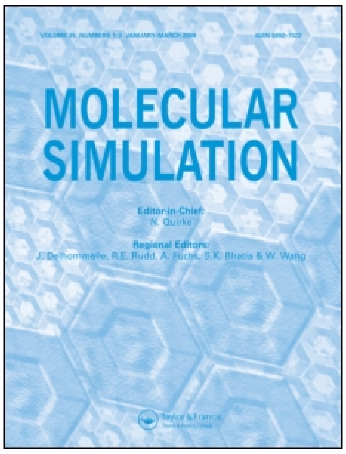

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

Molecular Dynamics on the Connection Machine using Fortran

B. Boltjes[ab]; S. W. De Leeuw[ab]
[a] Department for Physical Chemistry, University of Amsterdam, Amsterdam, WS, The Netherlands [b] Material Science Division, Argonne National Laboratory, Argonne, Ill, USA

To cite this Article Boltjes, B. and De Leeuw, S. W.(1991) 'Molecular Dynamics on the Connection Machine using Fortran', Molecular Simulation, 7: 1, 1 — 42
To link to this Article: DOI: 10.1080/08927029108022446
URL: http://dx.doi.org/10.1080/08927029108022446

Molecular Simulation, 1991, Vol. 7, pp. 1–42
Reprints available directly from the publisher
Photocopying permitted by license only

MOLECULAR DYNAMICS ON THE CONNECTION MACHINE USING FORTRAN

B. BOLTJES[†‡] AND S.W. DE LEEUW[†]

[†]*Department for Physical Chemistry, University of Amsterdam, Nieuwe Achtergracht 127, NL-1018 WS Amsterdam, The Netherlands*
[‡]*Material Science Division, Argonne National Laboratory, Argonne, Ill. 60439, USA*

(*Received September 1990, accepted October 1990*)

A new algorithm for molecular dynamics on the Connection Machine is presented and compared to an algorithm known from literature. The algorithms have been in implemented in CM Fortran 8x. Their performances, measured in CPU time, are compared and discussed for the simple Lennard-Jones system and for a liquid charged particle system. The implementation and performance of the Ewald sum, used for the charged particle system, is presented and discussed. The new algorithm is suitable for use with a cut off radius. Other advantages of the algorithm are discussed.

KEY WORDS: Molecular dynamics, connection machine, parallel algorithm.

1. INTRODUCTION

The development of massively parallel computers in recent years prompts the question on the significance of these computers in the field of Molecular Dynamics (MD). Of special interest is the Connection Machine (CM) with its large number of processors. The large number of processors and the capability of versatile interprocessor communication offers new solutions to old MD problems. The total number of particles in a MD simulation (N_t) has been restricted by the fact that the number of forces to be computed grows with N_t^2, or with N_t if one can use a cutoff radius in the force computation. Because in an MD simulation most of the time is consumed by the force computation, the time needed to do the simulation (CPU time) also scales with N_t^2 (or with N_t) on machines with one or a few microprocessors. Sub-square scaling of the CPU time in conventional MD simulations can be achieved using a cut-off radius for the forces and a link-cell algorithm [6]. Long range interactions, like the Coulomb interaction, can be treated separately and cause the CPU time to scale with $N_t^{1.5}$ [7] when optimized correctly. With some exceptions [1,2] the number of particles in a simulation is in the order of 10^3. Much more particles are needed if one wants to simulate phenomena like turbulence in the Rayleigh-Bénard cell [3] or long wavelength behavior in glasses for example. Studies of different algorithms on the CM have shown that the CPU time can scale linearly with the number of particles [4,5] without the use of a cutoff radius, as long as the number of particles is less than the number of processors.

The release of a good enough CM Fortran 8x compiler (version 0.6) enabled us to implement the MD algorithm of Greenwell *et al.* [5] which they wrote in *C**, CM verison of C (run with a SUN front end). The CPU times they give for a Lennard-

Jones program running on a CRAY-2 are used in order to estimate the performance of our algorithms. All computations have been carried out on the 16K CM-2 at Argonne with a VAX 8350 front end, unless stated otherwise.

This study has been carried out to examine the performance of the CM-2 under Fortran, with a full grown MD simulation of a Lennard-Jones system and a charged particle system (KCl), using constant temperature dynamics [8] and the Ewald sum as given by de Leeuw *et al.* [9]. The programs have been written in Fortran 8x using a few CM extensions. A new algorithm which is not only faster but also more suitable for MD on large systems than that of Greenwell *et al.* [5] will be presented and its performance for both particle systems will be discussed.

In section 2 the two algorithms and their key operations are described using the simple Lennard-Jones system as an example and compare their performances. The performances of both algorithms for the KCl system are compared in section 3. The advantages of the new algorithm are discussed in section 4. In section 5 we speculate that a 32K CM-2, with future releases of CM Fortran 8x, could have a better performance than a CRAY-2 in MD.

2. THE LENNARD-JONES SYSTEM

2.1 The linear Algorithm

Following Greenwell *et al.* [5] we started implementing their algorithm in Fortran for a Lennard-Jones system. This algorithm uses a linear geometry for laying out the particle positions, their time derivatives etc. on the CM. The linear geometry is such that the local memory if a processor i holds array elements A $(1,i)$, A $(2,i)$, A $(3,i)$ and the data of particles i-1 and $i + 1$ are in its neighboring processors. With a geometry like this a particle (x,y,z) position, velocity etc. is in the local memory of a processor. The layout of these arrays, defined with a shape A $(3,n_t)$, is A (:SERIAL, :NEWS). The first dimension along which the array elements are in the local memory of a processor is called serial. Communication between (physical) processor, along the parallel (second) dimension, takes place using the NEWS network. The shape of the arrays together with the way of interprocessor communication defines the geometry of the data layout on the CM. N_t is a power of 2 throughout this article, which is necessary in order to use NEWS communication (See [5] for the effect of using the slower ROUTER communication).

To compute the forces on the particles due to their interactions, the particle positions in array R $(3,N_t)$ are copied to a dummy array D $(3,N_t)$ whose elements are then shifted one element along the second dimension to the left (using the CSHIFT intrinsic function, see appendix B). This shift is circular: array elements in processor 1 go to processor N_t (see Figure 1). After this, processor i contains the (x,y,z) position of particle $i + 1$ in respectivley D $(1,i)$, D $(2,i)$, D $(3,i)$. Now the distances of the $(i,i + 1)$ particle pairs are computed and from that the force on each other. Notice that the forces for the pairs are computed in parallel. One can go on like this and shift array D once more and compute the forces from pairs $(i,i + 2)$ and add them to the previous ones, etc. Of course; having computed the force on particle i caused by particle j, this force can be shifted j-i fields back and subracted from the forces on particle j. So this algorithm has a loop for the computation of the forces which is performed $N_t/2$ times. This the reason why CPU time scales linear with N_t, as long as N_t is smaller or equal then the number of physical processors (N_p) available.

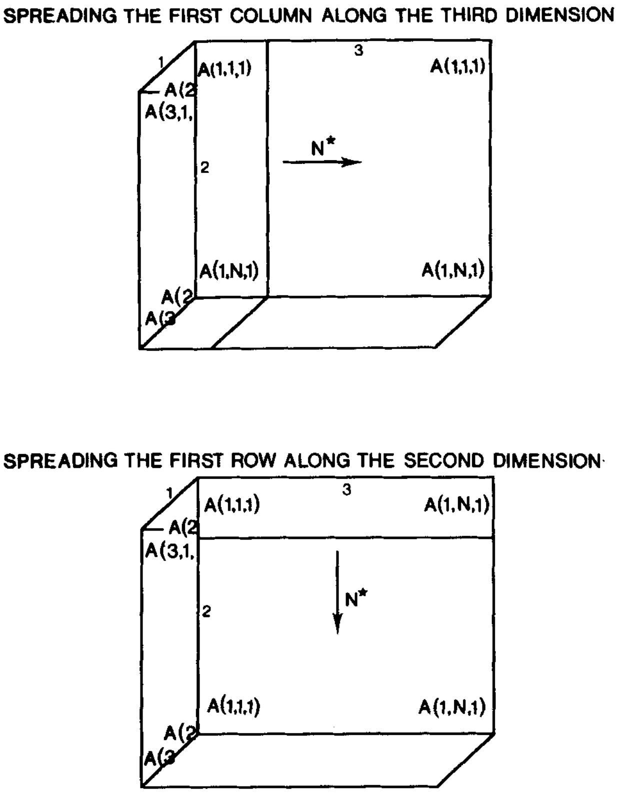

Figure 1 Illustration of the circular CSHIFT of an array used in the force computation in the linear algorithm. The 'wheels' hold the particle (x,y,z) position serially along the 'spokes'. The upper 'wheel' is kept fixed while the lower one is turned. Each processor holds array elements which are above each other.

If N_t is bigger than N_p each physical processor has to simulate $n = N_t/N_p$ virtual processors (called the VP ratio). If we have $N_t = 2 * N_p$ the CPU time will then be about twice as long. We will return to this effect in sections 2.3 and 4.

2.2 The Square Algorithm

Computing the forces by shifting the particle positions from processor to processor is not a very efficient way when N_t is in the order of the square root of N_p. The use of a N*N square geometry, instead of a linear one, enables one to compute the force of N^2 particle pairs in parallel. For this we define the shape as A $(3,N,N)$ with a layout A (:SERIAL, :NEWS, :NEWS). The number of particles in a parallel column in the arrays (N) could be up to 512 on the 16K machine at Argonne. This gives a VP ratio of $512 * 512/16\text{K} = 16$, every virtual processor has then only $8\text{K}/16 = 512$ bytes of memory. This sets the limit of N because of the number of necessary arrays residing on the CM. Using this geometry one can have $N_t = 512 * 512 = 256\text{K}$ particles at maximum when each parallel column contains $N = 512$ particles. (This maximum is the same for the linear algorithm; the linear algorithm then also has a VP ratio 16 on a 16K machine.)

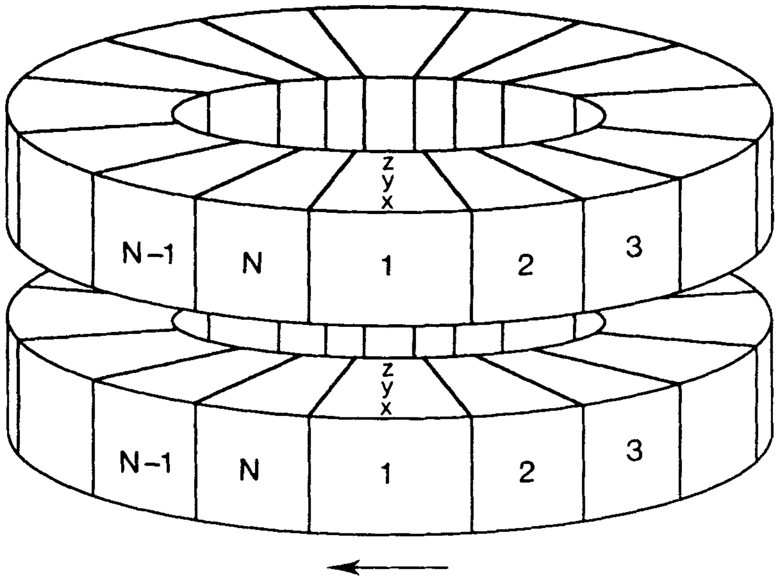

Figure 2 Illustration of SPREAD as used in the square algorithm. Along the sides are the dimension numbering. Dimension 1 is serial, 2 and 3 are parallel. In the upper block we spread the first parallel column to all other columns. In the lower one the first parallel column is copied to the rows.

In order to compute the forces on the N particles residing in the first parallel column of the array containing the particle positions R $(3,N,N)$ caused by the particles in this column, the first column is copied to the N parallel columns of a dummy array $D1$ $(3,N,N)$ (See Figure 2). Then the (parallel) rows of a second dummy array $D2$ $(3,N,N)$ are filled with this first column (For the filling of these two arrays the intrinsic function SPREAD is used, see appendix B. This filling is therefore called spreading.). Subtracting the two dummy arrays gives the distances of all the first row particle pairs, from which the forces of these pairs are then computed. Summing the elements of a row of the array containing the forces of the pairs, gives the force on the particle residing on that row in the first column of R (For the summing along a dimension one uses the SUM intrinsic function, see appendix B). All rows of the force array are summed in parallel.

This way computing forces is much faster than the linear algorithm. This makes it also useful when N_t is much larger than N. The number of forces which can be computed in parallel this way is restricted by the number of processors. For the parallel computation of the forces on N particles caused by N (other) particles, N^2 (virtual) processors are needed. For a number of particles larger than N, as many columns of R as necessary are filled and a double loop is made over the columns in order to compute all the forces (potentials, radial distribution functions etc.).

In the example above D1 and D2 were filled with the same column of R. We now take a different column of R and $D2$ and compute the distances and forces of this subset of particle pairs. Summing along the rows of the array containing the particle pair forces, gives again forces on particles residing in the column of R which was spread to the columns of $D1$. In addition we now also sum along the columns of the force array. This gives forces on the particles residing in the column of R whose positions where spread to the rows of $D2$ (times -1). These forces are then added to forces which have been computed earlier.

Having particles in M columns and all particle interactions taken into account, the force loop must be performed:

$$L(M) = 0.5 * M^* (M - 1) + M$$

times. The CPU time will scale with the square of the number of columns of R containing particles. The time required to perform the summing along the rows and columns with length N is proportional to $\log_2(N)$ so the total CPU time scales as:

$$\mathrm{CPU}(N_t) \sim L(M) * (\log_2(N) + T_{sp}(N))$$

$T_{sp}(N)$ is the time needed for the spreading. The spreading makes an important contribution to the CPU time. In the double loop for the force computation $D1$ is kept and $D2$ filled repeatedly in the inner loop. From this we expected and found that with the increase of the number of columns of R containing paticles, the CPU time per MD time step divided by $L(M)$ becomes smaller compared to the CPU time having only particles in one column. This is because the time spent in the spread of $D1$ in the outer loop becomes less important. We made no separate quantitative estimate for the CPU time spent in the spreading because the 0.6 Fortran version doesn't spread in the most efficient way. New versions should improve this.

2.3 Comparison of the Two Algorithms

The CPU time per MD step as a function of N_t for the system with Lennard-Jones interaction is shown in Figure 3. The sizes of the MD systems where obtained by filling columns completely with either 256 or 512 particles. For instance; the 16K system was made from 64 columns each containing 256 particles. The 32K system was made out of $64 * 512$ particles. For the linear algorithm this means that the VP ratio goes from 1 to 2 (16K machine) when we go from 16K to 32K. In the square algorithm it goes from 8 to 16.

The increase by a factor 2.1 in CPU time in the linear algorithm comes from two contributions; the force loop is twice as long and for the arithmetic operations the VP looping is increased. The increase by a factor 2.8 in CPU time for the square algorithm arises from the increased VP looping and the time needed to do the spreads. Twice as many particle positions had to be spread although $L(M)$ stayed the same.

To have a better view of the timing at small number of particles a plot of same data is given on a logarithmic scale in Figure 4. Up to $N_t = 8 * 512 = 4\mathrm{K}$ the difference in CPU time for the two algorithms is significant. The two curves converge, but at 32K the square algorithm is still 1.5 times faster than the linear algorithm.

To see where the CPU time is consumed in the 32K system, all arithmetic operations were taken out of the force loop with all interprocessor communication operations (shifting, spreading and summing) left intact. For the linear algorithm the CPU time per MD time step was then 326 sec., for the square algorithm 267 sec. Implying that the data in the square algorithm is more efficiently laid out for the communication operations. Subtracting these times from the total CPU times gives the times spend in the arithmetic operations. The linear algorithm spends 201 sec., the square algorithm 96 sec. More than a factor two difference. This can be explained from the fact that in the square algorithm the arithmetic operations become more

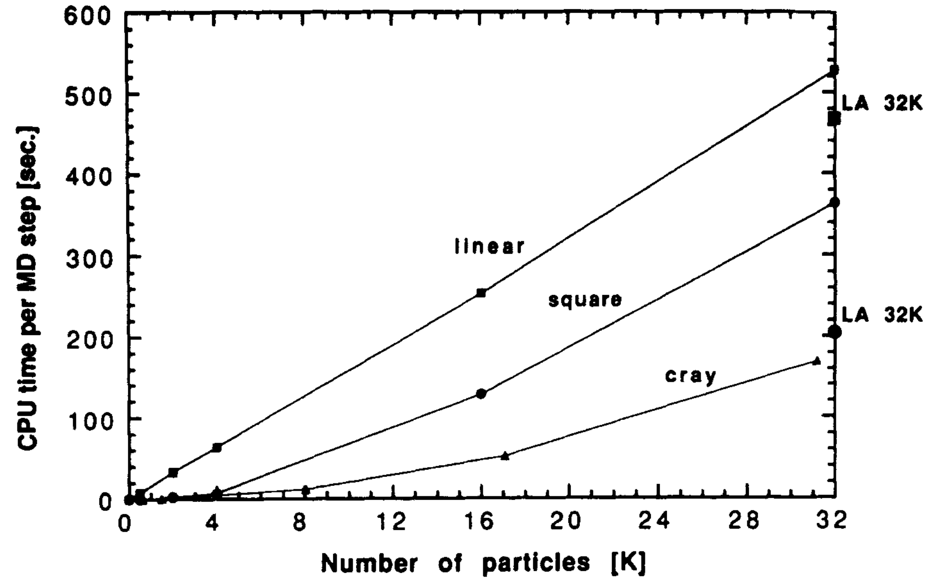

Figure 3 CPU time per MD step as a function of the total number of particles with Lennard-Jones interaction. The LA 32K points are the CPU times we got on the machine at Los Alamos using 32K processors and a faster front end than at Argonne, the square algorithm (round) then approaches CRAY-2 performance (triangles).

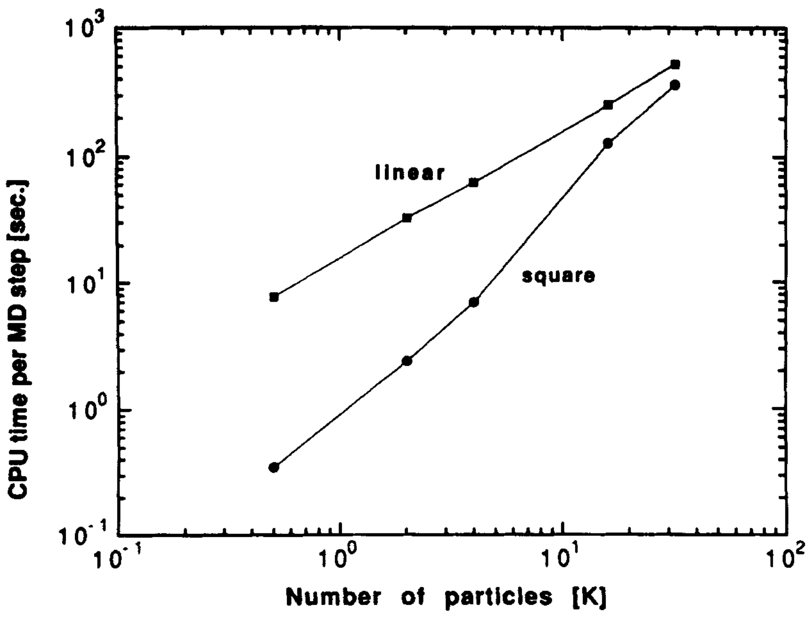

Figure 4 Same data as in Figure 3 for the linear and square algorithm on a logarithmic scale.

pipelined due to the high VP ratio. The CM can perform 8 times as much of the same computations in the square algorithm ($VP = 16$) as in the linear one ($VP = 2$).

In Figure 3 is also the CPU time which can be achieved on a CRAY-2 (taken from [5]). The CRAY-2 is still superior to the square algorithm run on th 16K machine at Argonne. However we also ran our Lennard-Jones programs on a 32K CM-2 at Los Alamos (The points marked a s LA 32K, the front end at Los Alamos is faster than the one at Argonne). The square algorithm then nearly matches the CRAY-2. This is not only at the 32K particle system but also at smaller ones, the parabola becomes less steep.

3 SIMULATION OF KCl

Liquid KCl has been chosen to simulate a more complicated physical system, using the potential of Tosi and Fumi [10]. The Ewald sum was computed as given by de Leeuw *et al.* [9], with a spherical macroscopic shape. The necessary arrays for the k-space part of the Ewald sum could stay on the front end, thus requiring no memory space on the CM. The k-space part was computed by looping over the k-vectors, the sums rewritten in the local form. Although in principle the k-space part can be computed in parallel, in practice it was not possible because of the insufficient number of processors. No use has been made of the symmetry of the k-space. The number of k-vectors (N_k) was 11^3 and the convergence factor 5.67, so only one term of the real space was required. With the use of the symmetry of the k-space and recursion relations, the contribution of the k space part to the CPU time could become a factor 8 smaller.

Computed this way, the contribution of the k-space part to the CPU time in the linear algorithm scales as $2 * N_k * \log_2 (N_t)$, because two summations for every k-vector are required (see the program listings in appendix B). For the 512 particle system this contribution to the CPU time was 10.3 sec. per MD time step, which is quite long.

For the square algorithm the contribution to the CPU time per MD time step is dependent on N, though independent of N_t (less or equal than $N * N$) and scales as $4 * N_k * \log_2 (N)$. The summing (appendix B) is performed over all columns whether there are particles in these columns of R or not (19.7 sec for a $512 * 512$ geometry). The k-space part was computed in the same way in both algorithms (except for the geometries). The forces in real space were computed as explained in sections 2.1 and 2.2

Figure 5 shows that for a 32K system the CPU time per MD time step for the square algorithm is a factor 2 smaller than that for the linear one. This factor is slightly larger than that for the Lennard-Jones system and is due to pipelining of the arithmetic operations, as explained above, which are more abundant in this more complex potential. Figure 6 shows that most of the CPU time for the square algorithm at $N_t = 512$ comes from the computation of the k-space part. After 4K the computation of the real space part becomes the major contribution.

4 DISCUSSION: APPLICATION OF THE SQUARE ALGORITHM

At the start of the simulations of the Lennard-Jones and KCl systems, the particles are defined on a cubic grid. In the square algorithm the MD cell is divided into M

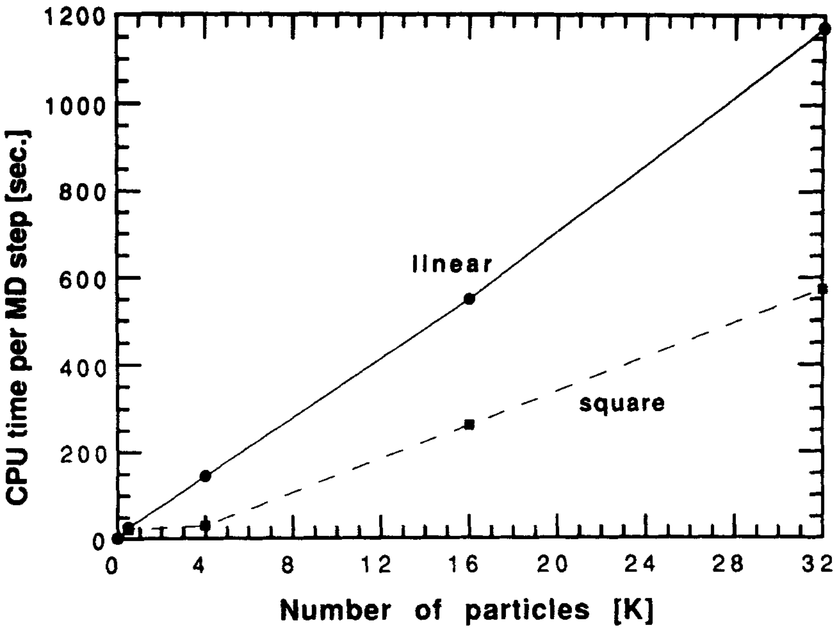

Figure 5 CPU time per MD step as a function of the total number of particles in the KCl system.

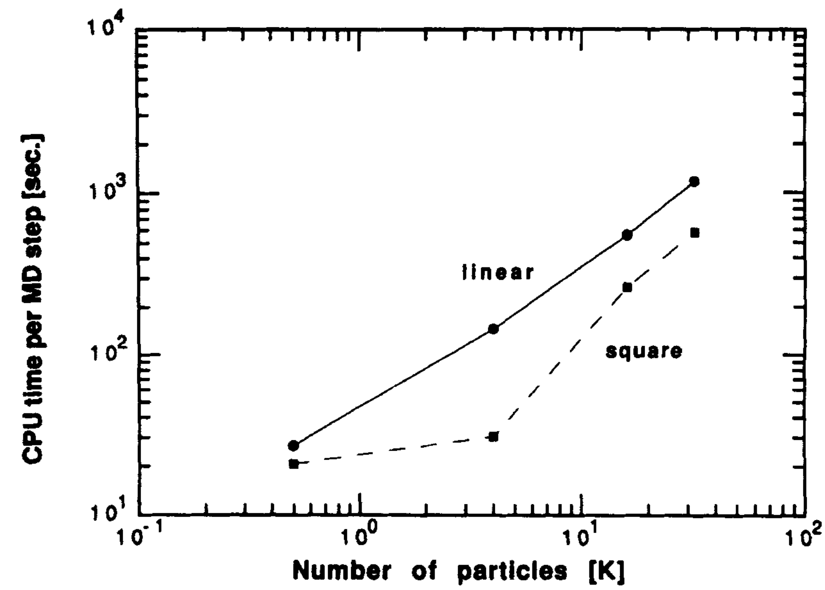

Figure 6 Same data as in figure 5 on a logarithmic scale.

subcells. Each column of the position array R then holds particles residing in subcell. If the particles would stay in their initial subcell only forces caused by particles in neighboring subcells would have to be taken into account in the case of the Lennard-Jones interaction and other interactions which fall off fast enough. Each subcell has 26 neighboring subcells when $M \geqslant 27$. So, simulating a solid instead of a gas or liquid, the CPU time would scale as;

$$L'(M) = 0.5*26*M + M = 14*M \; (M \geqslant 27)$$

Of course the data in the position array in the linear algorithm could also be viewed this way (different sections hold different subcells). But because of the data layout with the use of the shift operation on decrease in CPU time is gained by neglecting interactions between particles in non neighboring subcells.

In the square algorithm only a slight modification in the neighbor listing for the subcells has to be made. Before starting the simulation one decides which subcell (column) has interation with what other subcells (columns). This is only the case for solids, for liquids and gases the problem is more complex. When particles wander out of their subcell, the new subcell (column) to which they belong has to be found before they come close to subcells (columns) with which they should have interaction. This problem deserves to be studied further (see section 1). Because of fluctuations in the number of particles in a subcell, the number of particles in a subcell could become larger than the number of available places in a column. This is however not possible. The most simple solution is to decrease the size of the subcell in order to put less particles in a column than the number of available places in a column. The number of free places should be then so large as to make it unlikely that the number of particles put in a column would exceed the number of available places. The number of free places in each column is problem dependent however. Although a solution is necessary in order to do simulations with 10^4 or more particles, it does not change the algorithm here proposed.

Another advantage of the square algorithm is its capability of handling system with a variety of sizes more efficiently. In order to make use of the fast NEWS communication network one has to use geometries in which the length of dimensions are a power of two. Using the linear algorithm in MD with a prescribed number of particles (for instance proteins), one sometimes has to put dummy particles in the position array. Many of these can be avoided with the square algorithm. Supposing $N_t = 2*10^4$, the linear algorithm will need arrays with length of 32K with 12768 dummy particles. Using the square algorithm one needs 40 columns of length 512, having only 480 dummy particles.

With the square algorithm one can tune the VP ratio. This effect is shown in Figure 7. For Figure 7 three column lengths in the square gemoetry are used ($N = N_t = 128$, 256, 512). The $128*128$ and the $256*256$ are both run using 8K and 16K processors. The $512*512$ could be run only using 16K processors. Thus we were able to make 5 VP ratios. The $128*128$ geometry has a VP ratio 2 on a 8K machine. If the front end were fast enough and the start up time for force computation were small one would expect the CPU time of a 16K machine to be smaller than that of a 8K machine by a factor of two. In practice this factor is much smaller. The same effect occurs when the size is increased to $256*256$. The CM utilizations, a measure of how much of the CPU time is spent on the CM, for the VP ratios 1, 2, 4, 8, 16 were respectively 27, 43, 67, 87 and 97 percent.

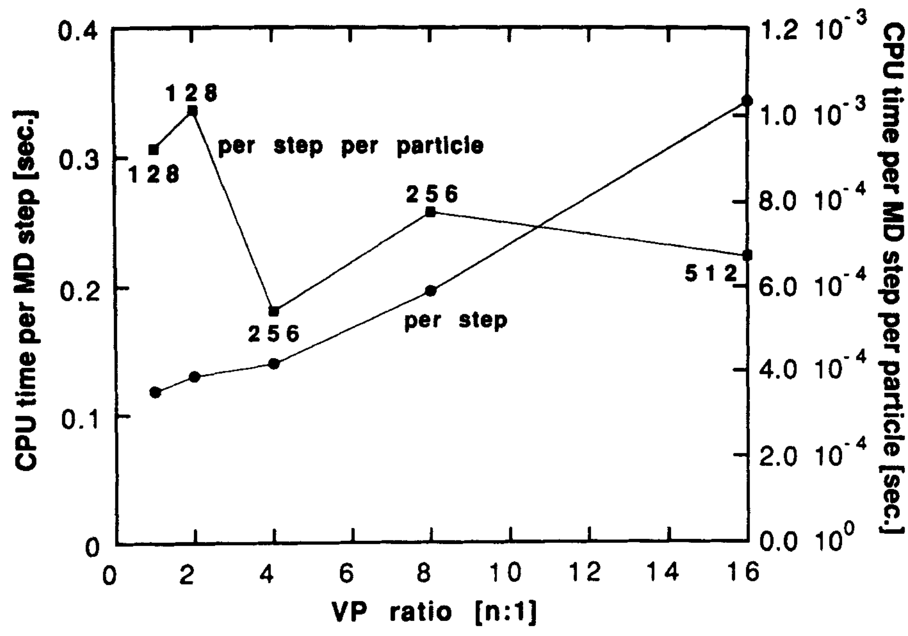

Figure 7 CPU timer per MD step and CPU time per MD step per particle versus the VP ratio using the square algorithm (Lennard-Jones interaction).

Dividing the CPU time per MD time step by the number of particles, gives a measure for the efficiency of the data layout with the machine used. A VP ratio 4 appears in Figure 7 as the most efficient for the machine at Argonne. However a VP ratio 4 is not much more efficient than a VP ratio 16. When one simulates a 512 particle system using a 256 * 256 geometry, L (M) becomes larger so that the advantage is lost. With different front ends and different amounts of physical processors one can change the lengths of the dimensions and the number of columns containing particles easily in the square algorithm in order to decrease the CPU time.

5 CONCLUSIONS

In this paper we have tested two algorithms for MD simualtions on a massively parallel machine. The square algorithm makes better use of the parallel architecture than the linear algorithm. It uses the more parallel instructions SPREAD and SUM instead of CSHIFT for the force computation.

The squre algorithm, implemented in CM Fortran version 0.6 on a 32K machine with a fast front end, nearly matches a CRAY-2. It might be expected that with future releases of Fortran, a CM-2 with 32K processors can out perform a CRAY-2 in the field of molecular dynamics.

We have pointed out that the square algorithm has a data layout which can be used in conjunction with link-cell algorithms. It can avoid many of the dummy particles previously needed to create the right geometry and it enables one to be more flexible in the number of particles and in the VP ratio.

For more complex MD simulations on the CM, more memory space per physical processor is needed, rather than more processors.

APPENDIX A

In the listing of the square algorithm for KCl in appendix B is an algorithm for computing the radial distribution functions (RDF) and the coordination numbers of K and Cl. In convetional MD one makes a double loop over the particles to compute their distances which are assigned a bin in a histogram which later becomes the RDF. On massively parallel and vector machines it is better to compute many particle pair distances first and then loop over the bins of the histogram.

The distances of particles residing in a pair of columns, of array R containing the positions, are computed first as explained in section 2.2. The array containing the particle pair distances is then divided by the bin width (the resulting integer array is called IDR). The number of one's, two's etc. in IDR are then counted, and these numbers are put in bin one, two etc. of the histogram. If enough processors are available, the counting can be done completely in parallel using the non circular SHIFT (The required number of processors is; N^2 times the number of bins. Here this would be $512^2 * 128$.). The counting is done partially serial, making a loop over the bin numbers. With the WHERE statement (a parallel version of IF) a 1 is assigned to fields in a dummy array where elements of IDR are the same as the bin counter. Summing all the elements of the dummy array then gives the total number of elements of IDR which are equal to the bin counter. In case of partial distribution functions one should also insure that a one is assigned for the correct particle paris.

When there are particles in more then one column of R one has to make a double loop over the columns, just like in the force computation. In Figure 8 an example of such RDF's is given for a molten KCl system ($N_t = 512$). The statistics is not good because it is only sampled over one configuration and serves merely as an example. However all the features to be expected from such RDF's of molten KCl are present [11].

APPENDIX B

In this appendix we list parts of the linear and square programs for KCl and explain some intrinsic functions (CSHIFT, SPREAD and SUM) used. Comments written in italic represent standard MD operations and are not listed in full. The shape and layout of the arrays in the examples given below are the same as defined in sections 2.1 and 2.2. To be consistent with the program listings we maintain the first (serial) dimension in the examples although it can be neglected here.

The essential intrinsic function in the linear algorithm is the CSHIFT. The CM-fortran expression for circular shifting all three array elements of the serial (first) dimension of an array A along the second (parallel) dimension by one element (processor) is:

$$B = \text{CSHIFT}\,(A, 2,1)$$

The elements B (1:3,i) now contain A (1:3,i + 1). The expression "1:3" means the

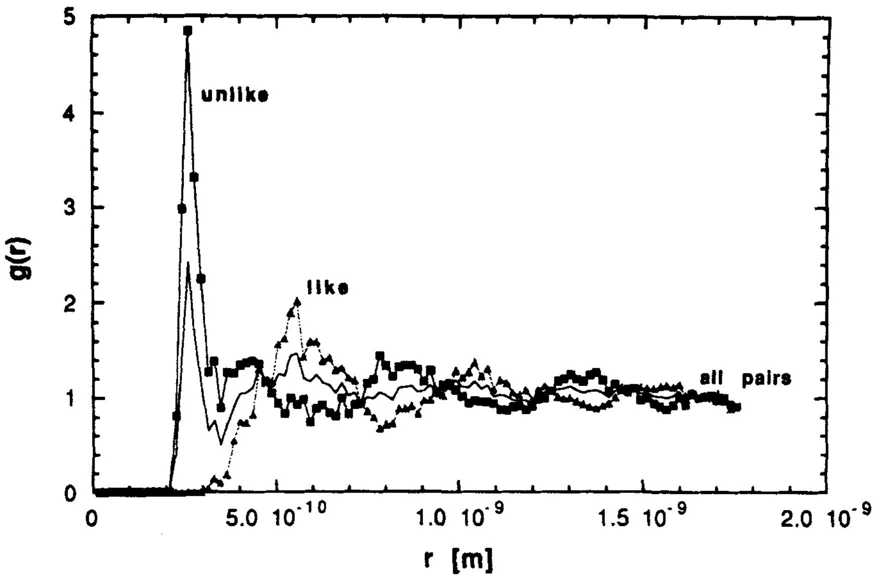

Figure 8 Total and partial radial distribution functions of KCl (ρ = 1.54 g/cm^3; T = 1073K). Unlike (squares), like (triangles) and total (no marks) RDF's from one configuration (512 particles).

elements in that dimension with counters 1 to 3. If the numbers are omitted, all elements in that dimension are meant. So:

$$A\,(:,:) \;=\; A\,(:,:) + 1$$

means that we add the scalar 1 to all array elements of array A and can be written in short as:

$$A \;=\; A + 1$$

In the square algorithm, we use the intrinsic SPREAD. For instance:

$$B \;=\; \text{SPREAD}\,(A(:,:,1)\;,\;2\;,\;N)$$

The other essential intrinsic function in the square algorithm is the SUM function. The operation:

$$B\,(:,:,1) \;=\; \text{SUM}\,(A,3)$$

means summing the elements in the (parallel) rows of A ('summing along dimension 3') and put the resulting column in the first (parallel) column of B.

We apologize that many arrays in the listings have names which are sometimes not very obvious. This is due to the fact that in the Fortran 0.6 version, EQUIVALENCE was not available for arrays with a parallel dimension and arrays had to be 'recycled' because one has to careful with the memory space available. Further we assume that the reader is familiar with implementing the standard predictor-corrector integration algorithm and the Ewald sum [12].

```
      program kcllin
c
c This program simulates a KCL system using the linear algorithm
      implicit integer (i,j,k,l,m,n)
      implicit real (a-h,o-z)
c Set parameters
      parameter (npart=4096,nrep=5,nrep2=2*nrep
     1          ,nrep21=nrep2+1,nkpnt=nrep21**3,nreppl=nrep+1)
c Define array's
      real, array(3,npart) ::      x0,x1,x2,x3,x4,x5,x2p,dx2c,
     1                             x0shift,
     2                             dr,dxsq,dx6,dx8
      real            erfc(npart),erfc2(npart),sqrdr(npart),
     1                qi(npart),qj(npart),exped(npart)
      real            fkk(3,nkpnt),gspace(3,nkpnt),
     1                fksc(nkpnt)
      integer         idr(npart)
      logical, array(npart) ::     lspecies
c
c Define geometry
CMF$  layout x0( :serial, :news)
CMF$  align x1(i,j)       with x0(i,j)
CMF$  align x2(i,j)       with x0(i,j)
CMF$  align x3(i,j)       with x0(i,j)
CMF$  align x4(i,j)       with x0(i,j)
CMF$  align x5(i,j)       with x0(i,j)
CMF$  align x2p(i,j)      with x0(i,j)
CMF$  align dx2c(i,j)     with x0(i,j)
CMF$  align x0shift(i,j)  with x0(i,j)
CMF$  align dr(i,j)       with x0(i,j)
CMF$  align dx6(i,j)      with x0(i,j)
CMF$  align dx8(i,j)      with x0(i,j)
```

```
CMF$  align erfc(j)       with x0(+0,j)
CMF$  align erfc2(j)      with x0(+0,j)
CMF$  align sqrdr(j)      with x0(+0,j)
CMF$  align qi(j)         with x0(+0,j)
CMF$  align qj(j)         with x0(+0,j)
CMF$  align exped(j)      with x0(+0,j)
CMF$  align lspecies(j)   with x0(+0,j)
CMF$  align idr(j)        with x0(+0,j)
c
c  Predictor-corrector coeff.
      fc0=3.0/16.0
      fc1=251.0/360.0
      fc2=1.0
      fc3=11.0/18.0
      fc4=1.0/6.0
      fc5=1.0/60.0
c
c  Error-function constants: Abramowitz 7.1.26
      eta=5.67
      perf=0.3275911
      a1=0.254829592
      a2=-0.284496736
      a3=1.421413741
      a4=-1.453152027
      a5=1.061405429
c
c  Numerical constants
      othird=1.0/3.0
      pi=4.0*atan(1.0)
      twopi=2.0*pi
```

```
c
c  Natural constants
      boltzman=1.380622e-23
      atomic_mass_u=1.660531e-27
      echarge=1.6021917e-19
      spofl=2.99794562e8
      ezero=1.0/(spofl*spofl*4.0*pi*1.0e-7)
c
c Woodcock & Singer Potential constants
      sigwsii=2.926e-10
      sigwsij=3.048e-10
      sigwsjj=3.170e-10
      bsmalii=0.423e-19
      bsmalij=0.338e-19
      bsmaljj=0.253e-19
      bbig=2.97e10
      cwsii=1.15949e-13
      cwsij=1.29879e-13
      cwsjj=1.52240e-13
      dwsii=6.27374e-13
      dwsij=7.20976e-13
      dwsjj=8.40896e-13
c
c Define or read initial positions (x0) and velocities (x1) and
c  charges (qi) of the particles, the Nosé variables (sn0, sn1),
c  temperature (tempdsrd), density (densdsrd) etc.
c lspecies(i)=.true. ==> particle i is a K atom (Cl otherwise)
c
      print *,' Give the number of integration steps '
      read *,nintstep
      print *,' Give the reduced time step (0.0025 ~1e-15 sec)'
      read *,rddeltat
      print *,' Give the Q>0 factor (Reduced Nosé mass)'
      read *,qfactor
c
c System constants
      epsilon=10.22*boltzman
      sigma=2.56e-10
      pmassk=39.102
      pmasscl=35.453
      pmass=0.5*(pmassk+pmasscl)
      partmass=pmass*atomic_mass_u
      prtmassk=pmassk*atomic_mass_u
      prtmasscl=pmasscl*atomic_mass_u
      ekinkcnst=0.5*prtmassk
      ekinclcnst=0.5*prtmasscl
      tempscale=2.0/(3.0*boltzman*float(npart))
      dens=densdsrd/(sigma*sigma*sigma)
      deltat=rddeltat*sqrt(partmass*sigma)*sqrt(sigma/epsilon)
c  this SQRT() has been split because it returns a 0.0 otherwise
c  ( tested for single precision )
      print *,' Real time step=',deltat
c
      x2=0.0
      x3=0.0
      x4=0.0
      x5=0.0
c
c Nose variables
      sn2=0.0
      sn3=0.0
      sn4=0.0
      sn5=0.0
      gkt=3.0*float(npart)*boltzman*tempdsrd
      dtsq=deltat**2
      print *,' gkt=',gkt,'  npart=',npart,'  boltzman',boltzman,
     1        '  tempdsrd=',tempdsrd,'  dtsq=',dtsq
      qnose=300.0*gkt*qfactor
      print *,' qnose=',qnose
c
c Scaled costants for the force and energy computation
       side=(volume**othird)/sigma
       sidi2=2.0/side
       fconstant=48.0*0.5*(epsilon/partmass)*(deltat/sigma)**2
       fcnstk=48.0*0.5*(epsilon/prtmassk)*(deltat/sigma)**2
       fcnstcl=48.0*0.5*(epsilon/prtmasscl)*(deltat/sigma)**2
       sigmai=1.0/sigma
       fnpart=float(npart)
       fnparti=1.0/fnpart
       qnosi=1.0/qnose
       potencnst=4.0*epsilon*0.5
       dtsqinv=1.0/(deltat*deltat)
       ekinkcnst=prtmassk*dtsqinv*0.5*sigma*sigma
       ekinclcnst=prtmasscl*dtsqinv*0.5*sigma*sigma
       gktdtsq=gkt*dtsqinv
c
c Scale position and velocities
      x0=x0*sigmai
      x1=x1*sigmai*deltat
      nparth=npart/2
```

```
c
c  Scaled Ewald sum constants
      erel=1.0
      eta=eta/side
      perfeta=perf*eta
      etasq=eta*eta
      einfinite=4.0*pi*erel*(ezero/echarge)*(sigma/echarge)
      vol=side**3
      f1c1=(1.0/einfinite)
      v1c1=(0.5/einfinite)
      v1c2=-eta/(einfinite*sqrt(pi))*dotproduct(qi,qi)
      f1c2=2.0*eta/sqrt(pi)
      f2c1=(2.0/vol)/einfinite
      v2c1=(0.5/(pi*vol))/einfinite
      v3c1=-((pi/vol)/einfinite)*othird           ! a=b=c=1
      f3c1=((4.0*pi/vol)/einfinite)*othird        ! bx=by=bz=1/3
      fsck=(0.5/prtmassk)*((deltat/sigma)**2)
      fsccl=(0.5/prtmasscl)*((deltat/sigma)**2)
c
c Set up the k-space grids
      etasqi=-pi*pi/etasq
      sidi=1.0/side
      sidisq=sidi**2
      twpisq=twopi**2
      twpds=twopi*sidi
      i=0
      do kz=-nrep,nrep
         zk=sidi*real(kz)
         do ky=-nrep,nrep
            yk=sidi*real(ky)
```

```
          do kx=-nrep,nrep
             xk=sidi*real(kx)
             fksq=sidisq*real((kx**2)+(ky**2)+(kz**2))
             i=i+1
             if (fksq.gt.0.0) then
                fksc(i)=exp(etasqi*fksq)/fksq
                fkk(1,i)=fksc(i)*xk
                fkk(2,i)=fksc(i)*yk
                fkk(3,i)=fksc(i)*zk
                gspace(1,i)=twopi*xk
                gspace(2,i)=twopi*yk
                gspace(3,i)=twopi*zk
             else
                fksc(i)=0.0
                fkk(1,i)=0.0
                fkk(2,i)=0.0
                fkk(3,i)=0.0
                gspace(1,i)=0.0
                gspace(2,i)=0.0
                gspace(3,i)=0.0
             endif
          end do
       end do
    end do
c
c Scale T&F Potential and force parameters
c (ii => K-K; ij => K-Cl; jj => Cl-Cl)
    sigwsii=sigwsii/sigma
    sigwsij=sigwsij/sigma
    sigwsjj=sigwsjj/sigma
```

```
      bbig=bbig*sigma
      bfii=bsmalii*bbig
      bfij=bsmalij*bbig
      bfjj=bsmaljj*bbig
      sixcii=6.0
      sixcij=6.0
      sixcjj=6.0
      eigdii=8.0
      eigdij=8.0
      eigdjj=8.0
      cwsii=(cwsii/sigma)**2
      cwsij=(cwsij/sigma)**2
      cwsjj=(cwsjj/sigma)**2
      dwsii=(dwsii/sigma)**2
      dwsij=(dwsij/sigma)**2
      dwsjj=(dwsjj/sigma)**2
c
c Reset and start the timer
      call CM_reset_timer
      call CM_start_timer
c
c                              START MD-LOOP
      do istep=1,nintstep
c
c Predictor step
      x0= x0+    x1+     x2+     x3+     x4+      x5
      x1=        x1+2.0*x2+3.0*x3+4.0*x4+5.0 *x5
      x2p=               x2+3.0*x3+6.0*x4+10.0*x5
      x3=                        x3+4.0*x4+10.0*x5
      x4=                                x4+5.0 *x5
```

```
c      x5=                                        x5

       sn0=sn0+  sn1+     sn2+     sn3+     sn4+      sn5

       sn1=      sn1+2.0*sn2+3.0*sn3+4.0*sn4+5.0 *sn5

       sn2p=              sn2+3.0*sn3+6.0*sn4+10.0*sn5

       sn3=                        sn3+4.0*sn4+10.0*sn5

       sn4=                                 sn4+5.0 *sn5

c      sn5=                                           sn5

c

c Compute x2 & snose2; acceleration*dt*dt*0.5

       x2=0.0

       esum=0.0

       v1c=0.0

       v2c=0.0

       v3c=0.0

       x0shift=x0

       qj=qi

       iforces=0

       do i=1,nparth

c

c Compute distances of the particles pairs (j,j+i) using CSHIFT

          x0shift=cshift(x0shift,2,1)

          dx2c=x0-x0shift

          dx2c=dx2c-side*aint(sidi2*dx2c)

          dr=dx2c**2

          dr(1,:)=dr(1,:)+dr(2,:)+dr(3,:)

          sqrdr=sqrt(dr(1,:))

          qj=cshift(qj,1,1)

c

          dr(2,:)=qi*qj

c
```

```
c Coulomb forces and potential from central cell in real space
         erfc=1.0/(1.0+perfeta*sqrdr)
         erfc2=erfc**2
         exped=exp(-etasq*dr(1,:))
         erfc=erfc*(a1+a2*erfc+a3*erfc2+a4*erfc*erfc2
     1             +a5*(erfc2**2))*exped
         erfc2=dr(2,:)*erfc/sqrdr
         v1c=v1c+sum(erfc2)
         erfc=f1c1*(erfc+f1c2*sqrdr*exped)/(sqrdr*dr(1,:))
         dx8(1,:)=dr(2,:)*dx2c(1,:)
         dx8(2,:)=dr(2,:)*dx2c(2,:)
         dx8(3,:)=dr(2,:)*dx2c(3,:)
         dx6(1,:)=dx8(1,:)*erfc
         dx6(2,:)=dx8(2,:)*erfc
         dx6(3,:)=dx8(3,:)*erfc
c
c Forces and potential by macroscopic shape part part of the E-sum
         dx8=dx6+f3c1*dx8
         erfc=dr(2,:)*dr(1,:)
         v3c=v3c+sum(erfc)
c
c Forces and potential by Core and multipole interactions (T&F)
       dr(2,:)=1.0/dr(1,:)
c
c K-K interactions
       where (lspecies.and.qi.eq.qj)
           dr(3,:)=exp(bbig*(sigwsii-sqrdr))
           dx6(1,:)=bfii*dr(3,:)*sqrdr(:)
           dr(3,:)=bsmalii*dr(3,:)
           dr(3,:)=dr(3,:)-((cwsii*dr(2,:))**3)
     1                        -((dwsii*dr(2,:))**4)
            dx6(1,:)=dx6(1,:)-(sixcii*(cwsii*dr(2,:))**3)
     2                          -(eigdii*(dwsii*dr(2,:))**4)
            dx6(3,:)=dx6(1,:)*dr(2,:)*dx2c(3,:)
            dx6(2,:)=dx6(1,:)*dr(2,:)*dx2c(2,:)
            dx6(1,:)=dx6(1,:)*dr(2,:)*dx2c(1,:)
      end where
c
c K-Cl interactions
      where (qi.ne.qj)
            dr(3,:)=exp(bbig*(sigwsij-sqrdr))
            dx6(1,:)=bfij*dr(3,:)*sqrdr(:)
            dr(3,:)=bsmalij*dr(3,:)
            dr(3,:)=dr(3,:)-((cwsij*dr(2,:))**3)
     1                        -((dwsij*dr(2,:))**4)
            dx6(1,:)=dx6(1,:)-(sixcij*(cwsij*dr(2,:))**3)
     2                          -(eigdij*(dwsij*dr(2,:))**4)
            dx6(3,:)=dx6(1,:)*dr(2,:)*dx2c(3,:)
            dx6(2,:)=dx6(1,:)*dr(2,:)*dx2c(2,:)
            dx6(1,:)=dx6(1,:)*dr(2,:)*dx2c(1,:)
      end where
c
c Cl-Cl interactions
      where (.not.lspecies.and.qi.eq.qj)
            dr(3,:)=exp(bbig*(sigwsjj-sqrdr))
            dx6(1,:)=bfjj*dr(3,:)*sqrdr(:)
            dr(3,:)=bsmaljj*dr(3,:)
            dr(3,:)=dr(3,:)-((cwsjj*dr(2,:))**3)
     1                        -((dwsjj*dr(2,:))**4)
            dx6(1,:)=dx6(1,:)-(sixcjj*(cwsjj*dr(2,:))**3)
     2                         -(eigdjj*(dwsjj*dr(2,:))**4)
              dx6(3,:)=dx6(1,:)*dr(2,:)*dx2c(3,:)
              dx6(2,:)=dx6(1,:)*dr(2,:)*dx2c(2,:)
              dx6(1,:)=dx6(1,:)*dr(2,:)*dx2c(1,:)
      end where
         esum=esum+sum(dr(3,:))
         dx6=dx8+dx6
         x2=x2+dx6
c
c Cshift the forces on the particles i places back and subtract
         dx6=cshift(dx6,2,-i)
         x2=x2-dx6
c
 11   end do
c
c Forces and potential by k-space part of the E-sum
      dx8=0.0
      do ik=1,nkpnt
         dx6(3,:)=gspace(1,ik)*x0(1,:)
     1            +gspace(2,ik)*x0(2,:)
     2            +gspace(3,ik)*x0(3,:)
         dx6(1,:)=qi*sin(dx6(3,:))
         dx6(2,:)=qi*cos(dx6(3,:))
         sskj=sum(dx6(1,:))
         sckj=sum(dx6(2,:))
         dx6(3,:)=sckj*dx6(1,:)
         dx6(3,:)=dx6(3,:)-sskj*dx6(2,:)
         dx8(1,:)=dx8(1,:)+fkk(1,ik)*dx6(3,:)
         dx8(2,:)=dx8(2,:)+fkk(2,ik)*dx6(3,:)
         dx8(3,:)=dx8(3,:)+fkk(3,ik)*dx6(3,:)
         v2c=v2c+fksc(ik)*(sskj**2+sckj**2)
      end do
c
      x2=x2+f2c1*dx8
c
      v1c=2.0*v1c1*v1c
      v2c=v2c1*v2c
      v3c=2.0*v3c1*v3c
      poten=esum
c
c Compute accelerations
      where (lspecies)      x2(1,:)=fsck*x2(1,:)
      where (lspecies)      x2(2,:)=fsck*x2(2,:)
      where (lspecies)      x2(3,:)=fsck*x2(3,:)
      where (.not.lspecies) x2(1,:)=fsccl*x2(1,:)
      where (.not.lspecies) x2(2,:)=fsccl*x2(2,:)
      where (.not.lspecies) x2(3,:)=fsccl*x2(3,:)
c
      x2=x2-(0.5*sn1/sn0)*x1
c
c Compute temperature
      dotkx1=0.0
      dx6=0.0
      where (lspecies) dx6(1,:)=x1(1,:)
      where (lspecies) dx6(2,:)=x1(2,:)
      where (lspecies) dx6(3,:)=x1(3,:)
      do i=1,3
         dotkx1=dotkx1+dotproduct(dx6(i,:),dx6(i,:))
      end do
      dotclx1=0.0
      dx6=0.0
      where (.not.lspecies) dx6(1,:)=x1(1,:)
      where (.not.lspecies) dx6(2,:)=x1(2,:)
      where (.not.lspecies) dx6(3,:)=x1(3,:)
      do i=1,3
         dotclx1=dotclx1+dotproduct(dx6(i,:),dx6(i,:))
      end do
      ekin=ekinkcnst*dotkx1+ekinclcnst*dotclx1
      temp=tempscale*ekin
c
      sn2=sn0*(2.0*ekin-gkt)*qnosi+(sn1*sn1)/sn0
      sn2=0.5*sn2
c
c Corrector step
      dx2c=x2-x2p
      x0=x0+fc0*dx2c
      x1=x1+fc1*dx2c
      x3=x3+fc3*dx2c
      x4=x4+fc4*dx2c
      x5=x5+fc5*dx2c
c
      ds2c=sn2-sn2p
      sn0=sn0+fc0*ds2c
      sn1=sn1+fc1*ds2c
      sn3=sn3+fc3*ds2c
      sn4=sn4+fc4*ds2c
      sn5=sn5+fc5*ds2c
c
      x0=x0-side*aint(sidi2*x0-1.0)
c
```

```
      eknose=qnose*0.5*(sn1/sn0)**2
      ponose=gkt*log(sn0)
      ham=(ekin+poten+eknose+ponose+v1c+v2c+v3c)*epsi*fnparti
c
c Preform optional statistics, printing, computation of RDF etc.
c
      end do
c                                 END   MD_LOOP
c
c Stop timer and print wall clock time and estimated CPU time
      call CM_stop_timer
c
c  Write all variables to tape etc.
c
      end
```

End of the listing of the linear algorithm for KCl

```
       program kclsq
c
c This program simulates a KCL system using the square algorithm
      implicit integer (i,j,k,l,m,n)
      implicit real (a-h,o-z)
c Set parameters, nrows is the number of columns with particles
c So the total number of particles is here 512*8=4096
      parameter (npart=512,nrows=8,nrep=5,nrep2=2*nrep
     1          ,nrep21=nrep2+1,nkpnt=nrep21**3,nrepp1=nrep+1)
c Define Array's
      real, array(3,npart,npart) :: x0,x1,x2,x3,x4,x5,x2p,dx2c,
     2                              dr,dx6,dx8,
     4                                x0sp,x2sum
      integer, array(nrows,nrows) :: nbrlst
c
      real, array(npart,npart)   :: erfc,erfc2,sqrdr,
     1                               qi,qiqj,exped
      real, array(nkpnt) :: fksc
      real, array(3,nkpnt) :: fkk,gspace
      logical, array(npart,npart) :: lspecies,lspesp
c
c Define geometry
CMF$  layout x0( :serial, :news, :news)
CMF$  align x1(i,j,k)      with x0(i,j,k)
CMF$  align x2(i,j,k)      with x0(i,j,k)
CMF$  align x3(i,j,k)      with x0(i,j,k)
CMF$  align x4(i,j,k)      with x0(i,j,k)
CMF$  align x5(i,j,k)      with x0(i,j,k)
CMF$  align x2p(i,j,k)     with x0(i,j,k)
CMF$  align dx2c(i,j,k)    with x0(i,j,k)
CMF$  align dr(i,j,k)      with x0(i,j,k)
CMF$  align dx6(i,j,k)     with x0(i,j,k)
CMF$  align dx8(i,j,k)     with x0(i,j,k)
CMF$  align x0sp(i,j,k)    with x0(i,j,k)
CMF$  align x2sum(i,j,k)   with x0(i,j,k)
CMF$  align erfc(j,k)      with x0(+0,j,k)
CMF$  align erfc2(j,k)     with x0(+0,j,k)
CMF$  align sqrdr(j,k)     with x0(+0,j,k)
CMF$  align qi(j,k)        with x0(+0,j,k)
CMF$  align qiqj(j,k)      with x0(+0,j,k)
CMF$  align exped(j,k)     with x0(+0,j,k)
CMF$  align lspecies(j,k)  with x0(+0,j,k)
```

```
CMF$  align lspesp(j,k)    with x0(+0,j,k)
c
c Read or initialize positions velocities etc., define constants etc.
c Set up the k-space grids etc. See the listing of the linear algorithm
c above.
c
c Set up the neighbour listing (All columns have interaction here)
      do icell=1,nrows
         do inbr=1,nrows
            nbrlst(icell,inbr)=0
         end do
      end do
      do icell=1,nrows
         i=1
         do inbr=icell,nrows
            nbrlst(icell,i)=inbr
            i=i+1
         end do
      end do
c
c Reset and start the timer
      call CM_reset_timer
      call CM_start_timer
c
c                                START MD-LOOP
      do istep=1,nintstep
      x0= x0+    x1+     x2+     x3+     x4+     x5
      x1=        x1+2.0*x2+3.0*x3+4.0*x4+5.0 *x5
      x2p=               x2+3.0*x3+6.0*x4+10.0*x5
      x3=                        x3+4.0*x4+10.0*x5
      x4=                               x4+5.0 *x5
c     x5=                                      x5
c
      sn0=sn0+  sn1+    sn2+    sn3+    sn4+     sn5
      sn1=      sn1+2.0*sn2+3.0*sn3+4.0*sn4+5.0 *sn5
      sn2p=             sn2+3.0*sn3+6.0*sn4+10.0*sn5
      sn3=                      sn3+4.0*sn4+10.0*sn5
      sn4=                              sn4+5.0 *sn5
c     sn5=                                       sn5
c
c Compute x2 & snose2; accelaration*dt*dt*0.5 and potentials
      x2=0.0
      esum=0.0
      v1c=0.0
      v2c=0.0
      v3c=0.0
c
c Loop over all array rows (subcell)
      do icell=1,nrows
         x0sp=spread(x0(:,:,icell),2,npart)
         lspesp=spread(lspecies(:,icell),1,npart)
c
c Loop over the columns considered as neighbours of the subcell
         x2sum=0.0
         do inbr=1,nrows
            nnbr=nbrlst(icell,inbr)
            if (nnbr.eq.0) goto 111
c
c Compute the distances
            dx2c=spread(x0(:,:,nnbr),3,npart)
            dx2c=x0sp-dx2c
            dx2c=dx2c-side*aint(sidi2*dx2c)
            dr=dx2c**2
            dr(1,:,:)=dr(1,:,:)+dr(2,:,:)+dr(3,:,:)
            sqrdr=sqrt(dr(1,:,:))
c
c Compute the charge pair product qi*qj
            qiqj=spread(qi(:,nnbr),2,npart)
            where (.not.lspesp) qiqj=-1.0*qiqj
c
c Coulomb interaction with the T&L convergence factor
            dx6=0.0
            dx8=0.0
            erfc=0.0
            erfc2=0.0
      where (dr(1,:,:).ne.0.0)
            erfc=1.0/(1.0+perfeta*sqrdr)
            erfc2=erfc**2
            exped=exp(-etasq*dr(1,:,:))
            erfc=erfc*(a1+a2*erfc+a3*erfc2+a4*erfc*erfc2
     1                +a5*(erfc2**2))*exped
            erfc2=qiqj*erfc/sqrdr
            erfc=f1c1*(erfc+f1c2*sqrdr*exped)/(sqrdr*dr(1,:,:))
            dx8(1,:,:)=qiqj*dx2c(1,:,:)
            dx8(2,:,:)=qiqj*dx2c(2,:,:)
            dx8(3,:,:)=qiqj*dx2c(3,:,:)
            dx6(1,:,:)=dx8(1,:,:)*erfc
            dx6(2,:,:)=dx8(2,:,:)*erfc
            dx6(3,:,:)=dx8(3,:,:)*erfc
      end where
```

```
              v1csum=sum(erfc2)
c
c Forces and potential by macroscopic shape part of the E-sum
              dx8=dx6+f3c1*dx8
              erfc=qiqj*dr(1,:,:)
              v3csum=sum(erfc)
              v3cdum=v3c1*v3csum
c Keep dx8
c
c Core and multipole W&S interactions
              dr(3,:,:)=0.0
              dx6=0.0
              dr(2,:,:)=0.0
       where (dr(1,:,:).ne.0.0) dr(2,:,:)=1.0/dr(1,:,:)
c
c  K-K interactions
       where (lspesp.and.qiqj.gt.0.0.and.dr(1,:,:).ne.0.0)
              dr(3,:,:)=exp(bbig*(sigwsii-sqrdr))
              dx6(1,:,:)=bfii*dr(3,:,:)*sqrdr
              dr(3,:,:)=bsmalii*dr(3,:,:)
              dr(3,:,:)=dr(3,:,:)-((cwsii*dr(2,:,:))**3)
     1                              -((dwsii*dr(2,:,:))**4)
              dx6(1,:,:)=dx6(1,:,:)-(sixcii*(cwsii*dr(2,:,:))**3)
     2                                -(eigdii*(dwsii*dr(2,:,:))**4)
              dx6(3,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(3,:,:)
              dx6(2,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(2,:,:)
              dx6(1,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(1,:,:)
       end where
c
c  K-Cl interacions
```

```
      where (qiqj.lt.0.0.and.dr(1,:,:).ne.0.0)
            dr(3,:,:)=exp(bbig*(sigwsij-sqrdr))
            dx6(1,:,:)=bfij*dr(3,:,:)*sqrdr
            dr(3,:,:)=bsmalij*dr(3,:,:)
            dr(3,:,:)=dr(3,:,:)-((cwsij*dr(2,:,:))**3)
     1                         -((dwsij*dr(2,:,:))**4)
            dx6(1,:,:)=dx6(1,:,:)-(sixcij*(cwsij*dr(2,:,:))**3)
     2                           -(eigdij*(dwsij*dr(2,:,:))**4)
            dx6(3,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(3,:,:)
            dx6(2,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(2,:,:)
            dx6(1,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(1,:,:)
      end where
c
c Cl-Cl interactions
      where (.not.lspesp.and.qiqj.gt.0.0.and.dr(1,:,:).ne.0.0)
            dr(3,:,:)=exp(bbig*(sigwsjj-sqrdr))
            dx6(1,:,:)=bfjj*dr(3,:,:)*sqrdr
            dr(3,:,:)=bsmaljj*dr(3,:,:)
            dr(3,:,:)=dr(3,:,:)-((cwsjj*dr(2,:,:))**3)
     1                         -((dwsjj*dr(2,:,:))**4)
            dx6(1,:,:)=dx6(1,:,:)-(sixcjj*(cwsjj*dr(2,:,:))**3)
     2                           -(eigdjj*(dwsjj*dr(2,:,:))**4)
            dx6(3,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(3,:,:)
            dx6(2,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(2,:,:)
            dx6(1,:,:)=dx6(1,:,:)*dr(2,:,:)*dx2c(1,:,:)
      end where
            pcsum=0.5*sum(dr(3,:,:))
            dx8=dx8+dx6
c
            x2sum=x2sum+dx8
```

```
c
            v1c=v1c+v1csum
            esum=esum+pcsum
            v3c=v3c+v3csum
            if (inbr.ne.1) then
               dx2c(:,:,nnbr)=sum(dx8,3)
               x2(:,:,nnbr)=x2(:,:,nnbr)-dx2c(:,:,nnbr)
               v1c=v1c+v1csum
               esum=esum+pcsum
               v3c=v3c+v3csum
            end if
         end do
 111     dx8(:,:,icell)=sum(x2sum,2)
         x2(:,:,icell)=x2(:,:,icell)+dx8(:,:,icell)
      end do
c
c Forces and potential by k-space part of the E-sum
c The E-sum has been rewritten to the local form, avoiding complex
c  numbers, which were not available in the 0.5 release of CM-Fortran
c qiqj is not a charge product here but the speaded qi's
c
c forces (i,j) and potential by k-space part
      x0sp=0.0
      v2c=0.0
      do ik=1,nkpnt
            dx6(1,:,:)=gspace(1,ik)*x0(1,:,:)
     1                 +gspace(2,ik)*x0(2,:,:)
     2                 +gspace(3,ik)*x0(3,:,:)
            dx6(2,:,:)=qi*sin(dx6(1,:,:))
            dx6(3,:,:)=qi*cos(dx6(1,:,:))
            sskj=sum(dx6(2,:,:))

            sckj=sum(dx6(3,:,:))

            dx8(2,:,:)=sckj*dx6(2,:,:)

            dx8(2,:,:)=dx8(2,:,:)-sskj*dx6(3,:,:)

            x0sp(1,:,:)=x0sp(1,:,:)+fkk(1,ik)*dx8(2,:,:)

            x0sp(2,:,:)=x0sp(2,:,:)+fkk(2,ik)*dx8(2,:,:)

            x0sp(3,:,:)=x0sp(3,:,:)+fkk(3,ik)*dx8(2,:,:)

            v2c=v2c+fksc(ik)*(sskj**2+sckj**2)

         end do

      x2=x2+f2c1*x0sp

c

      v1c=v1c1*v1c

      v2c=v2c1*v2c

      v3c=v3c1*v3c

      poten=esum

c

c Compute the accelerations from the forces

      where (lspecies)      x2(1,:,:)=fsck*x2(1,:,:)

      where (lspecies)      x2(2,:,:)=fsck*x2(2,:,:)

      where (lspecies)      x2(3,:,:)=fsck*x2(3,:,:)

      where (.not.lspecies) x2(1,:,:)=fsccl*x2(1,:,:)

      where (.not.lspecies) x2(2,:,:)=fsccl*x2(2,:,:)

      where (.not.lspecies) x2(3,:,:)=fsccl*x2(3,:,:)

c

      x2=x2-(0.5*sn1/sn0)*x1

c

c Compute the temperature

      dotkx1=0.0

      dx6=0.0

      where (lspecies) dx6(1,:,:)=x1(1,:,:)
      where (lspecies) dx6(2,:,:)=x1(2,:,:)
      where (lspecies) dx6(3,:,:)=x1(3,:,:)
c
      do icell=1,nrows
         do i=1,3
            dotx1=dotproduct(dx6(i,:,icell),dx6(i,:,icell))
            dotkx1=dotkx1+dotx1
         end do
      end do
      dotclx1=0.0
      dx6=0.0
      where (.not.lspecies) dx6(1,:,:)=x1(1,:,:)
      where (.not.lspecies) dx6(2,:,:)=x1(2,:,:)
      where (.not.lspecies) dx6(3,:,:)=x1(3,:,:)
      do icell=1,nrows
         do i=1,3
            dotx1=dotproduct(dx6(i,:,icell),dx6(i,:,icell))
            dotclx1=dotclx1+dotx1
         end do
      end do
      ekin=ekinkcnst*dotkx1+ekinclcnst*dotclx1
      temp=tempscale*ekin
c
      sn2=sn0*(2.0*ekin-gkt)*qnosi+(sn1*sn1)/sn0
      sn2=0.5*sn2
c
      dx2c=x2-x2p
      x0=x0+fc0*dx2c
      x1=x1+fc1*dx2c
      x3=x3+fc3*dx2c
      x4=x4+fc4*dx2c
      x5=x5+fc5*dx2c
c
      ds2c=sn2-sn2p
      sn0=sn0+fc0*ds2c
      sn1=sn1+fc1*ds2c
      sn3=sn3+fc3*ds2c
      sn4=sn4+fc4*ds2c
      sn5=sn5+fc5*ds2c
c
      x0=x0-side*aint(sidi2*x0-1.0)
c
      eknose=qnose*0.5*(sn1/sn0)**2
      ponose=gkt*log(sn0)
      ham=(ekin+poten+eknose+ponose+v1c+v2c+v3c)*epsi*fnparti
c
c Perform optional statistics, printing, etc.
c
      end do
c                              END   MD_LOOP
c
c Stop timer and print wall clock time and estimated CPU time
      call CM_stop_timer
c
      print *,' Computing the radial dist. func.'
c
      do i=1,101
         gofr(1,i)=0.0
         gofr(2,i)=0.0
         gofr(3,i)=0.0
      end do
      ds=side*0.005
      dsi=1.0/ds
c
c Compute the RDF and the coordination numbers
      fcnkk=0.0
      fcnkcl=0.0
      fcnclcl=0.0
      fcnclk=0.0
      dcldum=dcldum*1.3
c
      do icell=1,nrows
         x0sp=spread(x0(:,:,icell),2,npart)
         lspesp=spread(lspecies(:,icell),1,npart)
c
c Loop over the neighbours of the subcell
         x2sum=0.0
         do inbr=1,nrows
            nnbr=nbrlst(icell,inbr)
            if (nnbr.eq.0) goto 1111
c
c Compute the distances
            dx2c=spread(x0(:,:,nnbr),3,npart)
            dx2c=x0sp-dx2c
            dx2c=dx2c-side*aint(sidi2*dx2c)
            dr=dx2c**2
            dr(1,:,:)=dr(1,:,:)+dr(2,:,:)+dr(3,:,:)
            sqrdr=sqrt(dr(1,:,:))
c
c Compute the charge pair product qi*qj
```

```
          qiqj=spread(qi(:,nnbr),2,npart)
          where (.not.lspesp) qiqj=-1.0*qiqj
c
c Compute the coordination number of K and Cl; dcldum from cubic latice
c  interparticle distance.
          erfc=0.0
          where (lspesp.and.sqrdr.le.dcldum.and.qiqj.gt.0.0
     1             .and.sqrdr.ne.0.0)
     1           erfc=1.0
          fcnkk=fcnkk+sum(erfc)
          erfc=0.0
          where (lspesp.and.sqrdr.le.dcldum.and.qiqj.lt.0.0
     1             .and.sqrdr.ne.0.0)
     1           erfc=1.0
          fcnkcl=fcnkcl+sum(erfc)
          erfc=0.0
          where (.not.lspesp.and.sqrdr.le.dcldum.and.qiqj.gt.0.0
     1             .and.sqrdr.ne.0.0)
     1           erfc=1.0
          fcnclcl=fcnclcl+sum(erfc)
          erfc=0.0
          where (.not.lspesp.and.sqrdr.le.dcldum.and.qiqj.lt.0.0
     1             .and.sqrdr.ne.0.0)
     1           erfc=1.0
          fcnclk=fcnclk+sum(erfc)
c
          idr=int(sqrdr*dsi)-1
          igsumfac=2.0
          if (inbr.eq.1) igsumfac=1.0
          do ig=1,101
```

```
c RDF of all pairs
               erfc=0.0
               where (idr.eq.ig) erfc=igsumfac
               gofr(1,ig)=gofr(1,ig)+sum(erfc)
c RDF of like pairs
               erfc=0.0
               where (idr.eq.ig.and.qiqj.gt.0.0) erfc=igsumfac
               gofr(2,ig)=gofr(2,ig)+sum(erfc)
c RDF of unlike pairs
               erfc=0.0
               where (idr.eq.ig.and.qiqj.lt.0.0) erfc=igsumfac
               gofr(3,ig)=gofr(3,ig)+sum(erfc)
            end do
 1111    end do
      end do
c
      fcnkk=fcnkk*2.0/real(npart*nrows)
      fcnkcl=fcnkcl*2.0/real(npart*nrows)
      fcnclcl=fcnclcl*2.0/real(npart*nrows)
      fcnclk=fcnclk*2.0/real(npart*nrows)
      write (*,90) dcldum
      write (*,91) fcnkk
      write (*,92) fcnkcl
      write (*,93) fcnclcl
      write (*,94) fcnclk
c
c Normalize the RDF
         fnorm1=4.0*pi*densdsrd*ds*ds*ds*othird
     1              *real(isamp*npart*nrows)
         fnorm2=4.0*pi*densdsrd*ds*ds*ds*othird
     1             *real(isamp*npart*0.5*nrows)
        fnorm3=4.0*pi*densdsrd*ds*ds*ds*othird
     1             *real(isamp*npart*0.5*nrows)
c
c The array gofr containing the RDF is kept on the front end
        do i=1,101
           gofr(1,i)=gofr(1,i)/(real(i**3-(i-1)**3)*fnorm1)
           gofr(2,i)=gofr(2,i)/(real(i**3-(i-1)**3)*fnorm2)
           gofr(3,i)=gofr(3,i)/(real(i**3-(i-1)**3)*fnorm3)
        end do
c
c Write positions, velocities, sn0, sn1 etc. and
c  gofr and the coordination numbers to tapes
c
 90   format(1x,' Outer radius for the coordination number is ',e10.4)
 91   format(1x,' The average coordination number of K-K is   ',e10.4)
 92   format(1x,' The average coordination number of K-Cl is  ',e10.4)
 93   format(1x,' The average coordination number of Cl-Cl is ',e10.4)
 94   format(1x,' The average coordination number of Cl-K is  ',e10.4)
c
      end
```

Acknowledgement

The authors would like to thank the Mathematics and Computer Science Division at Argonne for their support, specially David Levine. We also would like to thank Mike Minkoff at Argonne for giving him access to the machine at Los Alamos. B. Boltjes thanks the Thinking Machines Corporation for letting him use their machine in times of need. We thank the John van Geunsfonds and the Argonne National Laboratory for their financial support.

References

[1] F.F. Abraham, W.E. Rudge, D.J. Auerbach and S.W. Koch, "Molecular dynamics of the incommensurate phase of Krypton on graphite using more than 100 000 atoms", *Phys. Rev Lett.* **52**, 445 (1984).

[2] A.F. Bakker, C. Bruin and Hilvorst, "Orientational order at the two dimensional melting transition", *Phys. Rev. Lett.* **52**, 449 (1984).

[3] F.F. Abraham, "Computational statistical mechanics. Methodology, applications and super-computing", *Adv. Phys.* **35**, 1 (1986).

[4] B.G.J.P.T. Murray, P.A. Bash, and M. Karplus, *Molecular dynamics on the Connection Machine*, Technical Report CB88-3 5/88 Thinking Machines Corporation.

[5] D.L. Greenwell, R.K. Kalia, J.C. Patterson and P.D. Vashishta, "Molecular dynamics on the Connection Machine" in *Proc. Scientific Applications of the CM*, (1988) ed. H.D. Simon, pub. World Scientific.

[6] R.W. Hockney and J.W. Eastwood, *Computer Simulations Using Particles*, (1981) Pub. McGraw-Hill Int. Book Comp.

[7] J.W. Perram, H.G. Petersen and S.W. de Leeuw, "An algorithm for the simulation of condensed matter which grows as the 1.5 power of the number of particles", *Mole. Phys.* **65**, 875 (1988).

[8] S. Nose, "A unified formulation of the constant temperature molecular dynamics", *J. Chem. Phys.* **81**, 511 (1984).

[9] S.W. de Leeuw, J.W. Perram and E.R. Smith, "Simulation of electrostatic systems in periodic boundary conditions. I Lattice sums and dielectric constants", *Proc. R. Soc. Lond. A* **373**, 27 (1980).

[10] F.G. Fumi and M.P. Tosi, "Ionic sizes and Born repulsive parameters in the Na-Cl type Alkali-Halides I", *J. Phys. Chem. Solids*, **25**, 31 (1964).

[11] L.V. Woodcock and K. Singer, "Thermodynamic and structual properties of liquid ionic salts by Monte Carlo simulation", *Trans. Far. Soc.* **67**, 12 (1971).

[12] M.P. Allen and D.J. Tildesley, *Computer Simulation of Liquids*, Oxford: Clarendon (1987).