Parallelizing a Molecular Dynamics Algorithm on a Multiprocessor Workstation Using OpenMP

Konstantin B. Tarmyshov*,† and Florian Müller-Plathe†

School of Engineering and Science, International University Bremen, P.O. Box 750 561, 28725 Bremen, Germany

Received April 12, 2005

The atomistic molecular dynamics program YASP has been parallelized for shared-memory computer architectures. Parallelization was restricted to the most CPU-time-consuming parts: neighbor-list construction, calculation of nonbonded, angle and dihedral forces, and constraints. Most of the sequential FORTRAN code was kept; parallel constructs were inserted as compiler directives using the OpenMP standard. Only in the case of the neighbor list did the data structure have to be changed. The parallel code achieves a useful speedup over the sequential version for systems of several thousand atoms and above. On an IBM Regatta p690+, the throughput increases with the number of processors up to a maximum of 12−16 processors depending on the characteristics of the simulated systems. On dual-processor Xeon systems, the speedup is about 1.7.

1. INTRODUCTION

The investigation of polymeric materials by means of Molecular Dynamics (MD)[1] simulations usually requires simulation times in the order of nanoseconds and thus demands big amounts of CPU time. Therefore, the efficiency of MD codes becomes an acute problem. In addition, one also tries to parallelize MD calculations and thus to decrease the turn around time of a simulation.

In a typical application roughly 90% of the total CPU time is consumed by the calculation of the forces between nonbonded atoms (nonbonded forces). This part has been the target of most of the efforts aimed at speeding up MD calculations.

The evaluation of additional terms associated with bond stretching, valence angle bending, torsional deformations, and inversions (internal forces), on the other hand, has received comparatively little attention (see, however, refs 2 and 3). While the computational cost of evaluating the internal interactions tends to be smaller than that of nonbonded interactions, it is, however, by no means negligible.

In addition, one often tries to increase the length of the time step by freezing out high-frequency motions, which in general arise from vibrations of chemical bonds. This is most conveniently accomplished by modifying the equation of motion to include bond constraints[4,5] and thereby keeping the bond lengths at fixed values. For general molecular systems this is done by an iterative procedure (SHAKE).[4,6,7] Again, this step takes a minor but not negligible part of the overall computing time.

The optimization of the number of operations to be performed is the first step toward increasing the efficiency of a method. Most methods used to perform nonbonded forces calculations can be classified into two categories: neighbor-list or link-cells algorithms or hybrids of the two. In the neighbor-list method[8] all possible pairs of nonbonded atoms are examined at intervals of a fixed number of time steps and a list is made up, which contains all pairs of atoms that are closer than a specified cutoff radius r_{cut}. Only forces between pairs on the list are then considered. The neighbor list is roughly of the length $4\pi r_{cut}^3 \rho N/6$, where ρ is the particle density and N is the number of atoms. Hence, for a given density, the time for calculating the nonbonded forces from a neighbor list is proportional to the number of atoms, rather than to its square. Updating the neighbor list by searching all possible atom pairs still is an N^2 operation, but it is only done every few time steps.

In the link-cells method[9] the system is geometrically divided into cells. For each cell it is established which atoms it contains. Nonbonded interactions are then calculated only between atoms in adjacent cells. For a large enough system, the computational cost is proportional to the number of atoms.

The relative merits of both methods have been discussed extensively in the literature (see e.g. refs 10 and 11), and the discussion is not to be repeated here. The link-cells algorithm becomes competitive if the cutoff radius used in calculations is small compared to the system dimensions. For dense molecular systems of 500−1000 atoms, the neighbor list appears to be the most efficient method, while the link cells is the method of choice for very large systems.[12] The MD code, whose parallelization is described in the present paper, switches between the two methods (neighbor list and hybrid link cells/neighbor list) automatically depending on the size of the simulation box and the cutoff radius used to find neighboring atoms.

Since the number of bonds in the system is fixed, the time for the calculation of the internal forces is proportional to the number of atoms. Here, the main consumers of CPU time are the calculations of trigonometric functions for bond and torsional angles and the constraints implemented through the multicolor SHAKE[7] algorithm.

* Corresponding author phone: 0421-200-3561; fax: 0421-200-3249; e-mail: k.tarmyshov@iu-bremen.de.

† Present address: Eduard-Zintl-Institute for Inorganic and Physical Chemistry, Technical University Darmstadt, Petersenstrasse 20, 64287 Darmstadt, Germany; phone: 06151-16-6523/4; fax: 06151-16-6526.

10.1021/ci050126l CCC: $30.25
Published on Web 10/11/2005

The second step in performance improvement is the exploitation of techniques for parallelization of calculations on different levels, which have become available with present-day computers and operating systems: vector and pipelining capabilities as well as multiprocessor (MP) parallel calculations. There exist two parallel architectures: shared memory and distributed memory. In the first model many processors have access to a common (shared) part of memory, and this memory space can be used for data exchange or for storage of common data. In the second, all processors have only their own memory and have to communicate with each other, to keep up to date the data they operate with during calculations. An intermediate model is the so-called nonuniform memory access (NUMA), where a processor has fast access to "its" memory and slower access to the memory owned by other processors, whereas the shared-memory paradigm is formally preserved. Programs on NUMA-architectures have to account for the memory hierarchy to achieve optimum performance.

These two models—shared memory through OpenMP[13] and distributed memory through MPI (Message-Passing Interface)[14]—have been extensively used for parallelizing MD algorithms.[15−24] These parallelized MD algorithms are available within many different software packages such as GROMACS,[15] IMD,[16] M.DynaMix,[17] DL_POLY,[18] and others. Additionally, some interesting work has been done aimed at increasing of ease of programming and modularity of the code using object-oriented[22] and functional[25] approaches.

Within the shared memory paradigm there is no big difference between operating on shared or private (to each processor) memory, particularly if implemented using OpenMP. Typically, differences are limited to declaration and allocation of variables and arrays. The largest distinction and complication lies in controlling the threads used for parallel calculation if they are used explicitly.[26] If, on the other hand, one concentrates only on the parallelization of the most CPU-time-consuming cycles, this implies relatively small changes to the code.[19,23,26]

Two ways of the implementation of the shared memory model into an MD code are possible. The first requires changes to be done in the sequential code thereby operating (creating, synchronizing, assigning task, and etc.) with threads is done explicitly.[33] Although, such a code can be fully compatible with the parent sequential one, it becomes parallel intrinsically (see for instance ref 26). The second way needs only small changes in the code, which is still sequential in its essence. Instructions used to parallelize selected sections of the code are given in specific format in blocks, which enclose these sections. Even if explicit parallel statements (calls to some specific functions) are necessary in addition to the parallelization blocks, they still can be simply masked by directives (e.g. #ifdef/#endif directives in C/C++). The OpenMP standard[13] supports the parallelization of code through special comment block directives. Moreover, MD codes parallelized via OpenMP can be easily ported to other platforms.

In this contribution, the implementation of the most CPU-time-consuming parts of the MD code YASP[32] using OpenMP is described: nonbonded forces, the neighbor-list scheme, constraints (multicolor SHAKE[7]), dihedral and bond angle forces. The parallel implementation is done without the help of software tools such as autoparallelizing compilers, as previous experience suggests that hand-crafted parallelization usually achieves better performance. It is also explained how to build a parallel MD algorithm from its sequential robust version. Performance tests of the parallelized MD code were done on the architectures we had access to: a multiprocessor IBM Regatta p690+ under AIX 5.2 and a dual processor IBM PC (Intel Xeon 2.8 GHz, 1GB Memory) under Linux.

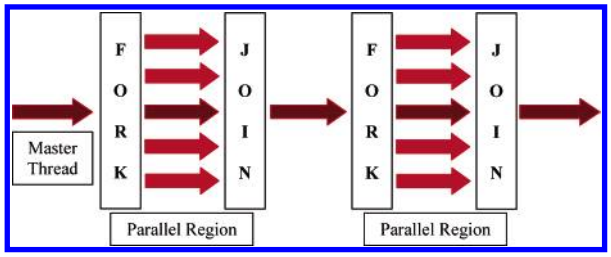

Figure 1. The "fork-join" model used by OpenMP.

2. SOME FEATURES OF OPENMP

2.1. The Model. A shared-memory process consists of multiple threads that run in the same memory space. OpenMP is based upon multiple threads. It is an explicit (not automatic) programming model, which employs so-called "fork-join" model (Figure 1) and offers a programmer full control over parallelization.

All OpenMP programs start as single process: the master thread. The master thread executes sequentially as any other program until the first parallel region is encountered. At this point the master thread creates a team of parallel threads (FORK), which then execute in parallel the program statements enclosed by the parallel region. When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread executing further (JOIN).

Virtually all OpenMP parallelism is specified by special compiler directives which are embedded into the C/C++ or Fortran source code.

2.2. Parallel Regions and Work-Sharing. A parallel region in an OpenMP program is always enclosed by the directives *parallel* and *end parallel*. All directives, which are used to share work between parallel threads, synchronize threads, and others, are placed only within these blocks. While specifying the parallel block, one also identifies those variables (scalars, arrays, etc.) that are accessed by all threads (shared) and those, which are local to each thread of the team and are not affected by other threads (private). The list of private variables can be extended specifically for some parallel construct when specifying the construct inside the parallel block.

As the main goal of the parallelization of a serial program is the reduction of the real execution time, OpenMP supports special directives to share the total amount of work between the team threads. There exist three different types of specification, which are used to share the work of the program section (work-sharing construct): *DO*, *SECTIONS*, and *SINGLE*. Since only the *DO* work-sharing construct was used, the function of *SECTIONS* and *SINGLE* constructs will not be described here.

By means of the *DO* work-sharing directive one can divide iterations of the immediately following loop into sets, which

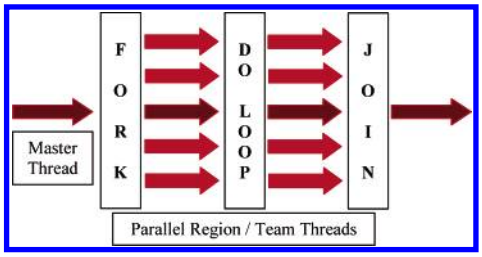

Figure 2. Schematic representation of the execution of a parallel region specified by the *DO* work-sharing construct.

are then executed in parallel by the team threads (Figure 2). It is obvious that all iterations of the loop must be absolutely independent, and program correctness must not depend on which thread executes a particular iteration.

2.3. Load Balancing. While performing any loop in parallel, it is crucial to keep all team threads working as much as possible. One needs to tune the work load balancing in order to reduce idle time of the team threads. Within OpenMP, load balancing of the parallelized loops is determined by special clauses of the *DO* work-sharing directive. There exist three different strategies for load balancing in the OpenMP standard: static load balancing, dynamic balancing, and guided balancing. In addition, IBM has introduced an extension for the Power PC IBM Aix Platform (Fortran, C/C++)—"affinity" load balancing.

Static Load Balancing. The loop iterations are divided into partitions and are then statically assigned to the team threads. Each partition is executed by the thread it was initially assigned to.

Dynamic Load Balancing. The loop iterations are split into chunks, and these chunks are dynamically scheduled among the threads. When a thread finishes one chunk, it is dynamically assigned another.

Guided Load Balancing. It is similar to dynamic load balancing. However, the size of the first chunk is taken to be [(number of loop iterations)/(number of the team threads)]. The size of each successive chunk is reduced exponentially until it reaches a specified minimum.

"**Affinity" Load Balancing.** In this hybrid scheme, iterations are initially divided into partitions equal to the number of the team threads and equal in size. Each partition is initially assigned to a thread and is then subdivided into chunks of specified size. When a thread becomes free, it takes the next chunk from its initially assigned partition. If there are no more chunks in that partition, then it grabs the next available chunk from a partition that was initially assigned to another thread.

2.4. Critical Sectioning. When one deals with multithreaded programming and common shared resources, in particular shared variables and arrays, the support of critical sectioning within parallel region becomes necessary. If during parallel execution of many threads each thread needs sometimes exclusive access to a resource common for all threads, then critical sectioning guarantees it. E.g. threads fill an array, to whose elements some value is added in each iteration. There is a situation possible when this operation is done by different threads for the same array element. In this case, the thread, which adds a value to the array element, needs to lock the access to the element for reading and writing. Otherwise two different threads could read the value of the same element simultaneously, add to it their values,

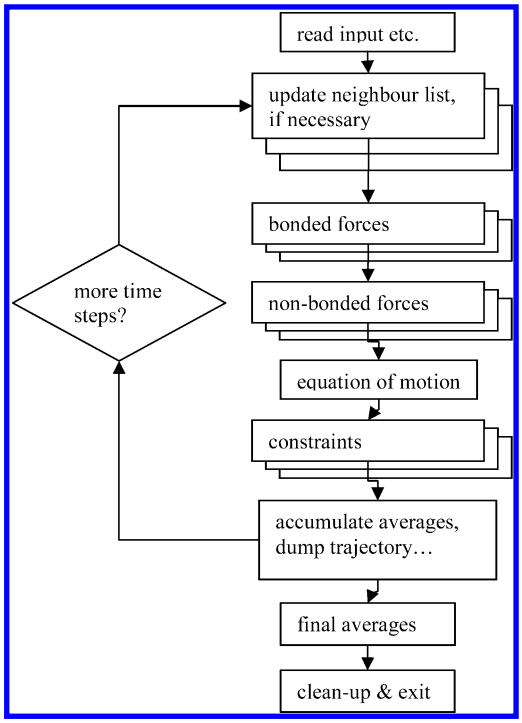

Figure 3. Schematic representation of the OpenMP molecular dynamics algorithm described in this article. The following parts of the MD code are parallelized: neighbor list, part of the bonded forces (dihedral angles, bond angles), nonbonded forces, and constraints.

and then write the results back to the array. As the same previous value of the element was read by both threads, the contribution of the first of the calculated values will be lost. The second thread will overwrite it. The block of code of the program that needs to be performed only by one thread at a time is enclosed between special clauses of OpenMP, which define the critical section.

3. PROGRAM AND DATA STRUCTURE

3.1. The Parallelization Strategy. The sequential YASP molecular dynamics program[32] has an overall structure, which is typical for many MD codes. After reading the input data and performing necessary initializations the main MD cycle is entered. Every few time steps the neighbor list is updated. In every step, the bonded and nonbonded forces are evaluated. After forces are completed, the equations of motions are integrated, and the positions evolve. If necessary, bond constraints are applied. Optionally, thermodynamic or other averages are accumulated, and the current configuration is dumped into a trajectory file. Then the new cycle step begins. When the program has executed the required number of time steps, it performs finalization (calculation of averages and fluctuations, closing of files) and exits.

In the parallel OpenMP implementation this structure is retained (Figure 3), and most parts of the program are unchanged. The majority of the tasks are performed by the master thread. The most CPU-time-consuming parts of the code (neighbor list, nonbonded forces, constraints, and some of bonded forces) are executed in parallel by all team threads. This has several advantages over a full parallelization. The most important one is the ease of implementation and of keeping a full compatibility with sequential version and the rest of the YASP package.

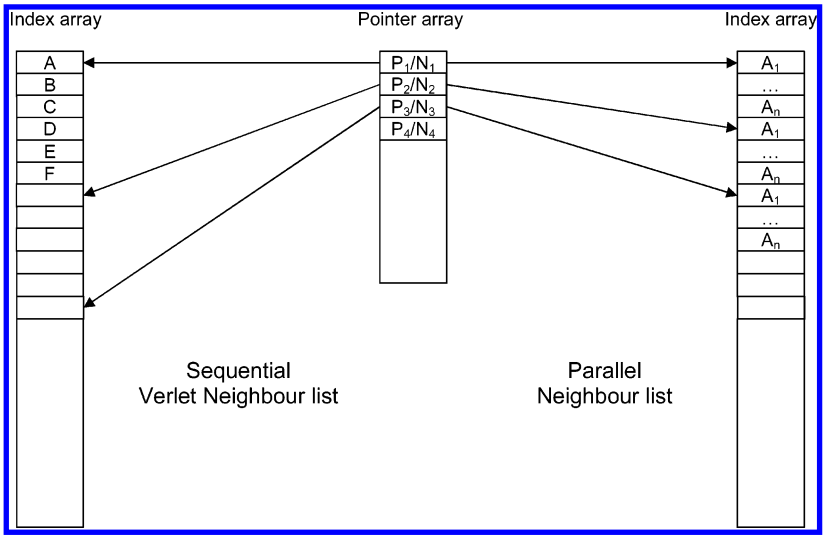

Figure 4. Modifications to the Verlet neighbor-list structure. In the parallelized version, each atom has a fixed and equal number of elements for its neighbors to be stored.

3.2. Neighbor List. Some changes are necessary in order to perform the neighbor-list update in parallel. Due to possible density fluctuations one cannot predict in advance precisely how many nonbonded neighbors every atom will have. It becomes obvious that each atom must have a reserved number of elements in the neigbor-list array where indices of neighboring atoms are stored (Figure 4). The number of elements must be large enough to be able to save all found neighbors of the atom.

For keeping neighbors for each atom in the sequential Verlet neighbor list two arrays are used (Figure 4). The index array keeps indices of neighboring atoms. The second is used to keep pointers P_i, which refer to that element in the index array, from which one the indices of the neighbors for the atom in question are stored. The number of neighbors kept for an atom i is easily calculated as $N_i^{\text{neighbors}} = P_{i+1} - P_i$. Since in the parallel version every atom has the same number of elements n reserved to keep the indices of neighbors, the start element can be computed as $P_i = n*(i - 1) + 1$ (the formula implies that array and atom indices start from 1). The pointer array is then used to keep actual number of neighbors found for each atom $N_i^{\text{neighbors}}$.

In the sequential version all atoms are examined sequentially. Therefore, the array must be long enough to keep all neighbors of all atoms. There is an imbalance in the number of neighbors. The first atom (with order number 1) has usually twice as many as atoms in the center. The last atom has no neighbors at all (Figure 5).

As, in the parallel implementation, every atom has the same number of elements reserved in the neighbor-list array, there are unused gaps, and the array is approximately twice as long as in the sequential version. The length of the parallel neighbor list is approximately $4\pi r_{\text{cut}}^3 \rho N/3$, whereas the sequential list is roughly $4\pi r_{\text{cut}}^3 \rho N/6$. According to this change of the neighbor structure, the part of the code that calculates nonbonded forces was also accordingly slightly modified. The neighbors search algorithm in this case becomes worse (i.e. doubled length of the neighbor-list array and unused gaps within it).

In principle, one could use the Brode-Ahlrichs scheme[34] for the neighbor-list construction, which does not waste memory. It is, however, difficult to integrate this scheme with the neighbor-list construction by link cells, which is used by YASP, when the system exceeds a few thousand atoms, i.e., in most cases. It would also slow the calculations, as additional if-branches would be required in the innermost loop. We, therefore, opted for the faster execution and easier implementation instead of memory economy.

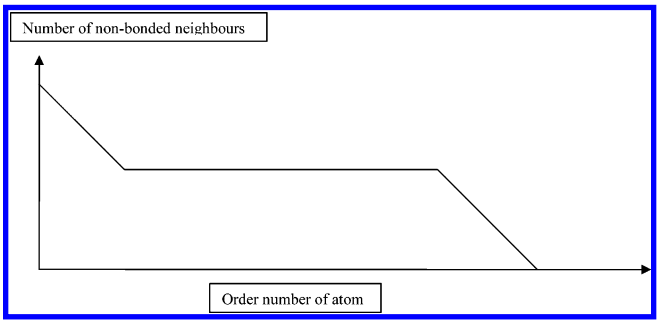

Figure 5. Schematic representation of contents of the neighbor-list array. The first atom has about twice as many neighbors as an average atom. The last atom does not have any neighbors, as it was examined last and, therefore, was already included in all possible atom pairs.

3.3. Atomic Forces. All parts of the program that evaluate atomic forces, bonded as well as nonbonded, have one thing in common. There is one cycle, which iterates over the atoms, the flexible bond angles or the dihedral angles of all molecules, calculates forces for all atoms participating in these interactions, and adds them into arrays. These arrays are used to keep the Cartesian components of the force on each atom and to integrate the equations of motion. This cycle consumes most of the CPU time of the subprogram it belongs to. The details of calculation are different in different subprograms. Chart 1 schematically presents the sequential version of a loop, which evaluates one of three types of forces—bonded angles, bonded dihedral angles, or nonbonded.

Chart 1 shows the calculation of all three components of the force (*fix*, *fiy*, *fiz*) between nonbonded atoms. These computed forces are then added into the arrays that keep atomic forces for all atoms—*fxatom*, *fyatom*, and *fzatom*. The array *nlptr* (lines *02*, *03*) is the array, which keeps the pointer to the first element of the array of neighbors *nllist* (see Figure 4). The evaluation of forces of bond angles and dihedral

Chart 1. Schematic Representation of the Part of the Sequential Code that Evaluates Atomistic Nonbonded Forces

```
! Loop over all atoms
01        DO i=1, N
            . . .
02          nlfirst = nlptr(i)
03          nllast = nlptr(i+1) - 1
04          DO m = nlfirst, nllast
05            j = nllist(m)
06            fx = <<evaluate x-component of force (i,j)>>
07            fy = <<evaluate y-component of force (i,j)>>
08            fz = <<evaluate z-component of force (i,j)>>
09            fix = fix + fx
10            fiy = fiy + fy
11            fiz = fiz + fz
12            fxatom(j) = fxatom(j) - fx
13            fyatom(j) = fyatom(j) - fy
14            fzatom(j) = fzatom(j) - fz
15          END DO
            . . .
16          fxatom(i) = fxatom(i) + fix
17          fyatom(i) = fyatom(i) + fiy
18          fzatom(i) = fzatom(i) + fiz
            . . .
19        CONTINUE
      . . .
```

angles is simpler, as they proceed along a predefined static list of interactions. Forces within each iteration are evaluated for three (bonded angles) and four atoms (dihedral angles). It needs also to be mentioned that YASP uses the "reaction field" algorithm to estimate electrostatic forces. In this case dispersive attraction described by Lennard-Jones potential and electrostatic interactions can be merged into one cycle.

Chart 2represents schematically the parallelized version of this cycle.

Similar to Chart 1, the cycle is given only for the calculation of nonbonded forces. Line *01* calculates the maximum number of elements in the neighbor-list *nmaxnl* array that each atom can have, where *mnl* is the total length of the array. Line *02* of Chart 2 defines the size of chunks, into which the whole cycle will be split later. The total number of chunks is equal to the number of threads N_{threads} multiplied by 5. The number of threads is returned by the function *OMP_GET_MAX_THREADS*. The final number of chunks can be one more than 5 times the number of threads because the number of iterations *N* might be not divisible by $5N_{\text{threads}}$. This will result in a small residual chunk.

The next two lines (*03*, *04*) open the parallel region, which is performed by all threads simultaneously. The clause *default*(*shared*) determines that, if not specified otherwise, all variables declared previously are shared by all threads. The clause *private*(*fxa_local*, *fya_local*, *fza_local*) means that local private copies of the arrays (referred to in the brackets) will be created for each thread. The number of elements in these arrays and in the shared forces arrays *fxatom*, *fyatom*, and *fzatom* is equal to the number of atoms. The local forces arrays are set to zero before the start of the main loop.

Chart 2. Schematic Representation of the Parallel Version of the Code that Evaluates Atomic Nonbonded Forces

```
      . . .
01      mmaxnl = mnl / (natom - 1)
      . . .
02    nomp_chunk = N_iter / (OMP_GET_MAX_THREADS()*5)
03 !$omp  parallel default(shared)
04 !$omp& private(fxa_local, fya_local, fza_local)

! Initialize arrays of forces
05      fxa_local = 0
06      fya_local = 0
07      fza_local = 0

08 !$omp  do private(<<variables list>>)
09 !$omp& reduction(+:<<variable list>>)
10 !$omp& schedule(dynamic, nomp_chunk)
! Loop over or all atoms
11      DO i=1, N
          . . .
12        nlfirst = (i-1)*nmaxnl + 1
13        nllast = (i-1)*nmaxnl + nlptr(i)
14        DO m = nlfirst, nllast
15          j = nllist(m)
16          fx = <<evaluate x-component of force (i,j)>>
17          fy = <<evaluate y-component of force (i,j)>>
18          fz = <<evaluate z-component of force (i,j)>>
19          fix = fix + fx
20          fiy = fiy + fy
21          fiz = fiz + fz
22          fxa_local(j) = fxa_local(j) - fx
23          fya_local(j) = fya_local(j) - fy
24          fza_local(j) = fza_local(j) - fz
25        END DO
          . . .
26        fxatom(i) = fxatom(i) + fix
27        fyatom(i) = fyatom(i) + fiy
28        fzatom(i) = fzatom(i) + fiz
          . . .
29      CONTINUE
30 !$omp end do nowait

31 !$omp critical (forces_add_lock)
32      fxatom = fxatom + fxa_local
33      fyatom = fyatom + fya_local
34      fzatom = fzatom + fza_local
35 !$omp end critical (forces_add_lock)

36 !$omp end parallel
      . . .
```

When the local arrays are initialized, the team threads start iterations of the outer loop. The comment block, which

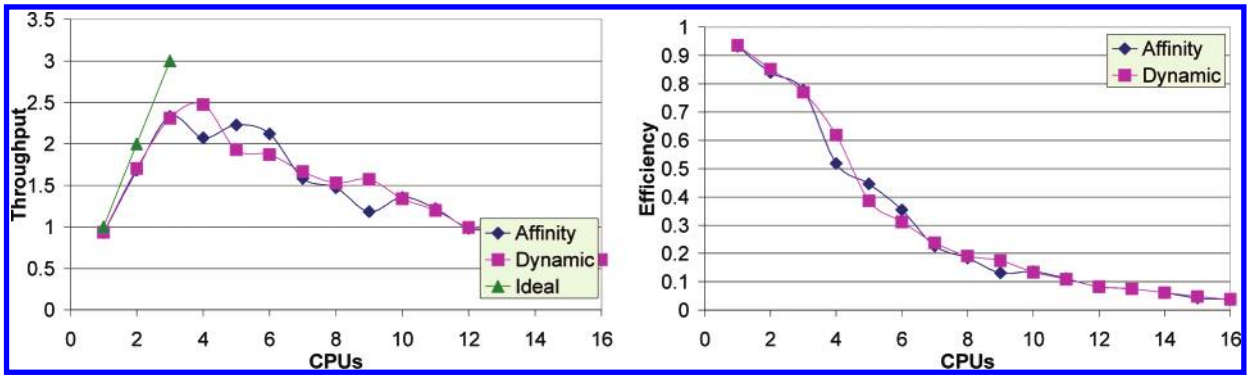

Figure 6. Throughput (left) and efficiency (right) for the system of 300 SPC/E water molecules.

precedes the cycle, establishes the parallel environment for performing the cycle and defines the strategy to execute iterations. Line *08* declares the cycle as parallel, and it defines the list of variables that should be private to each thread (in the same way as in line *04*). Line *09* declares shared variables, which serve as accumulators, i.e., in each iteration a value is added and the final outcome is the total sum for all iterations done. Examples are the virial and different energies. The last line *10* stipulates dynamic load balancing (see section 2.3).

Lines *02*−*03* of Chart 1, which define start and end elements of the neighbor-list array where neighbors of the current atom are stored (see Figure 4), are now changed and turned into lines *12*−*13* of Chart 2.

Line *30* closes the parallel cycle and also controls the behavior of the team threads upon the completion of the cycle. When a thread finishes all available chunks of iterations of the cycle, it does not wait for other threads and proceeds to the next statement following the cycle (*nowait option*).

The following code section (lines *31*−*35*) collects the forces from all threads, which are kept in the local arrays *fxa_local*, *fya_local*, and *fza_local*, into the shared arrays *fxatom*, *fyatom*, and *fzatom*, respectively. In this reduction, critical sectioning must be used in order to avoid data loss (see section 2.4). This is done by enclosing the piece of the code between lines *$omp critical* (*forces_add_lock*) and *$omp end critical* (*forces_add_lock*). The first line starts the critical section named *forces_add_lock*. The second one closes this critical section.

In the implementation of YASP, all OpenMP instruction comment blocks and additional parallel code outside these blocks are enclosed within *#ifdef/#endif* preprocessor directives. Therefore, it is possible to choose the parallel or sequential version of the MD code through compiler options.

3.4. Constraints. The multicolor SHAKE[7] is implemented in the YASP package.[32] Its idea is to split the list of constraints into sublists in such a way that no atom appears more than once in each of the sublists. This makes the algorithm vectorizable and parallelizable. The outermost cycle iterates until all constrained distances are converged. The intermediate cycle runs over the sublists of independent constraints. The innermost loop solves constraints in the current sublist and performs necessary changes to atom positions. Only this cycle is parallelizable.

While all other cycles, such as building a nonbonded neighbor list or calculation of nonbonded and bonded forces, were parallelized using the dynamic load balancing option, the cycle of constraints solver was parallelized with static balancing. Since all iterations in the innermost cycle require an equal amount of work, they are divided into partitions of the same size.

An alternative could be the Gauss-Seidel and Jacobi iterations.[35,36] These methods are the constraints solvers, which are intrinsically parallel (within one iteration), and they offer substantiated performance improvements over the standard SHAKE. As we are using the parallelizable multicolor SHAKE, we expect only a limited possible speedup from using the Gauss-Seidel and Jacobi iterations. In the view of the larger operations count of these methods, we prefer to use the simpler multicolor SHAKE.

4. ASSESSMENT OF PERFORMANCE

For the water systems, simulations were carried out for 1000 time steps (2 fs per step), and the neighbor list was updated every 15 time steps for benchmark purposes. The simulation of polyamide was done for 5000 time steps (2 fs per step), the neighbor list was updated every 30 time steps. All systems were equilibrated first. The arrangement of water molecules and polyamide chains in all systems is amorphous. All computations were performed in 64-bit precision. No other computations were running on the computers, while the benchmarks were taken.

Four work load balancing schemes for calculating the forces—static, dynamic, guided, and "affinity"—were investigated. The performance for static and guided balancing turned out to be unacceptable even for two processors and is, therefore, not reported here.

Defining W as the amount of work done, and t as the corresponding execution time, the throughput with respect to some reference case can be defined as $(W/W_{ref})(t_{ref}/t)$, where W_{ref} and t_{ref} are the work done in the reference case and time it takes, respectively. The reference is the sequential version of YASP, not the parallel version running on 1 CPU. Since the amount of work is the same for the sequential and the parallel version, W/W_{ref} is 1. The efficiency is obtained as the throughput divided by number of processors. The following figures (Figure 6−10) show throughput and efficiency as a function of the number of processors for all benchmark systems.

Each figure displaying throughput has an orientation line (triangles) that represents the ideal case of a 100% efficient parallel computer. In Figures 6−10 throughput and efficiency of affinity (rhombus) and dynamic (squares) workload balancing strategies are shown.

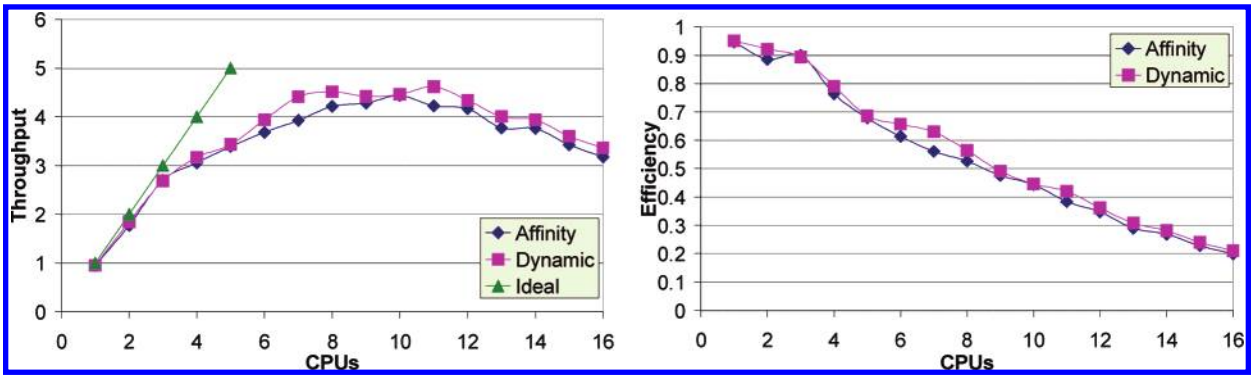

Figure 7. Throughput (left) and efficiency (right) for the system of 3000 SPC/E water molecules.

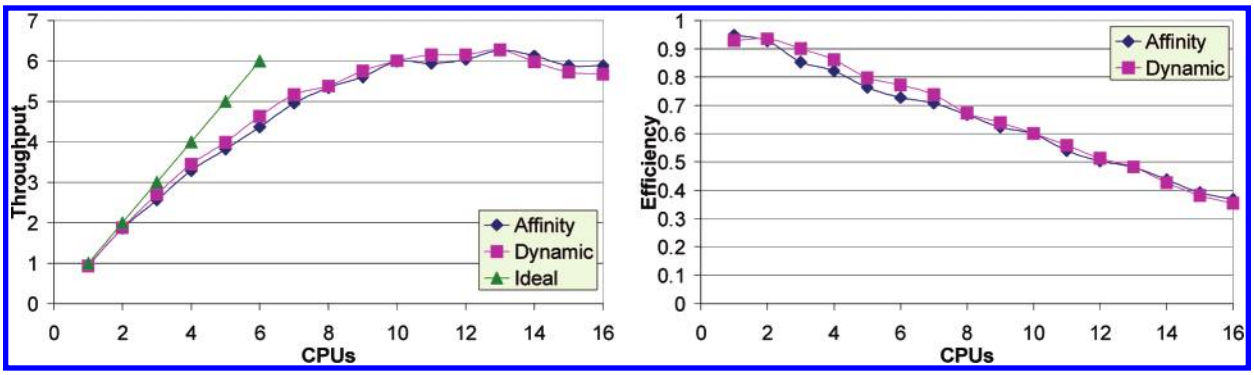

Figure 8. Throughput (left) and efficiency (right) for the system of 9000 SPC/E water molecules with cutoff radius 0.75 nm.

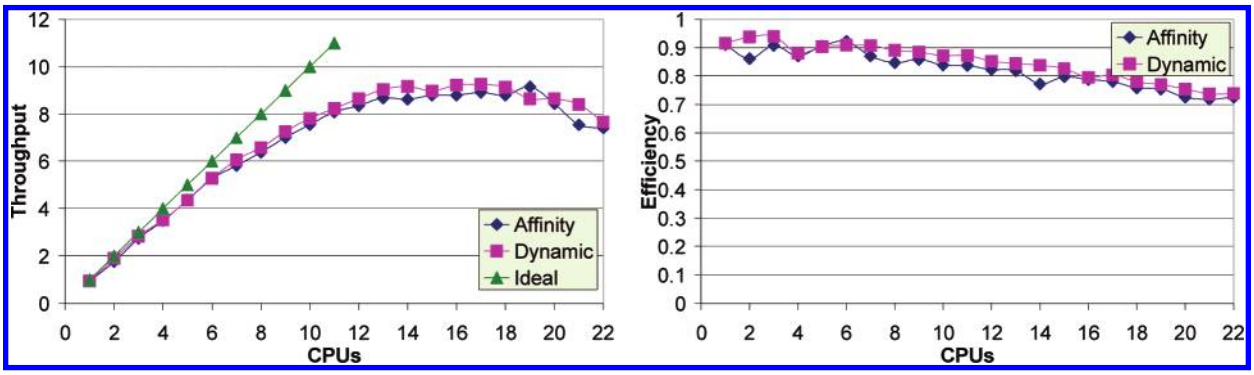

Figure 9. Throughput (left) and efficiency (right) for the system of 9000 SPC/E water molecules with cutoff radius 1.0.

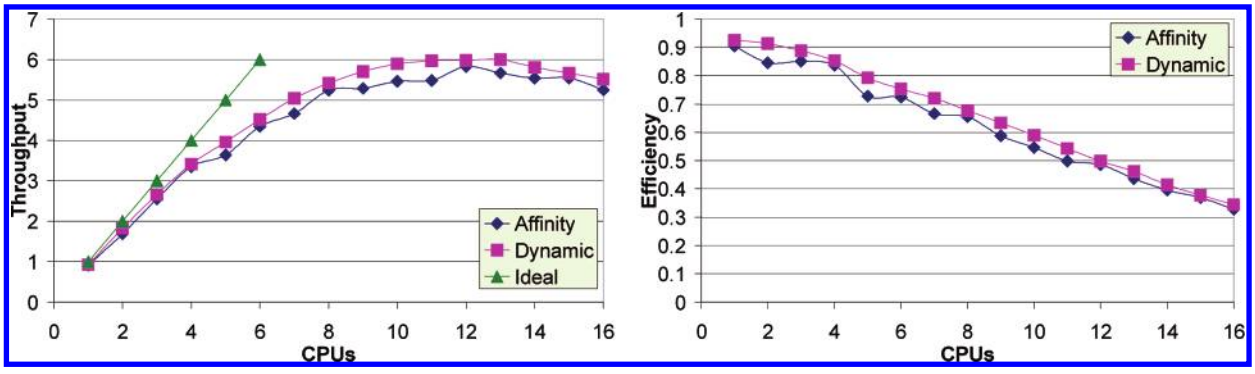

Figure 10. Throughput (left) and efficiency (right) for the system of 24 polyamide 6, 6 (nylon) molecules (totally 18 360 atoms).

For both parallelization strategies the efficiency drops with each new processor added. Except for the 300 water molecules, the decrease is approximately linear. The charts reveal that there also exists a limit of number of processors (LNP), beyond which throughput starts to decrease. Such behavior develops due to an increase of CPU overhead, such as synchronization of threads during access to critical sections of code; FORK/JOIN sections require more CPU time to treat more threads (especially in constraints solver cycle); distributing workload among the threads; inefficient memory access; memory and bus conflicts. Beside these factors, the sequential parts of the code, e.g. solution of the equations of motion, sampling averages, output of intermediate information into files, always take the same time. The CPU time

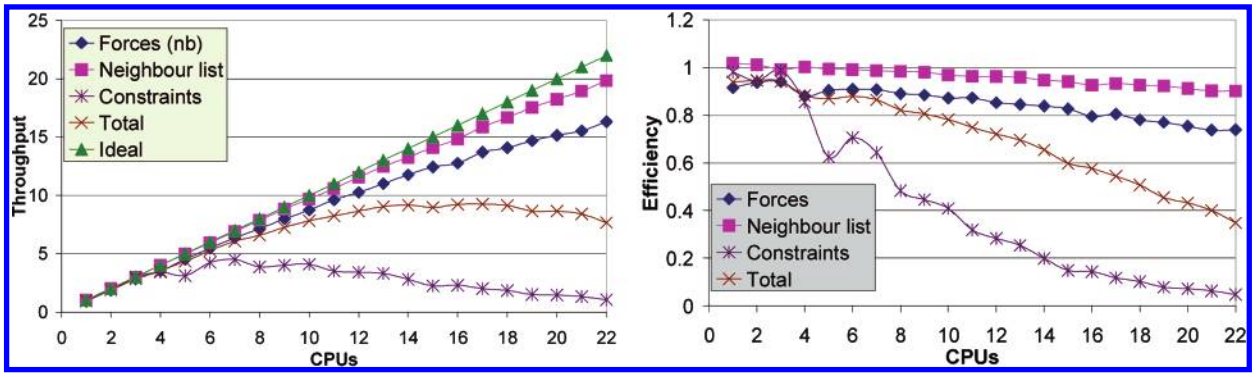

Figure 11. Separate throughput (left) and efficiency (right) of nonbonded-forces, neighbor list, and constraints for the system of 9000 SPC/E water molecules with cutoff radius 1.0. The throughput and efficiency of bonded forces are not shown, since SPC/E water molecule is rigid, and, therefore, does not have any bonded forces.

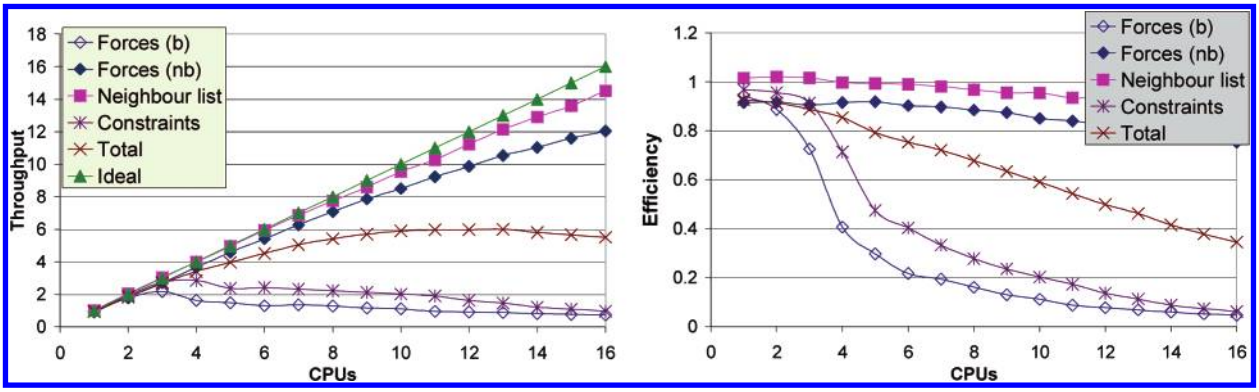

Figure 12. Separate throughput (left) and efficiency (right) of bonded forces, nonbonded-forces, neighbor list, and constraints for the system of 24 polyamide (6,6) (nylon) molecules.

required for the sequential code in the benchmarks is about 1−5% of the total time. Therefore, the maximum achievable speedup (for 5% of sequential part) in the ideal case of parallelization at the limit of an infinite number of processors according to Amdahl's law is 20.

It is seen from the graphs (Figures 6−10) that the LNP depends on the size of the molecular system being studied (Figures 6−8), atom (not mass) density, cutoff radius (Figures 8 and 9), and complexity of the molecules in the system (Figure 10).

The results shown in Figures 6−10 were extracted with all timings done on a dedicated node, so there were not any other programs running simultaneously. Under real conditions, when a node is shared with other running calculations, throughput and efficiency were found to be even more favorable.

Additionally, the separate throughputs for bonded forces, nonbonded forces, neighbor list, and constraints were investigated in order to understand the behavior shown in Figures 6−10. Since in almost all cases dynamic scheduling yielded slightly better results, affinity scheduling is not considered in the following. Two benchmarks were used to get separate timings of mentioned parts of the code: 9000 SPC/E water molecules with 1.0 nm cutoff radius and 24 molecules of polyamide.

The throughput (Figures 11 and 12) shows that the neighbor-list search algorithm, although not fully optimized, scales relatively well. The computing of nonbonded forces showed slightly worse but still useful scaling behavior. However, the parallel computation of bonded forces (Figure 12) shows an extremely poor increase in throughput, whose maximum is achieved with only 3 processors. In the case of constraints, the throughput also showed bad scaling with a maximum of 7 processors for water (Figure 11) and only 4 processors for polyamide (Figure 12). It is obvious that bonded forces and constraints are the parts of the parallelized code which define the LNP. It is achieved when the throughput decrease due to bonded forces and constraints overcomes the increase due to neighbor list and nonbonded forces.

In principle, the situation could be improved by applying the scheme of parallelization offered in ref 26, where bonded forces are calculated simultaneously with the computing of nonbonded forces (constraints can be resolved only after all forces are computed and the equation of motion is integrated). However, all timings of the polyamide benchmark demonstrated that the elapsed time taken by the calculation of bonded forces in the sequential version is always less than the elapsed time spent for the computation of nonbonded forces regardless of the number of processors used. Therefore, it is possible to estimate the total time t^n_{est} for an ideal parallel version on n processors ($n \geq 2$) as $t^n_{\text{est}} = t^n_{\text{tot}} - t^n_{\text{tot.f.}} + t^{n-1}_{\text{n.b.f.}}$, where t^n_{tot} is the total elapsed time, $t^n_{\text{tot.f.}}$ is the total elapsed time consumed for calculations of all forces, and $t^{n-1}_{\text{n.b.f.}}$ is the elapsed time spent for calculations of nonbonded forces. The ideal case implies that all possible additional time overheads are neglected.

Figure 13 shows the estimated throughput of the benchmark with polyamide in this hypothetical case. The LNP is still the same for both cases and is equal to 13 processors. The difference in throughput at this point is approximately 22%.

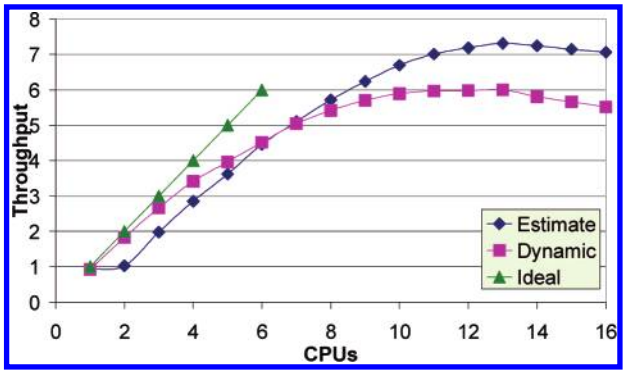

Figure 13. Estimation of throughput for the hypothetical case of simultaneous calculation of bonded and nonbonded forces for benchmark with polyamide (6,6) (nylon), compared to the throughput achieved without overlapping these tasks and using dynamic load balancing.

In principle, overlapping bonded and nonbonded forces could be done by enclosing function calls of bonded and nonbonded forces by the SECTIONS clause. This would be technically cumbersome and is not supported by all compilers, and one would have to use nonstandard constructs. For instance, the Portland Group Fortran 90 (PGF90) compiler does not allow a parallelized DO loop to be enclosed within SECTIONS clause. As the performance gain is limited, we have chosen not to implement a simultaneous calculation of all forces.

Another possible solution to increase the LNP for the whole simulation would be to freeze the number of processors used for constraints and bonded forces at their respective maxima. This was not successful. The system was that of 9000 of the SPC/E water molecules, the total number of processors was fixed at 22. The number of processors (threads) used to calculate constraints through the special option of the *$omp parallel do* clause—*num_threads*(*N*)—was systematically reduced from 22 down to 17. The total throughput continued to reduce for all steps (fewer threads, less throughput), although one would have expected an increase. The reasons are not yet understood.

The benchmarks were also run on a dual processor PC (Intel Xeon 2.8 GHz, 1GB Memory). The PGF90 compiler was used with the compiler flags "-fast −fastsse -Knoieee" for both the sequential and the parallel version. Table 2 represents results achieved on this machine.

It is seen from Table 2 that the throughput and efficiency of the parallel MD code depend on the system size very weakly. However, they have a noticeable dependence on the cutoff radius and even more on the chemical complexity of the system.

5. SUMMARY AND CONCLUSIONS

A partially parallelized MD code for shared memory computers is described, which achieves a substantial speedup over the sequential version of the program. The parallel implementation using OpenMP constructs is relatively easy because only the most CPU-time-consuming cycles, i.e., calculation of nonbonded forces, building nonbonded neighbor list, application of constraints (multicolor SHAKE), and evaluation of dihedral angle and bond angle forces, were parallelized. The rest of the code is taken from the sequential version. The approach takes a few small modifications to the structure of the nonbonded neighbors array and the implementation of the algorithm that fills it. Additionally, the evaluation of the forces (nonbonded, dihedral and bond angle) needs to be synchronized between the team threads. The resulting parallel version is fully compatible with the parent sequential version of the YASP program.

The efficiency and throughput of the parallel MD code was found to increase with the density and size of the simulated system. This means that the parallel version will be useful for simulating larger jobs. It has also been found that the throughput and efficiency are higher for systems of simpler molecules, such as molecular fluids.

For every given molecular system, there is a limit of the number of processors that can be usefully employed. It directly depends on the density, the system size, and the complexity of the molecular compounds studied, as these system characteristics determine the fraction of CPU time consumed by the parallelized cycles. The parallelized computing of nonbonded forces showed slightly worse scaling behavior than that one of the neighbor-list generation algorithm. The parallel calculation of bonded forces and constraints were found to be the main reasons, which limit the increase of the throughput and which even cause a

Table 2. Benchmark Systems Used To Test Parallel MD Code

	execution time, s	throughput	efficiency
Water, 300 K, 300 Molecules (900 Atoms)			
sequential	38	=1	=1
parallel, 1 CPU	42	0.91	0.91
parallel, 2 CPU	22	1.73	0.86
Water, 300 K, 3000 Molecules (9000 Atoms)			
sequential	442	=1	=1
parallel, 1 CPU	487	0.91	0.91
parallel, 2 CPU	248	1.78	0.89
Water, 300 K, 9000 Molecules (27 000 Atoms)			
sequential	1344	=1	=1
parallel, 1 CPU	1468	0.92	0.92
parallel, 2 CPU	769	1.75	0.87
Water, 300 K, 9000 Molecules (27 000 Atoms), $r_{cut} = 1.0$			
sequential	3085	=1	=1
parallel, 1 CPU	3269	0.94	0.94
parallel, 2 CPU	1697	1.82	0.91
Polyamide 6,6 (Nylon), 350 K, 24 Molecules (18 360 Atoms)			
sequential	5254	=1	=1
parallel, 1 CPU	7181	0.73	0.73
parallel, 2 CPU	3726	1.41	0.71

Table 1. Benchmark Systems Used To Test Parallel MD Code

	system size			cutoff radius, nm		neighbor-list length	
name	molecules	atoms	mass density, kg/m^3	potential	neighbor list	sequential total (per atom)	parallel total (per atom)
SPC/E water, 300 K	300	900	1017	0.75	0.8	118 074 (131)	236 149 (262)
SPC/E water, 300 K	3000	9000	1004	0.75	0.8	1 167 345 (130)	2 334 690 (260)
SPC/E water, 300 K	9000	27 000	1001	0.75	0.8	3 487 887 (129)	6 975 775 (258)
SPC/E water, 300 K	9000	27 000	1000	1.0	1.1	9 068 451 (336)	18 136 902 (672)
polyamide 6,6 (Nylon), 350 K	24	18 360	1083	0.9	1.0	5 092 068 (277)	10 184 136 (555)

decrease of the throughput beyond a certain number of processors.

The neighbor-list generation algorithm is not fully optimized, as it requires twice as much memory as the sequential version. However, measured timings revealed very good scalability of the parallelized version. Since memory is not a big issue on modern computers, especially on supercomputers, one can disregard this concern in most applications.

The speedup obtained with the OpenMP implementation is very similar to that found in similar problems, see e.g. refs 19 and 23. The speedup obtained on the dual-processor IBM PC (Intel Xeon 2.8 GHz, 1GB Memory, Linux) using the PGF90 compiler differs from that achieved on the Power PC IBM Aix 4.3. The efficiency and throughput are found notably smaller on the IBM PC. Still they are high enough to be useful for practical calculations.

In summary, the OpenMP technique has been found quite useful if one wants to parallelize an existing sequential version of a standard MD code, while keeping the effort to a minimum.

ACKNOWLEDGMENT

The authors are thankful to John von Neumann-Institut (NIC) für Computing for the granted time on IBM Regatta p690+ machines (JUMP) located at the ZAM Center, Jülich, Germany.

REFERENCES AND NOTES

(1) Allen, M. P.; Tildesley, D. T. *Computer Simulation of Liquids*; Oxford University Press: Oxford, 1987.
(2) Noid, D. W.; Sumpter, B. G.; Wunderlich, B.; Pfeffer, G. A. Molecular Dynamics Simulation of Polymers: Method for Optimal FORTRAN Programming. *J. Comput. Chem.* **1990**, *11*, 236−241.
(3) Teleman, O.; Svensson, B.; Jönsson, B. Efficiency in Statistical Mechanical Simulations of Biomolecules−Computer Programs for Molecular and Continuum Modelling. *Comput. Phys. Commun.* **1991**, *62*, 307−326.
(4) Ryckaert, J.-P.; Ciccotti, G.; Berendsen, H. J. C. Numerical Integration of the Cartesian equations of Motion of a System with Constraints: Molecular Dynamics of *n*-Alkanes. *J. Comput. Phys.* **1977**, *23*, 327−341.
(5) Ciccotti, G.; Ryckaert, J.-P. Molecular Dynamics Simulation of Rigid Molecules. *Comput. Phys. Rep.* **1986**, *4*, 346−392.
(6) McCammon, J. A.; Harvey, S. C. *Dynamics of Proteins and Nucleic Acids*; Cambridge University Press: Cambridge, 1987.
(7) Müller-Plathe, F.; Brown, D.; Multi-colour Algorithms in Molecular Simulation: Vectorization and Parallelisation of Internal Forces and Constraints. *Comput. Phys. Commun.* **1991**, *64*, 7−14.
(8) Verlet, L. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.* **1967**, *159*, 98−103.
(9) Hockney, R. W.; Eastwood, J. W. *Computer Simulation Using Particles*; McGraw-Hill: New York, 1981.
(10) Van Gunsteren, W. F.; Berendsen, H. J. C.; Colonna, F.; Perahia, D.; Hollenberg, J. P.; Lellouch, D. On Searching Neighbors in Computer Simulations of Macromolecular Systems. *J. Comput. Chem.* **1984**, *5*, 272−279.
(11) Morales, J. J.; Rull, L. F.; Toxvaerd, S. Efficiency Test of the Traditional and the Link-cell Methods. *Comput. Phys. Commun.* **1989**, *56*, 129−134.
(12) Rapaport, D. C. Multi-million Particle Molecular Dynamics I. Design Considerations for Vector Processing. *Comput. Phys. Commun.* **1991**, *62*, 198−216.
(13) Dagum, L.; Menon, R. OpenMP: an Industry Standard API for Shared-memory Programming. *IEEE Comput. Sci. Eng.* **1998**, *5* (1), 46−55.
(14) M. P. I. Forum. MPI: A Message-passing Interface Standard. *Int. J. Supercomput. App.* **1994**, *8* (3/4), 165−416.
(15) Berendsen, H. J. C.; van der Spoel, D.; van Drunen, R. GROMAmerican Chemical Society: A Message-passing Parallel Molecular Dynamics Implementation. *Comput. Phys. Commun.* **1995**, *91*, 43−56.
(16) Stadler, J.; Mikulla, R.; Trebin, H.-R. IMD: A Software Package for Molecular Dynamics Studies on Parallel Computers. *Int. J. Modern Phys.* **1997**, *8* (5), 1131−1140.
(17) Lyubartsev, A. P.; Laaksonen, A. M. DynaMix−a Scalable Portable Parallel MD Simulation Package for Arbitrary Molecular Mixtures. *Comput. Phys. Commun.* **2000**, *128* (3), 565−589.
(18) Forester, T. R.; Smith, W. *DL_POLY User Manual*; CCLRC, Daresbury Laboratory, Warrington, U.K., 1995.
(19) Couturier, R.; Chipot, C. Parallel Molecular Dynamics Using OpenMP on a Shared Memory Machine. *Comput. Phys. Commun.* **2000**, *124*, 49−59.
(20) Straatsma, T. P.; Philippopoulos, M.; McCammon, J. A. NWChem: Exploiting parallelism in Molecular Simulations. *Comput. Phys. Commun.* **2000**, *128*, 377−385.
(21) Roy, S.; Jin, R. Y.; Chaudhary, V.; Hase, W. L. Parallel Molecular Dynamics Simulations of Alkane/Hydroxylated α-Aluminium Oxide Interfaces. *Comput. Phys. Commun.* **2000**, *128*, 210−218.
(22) Trobec, R.; Sterk, M.; Praprotnik, M.; Janezic, D. Implementation and Evaluation of MPI-based Parallel MD Program. *Int. J. Quantum Chem.* **2001**, *84*, 23−31.
(23) Goedecker, S. Optimization and Parallelization of a Force Field for Silicon Using OpenMP. *Comput. Phys. Commun.* **2002**, *148*, 124−135.
(24) Özdogan, C.; Dereli, G.; Cagin, T. O(N) Parallel Tight Binding Molecular Dynamics Simulation of Carbon nanotubes. *Comput. Phys. Commun.* **2002**, *148*, 188−205.
(25) Scaife, N.; Hayashi, R.; Horiguchi, S. *IEICE Trans. Inf., Syst.* **2003**, *E86-D* (9), 1569−1576.
(26) Müller-Plathe, F. Parallelising a Molecular Dynamics Algorithm on a Multi-processor Workstation. *Comput. Phys. Commun.* **1990**, *61*, 285−293.
(27) Müller-Plathe, F.; Scott, W.; van Gunsteren, W. F. PARALLACS: a Benchmark for Parallel Molecular Dynamics. *Comput. Phys. Commun.* **1994**, *84*, 102−114.
(28) Janak, J. F.; Pattnaik, P. C. Protein Calculations on Parallel Processors. I: Parallel Algorithms for the Potential Energy. *J. Comput. Chem.* **1992**, *13*, 533−538.
(29) Janak, J. F.; Pattnaik, P. C. Protein Calculations on Parallel Processors. II: Parallel Algorithm for the Forces and Molecular Dynamics. *J. Comput. Chem.* **1992**, *13*, 1098−1102.
(30) Plimpton, S. J.; Hendrickson, B. A.; Heffelfinger, G. S. A New Decomposition Strategy for Parallel Bonded Molecular Dynamics. In *Proc. 6th SIAM Conference on Parallel Processing for Scientific Computing*; Reed, D. A., Ed.; Society for Industrial & Applied: 1993; pp 178−182.
(31) Clark, T. W.; Hanxleden, R. V.; McCammon, J. A.; Scott, L. R. Parallelization Using Spatial Decomposition for Molecular Dynamics. In *Scalable High Performance Computing Conference*; IEEE Computer Society Press: 1994; pp 95−102.
(32) Müller-Plathe, F. YASP: a Molecular Simulation Package. *Comput. Phys. Commun.* **1993**, *78*, 77−94.
(33) Rapaport, D. C. *The Art of Molecular Dynamics Simlations*, 2nd ed.; Cambridge University Press: New York, 2004.
(34) Brode, S.; Ahlrichs, R. An optimized MD program for the vector computer cyber 205. *Comput. Phys. Commun.* **1986**, *42* (1), 51−57.
(35) Barrett, R.; Berry, M.; Chan, T. F., Demmel, J.; Donato, J.; Dongarra, J.; Eijkhout, V.; Pozo, R.; Romine, C.; Van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed.; SIAM: Philadelphia, PA, 1994.
(36) Clark, T. W.; v. Hanxleden, R.; Kennedy, K.; Koelbel, C.; Scott, L. R. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*; Williamsburg, VA, 1992; pp 98−105.

CI050126L