4. Parallelizing a molecular dynamics code

4.1. 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 non-bonded atoms (non-bonded 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.[2,3]). While the computational cost of evaluating the internal interactions tends to be smaller than that of non-bonded 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 non-bonded forces calculations can be classified into two categories: neighbour-list or link-cells algorithms, or hybrids of the two. In the neighbour-list method[8] all possible pairs of non-bonded 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 neighbour 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 non-bonded forces from a neighbour list is proportional to the number of atoms, rather than to its square. Updating the neighbour 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. Non-bonded 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. ref.[10,11]), 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 to 1000 atoms, the neighbour-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 (neighbour-list and hybrid link-cells/neighbour list) automatically depending on the size of the simulation box and the cutoff radius used to find neighbouring 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 multi-colour SHAKE[7] algorithm.

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 multi-processor (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, in order to keep up to date the data they operate with during calculations. An intermediate model is the so-called non-uniform 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[27]. Although, such 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[28] using OpenMP is described: non-bonded forces, the neighbour-list scheme, constraints (multi-colour 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 multi-processor IBM Regatta p690+ under AIX 5.2 and a dual processor IBM PC (Intel Xeon 2.8 GHz, 1GB Memory) under Linux.

4.2. Some features of OpenMP

4.2.1. 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 and offers a programmer full control over parallelization (Fig. 4.1).

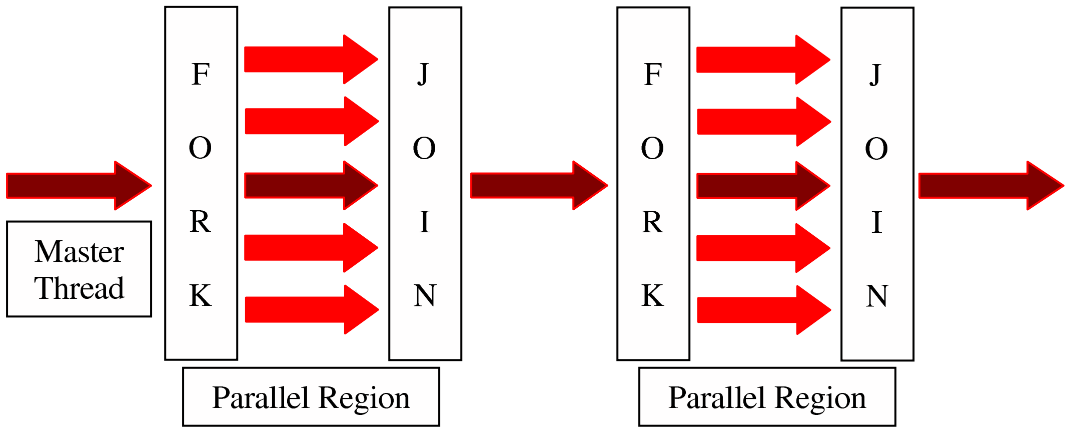


Figure 4.1. The "fork-join" model used by OpenMP.

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.

4.2.2. 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 are then executed in parallel by the team threads (Fig. 4.2). It is obvious that all iterations of the loop must be absolutely independent and program correctness must not depend upon which thread executes a particular iteration.

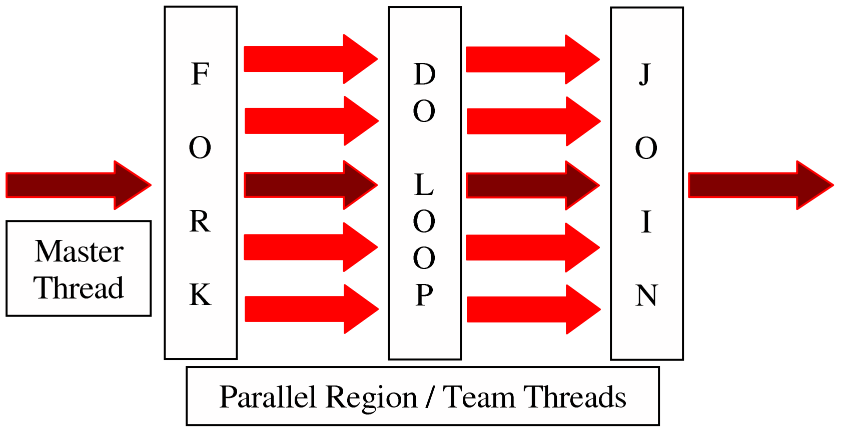


Figure 4.2. Schematic representation of the execution of a parallel region specified by the *DO* work-sharing construct.

4.2.3. 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. 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.

4.2.4. 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, 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.

4.3. Program and data structure

4.3.1. The parallelization strategy

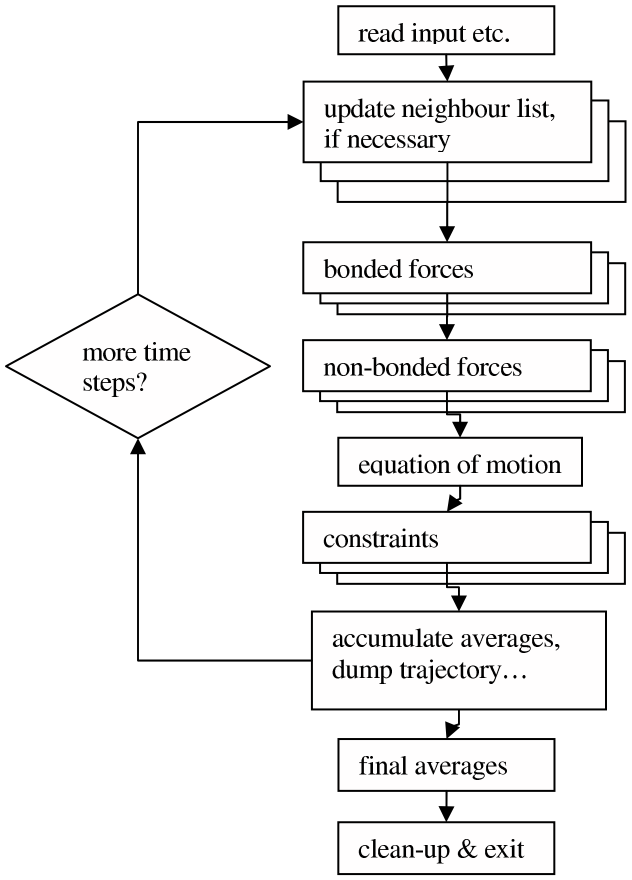


Figure 4.3. Schematic representation of the OpenMP molecular dynamics algorithm described in this article. The following parts of the MD code are parallelized: neighbour-list, part of the bonded forces (dihedral angles, bond angles), non-bonded forces, and constraints.

The sequential YASP molecular dynamics program[28] 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 neighbour list is updated. In every step, the bonded and non-bonded 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 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 (neighbour list construction, non-bonded forces, constraints and some of bonded forces) are executed in parallel by all team threads (Fig. 4.3). 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.

4.3.2. Neighbour list

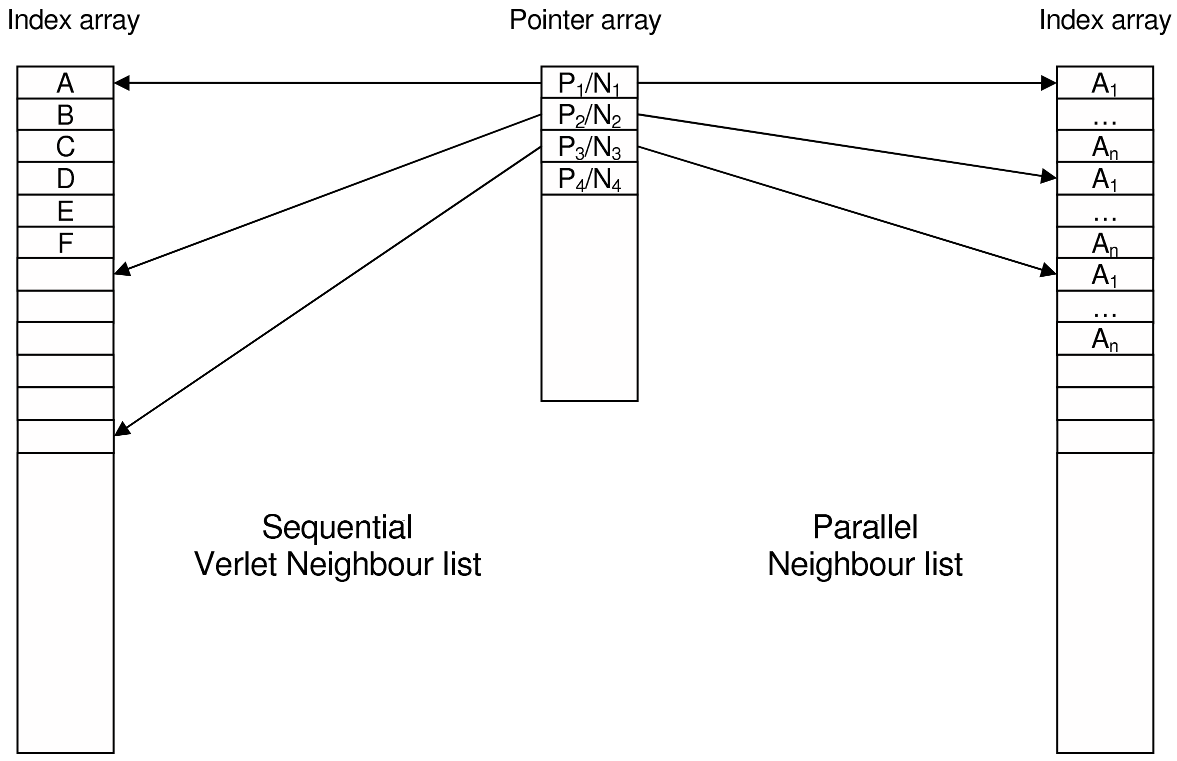


Figure 4.4. Modifications to the Verlet neighbour list structure. In the parallelized version, each atom has a fixed and equal number of elements for its neighbours to be stored.

Some changes are necessary in order to perform the neighbour list update in parallel. Due to possible density fluctuations one cannot predict in advance precisely how many non-bonded neighbours every atom will have. It becomes obvious that each atom must have a reserved number of elements in the neigbour list array where indices of neighbouring atoms are stored (Fig. 4.4). The number of elements must be large enough to be able to save all found neighbours of the atom.

For keeping neighbours for each atom in the sequential Verlet neighbour list two arrays are used (Fig. 4.4). The index array keeps indices of neighbouring atoms. The second is used to keep pointers P_i, which refer to that element in the index array, from which on the indices of the neighbours for the atom in question are stored. The number of neighbours kept for an atom i is easily calculated as $N_i^{neighbours} = P_{i+1} - P_i$. Since in the parallel version every atom has the same number of elements n reserved to keep indices of neighbours, the start element can be computed as $P_i = n*(i\text{-}1) + 1$ (the formula implies that array and atom indices start from 1). The pointer array is then used to keep actual number of neighbours found for each atom $N_i^{neighbours}$.
In the sequential version all atoms are examined sequentially. Therefore, the array must be long enough to keep all neighbours of all atoms. There is an imbalance in the number of neighbours. The first atom (with order number 1) has usually twice as many as atoms in the center. The last atom has no neighbours at all (Fig. 4.5). As, in the parallel implementation, every atom has the same number of elements reserved in the neighbour-list array, there are unused gaps and the array is approximately twice as long as in the sequential version. The length of the parallel neighbour list is approximately $4\pi r_{cut}^3 \rho N/3$, whereas the sequential list is roughly $4\pi r_{cut}^3 \rho N/6$. According to this change of the neighbour structure, the part of the code that calculates non-bonded forces was also accordingly slightly modified. The neighbours search algorithm in this case becomes worse (i.e. doubled length of the neighbour list array and unused gaps within it).

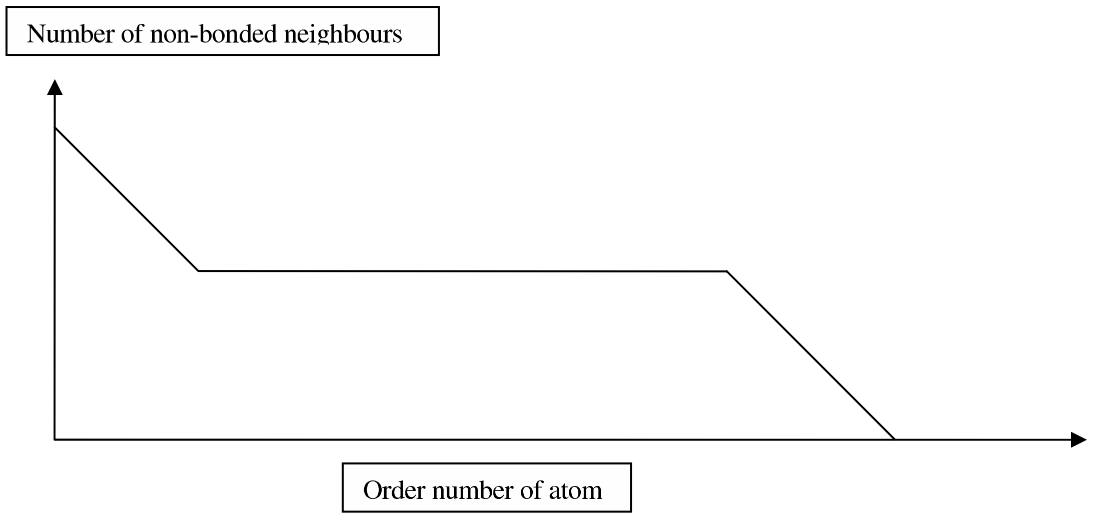


Figure 4.5. Schematic representation of contents of the neighbour-list array. The first atom has about twice as many neighbours as an average atom. The last atom does not have any neighbours, as it was examined last and, therefore, was already included in all possible atom pairs.

In principle, one could use the Brode-Ahlrichs scheme[29] for the neighbour-list construction, which does not waste memory. It is, however, difficult to integrate this scheme with the neighbour-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.

4.3.3. Atomic forces

All parts of the program that evaluate atomic forces, bonded as well as non-bonded, 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.

Listing 4.1. Schematic representation of the part of the sequential code that evaluates atomistic non-bonded forces.

```
      . . .
! Loop over all atoms
         DO i=1, N
           . . .
            nlfirst = nlptr(i)
            nllast = nlptr(i+1) - 1
            DO m = nlfirst, nllast
              j = nllist(m)
              fx = <<evaluate x-component of force (i,j)>>
              fy = <<evaluate y-component of force (i,j)>>
              fz = <<evaluate z-component of force (i,j)>>
              fix = fix + fx
              fiy = fiy + fy
              fiz = fiz + fz
              fxatom(j) = fxatom(j) - fx
              fyatom(j) = fyatom(j) - fy
              fzatom(j) = fzatom(j) - fz
            END DO
            . . .
            fxatom(i) = fxatom(i) + fix
            fyatom(i) = fyatom(i) + fiy
            fzatom(i) = fzatom(i) + fiz
            . . .
         CONTINUE
      . . .
```

Listing 4.1 schematically presents the sequential version of a loop, which evaluates one of three types of forces – bonded angles, bonded dihedral angles, or non-bonded. It shows the calculation of all three components of the force *(fix, fiy, fiz)* between non-bonded 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 neighbours *nllist* (see Fig. 4.4). The evaluation of forces of bond angles and dihedral 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.

Listing 4.2 represents schematically the parallelized version of this cycle. Similar to listing 4.1, the cycle is given only for the calculation of non-bonded forces. Line *01* calculates the maximum number of elements in neighbour list *nmaxnl* array that each atom can have, where *mnl* is the total length of the array. Line *02* of listing 4.2 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 $5N_{threads}$. 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.

When the local arrays are initialized, the team threads start iterations of the outer loop. The comment block, which 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 4.2.3).

Lines *02-03* of listing 4.1, which define start and end elements of the neighbour list array where neighbours of the current atom are stored (see Fig. 4.4), are now changed and turned into lines *12-13* of listing 4.2.

Line *30* closes the parallel cycle and also controls the behaviour 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.

Listing 4.2. Schematic representation of the parallel version of the code that evaluates atomic non-bonded forces.

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

! Initialize arrays of forces
      fxa_local = 0
      fya_local = 0
      fza_local = 0

!$omp  do private(<<variables list>>)
!$omp& reduction(+:<<variable list>>)
!$omp& schedule(dynamic, nomp_chunk)
! Loop over or all atoms
      DO i=1, N
        . . .
         nlfirst = (i-1)*nmaxnl + 1
         nllast = (i-1)*nmaxnl + nlptr(i)
         DO m = nlfirst, nllast
           j = nllist(m)
           fx = <<evaluate x-component of force (i,j)>>
           fy = <<evaluate y-component of force (i,j)>>
           fz = <<evaluate z-component of force (i,j)>>
           fix = fix + fx
           fiy = fiy + fy
           fiz = fiz + fz
           fxa_local(j) = fxa_local(j) - fx
           fya_local(j) = fya_local(j) - fy
           fza_local(j) = fza_local(j) - fz
         END DO
         . . .
         fxatom(i) = fxatom(i) + fix
         fyatom(i) = fyatom(i) + fiy
         fzatom(i) = fzatom(i) + fiz
         . . .
      CONTINUE
!$omp end do nowait

!$omp critical (forces_add_lock)
      fxatom = fxatom + fxa_local
      fyatom = fyatom + fya_local
      fzatom = fzatom + fza_local
!$omp end critical (forces_add_lock)

!$omp end parallel
   . . .
```

In this reduction, critical sectioning must be used in order to avoid data loss (see section 4.2.4). 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.

4.3.4. Constraints

The multi-colour SHAKE[7] is implemented in the YASP package[28]. 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 vectorisable 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 non-bonded neighbour list or calculation of non-bonded and bonded forces, were parallelized using dynamic load balancing option used, 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[30,31]. 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 multi-colour 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 multi-colour SHAKE.

4.4. Assessment of performance

For the water systems, simulations were carried out for 1000 time steps (2 fs per step), the neighbour 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 neighbour 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.

Table 4.1. Benchmark systems used to test parallel MD code.

Name	System size		Mass density [kg/m^3]	Cutoff radius [nm]		Neighbour list length	
	Molecules	Atoms		Potential	Neighbour list	sequential	parallel
						Total (per atom)	Total (per atom)
SPC/E Water, 300 K	300	900	1017	0.75	0.8	118074 (131)	236149 (262)
SPC/E Water, 300 K	3000	9000	1004	0.75	0.8	1167345 (130)	2334690 (260)
SPC/E Water, 300 K	9000	27000	1001	0.75	0.8	3487887 (129)	6975775 (258)
SPC/E Water, 300 K	9000	27000	1000	1.0	1.1	9068451 (336)	18136902 (672)
Polyamide 6,6 (Nylon), 350 K	24	18360	1083	0.9	1.0	5092068 (277)	10184136 (555)

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 $(W/W_{\mathrm{ref}})(t_{\mathrm{ref}}/t)$, 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 (Fig. 4.6-Fig. 4.10) show throughput and efficiency as a function of the number of processors for all benchmark systems.

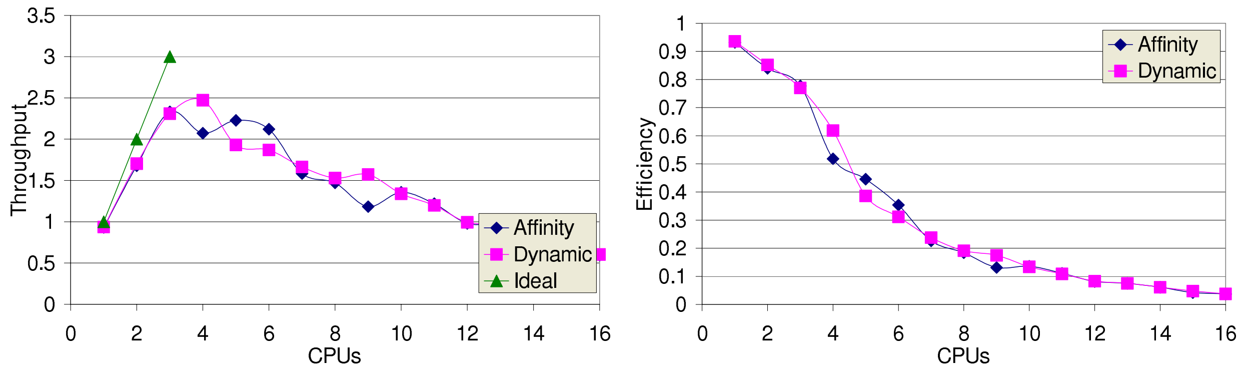


Figure 4.6. Throughput (left) and efficiency (right) for the system of 300 SPC/E water molecules.

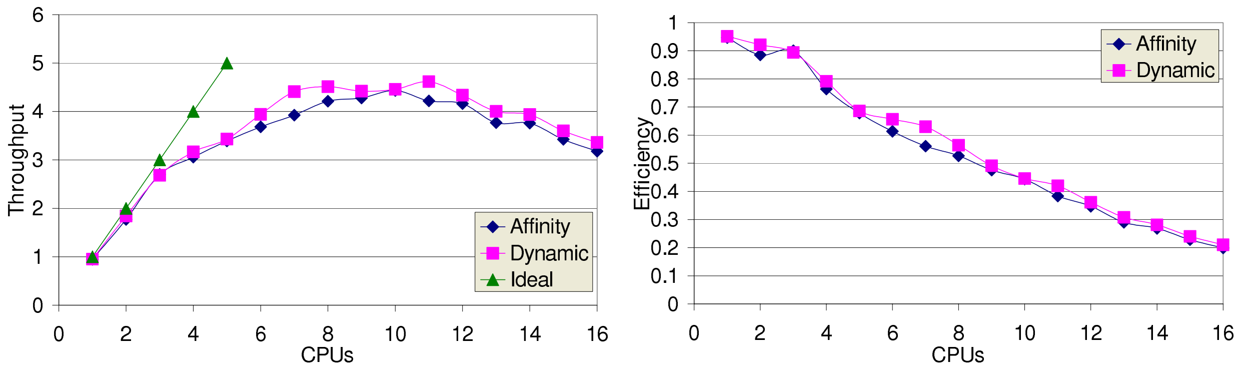


Figure 4.7. Throughput (left) and efficiency (right) for the system of 3000 SPC/E water molecules.

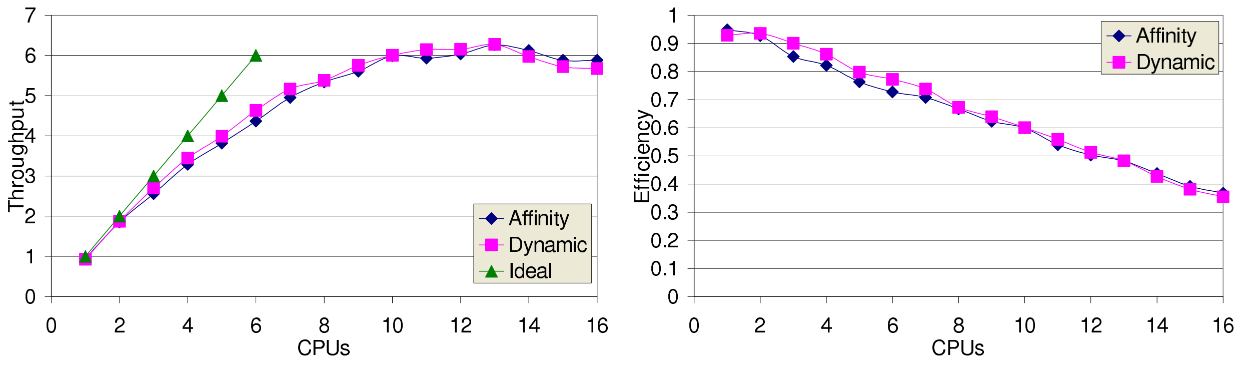


Figure 4.8. Throughput (left) and efficiency (right) for the system of 9000 SPC/E water molecules with cutoff radius 0.75nm.

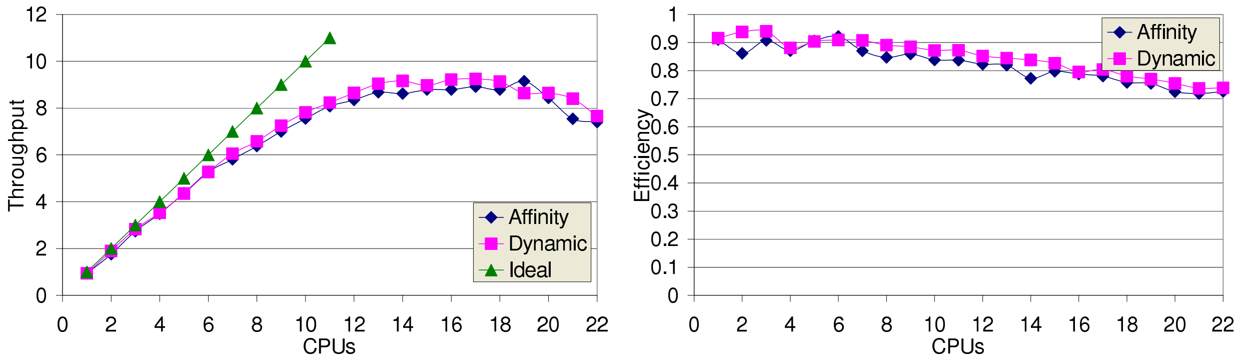
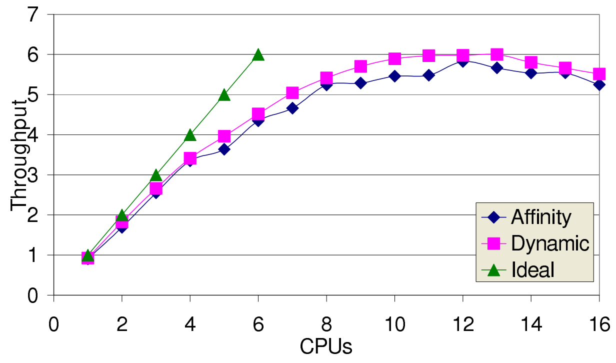


Figure 4.9. Throughput (left) and efficiency (right) for the system of 9000 SPC/E water molecules with cutoff radius 1.0.



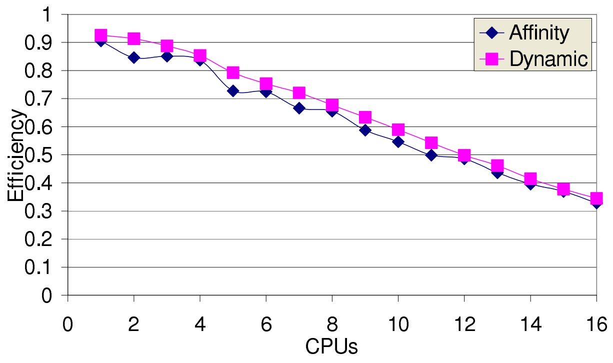


Figure 4.10. Throughput (left) and efficiency (right) for the system of 24 polyamide 6, 6 (nylon) molecules (totally 18360 atoms).

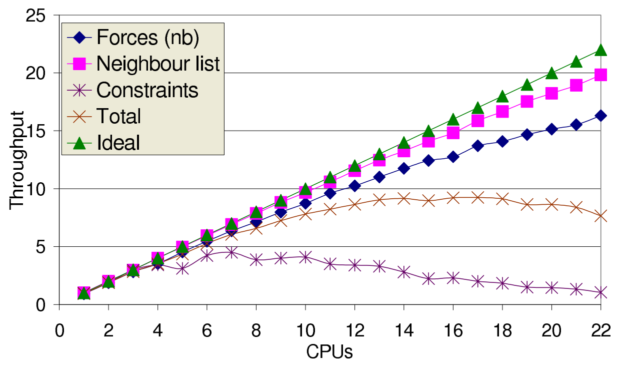
Each figure displaying throughput has an orientation line (triangles) that represents the ideal case of a 100% efficient parallel computer. In all figures (4.6-4.10) throughput and efficiency of affinity (rhombus) and dynamic (squares) work-load balancing strategies are shown.

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 behaviour develops due to 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 work-load 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 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 infinite number of processors according to Amdahl's law is 20.

It is seen from the graphs (Fig. 4.6 – Fig. 4.10) that the LNP depends on the size of the molecular system being studied (Fig. 4.6 – Fig. 4.8), atom (not mass) density, cutoff radius (Fig. 4.8 and Fig. 4.9), and complexity of the molecules in the system (Fig. 4.10).

The results shown in Fig. 4.6-4.10 were extracted with all timings done on a dedicated node, so there were not any other programs running simultaneously. Under real conditions, when node is shared with other running calculations, throughput and efficiency were found to be even more favourable.

Additionally, the separate throughputs for bonded forces, non-bonded forces, neighbour list, and constraints were investigated in order to understand the behaviour shown in Fig. 4.6-4.10. 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.



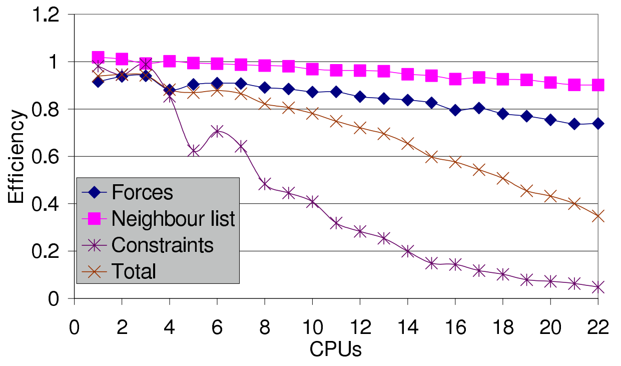
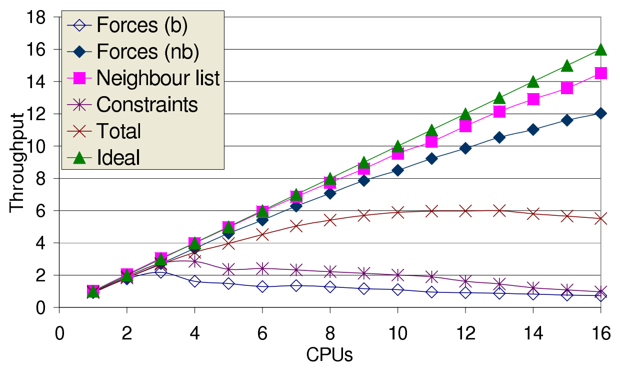


Figure 4.11. Separate throughput (left) and efficiency (right) of non-bonded-forces, neighbour list, and constraints for the system of 9000 SPC/E water molecules with cutoff radius 1.0. The throughput and efficiency of bonded forces are not shown, since SPC/E water molecule is rigid, and, therefore, does not have any bonded forces.



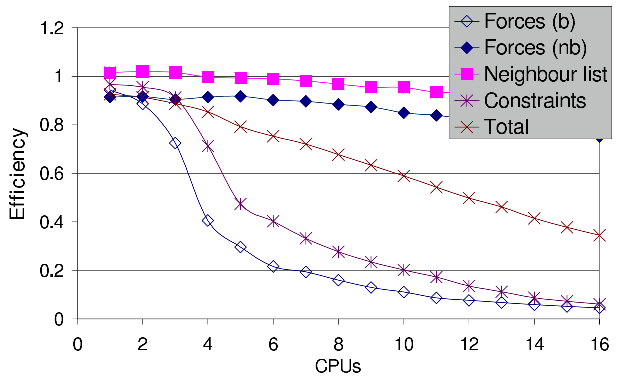


Figure 4.12. Separate throughput (left) and efficiency (right) of bonded forces, non-bonded-forces, neighbour list, and constraints for the system of 24 polyamide (6,6) (nylon) molecules.

The throughput (Fig. 4.11, 4.12) shows that the neighbour list search algorithm, although not fully optimized scales relatively well. The computing of non-bonded forces showed slightly worse but still useful scaling behaviour. However, the parallel computation of bonded forces (Fig 4.12.) shows an extremely poor increase in throughput, whose maximum is achieved with only 3 processors. In the case of constraints, the throughput showed also bad scaling with maximum of 7 processors for water (Fig 4.11) and only 4 processors for polyamide (Fig. 4.12). 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 neighbour list and non-bonded forces.

In principle, the situation could be improved applying the scheme of parallelization offered in ref[26], where bonded forces are calculated simultaneously with computing of non-bonded 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 non-bonded 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 $t^n_{est} = t^n_{tot} - t^n_{tot.f.} + t^{n-1}_{n.b.f.}$, where t^n_{tot} is the total elapsed time, $t^n_{tot.f.}$ is the total elapsed time consumed for calculations of all forces, and $t^{n-1}_{n.b.f.}$ is the elapsed time spent for calculations of non-bonded forces. The ideal case implies that all possible additional time overheads are neglected.

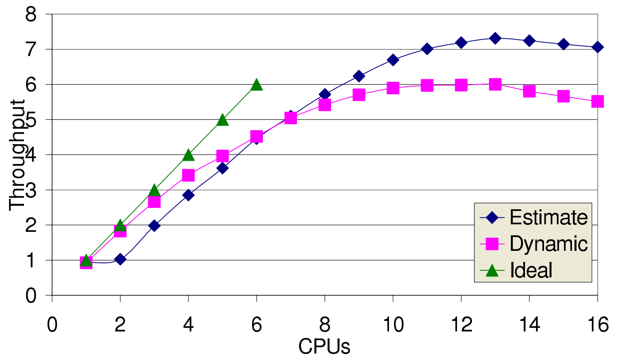


Figure 4.13. Estimation of throughput for the hypothetical case of simultaneous calculation of bonded and non-bonded forces for benchmark with polyamide (6,6) (nylon), compared to the throughput achieved without overlapping these tasks and using dynamic load balancing.

Fig. 4.13 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%.

In principle, overlapping bonded and non-bonded forces could be done by enclosing function calls of bonded and non bonded forces by the SECTIONS clause. This would be technically cumbersome and is not supported by all compilers, and one would have to use non-standard 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 SPC/E water molecules, the total number of processors was fixed at 22. The number of processors (threads) used to calculate constraints through special option of *$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 4.2 represents results achieved on this machine. It is seen 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.

Table 4.2. Benchmark systems used to test parallel MD code.

	Execution time [s]	Throughput	Efficiency
Water, 300K, 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, 300K, 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, 300K, 9000 molecules (27000 atoms)			
Sequential	1344	=1	=1
Parallel, 1 CPU	1468	0.92	0.92
Parallel, 2 CPU	769	1.75	0.87
Water, 300K, 9000 molecules (27000 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), 350K, 24 molecules (18360 atoms)			
Sequential	5254	=1	=1
Parallel, 1 CPU	7181	0.73	0.73
Parallel, 2 CPU	3726	1.41	0.71

4.5. Summary and conclusions

A partially parallelized MD code for shared memory computers is described, which achieves a substantial speed-up 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 non-bonded forces, building non-bonded neighbour list, application of constraints (Multi-colour SHAKE), 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 non-bonded neighbours array and the implementation of the algorithm that fills it. Additionally, the evaluation of the forces (non-bonded, 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 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 non-bonded forces showed slightly worse scaling behaviour than that one of the neighbour 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 decrease of the throughput beyond a certain number of processors.

The neighbour 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 speed-up obtained with the OpenMP implementation is very similar to that found in similar problems, see e.g. refs[19,23]. The speed-up 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.

4.6. Acknowledgements

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.

4.7. References

(1) Allen, M. P.; Tildesley, D. J. *Computer Simulation of Liquids*; Oxford University Press: Oxford, 1987.

(2) Noid, D. W.; Sumpter, B. G.; Wunderlich, B.; Pfeffer, G. A. *J. Comput. Chem.* 1990, *11*, 236.

(3) Teleman, O.; Svensson, B.; Jonsson, B. *Comput. Phys. Commun.* 1991, *62*, 307.

(4) Ryckaert, J. P.; Ciccotti, G.; Berendsen, H. J. C. *J. Comput. Phys.* 1977, *23*, 327.

(5) Ciccotti, G.; Ryckaert, J. P. *Comput. Phy. Rep.* 1986, *4*, 345.

(6) McCammon, J. A.; Harvey, S. C. *Dynamics of Proteins and Nucleic Acids*; Cambridge Univ. Press: Cambridge, 1987.

(7) Müller-Plathe, F.; Brown, D. *Comput. Phys. Commun.* 1991, *64*, 7.

(8) Verlet, L. *Phys. Rev.* 1967, *159*, 98.

(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. *J. Comput. Chem.* 1984, *5*, 272.

(11) Morales, J. J.; Rull, L. F.; Toxvaerd, S. *Comput. Phys. Commun.* 1989, *56*, 129.

(12) Rapaport, D. C. *Comput. Phys. Commun.* 1991, *62*, 198.

(13) Dagum, L.; Menon, R. *IEEE Comput. Sci. Eng.* 1998, *5*, 46.

(14) Forum, M. P. I. *International Journal of Supercomputer Applications and High Performance Computing* 1994, *8*, 165.

(15) Berendsen, H. J. C.; van der Spoel, D.; van Drunen, R. *Comput. Phys. Commun.* 1995, *91*, 43.

(16) Stadler, J.; Mikulla, R.; Trebin, H. R. *Int. J. Mod. Phys. C* 1997, *8*, 1131.

(17) Lyubartsev, A. P.; Laaksonen, A. *Comput. Phys. Commun.* 2000, *128*, 565.

(18) Forester, T. R.; Smith, W. *DL_POLY User Manual*; CCLRC, Daresbury Laboratory: Warrington, UK, 1995.

(19) Couturier, R.; Chipot, C. *Comput. Phys. Commun.* 2000, *124*, 49.

(20) Straatsma, T. P.; Philippopoulos, M.; McCammon, J. A. *Comput. Phys. Commun.* 2000, *128*, 377.

(21) Roy, S.; Jin, R. Y.; Chaudhary, V.; Hase, W. L. *Comput. Phys. Commun.* 2000, *128*, 210.

(22) Trobec, R.; Sterk, M.; Praprotnik, M.; Janezic, D. *Int. J. Quantum Chem.* 2001, *84*, 23.

(23) Goedecker, S. *Comput. Phys. Commun.* 2002, *148*, 124.

(24) Ozdogan, C.; Dereli, G.; Cagin, T. *Comput. Phys. Commun.* 2002, *148*, 188.

(25) Scaife, N.; Hayashi, R.; Horiguchi, S. *IEICE Trans. Inf. Syst.* 2003, *E86D*, 1569.

(26) Müller-Plathe, F. *Comput. Phys. Commun.* 1990, *61*, 285.

(27) Rapaport, D. C. *The Art of Molecular Dynamics Simlations*, 2nd ed.; Cambridge University Press: New York, 2004.

(28) Müller-Plathe, F. *Comput. Phys. Commun.* 1993, *78*, 77.

(29) Brode, S.; Ahlrichs, R. *Comput. Phys. Comm.* 1986, *42*, 51.

(30) 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.; PA: SIAM: Philadelphia, 1994.

(31) 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.

5. Ion binding to cucurbit[6]uril: structure and dynamics

5.1. Introduction

Much work is currently being done in the field of synthetic molecular receptors. Many have been synthesized and characterized. The inclusion of guest molecules into their active sites has the potential to lead to novel chemical transformations, to mimic enzymatic activity[1], to isolate reactive species[2], and, combined with controlled capturing, to allow drug delivery[3,4]. Along with the experimental research, molecular simulations of such receptors are being carried out in order to investigate the molecular mechanisms of host-guest complexing[5,6].

One class of synthetic molecular receptors are the cucurbiturils (cucurbit[*n*]uril, or CB[*n*]). Cucurbit[6]uril, which is investigated in this thesis, is a hexameric macrocyclic compound that has a cavity of ~0.55nm in diameter and high symmetry with two identical openings (Fig. 5.1). It was produced first in 1905[7] but fully characterized only in 1981 by Freeman *et al.*[8] (For recent reviews, see Lagona *et al.*[9] and Lee *et al.*[10]) The mechanism of complexation of cucurbituril with various guest compounds as well as other properties have been studied intensively[11-17]: complexation in aqueous and acidic solution[14]; co-crystals formed by cucurbituril[13]; assembly into nanotubes[15]; molecular and biomolecular recognition[17,18].

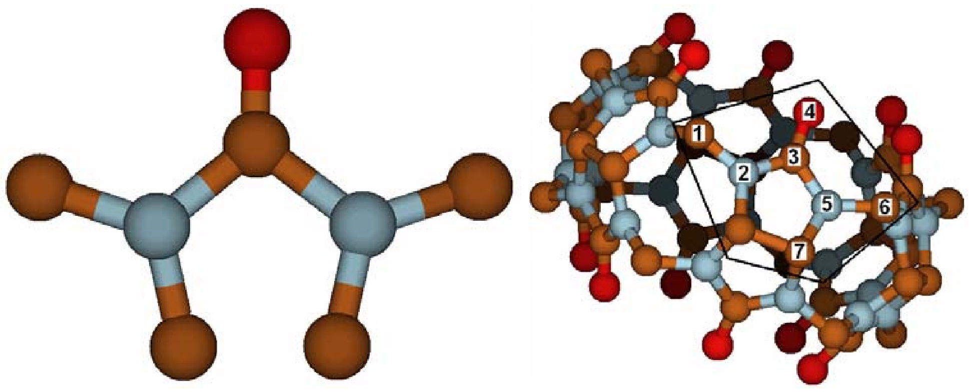


Figure 5.1. Tetramethylurea (left) and cucurbit[6]uril (right) molecules. Tetramethylurea was used as a building block of cucurbituril (outlined in the right figure).

Moreover, the interaction of cucurbituril with metal ions and their influence on properties and complexation behaviour under different conditions have also been considered[11-14,19,20]. In particular, the complexation of metal cations at the oxygens can result in ternary complexes composed of cucurbituril, a guest molecule, and a metal cation. It has been suggested that the cations function as "lids" sealing the portals and stabilizing the complex[12,21,22]. Furthermore, the encapsulation and release of the guest can be controlled by pH[22], which makes cucurbiturils applicable as molecular containers for drug delivery. Despite the great interest, relatively little is known about the microscopic dynamics and the quantitative effects of metal ion "lids" on the inclusion of the guest[14].

The investigation of different cyclic receptors by simulation methods and in particular molecular dynamics simulations (MD) is well established, e.g. cyclodextrin[23-25], calixarenes[26,27], crown ethers[27-29], and cryptands[30] (including complexes with metal cations[29,30]). Although cucurbituril complexes are being intensively studied experimentally, we are not aware of any such simulations for cucurbituril so far. In the current study, we prepare an MD model of cucurbit[6]uril. We report the results of simulations of CB[6] in water, and in aqueous solutions of potassium, sodium, and calcium chloride. Static properties such as structure, cation complexation and binding energies are discussed. The cation binding dynamics at the oxygens and the dynamics of capturing of water molecules in the CB[6] cavity are investigated.

5.2. Computational details

Since no cucurbituril model for MD simulations has been reported yet, its geometrical properties (bond lengths, angles, dihedrals) were derived from the structure optimized (energy minimization) by the PM3 semi-empirical method[31-35]. A density functional theory optimization (ADF software[36], VWN functional (LDA)[37], gradient corrections (GGA): Becke[38]-Perdew[39], double-ζ basis set, relativistic effects: Scalar Zora[40-44]), however, did not reveal big discrepancies: the maximum distance difference was 0.04Å, the maximum angle difference was 5º, and the maximum dihedral difference was 3º. Parameters of Lennard-Jones parameters and atomic partial charges have been taken from a force-field for tetramethylurea (TMU) in aqueous solution[45], as this molecule can be considered as a building block of cucurbituril (Fig. 5.1). The CH and CH_2 groups are connected to two nitrogens, as opposed to TMU. Therefore, their charges were set to double charges of TMU CH_3 to keep the total charge equal to zero. These parameters and the bond lengths are reported in tables 5.1 and 5.2 respectively. A united atom model has been used for the CH_n groups.

Table 5.1. Parameters of atoms: masses, Lennard-Jones parameters, partial charges.

Atom	**mass [au]**	**ε [kJ/mol]**	**σ [nm]**	**q [e]**
CH	13.007825	0.5356	0.37538	0.2202
N	14.0031	0.5314	0.325	-0.29605
C	12	0.4393	0.375	0.7954
O	15.9949	0.5314	0.2965	-0.6437
CH_2	14.01565	0.5356	0.37538	0.2202
K^+	39.09831	0.418454	0.3334	1.0
Na^+	22.98977	0.418454	0.2586	1.0
Ca^{2+}	40.078	0.418454	0.2872	2.0
Cl^-	35.4532	0.418454	0.4404	-1.0

Table 5.2. Bond constraints.

Bond	**length [nm]**
CH–CH	0.159141
CH–N	0.145851
N–C	0.137022
N–CH_2	0.145140
C–O	0.121137

Table 5.3. Bond angles.

bond angle	φ_0 **[º]**	**bond angle**	φ_0 **[º]**
CH-CH-N	102.423	N-C-N	105.777
CH-N-C	115.053	N-C-O	126.889
CH-N-CH_2	120.127	N-CH_2-N	111.597
C-N-CH_2	123.927		

Table 5.4. Dihedral angles.

dihedral angle	τ_0 **[º]**
CH_2-N-C-O (1-2-3-4, Fig. 5.1)	3.0
CH_2-N-C-O (6-5-3-4, Fig. 5.1)	-3.0
CH-N-C-O (7-5-3-4, Fig. 5.1)	173.0

The Lennard-Jones parameters are given for separate atoms. The united interaction parameters σ_{12} and ε_{12} between two atoms $U_{LJ} = 4\varepsilon_{12}\left[\left(\frac{\sigma_{12}}{r_{12}}\right)^{12} - \left(\frac{\sigma_{12}}{r_{12}}\right)^{6}\right]$ are calculated as follows: $\varepsilon_{12} = \sqrt{\varepsilon_1\varepsilon_2}$; $\sigma_{12} = \frac{\sigma_1 + \sigma_2}{2}$. Nonbonded interactions between atoms that are connected by fewer than four bonds were eliminated. Those are atoms participating in bonds, bond angles, and all possible dihedrals, not only those that are shown in Table 5.4.

Bond lengths have been kept rigid by the SHAKE algorithm[46-48] (Table 5.2). For both bond angles and dihedral angles, harmonic terms have been used. The CB[6] molecule is a stiff cage, and its conformation does not crucially depend on the precise values of the force constants. Therefore, we used one force constant for all bond angles (350 kJ/(mol·rad^2)) and one for all improper (harmonic) dihedral angles (170 kJ/(mol·rad^2)). The equilibrium values of bond angles φ_0 and dihedrals τ_0 are given in Tables 5.3 and 5.4.

CB[6] was simulated in pure water and also in aqueous salt solutions. All systems have approximately the same molarities – 0.0184 mol/L of CB[6] and 0.183 mol/L of salt. Each of the simulated systems contained 3000 water molecules and 1 CB[6]. 10 ion pairs (NaCl, KCl) or triplets ($CaCl_2$) were used to simulate salt solutions. CB[6] is poorly soluble in water under standard conditions. However, as only one molecule was simulated and, therefore, it was "forced" to dissolve. In presence of alkali salts cucurbituril becomes appreciably soluble[22].

The SPC/E model of water[49] and the models of potassium, sodium, and calcium chloride in aqueous solutions suggested by Koneshan *et al.* [50] (Table 5.2) have been used in our simulations. To account for electrostatic interactions, the reaction field method was used with the dielectric constant of pure water (72). That was done since all solutions were dilute and a small variation does not change the final results much.

All simulations were done with the program YASP[51-53] at constant temperature of 300K (coupling time 0.2ps) and constant isotropic pressure of 101.3 kPa (coupling time 3.0 ps) (NPT)[54]. The simulation time step was 2fs, with the sampling period 2ps. The neighbour list was updated every 15 time steps. The cutoff radius was 1.1nm for the neighbour list and 1.0nm for calculating the Lennard-Jones and the electrostatic potentials. All systems have been sampled for approximately 10-12ns after an equilibration period of about 1ns.

5.3. Results and discussion

5.3.1. Hydration of cucurbituril

The densities of the simulated systems were than that of pure water (Table 5.5), as one would have expected.

Table 5.5. Densities, self-diffusion coefficients, and rotational correlation times in the simulated systems (T=300K).

#	System	Density [kg/m^3]	Diffusion coefficient [$\cdot 10^{-5}$cm^2/s]				Rotational correlation times [ps]	
			Water	CB	Cation	Anion	τ_1 (C_1)	τ_2 (C_2)
1	Water, CB	1009.1	2.62	1.1	-	-	546	177
2	Water, CB, Na^+, Cl^-	1016.2	2.60	0.31	1.41	1.52	424	139
3	Water, CB, K^+, Cl^-	1016.8	2.72	0.48	1.49	1.69	699	232
4	Water, CB, Ca^{2+}, Cl^-	1025.0	2.42	0.25	0.59	1.48	435	142
5	Water, CB, K^0	1011.4	2.67	0.28	3.35 (K^0)		-	-

The analysis of radial distribution functions (RDF) of CB[6] in pure water (Fig. 5.2) showed a peak between oxygen of CB[6] and water oxygen [O(CB)-O(H_2O)] of about 1.5 at approximately 0.265nm. RDFs of other sites of CB[6] have the first peak of less than 1.0 at 0.483nm (nitrogen) and at 0.371nm (carbon). The higher first peak of the RDF for the crown CB[6] oxygen implies that only these sites are hydrophilic. Although the excluded volume of CH_n groups (n=0..2) is bigger than that of nitrogen (N), the water oxygen is notably further from the latter. This is explained by the fact that the nitrogens point inward the cavity of CB[6] and access to them is hindered by the surrounding CH_n

groups. Moreover, the methylene groups are pointed towards water and this also determines the peak at the closer distance.

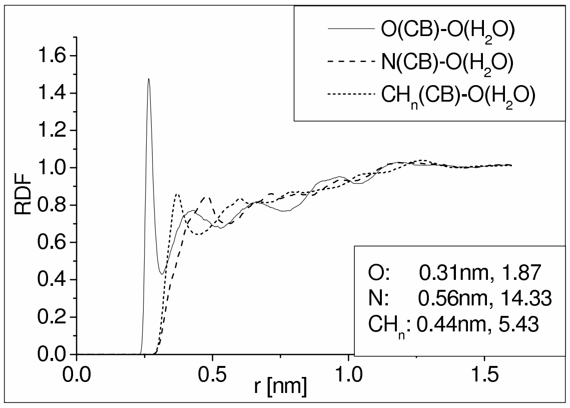


Figure 5.2. Radial distribution functions of CB[6] sites (O, N, CH_n) and the oxygen atoms of water molecules (O). The lower box in the figure indicates the positions of the first minima after the first peaks followed by the number of water molecules in the shell up to this distance for each site of CB[6].

The presence of salt ions in solution hardly affected the CB[6]-water RDFs. The first peak of the CB[6] oxygen-water RDF became a bit smaller; the first minimum after the peak increased slightly (Fig. 5.3). This can be explained by the fact that positive ions come close to the negatively charged CB[6] oxygens and displace some water molecules (Fig. 5.3). The other negative site of CB[6] is nitrogen. But for the same reasons as for water, cations do not bind to it.

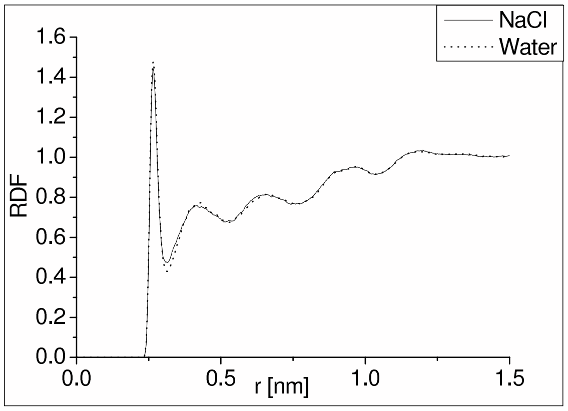


Figure 5.3. Radial distribution functions of CB[6] oxygen and water oxygen in NaCl solution and pure water.