

# Reduction on arrays: comparison of performances among different algorithms

Simone Meloni\*, Alessandro Federico, Mario Rosati  
CASPUR, Inter University Consortium for Supercomputing,  
Via dei Tizii 6/b, I-00185, Rome, Italy

August 31, 2003

## 1 Introduction.

In a reduction operation, we repeatedly apply a binary operator to a variable and some other value and store the result back in the variable [3]. The reduction operation may be ran in parallel. For this purpose, OpenMP provides the `REDUCTION` clause to the `DO` directive. According to the Fortran 2.0 OpenMP specification, the `REDUCTION` clause can be applied even to arrays. This new feature may be particularly useful in the context of Molecular Dynamics (MD) [4][2].

MD is a very important tool used to predict properties of materials and to understand the microscopic changes induced by different external conditions, such as temperature or pressure.

In classical MD the positions and velocities of the atoms at any time step are calculated from the inter-atomic forces. Usually, such forces are additive, that is the forces acting on the atoms are the sum of the many body forces due to the interactions of the atoms with their neighbors. Therefore, the total force of an atom is the reduction of the partial forces arising from the interactions with the neighbors.

In classical MD most of the CPU time is spent in the calculation of the forces acting on the atoms. Therefore an effective parallelization of this module is of paramount importance. Using OpenMP, a set of particles can be assigned to each thread, for example using a `PARALLEL DO` directive. This parallelization scheme is usually named **particle decomposition**. However, due to Newton's third law [1], several threads may try to write the force acting on the same atom, therefore it is necessary to "protect" such write instruction. This goal can be achieved using either the `REDUCTION` clause or the `ATOMIC` directive. In our experience, the "protection" of the write instruction applied to the force array strongly limits the scaling of molecular dynamics codes.

In this paper we compare a set of methods to implement the reduction on force array applied to our Classical MD code (CMPTool).

The paper is organized as follows: in the next section the force routine of the serial version of CMPTool will be presented; in section 2 straightforward parallelization techniques, based either on the `REDUCTION` clause or the `ATOMIC` directive will be presented; in section 4 a set of alternative schemes will be shown; in section 5 the results of all the methods will be compared and discussed and in section 6 some conclusions will be drawn.

## 2 CMPTool: the serial version of the force module.

CMPTool is a Molecular Dynamics code that implements a force field well suited for inorganic semiconductor named Stillinger-Webber [7]. In this force field, the potential energy is the sum of

two- and three-body terms:

$$\mathcal{E}[R_1 \cdots R_n] = \sum_{j>i} V_{2body}(|R_i - R_j|) + \sum_{k>j>i} V_{3body}(|R_i - R_j|, |R_i - R_k|, \theta_{kij}) \quad (1)$$

the sums run on all those pairs and triplets of atoms which satisfy the following condition:

$$|R_i - R_j| \leq R_{cutoff} \quad (2)$$

The calculation of the forces consists in a loop running on the atoms and two nested loops running on atom's neighboring:

```

do i = 1, n_atoms
  idx_atom1 = i
  do j = 1, n_neighbors
    idx_atom2 = neighbor_list(j,i)
C*** calculation of two-body terms ***C
    call twobody(idx_atom1, idx_atom2, tmp_a)
    do l = 1, 3
      forces(l, idx_atom1) = forces(l, idx_atom1) + tmp_a(l)

    do k = j+1, n_neighbors
      idx_atom3 = neighbor_list(k,i)
C*** calculation of three-body terms ***C
      call threobody(idx_atom1, idx_atom2, idx_atom3,
2         tmp_a, tmp_b)
      do l = 1, 3
        forces(l, idx_atom3) = forces(l, idx_atom3) + tmp_b(l)
        forces_tmp2(l) = forces_tmp2(l) + tmp_a(l)
        forces_tmp1(l) = forces_tmp1(l) - tmp_a(l) - tmp_b(l)
      end do
    end do
    do l = 1, 3
      forces(l, idx_atom2) = forces(l, idx_atom2) + forces_tmp2(l)
    end do
  end do
  do l = 1, 3
    forces(l, idx_atom1) = forces(l, idx_atom1) + forces_tmp1(l)
  end do
end do

```

where the `neighbor_list` array contains the indexes of all the atoms which satisfy the condition 2, the neighboring atoms. For the sake of simplicity, we omitted the zeroing of `forces_tmp1` and `forces_tmp2`.

### 3 Straightforward parallel versions.

A straightforward parallel version of the force module of the CMPTool can be obtained resorting to a `PARALLEL DO` directive and a `REDUCTION(+: forces)` clause around the external loop:

```

!$OMP PARALLEL DO
!$OMP& PRIVATE(idx_atom1, idx_atom2, idx_atom3)
!$OMP& REDUCTION(+: FORCES)
    do i = 1, n_atoms
        ...
    end do
!$OMP END PARALLEL DO

```

Hereafter, this version of CMPTool will be named `OMP REDUCTION`.

An alternative implementation is possible using the `ATOMIC` directive. The idea is to protect the updating of `forces` by means of the `ATOMIC` directive:

```

!$OMP PARALLEL DO
!$OMP& PRIVATE(idx_atom1, idx_atom2, idx_atom3)
    do i = 1, n_atoms
        ...
        do l = 1, 3
!$OMP ATOMIC
            forces(l, idx_atom3) = forces(l, idx_atom3) + tmp_b(l)
            forces_tmp2(l) = forces_tmp2(l) + tmp_a(l)
            forces_tmp1(l) = forces_tmp1(l) - tmp_a(l) - tmp_b(l)
        end do
    end do
    do l = 1, 3
!$OMP ATOMIC
        forces(l, idx_atom2) = forces(l, idx_atom2) + forces_tmp2(l)
    end do
    end do
    do l = 1, 3
!$OMP ATOMIC
        forces(l, idx_atom1) = forces(l, idx_atom1) + forces_tmp1(l)
    end do
    end do
!$OMP END PARALLEL DO

```

The semantic of the two approaches is different. The version which uses the reduction clause works as if the partial forces, calculated by the threads, were stored in private variables which are summed at the end of the parallel do. In the atomic directive based version, a thread acquires exclusive access to the single memory address corresponding to the force of an atom for the duration of the update.

In both cases the scaling with the number of threads is poor. For example, the scaling factor of the `OMP REDUCTION` version for a run on a system of 125000 atoms on a IBM 9076 375 MHz Power3 SMP High Node compiled with guidef77 4.0, is about 7.89 on 16 threads. The performance of the `ATOMIC` directive based version is even worse, being below 1 on 16 threads. Similar results were obtained for different problem sizes and number of threads. The bad performances of the version based on `ATOMIC` directive were ascribed to the Newton's third law [1]. It often occurs that a thread tries to write the force acting on an atom in the set of another thread. Sooner or later, the other thread will try to write the force either of the same atom or of an atom which belong to the same cache line. Because of the memory hierarchy, one of the two write instructions is hindered by the other. This effect take place only in the `ATOMIC` case because in the `OMP REDUCTION`, `DGEMV` and `OMP DO` versions (see below) the threads use either private arrays or different columns of the same array.

## 4 Alternative parallel versions of the force module.

An alternative approach consists in storing partial forces in an array which has an extra dimension corresponding to the threads. During the loop over the atoms each thread updates the entries in the column assigned to it of the force array, named `forces_priv` in our example. The updating step is represented in the figure 1/a.

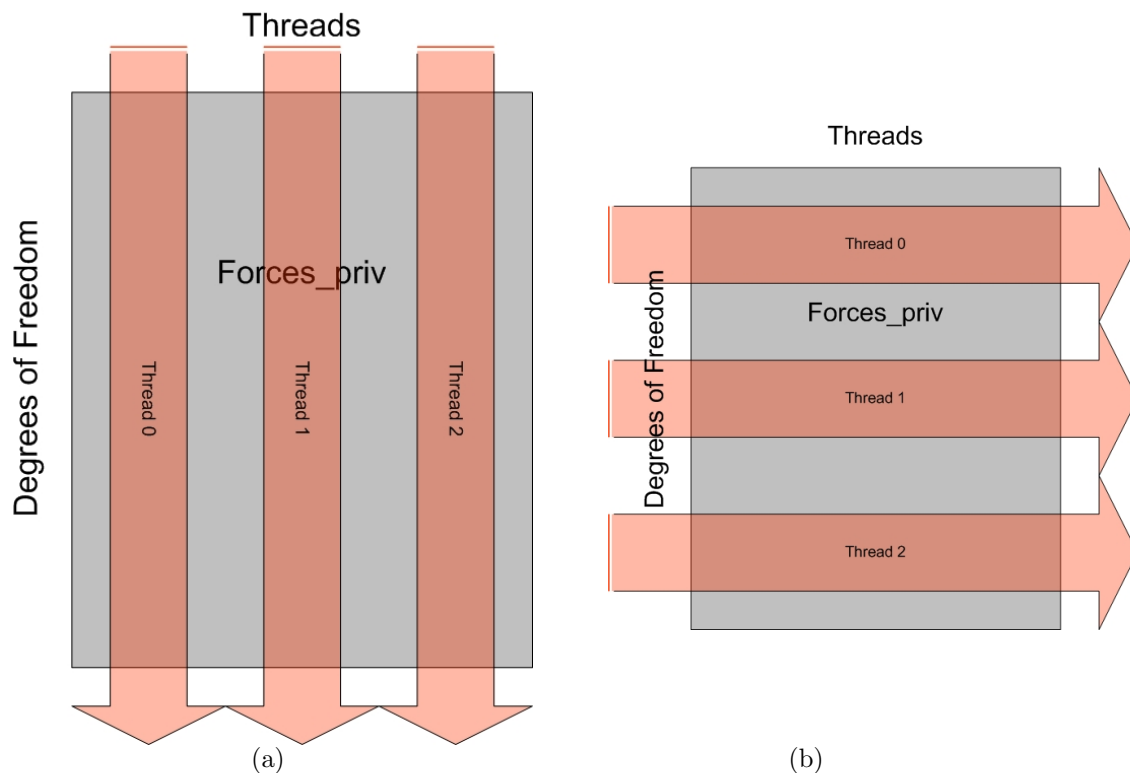


Figure 1: Updating (a) and parallel global sum (b) steps of the alternative versions of the force module.

The code which implements the first step of this scheme is the following:

```

!$OMP PARALLEL
!$OMP& PRIVATE( this_thread)
    this_thread = OMP_GET_THREAD_NUM() + 1

!$OMP DO
!$OMP& PRIVATE(idx_atom1, idx_atom2, idx_atom3)
    do i = 1, n_atoms
        ...
        do l = 1, 3
            forces_priv(l, idx_atom3, this_thread) =
2            forces_priv(l, idx_atom3, this_thread) + tmp_b(l)
            forces_tmp2(l) = forces_tmp2(l) + tmp_a(l)
            forces_tmp1(l) = forces_tmp1(l) - tmp_a(l) - tmp_b(l)
        end do
    end do
end do

```

```

        end do
    end do
    do l = 1, 3
        forces_priv(l, idx_atom2, this_thread) =
2         forces_priv(l, idx_atom2, this_thread) + forces_tmp2(l)
    end do
end do
do l = 1, 3
    forces_priv(l, idx_atom1, this_thread) =
2     forces_priv(l, idx_atom1, this_thread) + forces_tmp1(l)
end do
end do
!$OMP END PARALLEL DO

```

At the end of the updating step, the partial results calculated by the threads are summed up providing the total value of the forces. In this second step, the global sum step, the threads access the `forces_priv` array by rows and sum the partial values calculated in the previous step. The code which implements the global sum is pretty simple, it reads:

```

do k = 1, n_threads
    do i = 1, n_atoms
        do l = 1, 3
            forces(l, i) = forces(l, i) + forces_priv(l, i, k)
        end do
    end do
end do
end do

```

An equivalent result can be obtained by a matrix-vector multiplication, using an auxiliary vector with as many elements as the number of threads, all set to 1:

$$forces = forces\_priv \cdot aux \quad (3)$$

where *aux* is the auxiliary vector. This matrix-vector multiplication can be performed using `GEMV`, a BLAS routine [6]. As an example, if `forces_priv` and `forces` are double precision arrays, the global sum step using `GEMV` would read:

```

do i = 1, n_threads
    aux(i) = 1.d0
end do
call DGEMV('N', 3*n_atoms, n_threads, 1.d0, forces_priv, 3*n_atoms,
2     aux, 1, 0.d0, forces, 1 )

```

The advantage of this version of the global sum step is that very often the vendors provide an optimized version of `GEMV` in their high performance mathematical libraries. However, the number of floating point operations performed by `GEMV` are twice the number of operations strictly required. In fact, it performs `n_threads` multiplications and `n_threads - 1` additions per row against the `n_threads - 1` additions performed by the simple hand coded version of the global sum step. On those machines which feature fused multiply and add operations this extra floating point operations have no effect on the performance but on other machines the effect must be carefully evaluated.

A parallel version of the global sum step consists in assigning a set of atoms per thread, as shown in figure 1/b. Each thread sums all the entries in the rows corresponding to the coordinates of the atoms in its set. This version can be implemented in several ways. Here we present two versions, the first makes use of `GEMV` whereas the second one uses the OpenMP `DO` directive.

In the `GEMV` based version the master thread calculates the number of atoms to be assigned to each thread then every thread executes `GEMV` on its set of atoms:

```

        CHUNK = n_atoms / n_threads
!$OMP PARALLEL
        this_thread = OMP_GET_THREAD_NUM()

        initial_atom = CHUNK * this_thread + 1

        call DGEMV('N', 3*CHUNK, n_threads, 1.d0, forces_priv(1, initial_atom, 1),
2           3*n_atoms, aux, 1, 0.d0, forces(1, initial_atom), 1)

!$OMP END PARALLEL

```

This naive version can be easily extended to the case that `n_atoms` is not a multiple of `n_threads`. In the rest of the paper we shall refer to this version with the name `DGEMV`. Some vendors provide a multi-threaded parallel version of `GEMV` routine. In this case the parallel version of the global sum step can be obtained simply by linking the parallel version of the mathematical library.

In the second parallel version, the OpenMP `DO` directive is put just before the second loop, the one running over the atoms:

```

        do k = 1, n_threads
$OMP DO
        do i = 1, n_atoms
            do l = 1, 3
                forces(1, i) = forces(1, i) + forces_priv(1, i, k)
            end do
        end do
$OMP END DO
        end do

```

This version of `CMPTool` will be named `OMP DO`. The loops are ordered such that the access to `forces_priv` is stride one. This requires the insertion of the `DO` directive inside a loop, which could cause an overhead. It would be possible to invert the loops running on threads and atoms. However, in this case `forces_priv` would be accessed with stride `n_threads`. Alternatively, it would be possible to invert the loops and to change the shape of `forces_priv` at the same time (`forces_priv(3, n_threads, n_atoms)`). However, this latter approach affects the performance of the updating step. Tests shows that the `OMP DO` version provides better performances.

## 5 Results and discussion.

In order to compare the performances of the different schemes, we ran molecular dynamics simulations on three systems with different number of atoms: 125000, 250000 and 500000 atoms. The tests were run on two different SMP computers: IBM 9076-N81 375 MHz Power3 High Node (16 CPUs) and IBM 7040-681 1300 MHz Power4 (32 CPUs). The codes were compiled using `guidef77 4.0[5]` and `xlf 8.1.1.0`, the native IBM compiler. For the `DGEMV` based code we used the serial version of the `ESSL` mathematical library. The simulations were run binding the OpenMP threads to CPUs. However, the effect of binding is negligible, being about 2% of the scaling factor for all the problem sizes on both IBM Power3 and Power4 computers.

Table 1: Scaling factor of the various schemes for several problem sizes on an IBM Power4.  
125000 atoms

# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
2	1.82	1.87	1.84	1.86	1.78	1.85
4	3.34	3.64	3.55	3.48	3.42	3.56
8	5.66	6.78	6.59	5.42	6.51	6.63
12	7.33	9.61	9.27	6.29	9.28	9.33
16	7.95	11.23	11.16	6.24	11.46	11.54
24	9.25	14.76	13.98	5.32	14.50	14.41
32	9.29	12.84	12.54	4.08	14.46	14.31

250000 atoms

# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
2	1.80	1.89	1.78	1.80	1.81	1.87
4	3.30	3.66	3.45	3.38	3.51	3.62
8	5.17	6.99	5.72	5.37	6.74	6.67
12	7.28	9.74	9.00	6.26	9.45	9.46
16	8.26	12.28	11.12	6.16	11.88	11.74
24	9.41	16.87	14.54	5.43	15.23	14.98
32	9.83	14.27	12.09	4.19	16.45	15.80

500000 atoms

# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
4	3.37	3.63	3.58	3.41	3.47	3.63
8	5.63	6.90	6.71	5.43	6.56	6.75
12	7.37	9.65	9.26	6.19	9.15	9.30
16	8.23	11.52	11.25	6.14	11.31	11.40
24	9.40	15.31	14.20	5.35	14.20	14.25
32	9.76	15.62	14.42	4.11	15.44	14.95

Table 2: Scaling factor of the various schemes for several problem sizes on an IBM Power3.  
125000 atoms

125000 atoms						
# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
2	1.73	1.83	1.77	1.74	1.77	1.77
4	3.18	3.48	3.35	3.09	3.38	3.35
8	5.44	6.33	6.16	4.54	6.16	6.16
12	6.98	8.68	8.43	4.85	8.40	8.43
16	7.89	10.10	9.90	4.44	9.48	9.93

250000 atoms						
# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
2	1.89	1.86	1.67	1.81	1.88	1.91
4	3.45	3.54	3.19	3.25	3.58	3.63
8	5.86	6.53	5.92	4.89	6.61	6.69
12	7.40	9.00	8.21	5.32	9.09	9.01
16	8.16	10.51	8.83	4.93	10.53	10.59

500000 atoms						
# of Threads	guidef77			xlf		
	OMP REDUCTION	DGEMV	OMP DO	OMP REDUCTION	DGEMV	OMP DO
2	1.84	1.89	1.69	1.84	1.83	1.91
4	3.35	3.60	3.21	3.32	3.48	3.63
8	5.75	6.67	5.97	5.16	6.47	6.73
12	7.27	9.22	8.32	5.75	8.92	9.26
16	8.04	10.90	9.47	5.58	10.35	10.67

In table 1 and 2 the scaling factors for all the problem sizes and compilers are reported. Figure 2 provides a visual representation of the scaling factor of the MD simulations for a system with 500000 atoms performed on the IBM Power4 computer.

Both the DGEMV and the OMP DO schemes outperform the OMP REDUCTION. The scaling of the DGEMV and OMP DO versions compiled with the two compilers, guidef77 and xlf, are practically equivalent. In the case of the OMP REDUCTION version, the code compiled with guidef77 systematically outperforms the one compiled with xlf on both the IBM machines. For example, in the case of a simulation with 500000 atoms and 24 threads on Power4, the code compiled with guidef77 is roughly 2 times faster than the one compiled with xlf. The “stability” of performances in different contexts is a noteworthy feature of the DGEMV and OMP DO versions. The difference between OMP REDUCTION and either DGEMV or OMP DO increases with the number of threads. This fact confirms the assertion reported in the introduction that the reduction on the force array may become the limiting factor for a particle decomposition based parallelization of a MD code.

We ran some preliminary tests on a HP ES45 1250MHz (4 CPUs) computer using guidef77 and the serial version of the CXML mathematic library. The HP ES45 shows a behavior similar to that of the IBM machines. Even in this case, the performance of the DGEMV and OMP DO versions is better than that of the OMP REDUCTION one and the differences increase with the number of threads.

In our opinion, the DGEMV version is better than OMP DO because it maintains good performance, always close to the best, for all the computers, compilers and problem sizes. For example, although the performance of the OMP DO version compiled with xlf on IBM Power3 for a system of 250000 particles for 8 threads is 13.1% worse than the best run (OMP DO compiled with xlf), the DGEMV version is at most 2% slower in the worst case.

As a concluding remark, it has to be noted that in the MD case it was possible to find a simple

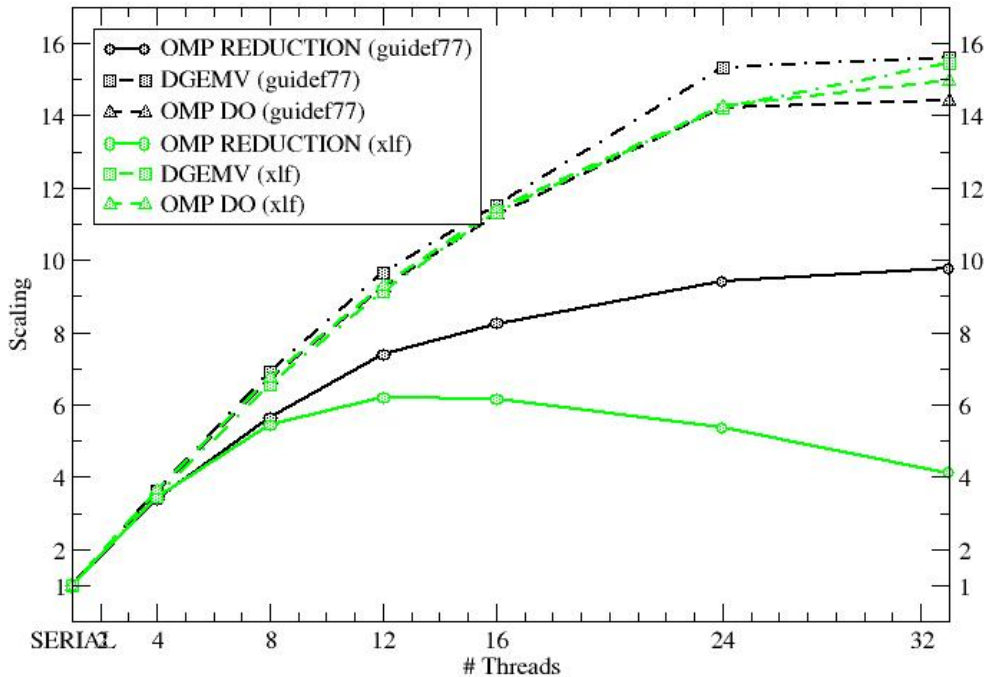


Figure 2: Scaling factor measured as a function of the problem size and reduction method.

scheme to obtain a good scaling, this may not be the case in a different context. This fact motivates us to plan the analysis of the difference of the hand coded and reduction based versions of CMPTool in the near future. This work could be done in the case of codes compiled with `guidef77`, which produces an instrumented source file which can be compared with the source files of the `DGEMV` or `OMP DO` versions.

## 6 Conclusions.

In this paper we presented several schemes for a particle decomposition based parallelization of a MD code. We have shown that the reduction on the force array may become the limiting factor running MD simulations on a large class of computers and problem sizes. The hand coded versions almost always outperform the “plain OpenMP” versions, those based on the `REDUCTION` clause and `ATOMIC` directive.

Our results show that the current compilers and runtime systems are inadequate to provide good performances, at least in the case of MD. We think that this topic should be carefully investigated also in other fields of scientific and technical computing.

## Acknowledgments.

We acknowledge Dr. Stefano Salvini for suggesting us the use of the GEMV routine. We also thank Dr. Giorgio Richelli for providing us a library to bind OpenMP threads to CPUs on IBM computers. A special thank goes to Prof. Massimo Bernaschi and Prof. Federico Massaioli for the useful discussions.

## References

- [1] **Newton third law.** if the atom  $\alpha$  exerts a force  $\mathbf{F}_\alpha^\beta$  on the atom  $\beta$  then the atom  $\beta$  exerts the same force but in the opposite direction on the atom  $\alpha$  ( $\mathbf{F}_\alpha^\beta = -\mathbf{F}_\beta^\alpha$ ).
- [2] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1987.
- [3] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publisher, 2001.
- [4] D. Frenkel and B. Smit. *Understanding Molecular Simulations, from algorithm to applications*. Academic Press, 1996.
- [5] <http://developer.intel.com/software/products/guide>.
- [6] <http://www.netlib.org/blas>.
- [7] F. A. Stillinger and T. A. Webber. *Physical Review B*, 31:5262, 1985.