two days of computational time on a 2GHz single processor PC running Linux. Each run is often just a component of a larger multi-scale simulation. It is therefore untenable to perform these simulations without the use of effective parallelization. Thus, for these classes of problems, parallelization is not a luxury but a necessity.

The specific class of problems addressed here involves the evaluation of mechanical properties, and should be clearly distinguished from a host of MD problems, where physical and thermodynamics properties, such as entropy and specific heat, are evaluated. The latter set of problems is usually formulated with periodic boundary conditions on all sides, representing samples of a microcanonical ensemble. Several independent simulations on the same set of atoms with differing starting states are then performed and the results are averaged in some sense to determine the physical properties. This leads to a trivially parallelizable scheme quite different from those cases when mechanical loads are applied, or when fracture can occur with no a-priori knowledge. Further, many mechanical property simulations involve a low number of atoms (e.g. 3000-10000) for long simulation times. In such cases, conventional spatial decomposition techniques often lead to fine granularity resulting in high communication overhead. Thus more care needs to be taken in the implementation to ensure that communication does not become a bottleneck.

The specific problem solved in this paper involves the determination of mechanical behavior of a CNT, either as a stand-alone unit or when connected to a matrix using chemical attachments; the latter case represents an important industrial application pertaining to CNT based composites. This is shown in Fig. 1. The CNT is pulled at a constant rate, while the polymer matrix remains fixed. One end of the attachment (which connects the CNT to the matrix) moves along with the CNT, while the other end is fixed to the matrix. The mechanical force vs. displacement response of these attachments then represents the mechanical behavior of the interface that we seek to understand. This figure also demonstrates that the physical domain is not homogeneous (it has an uneven distribution of attachments), and even for a case of 64 processors, the number of atoms per processor is of the order of only 50 for a 3000-atom CNT. Such fine granularity leads to communication cost being an important bottleneck to effective parallelization.

For the sake of completeness, and also to highlight the specifics of computational issues in MD, we present in Sect. 2, the problem description of the pull-out test, which should be carefully considered in the formulation and implementation of the parallelization scheme. In Sect. 3, we present the computational structure used, both in the serial and parallel versions of the code. In this section, we also present the details of the various schemes invoked to achieve parallelism.

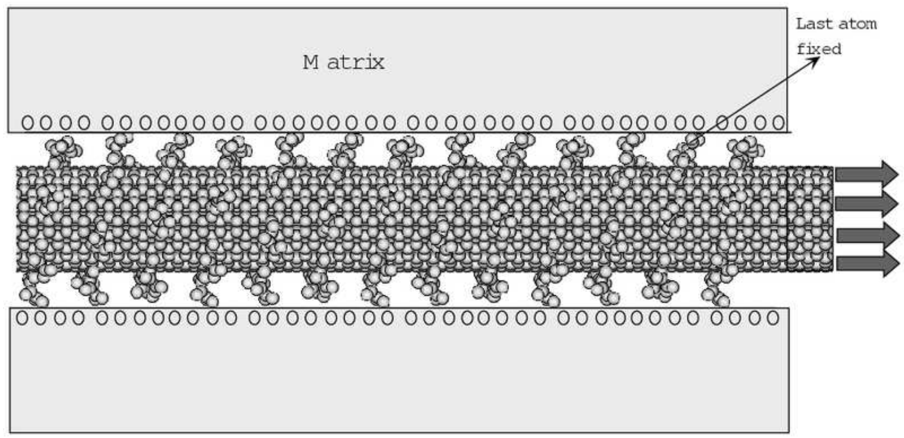

Fig. 1. Schematic of the boundary conditions applied in the pullout test simulation.

In Sect. 4, we present the timing studies that illustrate the effectiveness of the various implementation schemes, and then summarize our conclusions in Sect. 5.

2 Description of the MD Problem

Molecular dynamics is a computational simulation method that determines the position r_i and velocity v_i of every atom $i = 1, ..., N$, that is contained in a computational cell subjected to external boundary conditions (force, pressure, temperature or velocities). The basic equation of motion that solves for these $6N$ variables is given by,

$$F_i = \frac{\partial v_i}{\partial t}, \quad v_i = \frac{\partial r_i}{\partial t}, \tag{1}$$

where F_i is the force on an atom i having mass m_i. In a numerical scheme, ∂t is approximated by Δt. A variety of schemes is available to numerically integrate the above equation. Irrespective of the scheme used, the incremental time step is restricted to a very low value based on physical considerations of atomic level vibrations.

The only input needed to solve the above equation is the inter-atomic potential $E = E(r)$, which is related to force in the above equation through $F = \nabla E$. The potential energy function can be as simple as a pair-potential (forces between two atoms) or as complex as a four-body potential (three neighbor interactions) with chemical bonding effect. Truly, this function is what determines the accuracy of the MD solution. The simple Lennard-Jones potential defines a potential energy function that is based only on two-body interaction. However, in the case of CNTs considered in this work, a more complex but accurate reaction bond order potential, the Brenner potential, is required. It considers two, three,

- Initialization
- Loop over n iterations
  - Determine the neighbor list, if necessary
  - For each atom
    - Compute two-body forces, three-body forces, and four-body forces
    - Determine new positions, velocities, and its higher derivatives
  - Compute potential and kinetic energies
  - Apply thermostat: that is, change velocities to keep temperature constant.
  - Determine if neighbor list needs to be updated
- End loop

Fig. 2. Sequential algorithm for MD simulation of CNT interface.

and four-body interactions. These terms are very complex in nature and are expressed in terms of tables and implicit functions. These terms are all described in [1]. We wish to point out that the computational effort of one potential is quite different from that of others, and may require keeping track of a larger neighborhood (three neighbors compared to only one).

3 Computational Structure

In this section, we first describe the computational structure of the sequential algorithm and then discuss our optimizations. We then discuss the parallelization, and the issues that are addressed by our implementation.

3.1 Sequential Algorithm

The sequential algorithm for the simulation is given in Fig. 2. We examine the computational efforts required in each of the following components: (i) force, (ii) neighbor list, (iii) integration scheme, and (iv) thermostat computations.

Force and Neighbor List Computations. Brenner's potential can, in principle, require $O(N^2)$ time for two-body forces, $O(N^3)$ time for three-body forces, and $O(N^4)$ time for four-body forces. However, in practice, the number of neighbors of each atom is a constant, and so these take a constant time for each atom. This leads to O(N) time for the force computations, if we can determine all neighbors of an atom in constant time. This is typically accomplished through a neighbor list, which is an array that stores the neighbors of each atom. Though the list can theoretically change every time step, the physics of the problem dictates otherwise. Also, a neighbor list is re-computed only when some atom in the system has moved farther than a given threshold.

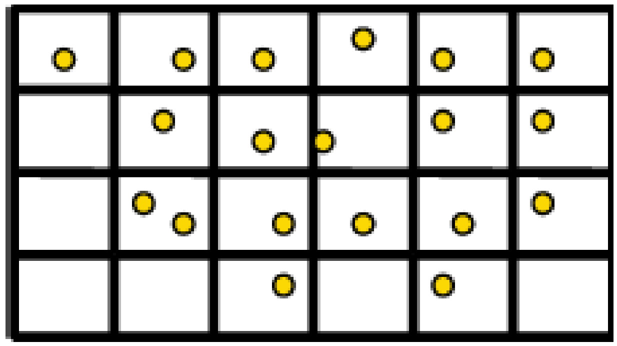

Fig. 3. Schematic of the cell-based approach to neighbor list computation.

Numerical Integration Scheme and Thermostating. Once the forces are known, acceleration can be computed from Newton's law. Positions of the atoms can be predicted by integrating the acceleration twice in succession using a numerical scheme. We use a third order Nordsieck predictor-corrector scheme. Themostating is the generic name given to maintaining a constant temperature. While generic MD uses a microcanonical ensemble (Number of atoms N, volume V, and total energy E fixed), we use a canonical ensemble (fixed N, V, and temperature T). The temperature is a linear function of the kinetic energy, which is determined by summing the squares of the velocity component of each atom. In order to maintain constant temperature, we keep the kinetic energy constant by modifying the velocities of atoms.

The force and the neighbor list computations consume more than 95% of the time taken by the algorithm. For a 3000-atom simulation, the neighbor list computation accounts for around 25% of the time taken. Computations other than the neighbor list take $O(N)$ time. The neighbor list in the original Brenner algorithm is determined by a simple scheme, which compares all pairs of atoms and determines if they are sufficiently close to be considered neighbors, leading to $O(N^2)$ time complexity. So, for a larger number of atoms, the proportion of time taken by neighbor list computations is even greater. We discuss below some of the approaches that have been taken to improve the neighbor list computations.

Srivastava [2] and Caglar [3] improve the neighbor list computation by using a cell-based approach, as shown in Fig. 3. The spatial domain studied is divided into a number of cells, with the length of a cell equal to the cutoff distance. Atoms are placed into cells based on their positions in space. A linked list is used in each cell to store the atoms located in that cell. In order to determine neighbors of an atom, while creating the neighbor list, one needs to search only neighboring cells. The worst case time complexity of $O(N^2)$ occurs if all the atoms are in the same cell. This does not happen in practice, because repulsive forces prevent such an agglomeration. In practice, the cells have only a few atoms at most, and so the time complexity of computing the neighbor list is $O(N)$.

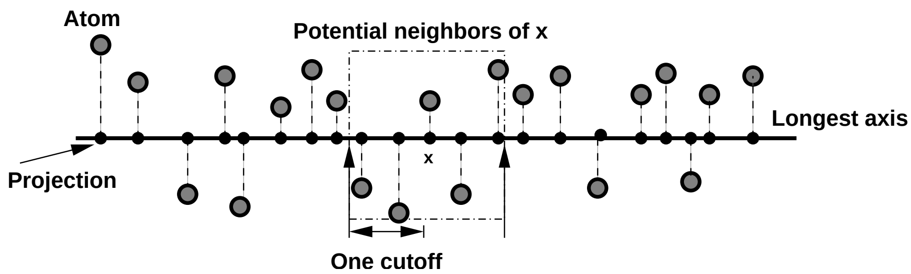

Fig. 4. Schematic of the projection-based approach to neighbor list computation.

In the problem studied here, the spatial region containing the functional groups (which connect the CNT to the polymer matrix) is sparsely populated with atoms. So, memory is often wasted, with several cells being empty. We observe that the neighbor list computation is just a range-search calculation, and so the use of a data structure such as k-d tree or range tree appears promising. But we found that in practice they are not as effective as the cell-based approach, as they actually attain their worst-case time complexities. For example, we implemented a k-d tree and found it to be much worse (by a factor of four for a 1000 atom CNT) than the cell-based approach, though it was still much better than the crude algorithm (by a factor of three). There is one other disadvantage to the cell-based algorithm, apart from memory usage; it uses indirection, due to access through pointers, and is thus less efficient than an array-based representation.

We address both these problems by observing that the CNT calculations involve long, thin regions in space. So we project the coordinates of the atoms along the axis of longest dimension (z-axis), and then sort the atoms using insertion sort based on this projection, as shown in Fig. 4. Neighbors of an atom can be determined by traversing this sorted array on both sides for a length of the cut-off distance. This not only avoids waste of memory, but also facilitates an array representation. While the worst-case time complexity is $O(N^2)$, in practice it only takes $O(N)$ time, since atoms change positions infrequently, and have only a small number of neighbors falling within the cutoff. We found that the average time for sorting and computing the neighbor list is approximately half that of the cell-based approach. There is an additional advantage to sorting. In irregular computations, data for neighboring atoms may be stored at unrelated memory locations, and so cache misses occur frequently. When we sort atoms, we move the entire data of the atoms to correspond to the sorted order, and so data of neighbors tend to be close to each other. Consequently, when data for some atom is brought into cache, it is very likely that data of its neighbors too are brought into cache, thereby reducing the likelihood of additional cache misses.

3.2 Parallelization

There has been considerable effort in parallelizing MD codes for CNT applications. In general, these efforts use domain decomposition techniques and divide the overall computational effort to multiple processors. The methods differ in the manner in which the domain is divided, with the overall objective of balancing the load, while minimizing the communication cost.

Srivastava [2] used a lexical decomposition method to parallelize the computations on a shared memory machine. Here, the atoms are divided equally amongst the processors based on their indices, in an attempt to balance the load. Auxiliary arrays were used to avoid cache misses due to false sharing. (False sharing implies that data for different atoms fall in the same cache line, and update of data for one atom invalidates that of others.) However, this random assignment of atoms does not make good use of memory bandwidth on NUMA architectures. The speedup reached in these computations was around 16 on 32 processors, with over 2500 atoms per processor.

Caglar [3] performed parallelization using a cell-based decomposition. Atoms are placed in cells, as in Fig. 3, and blocks containing equal numbers of adjacent cells are then assigned to different processors. Boundary cells on a processor require data from neighboring processors, which are obtained by message passing. Caglar obtained good speedups with granularity of less than 1000 atoms per processor. However, the geometry simulated was simple, not involving functional groups. With the use of a functionalized CNT, as in our case, the disparity in the number of atoms per cell will lead to greater load imbalance in their algorithm. In addition, the speedup results reported were for simple tensile tests. Also, the performance of the baseline sequential code used to measure speedup was quite slow. For example, their baseline sequential code took about thirty times that of our baseline sequential code. Even accounting for the difference in computing platforms (Cray T3E, versus IBM SP3 for us), the difference in speeds is substantial.

Furthermore, we solved a more complex science problem. Our MD simulations are performed at a constant temperature, which implies that the thermostat needs to be applied at every time step. Maintaining a constant temperature requires computing the global energies, which normally needs a reduction in the parallel code. Reduction is expensive in terms of communication cost. The design of the parallel code aims to achieve a good speedup with a fine granularity, even for a complex geometric domain. The parallel algorithm is shown in Fig. 5.

We project the coordinates of the atoms along the longest axis, as shown in Fig. 4. The processors can be logically considered as a linear array having two neighbors each, except at the two ends. The data of the sorted atoms is divided equally into P blocks of N/P atoms each, and each block is assigned to a different processor, where P is the number of processors. Though the initial sorting takes significant time, subsequent sorting consumes much less time, since the operations are local. Each processor needs data for not only atoms that it owns, but also for a buffer region from adjacent processors. The buffer region comprises atoms that are within three times the cut-off distance (third neighbor)

- Initialization: Sort data for atoms; each processor decides its own domain and buffer
- Loop over n iterations
  - Send and receive buffer data
  - Determine the neighbor list for local and buffer atoms, if necessary
  - For each local atom
    - Compute two-body forces, three-body forces, and four-body forces
    - Determine new positions, velocities, and its higher derivatives
  - Compute potential and kinetic energies (may need a reduction)
  - Apply thermostat: that is, change velocities to keep temperature constant.
  - Determine if neighbor list needs to be updated
- End loop

Fig. 5. Parallel algorithm for MD simulation of CNT interface.

for the border atoms, and every processor exchanges this buffer data with its immediate neighbor.

The neighbor list is recomputed for the local and buffer atoms on a processor if any local atoms moves more than a specified threshold. Also, if a neighboring processor performs a neighbor list computation, then the atoms in the buffer regions are sorted, and the neighbor list is recomputed. This is needed to ensure that neighboring processors have a consistent view of atoms they own. When a neighbor list computation is required due to local changes, a processor informs the neighbors of this, along with the message that sends the boundary data, so that the neighbors too can recompute their lists. This scheme assumes that atoms stored in each processor (other than the end processors) span a range of at least three cut-offs, in order to ensure correctness.

Apart from the computations of neighbor list updates, our algorithm has to cater to certain global operations involving total and kinetic energy of the system. Energy needs to be computed for two different reasons: (i) Energy is a system level quantity that needs to be monitored and reported. However, this information is needed only periodically, say once in 100 iterations. Therefore the reduction is performed only occasionally, and the communication overhead is small. (ii) The kinetic energy evaluation is also needed for thermostating, and this needs to be performed every iteration, which requires a reduction. In our most optimized implementation, we changed the thermostat to use the average kinetic energy of only the local atoms. This is acceptable, since thermostating is performed just to ensure that the set of velocities for a large number of atoms has the required distribution. If there are enough atoms per processor, then this leads to a scientifically acceptable result, even though the answers may not be identical to that of the sequential code.

We discuss a few more implementation issues below.

(i) *Replicating force computations:* There are symmetries in the forces being computed that can be used to reduce the computational effort. In two-body forces, for example, the force on one atom is the negative of the force on the other atom. Thus one computation is sufficient for each pair of atoms. However, if different processors own the atoms of a pair, then the processor that performed the force computation will need to communicate the value to the neighboring processor. This applies to multi-body forces also. In order to avoid this additional communication cost, we do not use symmetries for atoms close to the boundary, but make use of these symmetries for only local atoms.

(ii) *MPI Communication mode:* Use of non-blocking communication (MPI ISend/IRecv) improved communication cost by a factor of two over the blocking (MPI Send/Recv) version.

(iii) *Packing and unpacking data:* The code is mixed Fortran/C, with all the communication being performed in C routines. Instead of sending/receiving several arrays, we send a single message by packing and unpacking several arrays in each communication step. The arrays that contain the original data are defined in Fortran code, which is inherently stored in column-major order. Packing and unpacking arrays in this order resulted in a four-fold improvement in performance for this computational component, compared with our original code, which accessed the arrays in row-major order, as is natural for C code.

(iv) *Redundant computations:* One way to save on message startup cost is to perform redundant computations. Processors do not send messages to neighbors each time step, but only every k time steps. When they do send a message, they send not only the buffer, but also a buffer k times larger. After this step, a processor can perform accurate computations for its local atoms, and for atoms in $k-1$ buffer zones. In the next time step, without any additional communication, the same processor can perform accurate computations for all local atoms and $k-2$ buffer zones. Following this reasoning, communication needs to be performed only every k iterations. If message startup cost is significant, then this can lead to significant improvements in efficiency. However, this was not effective in our implementation for the following reason. Four-body forces imply that the buffer is fairly large – three cutoffs – instead of just one with two-body forces. The extra computation is not compensated for by the decrease in message startup cost on the IBM SP3.

4 Results

Pullout simulations are performed as shown in Fig. 1, while tensile tests do not include attachments to the matrix. A displacement of 0.05 Åis applied to the atoms at one end of the CNT, in a region about 15 Åin length. A similar region in the other end is held fixed. After each displacement, the system is equilibrated for 1500 time steps. The simulations are carried out until some of the hydrocarbon chains fail. Typically, a simulation lasts for 500,000 to 800,000 time steps. They were performed on an IBM SP3 with 375 MHz IBM Power3 processors. Each node consists of four processors, and has 2GB RAM.

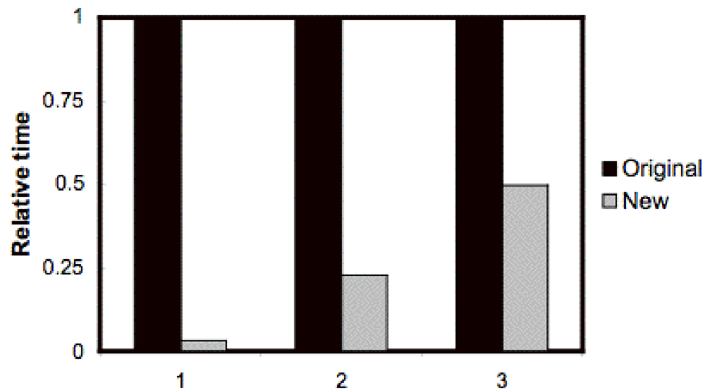

Fig. 6. Improvements in performance, relative to the original scheme. 1. Time for neighbor list computation on 3200 atoms, compared with Brenner's algorithm. The ratio (our time/Brenner's) decreases by a factor of two when the number of atoms doubles. The time relative to a cell-based algorithm is a little less than 0.5, and fairly independent of the number of atoms. 2. Time for packing and unpacking, using the cache-aware scheme, compared with a non-cache-aware one. The relative performance is independent of the number of atoms, for large numbers of atoms. 3. Time using non-blocking sends and receives, compared with blocking calls. The relative performance is independent of the number of calls and the number of atoms on an IBM SP3.

We first summarize improvements due to the optimizations in Fig. 6. Substantial improvements are obtained, especially for the neighbor list computations, which now ceases to be a bottleneck. The relative time for neighbor list computations is directly related to the number of atoms, and influences the speed of both the sequential and the parallel algorithms. The relative times for the other two techniques are independent of the number of atoms, and are relevant only to the parallel algorithm.

The speedup results are plotted below, for a 10000-atom simulation. The results shown are for tensile tests under isothermal conditions; pullout tests also show similar speedup with equivalent granularity. We can see that good efficiency is maintained up to 20 processors with a granularity of 500 atoms per processor. The efficiency reduces for a larger number of processors. An important reason for this is the replication of force computations on boundary atoms. This factor limits speedup to $P/(1 + 2bP/N)$, where $b = 50$ is the number of atoms in each buffer zone. This limits the maximum speedup on 20 processors for a 10000-atom CNT to 16.7, and on 32 processors to around 24.3. The loss in speedup beyond this is caused almost solely by the communication overhead; the load imbalance is negligible. As seen above, the code achieves a large fraction of the efficiency possible. In comparison, [2] requires a granularity greater than 2500 atoms per processor for good efficiency. Caglar [3] achieves good speedups at granularity less than 1000 atoms per processor too, but for simpler geometries

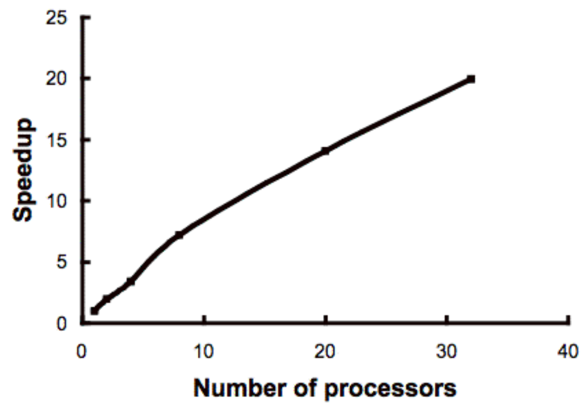

Fig. 7. Speedup results with 10000 atoms on an IBM SP3 for a tensile test.

and applications. Furthermore, the speedups in [3] are relative to a sequential code that are much slower.

5 Conclusions

We demonstrate significant improvements in sequential and parallel performance, and also explain the relative effectiveness of different optimization techniques.

One future work is to investigate the frequency with which different (two-, three-, and four-body) force components are recomputed. For example, if four-body forces change less frequently, then they need not be updated every time step. This can improve the performance of both, the sequential and the parallel codes. Such optimizations have been used by other groups, in other applications. The speedup of the parallel code can also be improved on the IBM SP3 by avoiding replication of force computations, at the expense of increased communication cost, since communication is quite fast on this machine when the MPI implementation uses the shared memory.

References

1. Brenner, D.W.: Empirical potential for hydrocarbon for use in simulating the chemical vapor deposition of diamond films. Physical Review B **42** (1991) 9458
2. Srivastava, D., Bernard, S.T.: Molecular dynamics simulation of large-scale carbon nanotubes on a shared-memory architecture. In: Proceedings of the IEEE/ACM SC1997 Conference, IEEE Computer Society (1997)
3. Caglar, A., Griebel, M.: On the numerical simulation of Fullerene nanotubes: $C_{100.000.000}$ and beyond! In et al., R.E., ed.: Molecular Dynamics on Parallel Computers, World Scientific (2000)