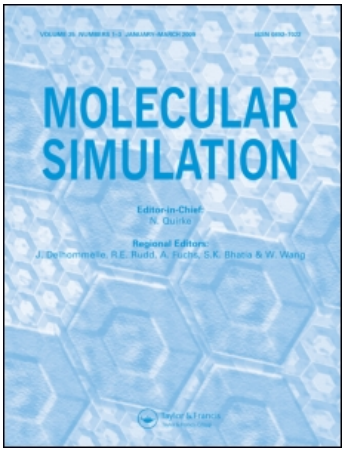

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

On the Efficiency of Vectorized Molecular Dynamics Algorithms of Order *N*

Zbigniew A. Rycerz[ab]; Patrick W. M. Jacobs[a]
[a] Department of Chemistry, The University of Western Ontario, London, Canada [b] Supercomputing Division, Hitachi Central Research Laboratory, Tokyo, Japan

To cite this Article Rycerz, Zbigniew A. and Jacobs, Patrick W. M.(1992) 'On the Efficiency of Vectorized Molecular Dynamics Algorithms of Order *N*', Molecular Simulation, 8: 3, 249 — 263
To link to this Article: DOI: 10.1080/08927029208022480
URL: http://dx.doi.org/10.1080/08927029208022480

Molecular Simulation, 1992, Vol. 8, pp. 249–263
Reprints available directly from the publisher
Photocopying permitted by license only

ON THE EFFICIENCY OF VECTORIZED MOLECULAR DYNAMICS ALGORITHMS OF ORDER N

ZBIGNIEW A. RYCERZ[+]* and PATRICK W.M. JACOBS*

Department of Chemistry, The University of Western Ontario, London, Canada N6A 5B7

(*Received October 1990, accepted March 1991*)

We discuss in this paper several factors which decide the efficiency of a vectorized Molecular Dynamics algorithm of order N, handling short-range interactions. General rules have been formulated and a fulfilment of these ought to ensure a very high speed of vector processing. The principles described make it possible to perform on presently available supercomputers the simulation of a 3-D system as large as 10^5 (or even more) particles in rather modest cpu time.

KEY WORDS: Molecular dynamics, condensed matter simulation, vector processing, supercomputing

1 INTRODUCTION

The Molecular Dynamics (MD) method is at present a well-established and powerful technique for studying fundamental processes in liquids and solids. Due to the continuously increasing power of vector and parallel computers this method continually finds new applications [1–3]. In such a situation it has become especially important to develop new efficient, vectorized and/or parallel algorithms suited to handling a very large number of particles, N, of the order of 10^5–10^6. There are several present and potential applications where the simulation of such large systems is required. Examples of this are the study of the dynamics of a low concentration of impurities (or defects) in solids, the calculation of physical quantities which involve a correlation over a large distance r, the study of hydrodynamics in terms of particle–particle interactions, and many others. Good reviews of both the computational and physical aspects of MD algorithms are given in refs. [4–6].

In any MD simulation the most expensive part, from the cpu time viewpoint, is the calculation of the interactions (forces) between the particles. Typically this part takes around 95% of the total cpu time. In general, these interactions can be divided into two parts. The first contains the short-range interactions (SRI), and the second one the coulombic type long range interactions (LRI). For a small system ($N < 500$), the second term (LRI) usually dominates the cpu time of a simulation. In a *classical*

*Associated with the Centre for Interdisciplinary Studies in Chemical Physics at the University of Western Ontario.

[+]Currently at Hitachi Central Research Laboratory, Supercomputing Division, P.O. Box 2, Kokubunji Tokyo 185, Japan (until Oct. 1991).

Particle–Particle (PP) simulation – where the interactions between all pairs are considered – the cpu time for the first part (SRI) grows with N^2, while for the LRI this dependency is, for large N, much closer to N than to N^2 (FFT $\sim N \ln N$; Ewald summation roughly $\sim N^{1.1-1.5}$). Therefore, for a system as big as $N \sim 10^5 - 10^6$, almost all the cpu time is spent in the SRI calculation and such a simulation would need several months or years of cpu time, even using the most powerful presently available supercomputers. For this reason the acceleration of the SRI calculation becomes the crucial point in the simulation of large systems. Consequently, a great deal of work has been done in this direction in recent years and several algorithms of order N (or nearly N) for handling SRI have been proposed. Usually these algorithms are oriented towards the architecture of a given computer (vector, parallel, serial) and for this reason it seems to have become rather impossible to develop any *general purpose* algorithm that would be reasonably efficient on all those different kinds of computers.

Examples of scalar algorithms are, for instance, the Neighbour List (NL) method [7], the Link Cells (LC) method [4], [8], the P^3M method [4] or the Scalar Pyramid (SP) method [9]. The main difference between these scalar algorithms consists in the method adopted for preselecting from all the particles in the system, only those ones which are the closest neighbours contained within a limited R_c (cut-off) sphere with respect to the particle under consideration (here called the *central* particle). The calculation of the forces remains essentially the same for all scalar algorithms. The algorithms given above differ significantly both in central memory requirement and in speed. Some of them allow one to simulate in a reasonable time on serial computers systems containing more than 10^5 particles.

Examples of vectorized algorithms for large N are the vector Link Cells method [10], the Method of Lights [11], the Monotonic Logical Grid method [12], hierarchical methods (e.g. [13]), the Layered Link Cells (LLC) method [14,15]) or Vector Pyramid [16] and Slab [17] methods.

In this paper we discuss several factors which determine the efficiency of a vectorized MD algorithm handling short-range interaction. All these factors and the suggested solutions have been tested in many different ways. On the basis of these tests we formulate the general structure of an MD $O(N)$ algorithm which, regardless of detail, should attain a very high efficiency on a vector computer. Most of the conclusions given here are derived from our extensive experience with the Vector Pyramid and Slab algorithms [16,17].

2 EFFICIENCY OF THE MOLECULAR DYNAMICS SIMULATION

In general it is not easy to compare vectorized MD algorithms performed on different supercomputers, due to the differences in architecture and the relative speed performance for particular operations. Such comparisons are always sophisticated. Even when making tests on the same supercomputer the results are not always clear, since some algorithms are optimized for a particular architecture and become less efficient for a different one. Therefore, all such comparisons – including those presented in this paper – should be considered with this factor in mind.

The Vector Link Cells (VLC) method is one well-known MD algorithm which achieves reasonable speeds on vector computers for large N. In contrast to some other methods, the VLC does not involve any special 'tricks' or specific routines related to

Table 1 Performance speeds in [p/sec] for the SLAB and VLC algorithms on the ETA-10P.

N	*SLAB*	*VLC*
256	23 600	
500	32 000	1 600
2 048	35 500	1 700
4 000	35 400	2 300
13 500	35 300	1 700
54 000	33 700	

a given supercomputer architecture, so it can be performed on any supercomputer. For this reason, we begin our consideration of MD efficiency by a comparison of the VLC with the SLAB algorithm [17]. In Table 1 are given the speeds in [p/sec] (number of particles per cpu sec and per time step) for these algorithms. Both programs were executed on the ETA 10 P supercomputer. As follows from the above table the SLAB is around 20 times faster than the VLC. The conditions of both simulations were not, however, exactly the same, so the above ratio of speeds is likely somewhat smaller. A more reliable speed ratio is rather $20/1.4 \sim 15$. This is, however, still more than one order of magnitude.

It is important then, to look more closely at the source of such big differences in the speed between these vector algorithms. As a matter of fact this is a more general problem, since it also concerns several other algorithms (see Tables 2 and 3).

Since the calculations of forces in an MD simulation takes $\sim 95\%$ of the total cpu time, our discussion in this and the following sections is limited to the calculation and speed-up of only this part of a MD simulation. Due to the fast decay of the short-range forces within a distance r, only a limited number of nearest neighbours (N_{nn}) make an essential contribution to the resultant force acting upon a given particle. Therefore, at each step and for each particle an MD program has to assign the neighbours contained in the R_c sphere and calculate the interactions between a central particle and its local neighbours. Taking this into account, the calculation of the mutual interactions between the particles in an $O(N)$ algorithm should be divided into two separate parts (additional arguments for this spearation will be given later in this section). For the purposes of this paper we shall, for brevity, call them:

1) CI – *calculation of interactions* (forces)(cpu time: t_{int})

2) ANN – *assignment of nearest neighbours* (cpu time: t_{ass})

The total cpu time per time step and per particle used for calculation of the forces is then:

$$t_{tot} = t_{int} + t_{ass} = N_{nn} \cdot T_{int} + N_{ex} \cdot T_{ass} \tag{1}$$

where N_{nn} is an average number of NN in the R_c sphere, N_{ex} is the number of particles examined by a program in order to assign the N_{nn}, T_{int} is the cpu time for calculation of the interaction between two particles (distance, force, potential etc.), and T_{ass} is the cpu time used for checking if a given particle belongs to the R_c sphere. In some methods N_{ex} is a function of N, as for instance in the Verlet NL method where $N_{ex} = N/2$. In a purely $O(N)$ algorithm the N_{ex} number has to be a constant.

In order to achieve high speed performance both parts have to be calculated as fast as possible; however, special attention has to be paid to the acceleration of the CI part.

This part then, would contain **exclusively** these operations that must be performed at each time step, that is the calculation of distances and force components. The main task of the second part (ANN) would then consist in a preparation of some vectors which make it possible to perform the CI part in a fully vectorized way over long vectors, and so with maximum available speed. The main reason for this approach is, that the ANN part does not need to be executed at each time step. By the use of a NN list it can be performed only every NTUPDA (typically 10 to 20) time steps. The ANN part would then include, an assignment of local neighbours in the R_c sphere, the creation of the NN list and the performing of some necessary manipulations on indices of vectors (e.g. removing data dependency). It is well known that for any MD algorithm which is of order N, the ANN part requires much more cpu time than the CI one. Therefore, the use of a NN list always increases considerably the efficiency of the simulation (cf Tables 2 and 3). The CI part consists of a very few simple operations of the type $+$, $-$, $*$ that are employed for the calculation of distances and force components, and regardless of the type of computer used it can – and must – always be calculated at the maximum speed possible.

The main advantage of vector processing is the possibility of significant acceleration of the calculations by vectorization of the most time-consuming parts of an algorithm. In order to achieve this, there are two conditions that have to be fulfilled. Firstly, the length of the vectors should be at least of order 10^2 (most vector instructions attain ~ 70–80% of maximum speed performance at a vector length of about 200 elements). Secondly, the vectors have to satisfy some requirements of vectorization (e.g. the indices of their elements should be devoid of data dependency that inhibits vectorization on some vector processors). As has been shown in paper [18], it is always possible to calculate the CI part in a fully vectorized way on any vector computer, over the long vectors ($\sim N$); including a full vectorization of the reaction forces calculation, and thus with the maximum speed available on a given computer. In any MD simulation the over-all speed t_{tot}^{-1} must be smaller than t_{int}^{-1}. Therefore, the latter one may be considered as the *limiting speed* of a simulation.

In order to compare the efficiency of different algorithms it is necessary to choose

Table 2 The C^{64} factor (in %) for vector algorithms that do not use any NN list. Computer symbols: E 10P denotes the ETA 10P and X-MP the Cray X-MP. For algorithm symbols see below.

N	*1* *PP* *X-MP*	*2* *VPYR1* *E 10P*	*3* *VLC* *E 10P*
256	11	8.0	
500	5.4	8.1	3.1
1 000	2.7	8.0	
1 372	2.0		2.7
2 048	1.3		3.5
4 000	0.7	7.5	4.7
6 912	0.4	7.5	3.9
13 500	0.2	7.4	3.4
19 625	0.14		
21 952	0.12	7.3	
32 768	0.08	7.1	

1 - *classical* PP (all pairs) method [13].
2 - Vector Pyramid method, version 1 [16].
3 - Vector Link Cells method [10].

the value of N_{nn}, and so R_c. An often used value is $R_c = 2.5\sigma$, for the L-J potential (e.g. [7,10,15]). It is, however, not too good a choice, since N_{nn} is related to a particular potential and it depends on the system's density ρ_N. It seems to us that the choice of $R_c^{64} = L\ (N = 64)/2$ is better, as it is the smallest system size giving a reasonable number of nearest neighbours in the R_c sphere (L is the side of simulation box). In this case $N_{nn}^{64} = 32\ \pi/3 \approx 33.5$ and this number of particles in the cut-off sphere remains the same regardless of the potential or the system considered. Let us denote by $1/t_{int}^*$ the maximum speed performance of the CI part available on a given computer. Therefore, the ratio:

$$C^{64} = \frac{t_{int}^*}{t_{tot}} = \left[\frac{t_{int}}{t_{int}^*} + \frac{t_{ass}}{t_{int}^*}\right]^{-1} = \frac{N_{nn}^{64} \cdot T_{int}^*}{t_{tot}} \tag{2}$$

represents the contribution of t_{int}^* to the total cpu time needed for the calculation of the forces in an MD simulation. Such a comparison seems to be useful since it refers the over-all speed of the algorithm to its own maximum MD speed characteristic for a given computer. For the fully vectorized CI part the speed of $(N_{nn}^{64} \cdot T_{int}^*)^{-1}$ is estimated as equal to about 50 000 [p/sec] on the ETA 10 P and 123 000 [p/sec] on the Cray X-MP. In Table 2 we present the value of the C^{64} factor given in % for different algorithms that do not employ any NN list. The conditions of the simulations for the data presented in Table 2 were somewhat different. The average number of nearest neighbours for column 2 was $N_{nn}^{64} = 33.5$, while for column 3, $N_{nn} = 47.1$. The values of C^{64} in column 3 were not adjusted to the N_{nn}^{64}, so their efficiency might be considered higher. As one can see the methods which do not employ any NN list spend less than 1/10 of the total cpu time for the calculation of essential interactions (cpu time t_{int}^*). In this sense all of them are inefficient. This poor efficiency is caused on one hand by the big contribution of t_{ass} to total cpu time (since the ANN is executed in each time step), and on the other hand, by the fact that none of these algorithms calculate the CI part in a fully vectorized way over long vectors, and so with maximum speed (in fact, an acceleration of the CI part becomes useless for these programs, since the over-all speed is dominated by the ANN part). Taking this into account there are essentially three ways to speed-up these algorithms:

1) make the calculation of interactions fully vectorizable (and so reduce T_{int} to T_{int}^*),
2) improve the assignment of NN,
   a) reduce the number of examined particles (N_{ex}), and/or
   b) reduce the T_{ass},
3) apply NN list and update it every NTUPDA time steps ($t_{ass}^u = t_{ass}/\text{NTUPDA}$).

The easiest solution is that in point 3, namely to use a NN list, which always gives considerable acceleration (typically 2–3 times [16]). The NN list requires a lot of additional memory but for not too large R_c this memory is still reasonable even for $N \sim 10^5$ or more. It should be noticed that the NN list typically contains only half of the nearest neighbours, since the algorithm calculates direct- and reaction- forces. In other words, the average number of elements in the NN list is equal to $N_{nn}/2$ per particle. The use of the classical Verlet approach for the creation of the NN list becomes useless for large systems because N_{cx} grows considerably with N ($N_{cx} = N/2$). For this reason it is necessary to employ an approach which involves a value of N_{ex} as small as possible, and, in addition, constant. There are several such algorithms available and we will discuss briefly some of them in section 4.1. After improving the

Table 3 The C^{64} factor (in %) for different vector algorithms that employ NN list(s). Computer symbols: CB 205 denotes the Cyber 205; E 10P and X-MP as in Table 2. For algorithm symbols see below.

N	*1* *SLAB* *E 10P*	*2* *VPYR3* *E 10P*	*3* *VPYR2* *E 10P*	*4* *VNLB* *CB 205*	*5* *VNL* *X-MP*	*6* *ML* *CB 205*	*7* *LLC* *X-MP*
256	47.2	45.0	36.2	29.8	20.4		
500	64.0	58.2	44.6	27.5	19.8		
1 000	71.0	62.2	45.6			14 (29)	
2 048	71.0	62.0	45.4	13.1	16.8		29 (47)
4 000	70.8	61.8	45.4				35 (57)
6 912				5.1[e]	11.3[e]		35 (57)
13 500	70.6	55.2	37.4	2.8[e]	7.6[e]		
21 952	70.4	56.2	34.2	1.7[e]	5.6[e]		
54 000	67.4	52.4	32.6	0.7[e]	2.8[e]		

1 - Slab method [17].
2 - Vector Pyramid method, version 3 [16].
3 - Vector Pyramid method, version 2 [16].
4 - Vector Neighbour List (Bit lists) method [19].
5 - Vector Neighbour List method [20].
6 - Method of Lights [11].
7 - Layered Link Cells method [15].

ANN part according to 2a and 2b and after applying the NN list (point 3) the t^{u}_{ass} cpu time usually becomes much smaller than t_{int}. Finally then, the CI speed decides the over-all speed of simulation. At this stage, the acceleration of t_{int} becomes the crucial point. Among others it requires the COLUMNWISE-ROWWISE (C–R) transformation ([18] and also section 4 here) and removal of index data dependency in the calculation of the reaction forces in ROWWISE mode [18]. The removal of data dependency in the reaction forces calculation results in a substantial speed-up of the CI part (around 4 times [18]) and in our experience this cannot be performed without keeping an integer NN list, since it requires some operations on the forces indices. The last statement constitutes the second reason for the necessity of using a NN list. In the next paragraphs we discuss the problems mentioned here in more detail.

Summarizing the above considerations one may say that the use of a NN list makes it possible to accelerate an MD $O(N)$ algorithm by around 10 times, where the factor $f_{a1} \approx 4$ comes from speed-up of the CI part [18] and the factor $f_{a2} \approx 2.5$–3.0 from updating the NN list every NTUPDA time step. Because the C^{64} value must be, by definition, smaller than 100%, this means that the algorithms presented in Table 2 cannot exceed the value of $C^{64}/(f_{a1} \cdot f_{a2})$, or $\sim$ 10% of the maximum speed performance.

In order to confirm these considerations we present in Table 3 the C^{64} factor for algorithms that do employ some NN list(s).

The conditions of the simulations for data in column 4 and 5 were: $R_c = 2.3\sigma$ and $\rho\sigma^3 = 0.6475$, which gives $N_{nn} = 33.0$ particles in the R_c sphere, and thus very close to the $N^{64}_{nn} = 33.5$ (columns 1,2,3). As stated above the improvement of the ANN part gives, for NTUPDA – 10–20, an acceleration of $f_{a2} \approx 2.5$–3.0. Therefore, the limiting efficiency (C^{64}) for this class of programs is 10%. $f_{a2} \approx 25$–30%, which agrees with the data in columns 4,5 and 6. The VNLB and VNL methods apply the classical Verlet approach for creation of the NN list, and so they lose their primary efficiency for larger N. The values with a superscript were estimated from data in refs. [19] and [20]. The authors in paper [11] give two cpu times, namely for updated (the first value in column 6, Table 3) and non-updated list (the value in brackets). The LLC program

(column 7) was tested for the L-J potential with $R_c = 2.5\sigma$ and $\rho\sigma^3 = 0.844$, which corresponds to the average number of nearest neighbours $N_{nn} = 55.3$ and so by a factor of 1.65 bigger than N_{nn}^{64}. First values in column 7 are given for original data from ref. [15], while the values in brackets are multiplied by 1.65. In fact the latter values should be used with some care, since an assumption of linearity between the speed and N_{nn} is not strictly true for the LC method. The SLAB algorithm, which has all the advantages discussed above, attains 70% of maximum efficiency, which is almost 10 times more than VPYR1 (Table 2). The VPYR3 algorithm also calculates the CI part with the maximum speed; however, its ANN part is less advanced (slower) than that for the SLAB. The next algorithm VPYR2 differs from VPYR3 only in that data dependency was not removed in the calculation of reaction forces. Consequently then, these forces are calculated in scalar mode and the CI part has a speed of about half the maximum rate. The calculation of the reaction forces in a scalar mode (VPYR2) saves significantly memory [16] but also slows down a simulation.

3 CALCULATION OF INTERACTIONS (FORCES)

In this section we discuss the reduction in cpu time through T_{int}, that is the first point quoted in section 2. A careful treatment of this part of a simulation is very important and might result in significant acceleration of an algorithm.

The calculation of the interactions consists of several very simple operations of the type of $+$, $-$, $*$. In a scalar notation it proceeds as follows:

- calculate components of distance (R)

$$\Delta x_{ij} = x_i - x_j, \qquad \Delta y_{ij} = \ldots, \qquad \Delta z_{ij} = \ldots \tag{3}$$

- periodic boundary conditions (PBC)

$$\Delta x_{ij} = \Delta x_{ij} - \text{int}\,(\Delta x_{ij}) - \text{int}\,(\Delta x_{ij}), \qquad \Delta y_{ij} = \ldots, \qquad \Delta z_{ij} = \ldots \tag{4}$$

$$R_{ij}^2 = \Delta x_{ij} \cdot \Delta x_{ij} + \Delta y_{ij} \cdot \Delta y_{ij} + \Delta z_{ij} \cdot \Delta z_{ij} \tag{5}$$

- cut-off criterion

$$\text{IF } (R_{ij}^2 > R_c^2) \quad \text{GO TO } \ldots \quad \text{(switch off)} \tag{6}$$

$$R_{ij} = \text{SQRT}\,(R_{ij}^2) \tag{7}$$

- calculate force and potential energy

$$F_{ij} = f(R_{ij}), \qquad U_{ij} = u(R_{ij}) \tag{8}$$

- calculate force components

$$F_{ij}^x = \frac{\Delta x_{ij}}{R_{ij}} F_{ij}, \qquad F_{ij}^y = \ldots, \qquad F_{ij}^z = \ldots \tag{9}$$

- accumulation of direct forces

$$F_i^x = F_i^x + F_{ij}^x, \qquad F_i^y = \ldots, \qquad F_i^z = \ldots \tag{10}$$

- accumulation of reaction forces

$$F_j^x = F_j^x - F_{ij}^x, \qquad F_j^y = \ldots, \qquad F_j^z = \ldots \tag{11}$$

Let us assume that all engaged vectors (positions, indices, forces...) are suficiently long and all vectors satisfy the requirements for vectorization. The points 3,4,5,9 and 10 can be easily vectorized and they would attain very high speed performance, since they consist only of simple +,−,* operations. Some attention should be devoted to the remaining points. The point (8) (F_{ij}, U_{ij}) contains usually several +,−,* operations and frequently some more time-consuming ones like: *exp, power, sqrt,* etc. Therefore, it should be replaced by look-up tables. The number of elements in the look-up table should be sufficiently large ($\sim 10^4$ or more) in order to avoid a time consuming interpolation. The point (7) should be excluded (or not) depending on the relative speed of the SQRT operation on a given computer. Consequently the look-up tables ought to be created linearly in either R or R^2. In addition, the forces in the look-up table should be stored in the units "F_{ij}/R_{ij}", which simplifies the operation (9) to the form: $F_{ij}^x = \Delta x_{ij} \cdot F_{ij}$. Depending on the computer used, a vectorization of operation (6) may slow down considerably the calculations (IF-THEN-ELSE construction). In that case a preferable solution is to exclude this operation and perform the calculations for all the neighbours from a NN list. In this case the look-up table should contain some surplus elements such that $F_{ij} \equiv 0$ for $R_{ij} > R_c$. Step (11) contains the indirectly addressed indices j (reaction forces) which prevents its vectorization on several contemporary vector processing machines (e.g. ETA 10, Cyber 205, Cray 1, Cray 2, older Cray X-MP, IBM 3090-VF, Hitac S-810) [18]. A potential solution of this problem strongly depends on the hardware and more precisely on the communication between the central memory and vector pipe. On the *memory access* computers like ETA 10 or Cyber 205 this problem cannot be solved, as a vector pipe is loaded directly from the central memory. On the other hand on *register* machines like a Cray it can be solved by hardware. For example, later models of the Cray X-MP line have a specific hardware gather/scatter feature which allows the vector registers to be loaded efficiently from central memory in such a way that each register contains data which are devoid of dependency. Anyway it is really important to calculate the reaction-forces loop in a vector mode, since this accelerates its performance by a factor of 5–6 (or more) and consequently results in the speed up of the entire MD simulation by a factor of around 2 [18]. As has been shown in [18] operation (11) can be effectively vectorized – for an MD program with NN list – on any vector processing computer by removing dependency for j indices. Removal of data dependency contributes only a few percent in additional cpu time (e.g. only 4% on the ETA 10P [18]) to the total MD cpu time. This means that in practice an MD simulation can be performed on any vector computer with efficiency similar to that on the Cray X-MP, which can vectorize such loops automatically.

It should be noticed that the calculation of PBC (point 4) can be made easier by changing the size of the simulation box (in dimensionless units) from that usually used $(-1, +1)$ to $(-1/2, +1/2)$, as suggested by Adams [21], but we found that this accelerates the MD simulation by only 1–2%, so that the improvement achieved is not significant.

The above calculations ought to be performed in a ROWWISE mode (sec. 4) and so over vectors of length N, which are much bigger than the required ~ 200 elements. If the ANN creates a NN list in which all central particles have the same (fixed) number of local neighbours then all vectors have had the same length equal to N. This can be achieved by the proper construction of the ANN part or by cutting off the top of the NN list after (or during) its creation (cf. VPRY2, 3 in [16]). In a more general solution we need to consider a local fluctuation of density in the R_c sphere, which

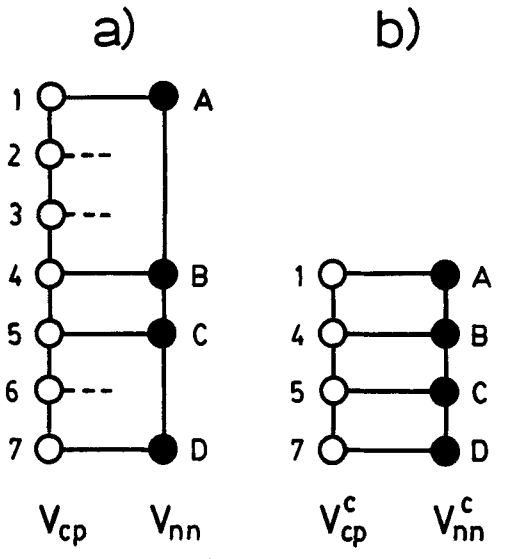

Figure 1 Representation of input vectors in the ROWWISE NN list for a row higher than NNN_{min}. V_{cp} denotes a vector of central particles and V_{nn} – a nearest neighbours vector. a) original input vectors, b) compressed vectors (*compact ladder*).

seems to be a more physical approach. In such a situation every central particle has some number of local neighbours that changes from NNN_{min} to NNN_{max}, with an average value of NNN_{av}. In other words not every central particle has a neighbour placed in a row higher than NNN_{min}, which means that between NNN_{min} and NNN_{max} corresponding vectors will be shorter than N. This situation is shown schematically in Figure 1a. By using bit masks an operation involving the two original vectors (Figure 1a) is still vectorizable. However, a more efficient solution is to move out the preparation of bit masks from the CI into the ANN part, since the cpu time there is divided by NTUPDA. Both the input vectors (V_{nn} and V_{cp}) should then be compressed to the form presented in Figure 1b (V^c_{nn} and V^c_{cp} denote compressed vectors). Such vectors are devoid of non-active elements although their lengths are smaller than N. Tests indicate that the second solution (Figure 1b) is always faster than the first one (Figure 1a) due to the characteristic structure of the *scattered* part of a real NN list in MD simulations.

Concluding the above considerations one may say that the best solution (from an efficiency viewpoint) seems to be to perform the calculation of the interactions between central particles and all their local neighbours, according to the scheme of a full *compact ladder* (i.e. with all rungs; Figure 1b). It is then hard to imagine any further substantial acceleration of the CI part, since its calculation is reduced to a few elementary operations like *subtraction* ($\Delta x_{ij} = x_i - x_j$), *multiplication* ($R^2_{ij} = \Delta x_{ij} * \Delta x_{ij} \ldots$) etc. Therefore, we consider that in any really fast MD simulation the calculation of interactions ought to be realized in this or a very similar manner. This solution has an additional important advantage, namely that such a structure for the vector operations (*compact ladder* with all parallel rungs) is very suitable for a multi-processor supercomputer. Each such ladder can be simply (and arbitrarily)

divided into smaller parts (sub-tasks) which can be calculated independently, and so at the same time, on different processors.

As mentioned earlier on the ETA 10P supercomputer (single processor, 21 ns clock, about 23 Megaflops) the CI procedure attains the speed of 50 000 [p/sec] at R_c^{64}, which means 2 cpu sec per time step for system of $N = 10^5$. Due to the reasons given above we expect that this speed should be at least one order of magnitude bigger on a really fast multi-processor supercomputer.

4 ASSIGNING OF NEAREST NEIGHBOURS AND REMOVING DATA DEPENDENCY

In this paragraph we discuss in more detail some problems related to the acceleration of the ANN part of the MD algorithm.

Let us come back to the NN list in which the number of local neighbours varies from NNN_{min} to NNN_{max}, with an average value for the entire list of NNN_{av}. Because the *reaction* forces are calculated in the CI part, so the NN list contains only those neighbours which are in one hemisphere of radius R_c. Apparently each *central particle* has no repeated neighbours, so the calculation of distances and direct forces can be easily vectorized in the COLUMNWISE mode (over local neighbours). However, in this mode the average length of vectors is very short $NNN_{av} = N_{nn}^{64}/2 \approx 17$) and in consequence, vectorization does not accelerate the calculation. Therefore, as mentioned before, in order to speed-up the CI it is necessary to change the order of the NN list from COLUMNWISE (in which the NN list is normally created) into ROWWISE (C–R transformation). Then the calculations will be performed over the vectors of length N instead of NNN_{av}.

The requirements of the CI part impose some duties on the ANN part. The main task of the ANN part consists in the preparation of such vectors for the CI part which will satisfy the next two conditions:

1) none of the vectors involved in any vector operation has a DO loop data dependency,
2) all vector operations in CI involving two vectors are performed according to the *compact ladder* (no *cross* or *empty* elements).

Point 1 concerns only those computers without the capability to vectorize indirectly-indexed DO loops with repeated indices. An ideal solution – from the point of view of cpu time – would be to create the NN list initially in a ROWWISE mode and it ought to be done in such a way that each row of neighbours would be devoid of DO loop data dependency. If such a procedure were fast enough it would be a really good solution. However, these requirements seem to us difficult to fulfill. As a matter of fact an index dependency always exists in a NN list (vertically – over local neighbours, or horizontally – over the rows of neighbours). This means physically that every particle is a neighbour to more than one central particle. Usually such a list is primarily created as a list of local neighbours in succession for each central particle (COLUMNWISE mode), so that it does not have vertical dependency, but over the rows of the NN list (the long vectors) such dependency does exist. As stated earlier, removing DO-loop dependency over long vectors is very important in the case of vector processing and gives a considerable acceleration of an MD algorithm [18]. Because we have not found as yet, any reasonably fast method that allows one to create a NN list without index

data dependency in ROWWISE mode, we apply another solution. This solution consists of three steps.

a) create the usual COLUMNWISE NN list,

b) perform the COLUMNWISE-ROWWISE transformation,

c) remove data dependency in the ROWWISE NN list.

For simplification, we will refer to the last (transformed) NN list (c) as LNRNNT. From a physical point of view it is unimportant, whether a given neighbour from NN list will be considered as, for instance, the i-th or the k-th neighbour of a central particle, so that the location of neighbours over the rows of the NN list is meaningless. Because of this, there are, in general, two ways in which data dependency in the *basic* ROWWISE LNRNNT list can be removed:

A) replace each original (horizontal) row of the NN list by a sequence of ordered sub-lists,

B) distribute the local neighbours of each central particle over the rows (in vertical direction) in such a way that there will be no repeated neighbours in any row.

Both these solutions have some positive and negative features. The first approach consists in the replacement of each horizontal row in the ROWWISE LNRNNT list, which goes through all N, by a sequence of ordered sub-lists. Each sub-list contains only nonrepeated indices and thus can be vectorized. However, such a solution requires additional cpu time and additional computer memory. During this operation the neighbours change their original locations in the NN list, so that it is necessary to remember their old locations. This requires the use of another list, which we call in refs. [16,17] the *list of ordered central particles* (LORDER). Finally then in this approach it is necessary to store two large integer NN lists (LNRNNT and LORDER). For R_c^{64} each such list needs around $25\text{–}30\,N$ words of memory. Therefore, the memory for the entire program is increased to $\sim 80\,N$ words.

It should be emphasized that the solution B is always possible, since the size of the NN list is $\geqslant \mathrm{NNN}_{\max}$, which is the maximum number of local neighbours for any central particle. The main advantage of that solution is the possibility of reducing considerably the required computer memory by the elimination of the LORDER array. This approach can be realized in several different ways and we have examined some of them. In all our attempts, solution B appeared to be significantly slower than the first one. However, we believe that the solution B can be improved by the adoption of a more refined approach to achieve the re-distribution of the neighbours over the rows of NN list. The method used from point A is described in detail in paper [18] and has been applied in [16,17]. An important feature of such an approach to the vectorization of indirectly indexed DO loops is that both solutions (A,B) are independent of the supercomputer used. They do not involve any specific predefined or software procedures, they are reasonably fast and in consequence they can be applied on any vector processing computer.

As mentioned earlier the ANN consists of three steps (points a,b,c) namely: assigning of nearest neighbours (cpu time: t_{ass}^0), C–R transformation (t_{tra}) and removing data dependency (t_{rem}). The C–R transformation is a simple and very fast operation taking usually less than 1% of the cpu time so Equation (1) can be rewritten as:

$$t_{\mathrm{tot}} = t_{\mathrm{int}} + (t_{\mathrm{ass}}^0 + t_{\mathrm{rem}})/\mathrm{NTUPDA} \tag{12}$$

For example, for the SLAB algorithm [17] on the ETA 10P these cpu times per one element of the NN list and for NTUPDA = 20 are as follows: $t^0_{ass} = 8.4\,\mu$sec/20 = 0.42 μsec (23% of the contribution to t_{tot}), $t_{rem} = 1.4\,\mu$sec/20 = 0.07 μsec (4%), $t_{int} = 1.3\,\mu$sec (73%), giving: $t_{tot} = 1.8\,\mu$sec (which is equivalent to a speed of more than 35.10^3 [p/sec] for R_c^{64}). On the other hand leaving the calculation of reaction forces in scalar mode, as for example in VPYR2 [16] decreases the overall speed of the simulation by a factor ~ 2 [18].

4.1 Creation of nearest neighbour list

The problems discussed so far concerned two points that play an important role in the efficiency of an MD simulation. Namely, the fast execution of the CI part and the application of the NN list. As follows from the considerations presented, both of them attain their limits from the point of view of possible acceleration. This means that on the one hand the CI part cannot be executed substantially faster, as all operations are performed over long vectors in a fully vectorized way, and on the other hand the value of NTUPDA cannot be too large. The third possibility of an algorithm acceleration is to reduce the t_{ass} cpu time (Equation 1). The speed-up from this possibility, in contrast to the previous ones, does not have such a clear speed-up limit since it can be realized in many different ways. As a matter of fact most of the MD algorithms published hitherto differ mainly in their approach to the assignment of nearest neighbours.

As follows from Equation (1) the acceleration of the ANN part can be achieved either by decreasing N_{ex} and/or T_{ass}. In order to analyze this problem it is convenient to rewrite Equation (1) as:

$$t_{tot} = N_{nn}(T_{int} + f_{en} \cdot T_{ass}) \tag{13}$$

where $f_{en} = N_{ex}/N_{nn}$. The factor f_{en} should be as close as possible to 1/2, since searching for NN might be limited to one hemisphere of radius R_c only. In the classical Verlet method this ratio is equal to $N/(2 \cdot N_{nn})$, and thus for $N \sim 10^5$ and R_c^{64}, it is about 1 500 and so too large by three orders of magnitude. A simple and good working solution is to divide the simulation box into smaller sub-cells, as for example in the LC method. In this method the side of a cell is $R_{cell} \approx R_c$ and typically $3 \times 3 \times 3/2$ adjacent cells are examined to establish N_{nn}, which gives a value of f_{en} order of 10. Effectively this number is smaller, as neighbours in adjacent sub-cells are common NN for all particles contained in a central sub-cell; however, additional cpu time is needd for sorting the neighbours. An important feature of the LC method is that the calculations are performed over very short vectors (length order of N_{nn}) which strongly restricts its speed on vector computers. Significant acceleration of the LC method can be attained by the layered decomposition of cells (LLC, [14]), which accelerates the assignment of nearest neighbours and makes it possible to calculate interactions over relatively long vectors (number of cells) instead of short ones (N_{nn}). The LLC approach when joined with the use of a NN list increases the speed of a simulation by a factor of 10 over the LC method [14,15]. Further improvement in the cell method can be achieved by using non-cubic cells or much smaller cubic cells that better circumscribe a hemisphere of radius R_c. In the latter case the small sub-cells constitute a pyramid which circumscribes a hemisphere. If sub-cells are sufficiently small then f_{en} is close to the minimum value (1/2). On the other hand, however, the sub-cells cannot be too small because then a program will spend a lot of cpu time checking non occupied sub-cells. We have analyzed the problem of the optimal size

of a sub-cell and we have found as an optimal solution, a model which we refer to as the 2 PC model (2 particles in cell). In this model there are at most two particles that may simultaneously occupy the same sub-cell and for this reason the model uses two associated 3–D sub-lists of particles. The 2 PC model works very efficiently on both, serial [9] and vector [16,17] computers.

The creation of a NN list by searching the local environment for each particle in the R_c hemisphere (e.g. in a pyramid) is a good solution on a serial computer. However, on a vector computer such an approach involves very short vectors which results in small (if any) vector/scalar acceleration for T_{ass} cpu time.

For vector purposes it is necessary then to replace the pyramid by another construction, which will involve operations on much longer vectors. Such a solution has been applied, for instance, in the SLAB algorithm [17]. The slab spreads over the entire MD box and therefore for large systems, its size is much bigger than the size of a pyramid. This allows one to reduce the cpu time T_{ass} additionally by a factor close to 2.

There are several other methods of assigning NN that can be used in the creation of NN lists. In general they choose the relevant neighbours by preselecting the individual numbers of particles or by sorting their coordinates. Some of them were mentioned in section 1, but we do not discuss their differences here, partly because that is not the purpose of this paper and partly due to their different objectives and applications. What we want to emphasize, is that – regardless of the detailed solution – any efficient method for large N has to search for NN among a very limited number of particles. Therefore, N_{ex} should be as close as possible to $N_{nn}/2$ or in other words, the searching area should be close to the R_c hemisphere. In this context we consider the cell model to be very useful.

5 SUMMARY AND CONCLUSIONS

We have discussed the factors that decide the high-speed performance of MD algorithms of order N on vector computers. It has been shown that there is some limiting speed for the algorithms without a NN list, and that this appears to be rather low. A significant acceleration of $O(N)$ MD algorithms may be attained only at the price of computer memory. In this context we emphasize the role of the NN list and the removal of data dependency from the calculation of mutual interactions. As has been shown, in order to attain a very high speed performance on a vector processing computer, an $O(N)$ MD algorithm has to obey the following three rules:

1) calculate interactions over long vectors ($\sim N$) in a fully vectorized way,
2) apply a NN list,
3) apply an $O(N)$ method (e.g. a cell method) for assignment of a NN list.

According to the arguments presented the $O(N)$ MD algorithms can be formally divided into four groups as shown in Table 4. Due to the variety of possible solutions the classification given in this table ought to be viewed with some tolerance.

The C^{64} in Table 4 is the over-all algorithm efficiency (Equation 2), while the efficiency of the CI part given in the last column is defined as the ratio t^*_{int}/t_{int} (Equations 1,2). Computer memory for group I may depend on the manner of removing data dependency (see points A and B in section 4).

A similar analysis, in terms of the C^{64} factor can be presented for scalar algorithms of order N. In this case there are only two groups of programs, namely those with and

Table 4 The efficiency and computer memory of different MD vector algorithms of order N. CMBAS denotes the basic computer memory, which is order of $20 \cdot N$. For more details see text.

Group	*Method*	*Total central memory [words]*	C^{64} *[%]*	*CI efficiency [%]*
I	use of NN list with all advantages (ROWWISE mode, fully vectorized CI)	$\sim 2 \cdot N_{nn} \cdot N$ + CMBAS or $\sim N_{nn} \cdot N$ + CMBAS	~ 70 (+)	100
II	as I, but with reaction forces left in scalar mode	$\sim N_{nn} \cdot N$ + CMBAS	up to ~ 50	~ 50–60
III	basic Verlet type NN list	$\sim N_{nn} \cdot N$ + CMBAS	up to ~ 30	up to ~ 50 (typically ~ 30)
IV	without NN list (COLUMNWISE mode)	CMBAS	up to ~ 10	typically ~ 20–30

without a NN list and they correspond to groups I and IV from Table 4. The first one achieves the efficiency up to about 80% (and so is close to group I here), while the second one achieves up to 30%. This means that an algorithm without NN list may attain, on a serial computer, up to around 4 times better efficiency than on a vector computer. In other words sequential processing is better suited for programs without a NN list than vector processing. The main reason for this is, that usually the vector/scalar speed-up of the CI part is much higher than the speed-up of the ANN part. This results in a higher contribution of ANN to the over-all speed of vector allgorithms and in consequence the C^{64} factor decreases.

Acknowledgement

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] B.J. Alder, in *Proc. of International School of Physics 'Enrico Fermi'*, course 97, on 'Molecular Dynamics Simulation of Statistical-Mechanical Systems', eds. G. Cicotti and W.G. Hoover, North-Holland, Amsterdam (1986).

[2] CCP5 Annual Meeting: *Industrial Applications of Molecular Simulation*, Birbeck College London, 6–8 January 1988. See *Molecular Simulation* **2** 4–6, **3** 1–3 (1989).

[3] D.C. Rapaport and E. Clementi, "Eddy Formation in Obstructed Fluid Flow: A Molecular Dynamics Study," *Phys. Rev. Lett.*, **57**, 695 (1986).

[4] R.W. Hockney and J.W. Eastwood, "Computer Simulation Using Particles," McGraw-Hill, New York (1981); Adam-Hilger, Bristol (1988).

[5] F.F. Abraham, "Computational Statistical Mechanics. Methodology, Applications and Supercomputing," *Advan. Phys.*, **35**, 1 (1986).

[6] D. Fincham, "Parallel Computers and Molecular Simulation", *Mol. Simul.*, **1**, 1 (1987).

[7] D. Fincham and B.J. Ralston, "Molecular Dynamics Simulation Using the Cray-1 Vector Processing Computer," *Comput. Phys. Commun.*, **23**, 127 (1981).

[8] J.N. Cape and L.V. Woodcock, "Glass transition in a soft-sphere model," *J. Chem. Phys.*, **72**, 976 (1980).

[9] Z.A. Rycerz and P.W.M. Jacobs, "Molecular Dynamics Simulation Program of Order N for Condensed Matter. I. MDPYRS1: Scalar Pyramid, Short-range Interactions," *Comput. Phys. Commun.*, **60**, 53 (1990).

[10] D.M. Heyes and W. Smith, "Cray Vectorized Link Cell Code," *IQCSP*. **26**, 68 (1987); D.M. Heyes,

"Correction to Cray Vectorized Link Cell Code," *IQCSP*, **28,** 63 (1988). The *IQCSP* abbreviation denotes – *Information Quarterly for Computer Simulation of Condensed Phases*, CCP5, SERC, Daresbury Laboratory, Daresbury, England.

[11] F. Sullivan, P.D. Mountain and J. O'Conell, "Molecular Dynamics on Vector Computers," *J. Comput. Phys.*, **61,** 138 (1985).

[12] J. Boris, "A Vectorized Near Neighbours Algorithm of Order N Using a Monotonic Logical Grid," *J. Comput. Phys.*, **66,** 1 (1986).

[13] L. Hernquist, "Hierarchical N-Body Methods", *Comput. Phys. Commun.*, **48,** 107 (1988).

[14] D.C. Rapaport, "Large-Scale Molecular Dynamics Simulation Using Vector and Parallel Computers," *Comput. Phys. Reports*, **9,** 1 (1988).

[15] G.S. Grest, B. Dunweg and K. Kremer, "Vectorized Link Cell Fortran Code for Molecular Dynamics Simulation for a Large Number of Particles," *Comput. Phys. Commun.*, **55,** 269 (1989).

[16] Z.A. Rycerz and P.W.M. Jacobs, "Vectorized Program of Order N for Molecular Dynamics Simulation of Condensed Matter. I. MDPYRV1: Vector Pyramid, Short-range Interactions," *Comput. Phys. Commun.*, **62,** 125 (1991).

[17] Z.A. Rycerz and P.W.M. Jacobs, "Vectorized Program of Order N for Molecular Dynamics Simulation of Condensed Matter. II. MDSLAB1: Slab, Short-range Interactions," *Comput. Phys. Commun.*, **62,** 145 (1991).

[18] Z.A. Rycerz, "Acceleration of Molecular Dynamics Simulation of Order N with Neighbour List," *Comput. Phys. Commun.*, **60,** 297 (1990).

[19] R. Vogelsang, M. Schoen and C. Hoheisel, "Vectorization of Molecular Dynamics Fortran Programs Using the Cyber 205 Vector Processing Computer," *Comput. Phys. Commun.*, **30,** 235 (1983).

[20] M. Schoen, "Structure of a Simple Molecular Dynamics Fortran Program Optimized for Cray Vector Processing Computers," *Comput. Phys. Commun.*, **52,** 175 (1989).

[21] D. Adams, "More on Neighbourhood Tables" *IQCSP*, **3,** 32 (1981).