VECTORIZED LINK CELL FORTRAN CODE FOR MOLECULAR DYNAMICS SIMULATIONS FOR A LARGE NUMBER OF PARTICLES

Gary S. GREST
Corporate Research Science Laboratory, Exxon Research and Engineering Company, Annandale, NJ 08801, USA

Burkhard DÜNWEG
Institut für Physik, Universität Mainz, D-6500 Mainz, Fed. Rep. Germany

and

Kurt KREMER

Received 15 March 1989

A highly efficient procedure for calculating forces in a molecular dynamics simulation on a vector computer is described. The algorithm is feasible for several thousand particles on currently available machines. The efficiency comes about by a combination of a Verlet table with a link cell algorithm. By a double list data structure, that treats the pairs of interacting particles in a symmetrical manner, the vectorization can be improved significantly. Moreover, the "layering algorithm" recently described by Rapaport can be incorporated. A standard Fortran formulation of the basic procedure is given. We test the algorithm on a Cray XMP/416 for two different Lennard-Jones fluids with interactions truncated at $2^{1/6}\sigma$ and $2.5\sigma$. The CPU time is shown to increase linearly with system size. Our link cell method proves to be more efficient than a straightforward search over all pairs as soon as the particle number exceeds about 500.

1. Introduction

With the availability of modern supercomputers, molecular dynamics simulations [1–3] have become very important in understanding the properties of many complex systems, including proteins, biological membranes, and polymers. These systems differ from simple atomic fluids, in that it is often necessary to include thousands of particles or degrees of freedom to properly describe their static and dynamic properties. Thus, it is very important to develop computer codes which take the maximum possible advantage of the particular computer architecture available. While in the future, this will probably involve developing algorithms for parallel and special purpose computers, today the most widely accessible supercomputers are the Cray XMP and Cyber 205, which are vector processors. In this paper, we will describe a number of modifications which when incorporated into standard molecular dynamics (MD) codes greatly increase their performance, particularly when the number of particles $N$ exceeds a few thousand.

In most simulations, each particle $i$ in the simulation cell can potentially interact with any of the remaining $(N-1)$ particles and it is necessary to determine in some efficient manner the subset of particles $j$ with which $i$ interacts. For systems with long range interaction, like Coulomb or dipole systems, this problem does not occur since particle $i$ interacts with each one of the remaining $N-1$ particles. However for short range interactions with a well-defined cutoff in the interaction, a significant amount of computer time is often used to simply determine which pairs of particles are close enough to interact. When $N$ is small, say less than about 100, it is sufficient to examine all
of the $N(N-1)/2$ pairs of particles $ij$ each time step to determine which ones have a nonzero force $F_{ij}$. While this simple, brute force technique works well for small $N$, it requires too much CPU time to carry out each step for large $N$. For this reason, Verlet [4] in 1967 introduced a neighbor table, in which those particles $j$ which are nearby particle $i$ are stored. Verlet [4] pointed out that by using a table containing all particle pairs which are separated by a distance less than $(r_c + r_s)$, where $r_c$ is the cutoff in the range of the interaction potential and $r_s$ is a small skin, significant reductions in computer time could be achieved. This is because the search over all pairs of particles, which scales as $N(N-1)/2$, need only be done at intervals. Between updates of the neighbor list, the program just has to check the particle pairs which appear in the list. Since the frequency at which the table must be updated depends on the thermodynamic state point, the time step, the range of the interaction $r_c$ and thickness of the skin $r_s$, the improvement in performance varies from system to system and has to be optimized. However in all cases it is quite significant, particularly as $N$ increases. This combination of an $N(N-1)/2$ search and a Verlet neighbor table has proven to be effective and is probably the most widely used programming technique for MD simulations [1–3].

However in many applications, it is necessary to increase $N$, in some cases to greater than $10^5$. For $N$ larger than about a thousand, searching over all pairs to determine which are within a distance $(r_c + r_s)$ is just too expensive, even on a supercomputer. Instead, an alternate method, the cell index or Link Cell (LC) method [1,3,5–8] is more effective. As seen from fig. 1 for a two-dimensional slice of our three-dimensional cell, in this scheme the simulation cell is broken into $N_c$ smaller sub-cells. The number of these cells depends on $r_c$ and $r_s$. If the edge length of each cell $L_c$ exceeds $r_c$, then all interacting pairs $ij$ are located within the same cell or in one of the 26 adjoining cells (8 in two dimensions). The time to find all the pairs of particles which interact now scales as $N$ compared to $N^2$ for the standard method. The crossover size $N'$ when the LC method becomes more efficient than straightforward search of all $N(N-1)/2$ pairs depends on the range of interaction and the type of computer used. As we will see in section 3, a typical value for a Lennard–Jones monatomic liquid with the interaction truncated at $r_c = 2.5\sigma$ on the Cray XMP is $N' \approx 500$.

Heyes and Smith [8] presented a LC code for use on a Cray that had the desired feature that the computer time increased linearly, not quadratically with $N$. They considered a monatomic Lennard–Jones liquid with an interaction potential $U(r) = 4\varepsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^6\right)$ truncated at $r_c = 2.5\sigma$. At temperature $k_BT = 0.72\varepsilon$ and density $\rho = 0.8442$ using a time step $\Delta t = 0.00462\tau$ ($\tau = \sqrt{m/\varepsilon}$, where $m$ is the particle mass), they obtained the results given in table 1 for CPU time per step on a Cray 1S. While their code used the LC method to determine which pairs of particles were interacting, they did not construct a Verlet table ($r_s = 0$). For $N = 6912$, their code took 1.59 seconds of CPU time per step. Recently, Schoen [9] showed that by constructing an optimized vectorized MD program especially for the Cray XMP using the simple search over all $N(N-1)/2$ pairs and a Verlet neighbor list was much faster than Heyes and Smith's LC program [8] (even accounting for the difference in performance of
The total CPU time per step for four values of the number of particles $N$ for $r_c = 2.5\sigma$. $N_c$ denotes the number of subcells into which the box was divided in the LC method of Heyes and Smith [8] as well as in this work and $r_c^{\text{max}} = L_c = r_c$

<table>
<thead>
<tr>
<th>$N$</th>
<th>CPU time per step (second)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ref. [8]</td>
</tr>
<tr>
<td>1372</td>
<td>0.45</td>
</tr>
<tr>
<td>2048</td>
<td>0.57</td>
</tr>
<tr>
<td>4000</td>
<td>--</td>
</tr>
<tr>
<td>6912</td>
<td>1.59</td>
</tr>
</tbody>
</table>

the Cray XMP and 1S). Schoen's results [9] for the same state point with the Verlet table updated every 20 steps are presented in table 1. For $N = 6912$, this code ran in 0.529 seconds per step, nearly 3 times faster than that of Heyes and Smith [8] even though it is quadratic in $N$ and not linear in $N$. Schoen's results suggested that $N'$ was greater than $10^4$.

In this paper, we show that the LC algorithm presented by Heyes and Smith [8] can be improved by incorporating a Verlet-type neighbor table and restructuring the forces loop. We find the results presented in table 1 for the same state point and updating the Verlet table every 20 steps. For $N = 6912$, our fastest program which incorporates the layered data structure first suggested by Rapaport, runs at the speed of 0.162 steps per second, nearly 10 times faster than Heyes and Smith's program [8] and 3.3 times faster than Schoen's [9]. We also found that by using the $N(N-1)/2$ search over all pairs for the Verlet table and our formulation of the forces loops, we could improve on Schoen's time by an average of 10%, which is useful for small $N$. We find that our LC programs become more efficient than our fastest $N(N-1)/2$ algorithm for $N' \approx 500$ for $r_c = 2.5\sigma$ and the value of $N'$ decreases as the range of interaction decreases (see section 5).

One of the reasons for the improvement Schoen [9] found, compared to the LC method presented by Heyes and Smith [8], is that he used a Verlet [4] neighbor list and calculated the force only between pairs with $r_{ij} < r_c$. Heyes and Smith determined the force between all pairs $ij$ which are in the same cell or one of the 26 adjoining cells, including those with $r_{ij} > r_c$. They then multiplied the resulting forces $F_{ij}$ by zero for all $r_{ij} > r_c$ in order to vectorize the loop which determines the forces. However, since the number of pairs which interact is much smaller than the number of particles in the 27 cells which must be checked (a ratio of $4\pi r_c^3/81 L_\sigma^3$), this results in a significant amount of extra computations, particularly if the interaction potential is complicated.

One obvious way to try to improve the performance of the LC method is to combine it with a Verlet neighbor table, taking care to assure that most if not all of the loops are vectorized. The main advantage of this way of constructing the code is that the table needs only to be updated at intervals, thereby greatly reducing the fraction of time the program spends in determining which pairs are close enough to interact. Since $F_{ij} = -F_{ji}$ only pairs with $j > i$ need to be stored. The usual method most are familiar with who use a Verlet neighbor table [1-4] is to construct a long one-dimensional list of size $M$, LIST(M), of particles $j > i$ ($M > N$). Holding then constructed the Verlet table in this manner, one can follow the procedure outlined by Schoen [9] to vectorize the forces loop. However, the generation of a neighbor table with this structure cannot be
done using the LC method in a way which can be vectored. It can only be done in a scalar mode. Here we show that by modifying the structure of the Verlet neighbor table to include two lists LIST1 and LIST2 of size M instead of just one, it is possible to use the LC method for building a neighbor table which can then be used to determine the forces between particles \(ij\) in a vectored manner. In this data structure, LIST1(K) and LIST2(K) are the two particles that form the Kth pair. This symmetric treatment allows vectorization over pairs instead of over neighbors of a given particle. In the LC method, as seen from fig. 2, all pairs \(ij\) within a volume \((3L_c)^3\) would be included in LIST1(M) and LIST2(M) \((3L_c)^2\) in two dimensions). By this construction, the two lists contain many more elements than when set up by a simple search over all particles. In fact, M increases by a factor of 81\(L_c^3/4\pi(r_c + r_c)^3 \geq 6.45\), as one has to assure that \(L_c \geq (r_c + r_c)\). This restriction comes from the fact that in the LC method it is essential that all pairs be in the same cell or in one of the 26 adjoining cells. For a particle on the edge of one cell, this restriction guarantees that particles two cells apart are separated by a distance greater than \((r_c + r_c)\). For \(N = 6912\), for the state point we have been using to compare to previous simulations \((\rho\sigma^3 = 0.8442\) and \(r_c = 2.5\sigma\)), the maximum number of sub-cells is \(N_c = 7^3\) with \(L_c = 2.88\sigma\), therefore the maximum value of \(r_c\) in 0.38\(\sigma\). In addition to the obvious increase in memory requirements, the extra elements in the list would significantly slow down the calculations of the forces. With this in mind, we added a vectored “stripping” procedure to the section of the code which constructs the neighbor table which eliminates all pairs with \(r_{ij} > r_c + r_c\). Thus, we use a vectored LC method to generate a list of potential interaction pairs and then prune this list down with a second vectorized loop, resulting in a very efficient generation of a Verlet neighbor table which can then be used to calculate the forces between particles \(ij\).

In this paper, we present two versions of a vectored LC program. In the first, we construct the neighbor table by first including all pairs which are in the same cell 1 and then the pairs with one particle in cell 1 and the other in one of the adjoining cells. In this version, the pairs \(ij\) are included in LIST1 and LIST2 in no particular order. Instead of calculating the forces between all pairs in the list and multiplying by zero all those which are beyond the interaction cutoff, as is often done, we create a temporary list each step of only those pairs with \(r_{ij} < r_c\). This can be easily done in a vectored loop and in the example given above reduces the number of elements in the list by 35%. This step is particularly important when the potential is time consuming to evaluate. The forces \(F_{ij}\) can then be evaluated in a vectored loop. In this version given in appendix A, only the summation of forces on particle \(i\), \(\sum_j F_{ij}\), remains in a short scalar loop. In spite of this one scalar loop, the program is already very fast, as seen from the results presented in table 1. However, analysis of the code showed that a significant fraction of the computer time (approximately \(\frac{1}{3} - \frac{1}{2}\)) was spent summing the forces. In appendix B, we present a modification of the basic LC code, the layered link cell (LLC) which eliminates this scalar loop. Following Rapaport [10], LIST1 and LIST2 are now organized into groups or packets. In each packet, particle \(i\) enters no more than once. This allows us to vectorize all of the forces loops. Since the size of each packet can be as large as \(N\), the vector loops are in general long and therefore quite efficient, as seen in table 1. As will become clear in the next section, the actual changes in the
code are not major and basically follow the logic which has been widely used in most MD codes and therefore should be straightforward to incorporate.

Though the LLC algorithm generally seems faster than the LC, this is not always the case. We have identified at least one case when the size of the packets is so small that the LC algorithm is actually faster. For a polymer melt [11], in which the monomers interact with a short range excluded volume interaction, the Verlet list is not very long since it is only necessary to save those pairs which are not connected along the chain. Those that are connected are saved in a separate list which only needs to be calculated once since the connectivity does not change. In this case, the LC method is about 20% faster than the LLC. We also found that for the case of polymers grafted with one end to an otherwise repulsive surface [12] the LC method was about twice as fast as the LLC method due to the fact that the density was highly non-uniform. In this case, some cells contained many monomers while others contained only a few.

In this paper, we will describe the basic structure of our MD Fortran code which we developed on a Cray XMP. The simulations were carried out on the HLRZ Cray in Jülich, a machine that has a special gather/scatter/compressed index hardware. We compiled our program with CFT, version 1.15. This version of the compiler turned out to be able to take full advantage of the special hardware, even when only standard Fortran code is programmed. One only has to be a little careful about the formulation of the loops. Hence, in our case no gain could be achieved by calls to special Cray routines like GATHER or SCATTER. For reasons of simplicity and portability, we therefore used the standard Fortran version. In fact, in at least one case, we found that an explicit call to the Cray subroutine WHENIGT, is actually about 3 times slower than the equivalent 6 line DO loop (see section 2).

The original code was developed for a scalar mainframe and was given to one of us (GSG) by A. Rahman many years ago. We then carried out a series of modifications to take advantage of the special features of the Cray. We believe that significant improvements on standard MD codes, of order 5–10, can be obtained with relatively little effort if the modifications suggested here are adopted. Further refinements on our modified code are still possible (see section 6), but by these we do not expect such a dramatic improvement.

The outline of the paper is as follows. In section 2, we describe how the LC method can be used to generate a modified Verlet neighbor list. In section 3, we discuss how the forces between particles with \( r_{ij} < r_c \) are determined. In section 4, we show that by incorporating Rapaport’s layered data structure, the section of the forces loop which was not vectorized in the algorithm presented in section 3, can be vectorized. In section 5, we present our results for a Lennard–Jones liquid and compare them to previously published results of Heyes and Smith [8] and Schoen [9]. Finally, in section 6, we outline our ideas how one could go on to introduce further slight speedups, while section 7 gives a short summary. In appendix A, we present the section of the computer code for the LC method, which generates the neighbor list and calculates the forces, while the appendix B we present the LLC code. In appendix C, we present a subroutine, LABELS, which generates a list of which cells are adjoining every other cell, taking into account the periodic boundary conditions. Finally, to give a rough overview, appendix D contains a sketched flow diagram of the overall program.

2. Computation of the neighbor list

The unvectorized version of the LC method has been fully discussed by Fincham and Heyes [6] and Smith [7]. The method [1] involves breaking the MD cell into smaller sub-cells of length \( L_c \). Since we want all particles to interact only with particles which are in the same cell or one of the 26 adjoining cells, \( L_c \geq r_c \) is required. If one wants to include a neighbor list then \( L_c \) must be larger, \( L_c \geq (r_c + r_s) \). The number of cells on a side then equals \( \lfloor L/L_c \rfloor \), where \( \lfloor \rfloor \) is the integer part. In most cases, we chose the number of cells \( N_c \) as large as possible consistent with the constraint \( L_c \geq r_c \) in order to reduce the excess number of pairs \( ij \) which must be considered. However, as shown in section 5, this is not always the case for
short range interactions, when it is more efficient
to choose a large value for \( r_s \) and thus fewer cells. 
Because the list contains more elements, it is up-
dated less frequently.

The section of our computer code which is used
to build the neighbor tables is given in the first
part of appendix A for the LC method. Because
the particles diffuse, the table has to be updated
periodically. Often, the interval between updates
of the tables is fixed at the beginning of the run
depending on the thermodynamic state of the
system, the time step and the values of \( r_s \) and \( r_c \).
However, this procedure can be automated by
monitoring the displacement of each particle since
the last update of the table. Fincham and Ralston
[13] and Thompson [14] suggested updating the
table when the sum of the magnitudes of the two
largest displacements exceeded \( r_s \). Blink and
Hoover [15] used a slightly different procedure
and updated the table when the largest displace-
ment exceeded \( 0.95 \left[ 1 - \frac{1}{(n_1 + 1)} \right] r_s \), where \( n_1 \) is
the number of steps since the last update of the
table. To compare to the work of Schoen, we
chose \( r_s = 0.3 \sigma \), provided \((L_x - r_s) > 0.3 \sigma \) and up-
dated the table every 20 steps for the data pre-
sented in table 1. For other cases studied, particu-
larly for very short range interactions, we used a
criterion similar to that suggested by Blink and
Hoover [15]. We updated the table whenever the
square of the largest displacement exceeded \( 0.60 \left[ 1 - \frac{1}{(n_1 + 1)} \right] r_s^2 \), to eliminate the need for comput-
ing the square root. We found that as the skin
thickness \( r_s \) increases, the interval between up-
dates was too long if a prefactor of \( 0.95^2 \) was
taken as in ref. [15]. This showed up by an un-
acceptable drift in the energy. With a factor of
0.60 we never ran into difficulties of that sort.

Before making the neighbor table, we first check
the periodic boundary conditions using the DO
loop 200 in appendix A. We also setup an array
XINN(I) = X0(I) which is used to monitor the net
displacement of each particle since the last update
of the table, for the cases when this is being
automated. The coordinates, which are in reduced
units so that \( -\frac{1}{2} < x_j < \frac{1}{2} \) are stored in the one-di-
rectional array X0(3N), which is equivalenced to
the two-dimensional array X0P(N,3). This allows
one to use longer DO loops of length 3N when
checking the periodic boundary conditions, as well
as in the predictor and corrector loops which are
used to carry out the numerical integration. The
two-dimensional array X0P(N,3) is used when the
x, y or z coordinates are needed, say in determin-
ing the distance between two particles or deter-
mining in which cell each particle resides. The
subroutine LABELS (see appendix C) is called in
the initialization phase of the program, so that the
indices of the cells which are adjoin cell I can be
found in LABEL(ISIZ3,27), where ISIZ3 = \( N_c \).
The periodic boundary conditions are taken into
account using the minimum image convention.
The array NBOX(I) is used to keep track of the
number of particles in cell I, while IBOX(I,J)
contains a list of which particles are in cell I. Thus
IBOX(345,3) = 427 means that the third particle
in box 345 is number 427.

Apparently, the second dimension of IBOX
must be greater than the maximum occupancy
of a cell. At a first glance, one might think that this
results in a large memory requirement. In our
case, however, the system has rather small local
density fluctuations, and hence it is sufficient to
estimate the maximum occupancy by the average
occupancy, multiplied by some “safety factor” (3
has proved to be absolutely sufficient for our
system; note that the local density fluctuations
increase as the cell size decreases). The main mem-
ory requirement does not come about by IBOX,
but by LIST1 and LIST2. Note that our imple-
mentation of IBOX is not a linked list in a strict
sense [5], but both conceptually and computa-
tionally a simpler data structure *.

* A linked list, as has been implemented by Heyes and Smith
[8] and described in the book by Allen and Tildesley [1], is
very convenient in all cases where memory is an issue (e.g. a
very huge number of particles, a small computer, large density
fluctuations), the main advantage being that the
memory used always equals the number of particles plus the
number of cells independently of the configuration. This
method consists of constructing a “chain” of particles for
each cell so that the \((n + 1)\)st particle resides at a memory
address given by the \(n\)th particle. The first particle of the
chain resides in a different array with memory address given
by the cell. The chain is finished by a zero. Obviously, this
data structure is inherently recursive. Therefore, extracting
out of these data the arrays LIST1 and LIST2 in a vectorized
manner is not as easy as using our simple IBOX.
We can now construct the two lists LIST1 and LIST2, which contain NLIST pairs of particles with \( r_{ij} < r_c + r_s \). Because many of the pairs \( ij \) which are in the same cell or in adjoining cells have \( r_{ij} > r_c + r_s \), we first make two temporary lists LIST3 and LIST4 of all pairs \( ij \) in which one of the particles is in cell I. Taking advantage of the fact that \( F_{ij} = -F_{ji} \), it is only necessary to check on 13 of the 26 cells adjoining cell I, so that the size of these temporary list scales as 13.5\( \rho L_c^3 \) (4.5\( \rho L_c^3 \) in two dimensions). At this point we then want to “strip” the number of pairs to only those with \( r_{ij} < r_c + r_s \). This can be done in several ways and we present two slightly different procedures in appendices A and B. Which is most efficient may depend on the hardware and software available. In each procedure, the distances between all MLL pairs in LIST3 and LIST4 are determined. In appendix A, we use this information to create another temporary array ITEMP1. ITEMP1(K) equals 1 if \( r_{ij} < (r_c + r_s) \) and 0 otherwise. This can either be done using the Cray library function CVMGP(0,1,R2-SANSQ), where \( R2 = r_{ij}^2 \) and \( SANSQ = (r_c + r_s)^2 \), or the statement

\[
\text{DIM(SIGN(1,(SANSQ-R2)),0)}
\]

The former is about 1% faster than the latter, while the latter is not Cray specific. Both are about twice as fast as writing the two lines

\[
\text{ITEMP1(K) = 0}
\]

\[
\text{IF(R2.LT.SANSQ)ITEMP1(K) = 1.}
\]

The ITEMP1(K) array is then used to generate a list ITEMP of those indices K for which ITEMP1(K) = 1. This could either be done with the Cray library subroutine WHENIGT(MLL, ITEMP1,1,0,ITEMP,LEFT) or with the six lines of code shown in appendix A (DO loop 1020). It turns out that on the HLRZ Cray XMP, with the CFT 1.15 compiler, the explicit call WHENIGT is actually about 3 times slower than the six lines of Fortran code used (test April 1989). With this version of the compiler, the loop vectorizes, though it did not for earlier versions of the compiler nor will it with the present version of the CFT77 compiler. We made the observation that vector compression loops of this kind vectorize with CFT only if the IF block contains variables that do not appear in other statements of the loop. Hence, we formulated our program in such a way that the vector compression loop contains only the IF block. The readers should check which kind of procedure is most appropriate for their machine.

In appendix B, we follow a similar procedure, except that we do not generate the array ITEMP1(K) but save a temporary array of the distances RSQ(K) which is then used (loop 1020, appendix B) to generate ITEMP(K). The version in appendix B is slightly faster than that given in appendix A. Had loop 1020 not been vectorized, then it would be necessary to use routine WHENIGT (or WHENFLE). The final step in making the neighbor table is to place the relevant pairs into LIST1 and LIST2 (loop 1030). This procedure is then repeated for all \( N_c \) cells. All the DO loops in this section vectorize, except the short loop 240, which tells each cell the particles it contains.

### 3. Calculation of the forces

In the latter half of appendix A, we present the relevant code for determining the forces on particle \( i \) from the remaining \( N - 1 \) particles for the LC method. Since we want to update the table as infrequently as possible, it is important to choose \( r_c \) as large as possible. However, as \( r_c \) increases, the fraction of pairs in LIST1 and LIST2, which actually contribute to the force on \( i \), \( r_c^3/(r_c + r_s)^3 \), decreases. On a scalar computer, one simply determines the distance \( r_{ij} \) between two particles and skips to the end of the DO loop if \( r_{ij}^2 > r_c^2 \). However, since this would cause the loop not to vectorize, an alternative procedure needed to be found. One simple solution, which has been used, is to simply calculate the force between all pairs, independent of the value of \( r_{ij}^2 \) and then multiply the result by zero if \( r_{ij}^2 > r_c^2 \). The loop will then vectorize and more than make up for the extra operations. One could also do as was done when the table was setup and eliminate all pairs with \( r_{ij} > r_c \) before calculating the forces. As the length of the arrays LIST1 and LIST2 may already be quite large, one may be concerned about the additional memory requirements to generate two new lists containing only those pairs which actu-
ally interact. However, there is no reason to go through all NLIST elements at one time. Instead, we found it just as efficient in CPU time and far less memory intensive to work on blocks of the NLIST elements. Since the maximum vector length on the Cray is only 64, we broke the NLIST elements into blocks which were multiples of 64, with the last block taking up the remainder. We tried 64, 640 and 6400 and found that the latter two were nearly equivalent, and faster than 64. We chose 640 to minimize the memory requirements. We also stored in temporary arrays $r_{ij}^2$, $x_i - x_j$, $y_i - y_j$ and $z_i - z_j$.

Since the elements of LIST1 and LIST2 contain the indices of the pairs $ij$, some indirect addressing is necessary to calculate the distance $r_{ij}$. Normally this would be done using the Cray library subroutine GATHER followed by a loop which involves only direct addressing. However, the latest versions of the CFT compiler now allow vectorization of loops with indirect addressing. That is, the loop

$$\text{DO 1 } K = 1, 1640$$
$$\text{XIJ} = \text{XOP(LIST1(K),1)} - \text{XOP(LIST2(K),1)}$$
$$\text{XIJ} = \text{XIJ - ANINT(XIJ)}$$

...  

1 CONTINUE

will now vectorize. In earlier versions, the first line of the loop would prohibit vectorization. It would be necessary to make an explicit call to GATHER to collect the coordinates of XOP(LIST1(K),1) into a temporary array X1P(K). We tested GATHER(N,A,B,INDEX) and the equivalent Fortran vectorizing loop

$$\text{DO 1 } I = 1, N$$
$$A(I) = B(INDEX(I))$$

1 CONTINUE

to see which was more efficient and found no difference on the HLRZ Cray. Since the structure of the code was both simpler and more easily portable without using the Cray library routines, we chose to avoid using them.

The structure of the forces calculation is actually rather simple. There are IPACK = [NLIST/640] + 1 groups, with the last group having NLIST-640(IPACK-1) elements if this number is greater than 0. Similar to the manner in which the table was pruned down to a minimum of elements, $r_{ij}$ is determined and a pointer array of 1’s and 0’s, ITEMPI is constructed. Two temporary lists LIST3 and LIST4 are then filled with only those pairs with $r_{ij} < r_c$. The forces and potential, in this case for a shifted Lennard–Jones interaction, are then determined in a vectorized loop. The forces $F_{ij}$ are stored in temporary arrays FXTMP(640), etc. The last step is to sum the forces on particles $i$ and $j$, which is done in a scalar loop. Even though this loop (1730) sums over the minimum number of elements and does only a single addition or subtraction, it takes between $\frac{1}{3}$ to $\frac{1}{2}$ of the total CPU time in the forces calculation. The longer the range of interaction, the larger the CPU time spent in this loop. However, in spite of this single scalar loop, the algorithm runs very fast, as described in detail in section 5.

Schoen [9] recently presented a program optimized for the Cray where he tried to maximize the use of vectorized loops and Cray library subroutines. His times [9] for a Lennard–Jones interaction truncated at 2.5σ are presented in table 1. He used a search over all pairs in the system to generate a standard Verlet neighbor table, with a one-dimensional LIST and a pointer array LPOINT(N). In the same spirit as just described, Schoen first removed all pairs with $r_{ij} > r_c$ before calculating $F_{ij}$. Using GATHER and SPAXPY statements extensively and taking advantage of the grouping of the elements in his LIST, he obtained a highly vectorized forces loop. However, even for $r_c = 2.5\sigma$, most of the loops are rather short except for the first few particles since $j > i$. For $\rho\sigma^3 = 0.8442$, the first particle interacts with about 60 others, but this number decreases rapidly as $i$ increases. We found that the $N(N-1)/2$ search combined with the forces calculation presented in the latter half of appendix A was actually faster than Schoen’s except for $N = 6912$, even though all the loops in his program vectorize. The difference is that the two list structure of our routine ensures that the length of the loops always remains fairly long throughout the calculation with only one scalar loop to sum up the forces. While all of the loops vectorize in Schoen’s program, the overhead in setting up short loops turns out to cost more than the time saved in vectorization.
This combination of an \( N(N-1)/2 \) search and the double list neighbor table is the fastest algorithm for \( N \) up to a few hundreds that we are aware of.

4. Layered link cell

As mentioned in section 3, the scalar loop which sums the forces on particle \( i \) took a large contribution of the total CPU time of the program. However, this is necessary because the simple procedure for constructing the neighbor table discussed in section 2 placed the pairs in the list without regard to order. Thus it is clear that to improve the performance of the code further, it is necessary to adopt a different procedure in the construction of the table, which will allow vectorization of the sum over forces. Such a procedure has recently been suggested by Rapaport [10]. In this section we present the modifications to the code described above in order to incorporate his layered data structure.

Rapaport's suggestion [10] is to split the two lists \( \text{LIST1} \) and \( \text{LIST2} \) into packets; in each packet particle \( i \) appears at most once. This means, that in each packet all values of \( \text{LIST1} \) are pairwise different; the same is true for \( \text{LIST2} \). However, a particle may appear in both \( \text{LIST1} \) and \( \text{LIST2} \) in the same packet. The sum over forces can then be done over each packet in a vectorized loop, since no entry appears more than once. This is done in the following manner. Before the particles are filled into the boxes, the array \( \text{IBOX} \) is set to zero; \( \text{MAXI} \), the current maximum number of particles in a cell is also determined. The idea is to first take all first particles out of all boxes, then all second particles out of all boxes, and so on, up to \( \text{MAXI} \). These we call the intra-layer interactions (loop 1100 in appendix B). When one reaches, say, the fifth particle in all boxes (\( J = 5 \)), one may encounter a number of non-particles (\( \text{IBOX}(I,J) = 0 \)). These, of course, must be eliminated by an IF statement in loop 1000 (which vectorizes in the same manner as discussed in section 2). In this formulation, it is essential that the loop over neighboring cells (DO 1100) is the outermost loop, since each “move” to a neighboring cell defines a new data packet. The distances between the MLL pairs in each packet are then computed and checked to see if the distance is less than \( r_+ + r_- \). The selected interactions are then added to the Verlet tables. \( \text{LPOINT}(K+1) \) is used to mark the last entry in \( \text{LIST1} \) and \( \text{LIST2} \) which is in packet \( K \). The same procedure is then carried out for the inter-layer interactions, i.e. first particle in cell with second, first with third, etc. Here, we check all neighboring cells including the central cell. By taking into account each of the \( \text{MAXI} \)(\( \text{MAXI}-1 \))/2 layer pairs only once, we assure that each particle pair occurs only once. For a given “move” to a neighboring cell, a data packet involves at most \( \text{MAXI} \) layer pairs. This is done in loop 1300, by a kind of “circle algorithm”. If one imagines the layer indices arranged in a circle like on a clock, one first takes an offset of 1 (1st layer−2nd layer, 2nd−3rd, \( \text{MAXI} \)th−1st), then an offset of 2 (1st−3rd, 2nd−4th, etc.) and so on until all the pairs are found. This way one gets a minimum number of packets and therefore a maximum vector length. For \( N = 4000 \) and \( r_+ = 2.5\sigma \), about 300 packets were used.

Once the table is constructed in this manner, the forces calculation can be done exactly as in appendix A, except that the data packets are of length \( \text{LPOINT}(K+1)\)-\( \text{LPOINT}(K) \) instead of 640. The 1720 and 1730 loops in appendix A can now be combined into one loop, which vectorizes. The compiler directive IVDEP (“ignore vector dependency”) is needed because of the indirect addressing on the left hand side. However, since within a packet each particle appears at most once in \( \text{LIST1} \) or \( \text{LIST2} \), this can be done safely. Since the two DO loops have now been merged into one, the temporary arrays \( \text{FXTMP}, \text{FYTMP} \) and \( \text{FZTMP} \) as well as the temporary lists \( \text{LIST3} \) and \( \text{LIST4} \) can now be replaced with scalar temporaries.

5. Comparison of results

To demonstrate the efficiency of the programs described above, we carried out a number of runs for a Lennard–Jones liquid with interactions truncated at \( r_+ = 2.5\sigma \) for \( k_B T = 0.72\epsilon \), \( \rho \sigma^3 = \ldots \)
0.8442 with a time step $\Delta t = 0.00462$. To compare with previous results, we chose $r_c = 0.3\sigma$ and updated the neighbor table every 20 steps, except for $N = 2048$. In this case, for the largest number of sub-cells, $N_c = 5^3$, the maximum value of $r_c$ equals $0.187\sigma$, in which case we updated the table every 10 steps. The numerical integration was carried out using a fifth order predictor–corrector loop. We also studied the same state point, with everything the same except the range of interaction $r_e = 2^{1/6}\sigma$; a purely repulsive fluid. This was done to test the interplay between $N_c$ and $r_e$. The program was run on the Cray XMP/416 at the HLRZ in Jülich, the same machine used by Schoen [9], except for a memory upgrade in the meantime, which did not affect our simulations. After equilibration, we ran up to 5000 steps for each case to determine the CPU time/step.

Table 1 and fig. 3 present our results for the LC and LLC method described in the previous sections, as well as the results of Heyes and Smith [8] and Schoen [9]. We also present results using the slower $N(N-1)/2$ search over all particles to create the neighbor table (updated every 20 steps) in combination with the forces calculation presented in section 3. The improvement over the first vectorized LC method of Heyes and Smith [8] is about a factor of 5 for our LC method and nearly 10 for the LLC (some of this comes from the difference in Cray 1S and XMP). The crossover from the Schoen algorithm [9] or our slightly improved version (using the procedure outlined in section 3) to the LC and LLC methods now occurs for $N$ approximately 500 instead of over 10,000 or more, consistent with what we observed on scalar computers. It may seem surprising but the LLC method is actually faster than the $N(N-1)/2$ method for $N \approx 500$, even though $N_c = 3$ and all pairs in the simulation cell are checked to determine which are within $r_e + r_c$. This is presumably because of the longer vector force loops compared to the relatively short vector force loops which arise from the $N(N-1)/2$ search. Comparison of the megaflop rates of our programs is shown in fig. 4. Note that while the megaflop rate is fairly constant for the LC and only increases slowly for the LLC method, it increases signifi-

![Fig. 3. Total CPU time per step as a function of the number of particles $N$ for $r_c = 2.5\sigma$; see table 1. Results for Heyes and Smith [8] (×), Schoen [9] (○), the LC method (□), LLC method (△) and the $N(N-1)/2$ method with the forces loop described in section 3 (△).](image)

![Fig. 4. Megaflop rate, in millions of floating point operations per second, for the LC method (□) and the LLC method (△) described in this work as well as for the $N(N-1)/2$ method (△).](image)

![Fig. 5. Total CPU time per step for $N = 4000$, $r_c = 2^{1/6}\sigma$ for six values of the skin thickness $r_e$ for LC (●) and LLC (○) method.](image)
The total CPU time per step for four values of the number of particles $N$ for $r_s = 2^{1/d} \sigma$, $N_c$ denotes the number of subcells which were used to give the most efficient program and $r_s$ is the skin thickness. More than one value for $N_c$ is given if they had the same CPU time/step. Results for the LC method are from code presented in appendix A, while LLC results use the code in appendix B.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N_c$</th>
<th>$r_s/\sigma$</th>
<th>CPU time/step</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LC</td>
</tr>
<tr>
<td>1372</td>
<td>$7^3$, $8^3$</td>
<td>0.557, 0.347</td>
<td>0.010</td>
</tr>
<tr>
<td>2048</td>
<td>$8^3$, $9^3$</td>
<td>0.557, 0.371</td>
<td>0.015</td>
</tr>
<tr>
<td>4000</td>
<td>$10^3$, $11^3$</td>
<td>0.557, 0.404</td>
<td>0.027</td>
</tr>
<tr>
<td>6912</td>
<td>$13^3$</td>
<td>0.428</td>
<td>0.048</td>
</tr>
</tbody>
</table>

Significantly as $N$ increases for the $N(N-1)/2$ algorithms. This is not surprising, since the CPU time necessary to construct the neighbor table increases quadratically with $N$ for the latter and linear for LC and LLC. Note that inspite of this increase in megaflop rate particularly for large $N$, the overall throughput (or CPU time per step) is about a factor of 3.3 worse for the $N(N-1)/2$ programs compared to the LLC program for $N = 6912$.

For short range interactions, as in the case of the purely repulsive Lennard–Jones interaction with $r_s = 2^{1/d} \sigma$, the best choice of $N_c$ and $r_s$ is not obvious. The larger $N_c$ is, the smaller $r_s$ must be in which case the table must be updated quite often. Similarly, the smaller $N_c$, the larger $r_s$ and the larger NLIST. In fig. 5, we present our results for the CPU time per step for $N = 4000$ for 6 values of $r_s = L_c - 2^{1/d} \sigma$, ranging from $N_c = 14^3$ down to $9^3$. The table is updated automatically, as described in section 2. The best performance was achieved for $N_c = 10^3$, $r_s = 0.557 \sigma$ (table updated every 28.6 steps in the average) and $N_c = 11^3$, $r_s = 0.404 \sigma$ (20.0 steps). In table 2, we present our results for four values of $N$. The crossover size $N'$ when the $N(N-1)/2$ search becomes more efficient decreases as $r_s$ decreases and is less than 400 for $\rho \sigma^3 = 0.8442$ and $r_s = 2^{1/d} \sigma$.

6. Further possible optimizations

In the preceding sections, we described the basic structure of our programs. We think that the most important efficiency gains possible with relatively simple modifications have been covered. Starting out from this basic structure, one can, of course, go on and implement further changes for optimization. Here we mention a few possible further modifications.

6.1. Treating the periodic boundary conditions by "ghost particles"

In our algorithm, the minimum image convention is taken into account by the statement $XIJ = XIJ - \text{ANINT}(XIJ)$ (and analogously for the $y$ and $z$ components). This operation occurs in the innermost loop where the interparticle distances are calculated (see, e.g., loop 1010, appendix A). If one has a very large system, however, most of the interparticle vectors (i.e. those in the "bulk") have already been calculated correctly by the previous statement, $\text{ANINT}(XIJ)$ having the value zero. Only a small fraction of surface vectors must really be corrected. Obviously, this fraction scales as $N^{-1/d}$, where $d$ is the dimensionality of the system. This means that this inefficiency is more important in a two-dimensional simulation than in a three-dimensional one. For very large systems, it is desirable to implement the minimum image in such a way that the number of operations required for that does not scale as $N$, but as $N^{(1-1/d)}$ (i.e. proportionality to the surface of the system). This can be achieved by the introduction of "ghost particles" located at the surroundings of the simulation cell. By the cell index method, it is easy to find out those particles that are located near the surface. Each of these particles are given image particles outside the simulation cell shifted by $+L$ or $-L$ in one or more directions. The Verlet table is then constructed in such a way that it contains both real particles and ghost particles. The indirect addressing in the distance calculating loop then automatically finds "correct" coordinates, and the statement $XIJ = XIJ - \text{ANINT}(XIJ)$ is no longer needed. The forces are at first distributed on both real and ghost particles; after that, the forces acting on the ghost particles are added to those acting on the real particles. In the differential equation solving part, only the real particles are moved according to the forces, whereas the
ghost particles are moved by exactly those dis-
placements by which the corresponding real par-
ticles have been moved. The real particles are not
put back into the simulation cell until a new
Verlet table and a new set of ghost particles is
constructed. Obviously, this algorithm involves a
pointer array IDENT(I) that contains the corre-
sponding real particle for ghost particle I. A con-
venient implementation is to treat the spatial di-
rections recursively: At first, all those real par-
ticles located near the two surfaces \( z = \text{constant} \) are
given image particles. After that, the two
surfaces \( y = \text{constant} \) are treated. Now, however,
not only real particles are given images, but also a
certain amount of those ghost particles that have
already been constructed. In the same way, finally,
the surfaces \( x = \text{constant} \) are treated. Hence, for
certain ghost particles, IDENT(I) does not point
to real particles but to other ghost particles. This
is necessary in order to fill the surrounding shell
of the simulation cell completely. The addition of
forces from ghost particles to real particles is done
also recursively, just in reversed order. For more
details and an explicit formulation, the reader is
referred to Rapaport's paper [10].

It is clear that this procedure will pay off only
for large system sizes. For a small number of
particles, the number of saved operations is too
small compared to the additional overhead. Hence,
there exists another crossover particle number \( N'' \)
able which the ghost particle method is more
efficient than usage of the ANINT function.
Moreover, it is clear that \( N'' \) increases with the
range of interaction. The reason for this is that a
larger range of interaction implies a larger number
of ghost particles and the surface corrections be-
come more important.

We did not systematically check the behavior
of the ghost particle algorithm compared to the
ANINT function. However, the results of some
tests treating both the 4000 particle system and
the 6912 particle system (short range interaction,
LLC method) indicate that the ghost particle
method saves about 10% of the computer time
needed for the ANINT function method. The
ghost particle method had a megaflop rate of
60–67 compared to 80–85 for the ANINT method.
This is easy to understand since the ANINT
method involves many arithmetic operations that
are fully vectorized whereas the ghost particle
method needs less operations but more indirect
addressing.

6.2. Changing the lattice symmetry of the sub-cells

An additional acceleration might be achieved
by using other than simple cubic lattices for the
construction of the cells. E.g. for a two-dimen-
sional system the change from the square to the
triangular lattice reduces the volume overhead
from \( 9/\pi \left( L_c/(r_c + r_s) \right)^2 \) to \( 13/(2^{2/3}\pi) \left( L_c/(r_c + \right.
\left. r_s \right) \right)^2 \approx 1.463 \left( L_c/(r_c + r_s) \right)^2 \). Even larger savings
can be gained in three dimensions. However, it
remains to be checked, for each specific symmetry,
whether this reduction of distance calculations
compensates for the more complicated sorting
procedure.

6.3. Tabulating the forces and potentials

Instead of calculating the forces and potentials
during each time step, the squared distance is
linearly transformed to an integer that is used as
an index for a table lookup. Tests that we did with
an earlier (less efficient) version of our program
on a short range system indicated that this yields a
speedup of only a few percent. Before using a
force table, one should be aware of the loss in
precision by discretization. If one has to do very
long runs with constant energy, this inaccuracy
may contribute to a systematic energy drift.

6.4. Changing the differential equation solver

In a simulation of a short range system, using a
highly optimized force calculation algorithm, the
solving of the equations of motion is no longer
completely negligible. One could try then to switch
to a less expensive algorithm, e.g. to replace a fifth
order predictor–corrector scheme by a third order
predictor–corrector or the Verlet algorithm. In
general, this gain is also paid by a loss accuracy. It
should be checked in each case which precision is
needed.
6.5. Pushing the vector lengths to their limits

In our program, all the computations in the force calculation have been split into packets. However, the logic of the method requires such a splitting only for certain operations like the force summation whereas other computations like the distance computation could be done in one large vectorized loop as well. This would be especially important on vector machines like the CYBER 205 that needs very long vectors for optimum performance. However, in trying to implement this on the HLRZ Cray, we ran into severe memory problems. On this machine, the gain would probably have been not dramatic, but such a modification should be kept in mind.

7. Conclusion and summary

We have presented algorithms for a vectorized version of molecular dynamics, confining ourselves to rather simple, generally known techniques. Whereas the LLC method proved to be most efficient for homogeneous systems, the LC method could deal better with inhomogeneities. The methods strongly rely on using indirect addressing (both gather and scatter) as well as index compression. Therefore, the algorithms are especially suited for vector machines that support those operations by special hardware or at least by an efficient software emulation. Basically, we optimized the MD along two lines. Firstly, we tried to save superfluous operations. This kind of optimization, of course, is also useful for scalar computers. Three ingredients of our programs can be mentioned in this context:

- Usage of a Verlet table in order to exploit the knowledge of the previous particle positions.
- Usage of a link cell algorithm to avoid an \( O(N^2) \) loop so that for our programs the CPU time per step is strictly proportional to the number of particles.
- Calculation of only those interactions that are within the interaction range instead of a multiplication with zero.

Secondly, we introduced data structures that are appropriate for vectorization:

- By a two-list Verlet table, it is possible to vectorize over bonds instead of over neighbors of a given particle.
- The layered data structure allows a vectorization of the force summation loop in spite of the involved random addressing. By this, we get a vector length that increases linearly with the number of particles.

Acknowledgements

We thank D.C. Rapaport for several helpful discussions on how to implement his layering algorithm, M. Robbins for a critical reading of the manuscript, M. Murat for testing the algorithm for polymers near a surface, and N. Pistoor for general discussions on optimized MD on vector computers. This work was supported in part by NATO travel grant No. 86/680, by HLRZ Jülich, and by the Deutsche Forschungsgemeinschaft.

Note added in proof

As J.S. Ho pointed out to us, the ordered data structure of LIST1 and LIST2 can also be achieved by a call to the Cray subroutine ORDERS after having used the ordinary LC method. This might be of use for very long range interactions where the number of subcells becomes very small.

References

Appendix A

C SHIFTED LENNARD-JONES INTERACTION, CUTOFF AT RC
C CUBIC INTERACTION CELL, CONSTANT VOLUME=XL*3
C
C PARAMETER
MOLX=6000, MOLY=1, MOLZ=1, MAX=15000, NCELL=39
C
DIMENSION X0(MOLS), XEP(MOLS,3), XSP(MOLS,3,3),
0 C (X0,XSP) EXPONENTIAL INBOUNDS, NR =NCELL-1, NR0=300,
0 XI(NR0)-XI00=0.5
C
C Coordiates X0 are in reduced units, from -0.5 to 0.5
C RANGE=RC**2, CUTOFF SQUARED
C RC=6.0 THICKNESS
C .SANQ=RC^2-0.5
C ADDROY= .SANQ^2-0.5 .SANQ^2-0.5
C ALL=LENGTH OF CELL, AL=4*XL
C IST=NUMBER OF CELLS ON A SIDE
C
C CALL LABE= JS15=ISIS=1 JS15=JS15=5 JS15=JS15=5
C
C OTHER INITIALIZATION OF THE SYSTEM, PREDICTOR LOOP
C
C IF FILX.SQ.C THEN NO Verlet TABLE IS COMPUTED
C T(PMX,PQ=0) GOTO 1000
C
C XI: COORDINATES OF THE PARTICLES
C THOSE PARTICLES HAVING LEFT THE BOX ARE PUT BACK
C INTO IT (PERIODIC BOUNDARY CONDITIONS)
C FOR MONITORING THE PARTICLE DISPLACEMENTS AS A CRITERION FOR
C LIST UPDATE, THE CURRENT CONFIGURATION IS SAVED ON XIN
C
C DO 200 I=1,MOLS
C X0(I)=X0(I)+XI(I)
C XEP(I)=XEP(I)+XI(I)
C 200 CONTINUE
C
C LIST1(I) IS THE CELL THAT CONTAINS PARTICLE I
C NBX(I) IS THE NUMBER OF PARTICLES IN CELL I
C IBX(I) IS THE FIRST PARTICLE IN CELL I
C DO 210 I=1,MOLS
C X0(I)+XI(I)+0.5001=X0(I)+XI(I)
C 210 CONTINUE
C
C THE FOLLOWING LOOP IS RECURSIVE AND THEREFORE SCALAR.
C DO 220 I=1,MOLS
C ,SANQ+0.5001=X0(I)+XI(I)
C 220 CONTINUE
C
C FIRST, TREAT THE INTRA-CELL INTERACTIONS
C DO 1000 1=1,N01000=1
C DO 1000 2=1,1000=1
C 1000 CONTINUE
C
C NOW, TREAT THE INTER-CELL INTERACTIONS
C DO 1100 1=1,N01000=1
C DO 1100 2=1,1000=1
C 1100 CONTINUE
C
C ALL POSSIBLE INTERACTIONS HAVE BEEN FOUND
C COMPUTE THE DISTANCES AND TAKE INTO ACCOUNT
C PERIODIC BOUNDARIES
C CHECK IF DISTANCE IS SMALLER THAN SANQ
C IF YES: I = 1 IF NO, I = 0 IF NO
C DO 1200 K=1,MOLS
C K=K+1, N=K+1, M=K+1, N=K
C 1200 CONTINUE
C
C ALTERNATIVE FORMULATION:
C IF XEP(I)+XEP(I)+XI(I)=SANQ, THEN
C XEP(I)+XEP(I)+XI(I)=SANQ, ELSE
C XEP(I)+XEP(I)+XI(I)=SANQ, END IF
C
C ALTERNATIVE FORMULATION:
C CALL GEOMETELL ITEMS(1,TOT,N0000=1)
C
C SET UP VERLET TABLES, LIST IS COUNTING VARIABLE GOING THROUGH THE LARGE TABLES
C NIL IS COUNTING VARIABLE GOING THROUGH THE TEMPORARY TABLES
C NIL IS LOW OVER TEM TERN TABLES
C
C .SANQ=0.5001
C DO 1300 I=1,MOLS
C I=I+1
C 1300 CONTINUE
C
C IF XI(I)+XI(I)+0.5001=0.0001, THEN
C XI(I)+XI(I)+0.5001=0.0001, ELSE
C XI(I)+XI(I)+0.5001=0.0001, END IF
C
C CINCLUDE DATA PACKETS OF LENGTH 640 (CHAR SPECIFIC)
C PACK IS NUMBER OF PACKETS (LAST PACKET IS SHORTER
C IF THERE IS EVEN MORE DEMAND)
G. S. Grest et al. / Molecular dynamics simulations for a large number of particles

283

C

IPACK=MLIST/640+1
C

LIST SETUP FINISHED
C

1500 CONTINUE
C

NOW THE FORCE CALCULATION STARTS
C

PERIODIC BOUNDARIES BOTH FOR X0 AND X1MM
C

DO 1600 I=1,MLIS
    X0(I)=X0(I)-XNM
    X1(I)=X1(I)-XNM
1600 CONTINUE
C

DO 1900 IS LOOP OVER THE PACKETS
C

DO 1500 XDNM=1,IPACK
    KMAX=KOFFS+1
    KMAX=640*XNM
    IF(XNM.EQ.1) GOTO 1510
C
C C US: COORDINATES OF THE PARTICLES
C LIST SETUP FINISH/SD C TEOSE PARTICLES HAVING LEFT THE BOX ARE
C BACK INTO IT (PERIODIC BC)
1500 CONTINUE C FOB MONITOE THE PARTICLE DISPLACEMENTS AND A CRITERION FOR
C LIST UPDATE, THE CURRENT CONFIGURATION IS SAVED ON XINN
C

S
C NOW THE FORCE CALCULATION STARTS DO 200 I1,MOLO3
C
BNBB(E)+SI)I(W
C PERIODIC BC
X(NI,0E)=X0(I)—XNM
C
C 2010 CONTINUE

DO 210 Il,MOLO
    XI(I)—ID(I)—NN C LIST(I,N(N)) IS THE CELL THAT CONTAINS PARTICLE IXINN(I)—XINN(I)—NN C NROX)I( IS THE NUMBER OF PARTICLES
1600 CONTINUE C IBOX(I,J) IS THE JTH PARTICLE IN CELL I
C
C DO 210 Il,MOLO
    IDONT((XOP)I,I),-0.RODO('ISIZEEl.UDU)
    DO 1900 IS LOOP OVER THE PACKETS O2IEONT~XUP)I,2(*0.5IDR)'IBIZO0l.ODQ(
4A/DINT((SOP)I,3(+I.DIEO('OSIZEEl.UDU)
    DO 0900 KIUMA/, IPACE LITI(O( IS THE CELL THAT CONTAINS PARTICLE IXINN)I)—XINN)I(—NN C NROX)I( IS THE NUMBER OF PARTICLES
1600 CONTINUE C IBOX(I,J) IS THE JTH PARTICLE IN CELL I
C
C DO 231 J—0,NBOXM2I
C COMPUTE DISTANCES AND SAVE ON TEMPORARY VECTORS DO 230 IA/,ISID3
C COMPARE WITH RANGE AB DONE OH LOOP 1010 IBOB(I,JLIL
C 230 CONTINUE

DO 1920
C THE FOLLOWING LOOP IS RECURSIVE AND THEREFORE SCALAR.
C
DO 240 T=1,MLIS
    IBOB(I,JL1=0.0
    IBOB(I,JL4=1
1920 CONTINUE
C FIND THE MAXIMUM OF ANY CELL CONTENTS
C
MAX=0
DO 250 11,ISIO3
    IF(ITEMP(I,0,1).GT.E) THON
        MAX=MAX+1
1920 CONTINUE
C COMPETE THE FORCES AND SAVE ON TEMPOSARY VECTORS
C
C C RET OF VEMLET TABLED.
C
C C IPACEO
C COMPETE THE FORCES AND SAVE ON TEMPOSARY VECTORS
C
C Cote
C COMPETE THE FORCES AND SAVE ON TEMPOSARY VECTORS
C
C C C SLEEP IMPORTANT energies
C C VIRTRA IS TOTAL VIRTUAL ENERGY
C
C DO 1720 X=1,LEFT
    LIST3(K)=LIST1(KOFFS + [ITEMP(K)]
    LIST4(K)=LIST1(KOFFS + ITEMP(K))
    S=1.0/RSQ(IITEMP(K))
    CCEL=RSQ(IITEMP(K))
    POTEN=POTEN+0.5000*S
    FTEMP(I)=FTEMP(I)+0.5000*IP(ITEMP(K))
    IP(ITEMP(K))=IP(ITEMP(K))+0.5000*IP(ITEMP(K))

1720 CONTINUE
C
C C THE FOLLOWING LOOP IS RECURSIVE AND THEREFORE SCALAR.
C
C DO 1730 X=1,LEFT
    FORCE(LIST3(K),1) = FORCE(LIST3(K),1) — FTEMP(K)
    FORCE(LIST4(K),2) = FORCE(LIST4(K),2) — FTEMP(K)
    FORCE(LIST4(K),3) = FORCE(LIST4(K),3) — FTEMP(K)

1730 CONTINUE
C
C C SHIF POTENTIAL
C
C POTEN=POTEN—LEFT+ADVPOT
C
1900 CONTINUE
C
C POTEN=POTEN+4.000
C
C MORE CODE, CORRECTOR LOOP AND ANALYSIS

Appendix B
C

C INITIALIZATION OF THE SYSTEM, PREDICTOR LOOP
C
C IF MFIX.EQ.0 THEN NO VERLET TABLE IS COMPUTED

IF(RFIX.EQ.0) GOTO 1500
C
X0: COORDINATES OF THE PARTICLES
C THESE PARTICLES HAVING LEFT THE BOX ARE PUT BACK
C INTO IT (PERIODIC BOUNDARY CONDITIONS)
C FOR MONITORING THE PARTICLE DISPLACEMENTS AS A CRITERION FOR
C LIST UPDATE, THE CURRENT CONFIGURATION IS SAVED ON XINN
C
DO 200 I=1,MLIS
    X0(I)=X0(I)-XNM
    X1(I)=X1(I)-XNM
200 CONTINUE
C
LISTO() IS THE CELL THAT CONTAINS PARTICLE I
C NB0X(I) IS THE NUMBER OF PARTICLES IN CELL I
C
DO 210 I=1,MLIS
    IF(I.INT/2K(I),0,0500*15IS2+1.000)
        NB0X(K(I))=NB0X(K(I))+1

LISTO(I)=11+70*15IS2+1*15IS2-15IS4

210 CONTINUE
C
DO 220 I=1,15IS4
    NB0X(I)=0
220 CONTINUE
C
DO 230 J=1,15IS4
    NB0X(J)=0
230 CONTINUE
C
THE FOLLOWING LOOP IS RECURSIVE AND THEREFORE SCALAR.
C
DO 240 T=1,MLIS
    NB0X(I)=NB0X(I)+1
1920 CONTINUE
C
C RETURN THE MAXIMUM OF ANY CELL CONTENTS
C
MAX=0
DO 250 11,ISIO3
    IF(ITEMP(I,0,1).GT.E) THON
        MAX=MAX+1
1920 CONTINUE
C
C CNOTE ON LARGE TABLES
C
C C FIRST THE EXTRA - LAYER INTERACTIONS
C
C DO 1100 N0V=2.14
    MLIS=0
    DO 1000 I=1,15IS4
        IF(I.NEQ.1) LABEL(I),MOV(LAY(I,0)]+1
            LEFT(MO)=I—LABEL(I),MOV(LAY(I,0)]
            LEFT(MO)=I—LABEL(I),MOV(LAY(I,0)]
            END IF
1100 CONTINUE
C
C C IF(NLGT,0).THEN
C
C C POSSIBLE INTERACTIONS HAVE BEEN FOUND
C
C C COMPETE THE DISTANCES AND TAKE INTO ACCOUNT
C PERIODIC BOUNDARIES
C
C DO 1010 K=1,MLIL
    XI=I—ANO(LIST7(K),1)
    YI=I—ANO(ITEMP(K))
    ZI=I—ANO(LIST7(K),3)
    XI=I—ANO(LIST7(K),1)
    YI=I—ANO(LIST7(K),2)
    ZI=I—ANO(LIST7(K),3)

1010 CONTINUE
C
C C CHECK IF DISTANCES ARE SMALLER THAN SANSQ
C
C LEFT=0
C
DO 1020 K=1,MLIS
    IF(NBOX(K,0,0).AND.SANSQ) THEN
        LEFT=LEFT+1
        ITEMP(LEFT)=K
        END IF
1020 CONTINUE
C
SAVE THE SELECTED INTERACTIONS ON THE LARGE VERLET TABLES
C
DO 1030 K=1,LEFT
    LIST(K)=LIST(I)+LIST3(ITEMP(K))
1030 CONTINUE

C IF LEFT.GT.0 THEN A NEW DATA PACKET HAS BEEN FOUND
C IF(LEFT.GT.0) THEN
SPACE=PACK-1
LPOINT=PACK-MLIST
END IF
C END IF
C 1100 CONTINUE
C NOW THE SAME PROCEDURE IS DONE FOR THE INTER-LAYER INTERACTIONS
C CHECK ALL THE NEIGHBORING CELLS AS WELL AS THE CENTRAL
C CELL ITSELF (MOV = 1,3,7)
C NPAIR IS THE NUMBER OF LAYER PAIRS
NPAIR IS NUMEROUS OF POTENTIAL DATA PACKETS FOR ONE VALUE OF MOV
THE FIRST PACKETS ALL HAVE NPAIR=1.
C THE LAST PACKET HAS MIN = MAX OR MAX/2 DEPENDING ON WHETHER
MAX IS EVEN OR ODD
C LAY1, LAY2 ARE THE INDICES OF A LAYER PAIR
C
C NPAIR=(MAX(1)+MAX(2))/2
SPACE=MAX(1)/2
DO 1251 MOV=1,7
NPAIR=SPACE
NPAIR=SPACE-SPACE
DO 1251 SPACE=0,SPACE-1
MLIST=SPACE/MLIST
NPAIR=SPACE/MLIST
IF(NPAIR.EQ.0) THEN
1251 CONTINUE
C DO 1250 MOV=1,7
NPAIR=SPACE
NPAIR=SPACE-SPACE
DO 1250 SPACE=0,SPACE-1
NPAIR=SPACE/MLIST
IF(NPAIR.EQ.0) THEN
1250 CONTINUE
C END IF
C DO 1260 MOV=1,7
NPAIR=SPACE
NPAIR=SPACE-SPACE
DO 1260 SPACE=0,SPACE-1
NPAIR=SPACE/MLIST
IF(NPAIR.EQ.0) THEN
1260 CONTINUE
C END IF
C COMPUTE THE FORCES. VECTORIZATION BY COMPILER DIRECTIVE.
C POTEN IS TOTAL POTENTIAL ENERGY
C VIBRON IS TOTAL VIBRATIONAL
C
C CALL VIBRON
C DO 1270 K=1,12
IF(REQ(K).LT.RAN(2)) THEN
LEFT=LEFT-
TEMP=LEFT-
END IF
C DO 1270 CONTINUE
C SHIFT POTENTIAL
C IF(REQ(K).LT.RAN(2)) THEN
1200 CONTINUE
C PUTEN-POTEN+1,000
C MORE CODE, CONVERGENCE LOOP AND ANALYSIS

Appendix C
C
C SUBROUTINE LABEL
C NCELL=SIZE**3 NUMBER SUB-COMS SYSTEM IS BROKEN INTO
C "LATTPEOM" IS X SIZE**2, "LATTPEOM" IS Y SIZE**2, "LATTPEOM" IS Z SIZE**2
C COMMON/LLL/LABEL
C (SIZE**2,LATTPEOM,LATTPEOM,LATTPEOM)
C (SIZE**2,LATTPEOM,LATTPEOM,LATTPEOM)
C (SIZE**2,LATTPEOM,LATTPEOM,LATTPEOM)
C K = -1,0,1
C (SIZE**2,LATTPEOM,LATTPEOM,LATTPEOM)
Appendix D

General sketched flow diagram for the program structure described in the text. The loop numbers refer to the DO loops in the listings of appendices A and B.