JOURNAL OF COMPUTATIONAL PHYSICS 106, 101–107 (1993)

Efficient Parallel Implementation of Molecular Dynamics on a Toroidal Network. Part I. Parallelizing Strategy

K. ESSELINK, B. SMIT, AND P. A. J. HILBERS

Koninklijke/Shell-Laboratorium, Amsterdam, Shell Research B.V., Badhuisweg 3, 1031 CM Amsterdam, The Netherlands

Received December 5, 1990; revised July 6, 1992

Molecular dynamics simulations require supercomputers. A specific class of supercomputers is that of parallel computers. We derive an implementation of molecular dynamics on a toroidal network of processors. First, we argue that for a fast algorithm the simulation universe has to be divided into regular cells, and we determine the best shape of these cells. For a parallel implementation, we choose to distribute cells rather than particles and we show how to assign the cells to processors, given certain restrictions on universe and network. The assignment is proven to be optimal with respect to communication cost. We go on to explain our implementation. Finally, we compare the timing results with those for computations performed on a Cray single-processor machine. The physical results obtained with the implementation are discussed elsewhere.

1. INTRODUCTION

Computer simulations have come to play an important role in statistical mechanics (see [2] for a historic overview). Since exact, quasi-experimental data can be generated on well-defined model systems, simulations can sometimes replace experiments as a means to test theories, and may enable new phenomena to be observed as well.

The capacity of the available computers places restrictions on the total number of particles and on the total simulation time. Therefore, much effort has been spent in designing algorithms to reduce the computer capacity required for a simulation [1]. This, however, concerns almost exclusively sequential and vector machines, and many optimal codes for molecular simulations on these machines are available today.

For parallel computers the developments are still in their infancy. Only a few articles have been published on the systematic design of parallel algorithms for molecular dynamics simulations [3, 5, 9–11]. The main idea of parallel computing is to distribute the work among the several (small) processors instead of using one (large) computer. In general, these small processors will be much cheaper than a large vector computer and easier to maintain also. This allows the study of large size problems at a fraction of the cost of a supercomputer, as is illustrated in [13]. Good performance can only be obtained if the load is distributed over the processors evenly and the amount of communication between processors is small.

The details of a parallel algorithm in general depend on the topology of the processor network. Therefore, it is important to be able to estimate a priori which mappings of an algorithm on a processor network will yield optimal performance. In this work we demonstrate how to approach this problem for molecular dynamics simulations of homogeneous systems. We will argue that it is more efficient to distribute cells than particles. We derive rules (Section 4) which, depending on the system to be simulated and the processor network, give the optimal division of the universe into cells, the optimal shape of these cells, and the optimal assignment of cells to the processors. As a result, it is no longer necessary to use only "heuristic" arguments in choosing a mapping. One of the conclusions is that for processor networks of size $32 * 32$ the common z-mapping is no longer optimal with respect to communication cost.

We illustrate this approach by a simulation of a Lennard–Jones fluid on a toroidal processor network. For this particular network we explain our implementation in somewhat more detail. Furthermore, we compare the timing results of several simulations on a Transputer network with those performed on a Cray single-processor machine.

This article is the first of a series of two articles describing our implementation. The second article ([4], hereinafter II) pays specific attention to potentials commonly used, and the problem one encounters with multi-particle potentials if these particles are in different cells and processors.

2. MOLECULAR DYNAMICS ALGORITHM

In this section we review the algorithm we have used for molecular dynamics simulations. We have restricted ourselves to cubic simulation boxes with periodic boundary conditions and to homogeneous systems with short-range interactions; i.e., the potential can be truncated at a distance

0021-9991/93 $5.00

which is significantly smaller than half the box length of the system being simulated. (For a more detailed description of potentials, please see II.) The algorithm is in essence a combination of the well-known neighborlist or Verlet-list method [14] and the linked-list method [6]. This combination was introduced by Auerbach *et al.* [2].

When we truncate the potential at a fixed cutoff radius R_c, a particle interacts only with those particles which are in a sphere with radius R_c around this particle. In order to prevent checking each time step whether a particle is in the sphere surrounding another particle, which is an order N^2 operation (where N is the total number of particles), a neighbor list is constructed. In this method the cutoff sphere of the potential is surrounded by a sphere with a larger radius R_L. For each particle a list is made of all the other particles in this larger sphere. After that this list is used for the evaluation of the force: only the particles that are in the neighbor list are considered (order N). In order to account for particles entering the sphere which were initially not in the neighbor list, this list needs to be updated from time to time (order N^2). So although this algorithm reduces the calculation time significantly, the computational work still has a component of order N^2.

An alternative approach for the evaluation of the force is the "linked-list" method, in which the simulation box is divided into a number of cells. If the size and the shape of the cells are chosen such that

- particles only interact with particles in the same cell or neighboring cells,
- the assignment of particles to cells can be determined easily,
- there is a maximum number of particles in a cell, independent of N,

then instead of looping over all particles it suffices to loop over nearby cells, resulting in an algorithm of order N. (Note that a maximum number of particles in a cell, independent of the total number of particles N, can be guaranteed for homogeneous systems and taking cells of fixed size. The *number* of cells *may* depend on N.)

Due to the above criteria a cell has to be a semi-regular space-filling polyhedron with a judiciously chosen edge length. Let the search space of a cell be those cells in which particles reside that have to be investigated for interaction. Before analyzing the volume of the search space for several polyhedra we note that

- without loss of generality we may assume that the cutoff length equals one,
- due to Newton's third law it suffices to consider only half the volume of the search space, and
- the search space for one particle equals half the volume of a sphere with radius one, hence $\frac{2}{3}\pi \approx 2.1$.

TABLE I

Comparison of the Polyhedra

Polyhedron	Edge length	Volume of cell	Number of neighbors	Volume of search space
Cube	1	1	26	13.5
Rhombic dodecahedron	1	$\frac{16}{9}\sqrt{3}$	18	29.3
Octahedron	$\sqrt{\frac{3}{2}}$	$\frac{2}{3}\sqrt{2}$	34	16.5
Truncated octahedron	1	$8\sqrt{2}$	14	84.9

Table I shows for each of the polyhedra its name, the necessary edge length (under the constraint that the neighboring cells be adjacent), the volume of a cell, the number of neighboring cells, and half the volume of the search space.

From Table I we conclude that although the cube is "less spherical" than the other polyhedra, it turns out to be the best. It has both a modest volume and a modest number of neighbors, whereas the other polyhedra either have a small volume and a large number of neighbors or a large volume and a small number of neighbors. Because of this analysis and the simple geometry of the cube we have opted for cubic cells. We note that it is also possible to let particles interact with other particles in the same cell, the neighboring cells, and the cells at distance two. A similar comparison table can be made for these options.

It is interesting to compare the linked-list method with the neighbor-list method. Using cubic cells in the linked-list method, half the volume of the search space is $13.5R_L^3$, which is approximately six times larger than the volume which contributes to the force ($\frac{2}{3}\pi R_c^3$). For the neighbor-list method the search space is $\frac{2}{3}\pi R_L^3$, but this algorithm is of order N^2 due to the construction of the list. This observation motivated Auerbach *et al.* [2] to combine these two algorithms, using the linked-list method to construct a neighbor list. This removes the order N^2 for the construction of the neighbor list without increasing the volume, which needs to be considered in the evaluation of the force by a factor of 6.

3. PARALLEL COMPUTING

In molecular dynamics simulations the calculation of the movements of the particles looks the same for all particles. From a computational point of view, these properties make molecular dynamics particularly interesting for parallel computing. Furthermore, a parallel machine has the advantage that it can be expanded if more computing power is needed. If, for instance, the number of particles and the number of processors are increased by a constant factor, the overall execution time should not be affected (Eq. (5)). This

offers opportunities for those research fields where large simulation universes are necessary.

This article deals with the construction of an implementation on a toroidal network of processors; i.e., all processors have connections to a north, east, south, and west neighbor, and the total network is a wrap-around mesh. In II we discuss the question whether this is indeed an efficient topology.

The main loop of the molecular dynamics calculation consists of two phases. In the first phase the forces on each particle are determined, and in the second phase the displacements of the particles are determined from the forces, together, perhaps, with some macroscopic properties of the system. Especially the latter phase is trivial to parallelize. The first one is usually more difficult, since the processors need to cooperate (exchange information) in order to compute the potentials.

Two techniques are most commonly used for exploiting parallelism, viz. particle parallelism and geometric parallelism.

3.1. *Particle Parallelism*

The first technique for exploiting parallelism assigns particles to processors [5, 7]. Continually, each processor calculates forces and the new positions for its own particles. The initial distribution of particles remains unchanged during the simulation and can be chosen such that the workload is evenly distributed. For multi-particle potentials this technique is particularly convenient if all particles involved in one potential are assigned to the same processor. The computation of, say, a torsion potential (II) usually involves a lot of manipulation with minor differences for the four particles. It is easy to take advantage of this if the particles belong to the same processor (although this cannot always be guaranteed, for example, for large chains and small processor networks). Implementation of the Lennard–Jones potential, however, poses a problem. It is quite possible that two particles encounter each other closely while initially they were very distant. Therefore, despite the short-range nature of the potential, it is necessary for each processor to communicate with all others to determine whether any two particles have become close. It is difficult to analyze the communication behavior of this technique a priori. It will in general also not be easy to achieve good scale-up properties if the processor network size increases. The disadvantage is quite serious, since usually the bulk of the computation consists of the evaluation of Lennard–Jones (or comparable) potentials. (See, e.g., [12].)

3.2. *Geometric Parallelism*

Geometric parallelism does not suffer from this particular disadvantage. It assigns space, not particles, to processors [10, 11]. During the computation, a processor calculates the trajectories of all particles it finds in its space. Because of the movement of the particles, some particles may enter a processor's space, others may leave. For this reason, processors continually need to redistribute the particles to make sure that each one has the right subset. The short-range nature of the Lennard–Jones potential however, can be turned into a real advantage. Since the interaction does not extend over distances larger than R_c, it is not necessary to exchange information over long distances, because distance in the simulation universe is related to distance in the processor network. The analysis of the previous section showed that cubes form an efficient subdivision of space. In the following section we will derive an efficient assignment of these cubes to the processors.

We should note that another approach to parallel programming consists of the "processor farm" idea: there is a master process dividing the total work among a number of slaves. This technique is only helpful for relatively large pieces of work. Implementations for MD usually yield bad scale-up [7].

4. ASSIGNMENT OF CELLS

One of the initial assumptions is the homogeneous distribution of particles, which means that the amount of work that has to be done is roughly the same for each cell. Many of the molecular dynamics problems deal with these systems. "Homogeneous" means that if one takes a time average, the number of particles at each processor will be equal. (However, at each individual time step we allow fluctuations of the order of 10%.) Therefore we will assign an equal number of cells to all processors. The problem is to find out which cells are to be assigned to which processor. Suppose we have a square torus of $Q * Q$ processors and a cubic universe of $n * n * n$ cells, in which the size of the cells is at least the cutoff R_c. The most "natural" mapping (which is also the most used) is the orthogonal projection of the universe onto the torus of processors. However, this z-mapping is not always the best. By considering a specific class of mappings we show for which sizes of systems the z-mapping is indeed efficient and for which sizes it is not.

The choice of the mapping determines the communications cost, which depends on both

- the number of cells that have to be communicated between two processors and
- the distance between two communicating processors.

Moreover, we want this communications cost to be equal and minimal for all processors. First we investigate the information needed for one cell.

In Fig. 1 we see a central cell "C" surrounded by 13 other cells, the "interaction set." With the particle information from the interaction set, the central cell can calculate all the

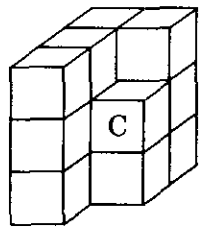

FIG. 1. One cell ("C") with 13 neighbors.

necessary interactions. Interactions between the central cell and the other 13 neighbors (not drawn in the figure) will be taken into account by these 13 other cells. (The 13 cells in the interaction set can be chosen freely from the 26 candidates as long as no two opposite cells are in the set.) While mapping the cells onto the network, we have to take into account the communications between the central cell and its interaction set. It is important to minimize the distance between these cells in the network.

But there is another aspect; see Fig. 2.

In this figure, $2 * 2 * 2$ central cells and their interaction sets are shown. Note that only 35 cells are needed to form the interaction sets of the 8 central cells. 35 is considerably less than $8 * 13$ and yet it is sufficient, since the 8 cells can share information from the surrounding cells. This also means that it is important to assign contiguous cells (a cluster), not just randomly chosen cells, to processors. In the analysis below we have made use of a further permissible reduction, using only half of the 56 cells surrounding the 8 central cells; it should be noted though, that, in consequence, the interaction sets of these 8 cells are no longer of equal size.

We will derive formulae describing the communication costs of a class of mappings of clusters. These clusters have equal x- and y-sizes, and a variable z-size; see, for example, Fig. 3.

Suppose that the x- and y-sizes of a cluster is m cells. We determine the z-size as follows. In our $Q * Q$ torus network, we make sub-squares of size $q * q$. (Suppose q divides Q and $q > 1$.) See Fig. 4.

Within these $q * q$ squares, we make a ring of processors.

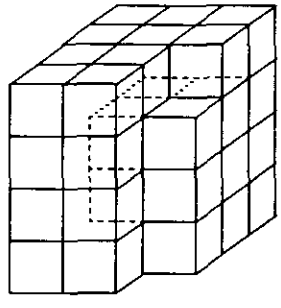

FIG. 2. Eight central cells with 35 neighbors.

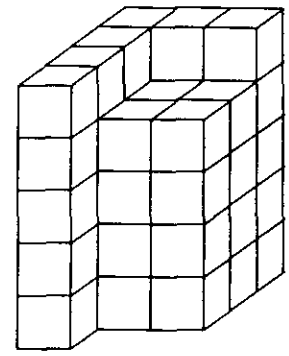

FIG. 3. A cluster with x- and y-size 2, z-size 3, and 34 neighbors.

We can guarantee for each of the $q * q$ processors in the ring that its two neighbors in the ring are at a distance one in the $q * q$ mesh, apart from one connection which has a distance of two at most. For reasons of simplicity we take this cost to be one. On these $q * q$ processors we map a column of the universe of size $m * m * n$. This means that each processor gets a sub-universe of $m * m * (n/q^2)$ cells. (We define $l = (n/q^2)$.) The universe consists of $(n/m)^2$ such columns and we have $(Q/q)^2$ sub-squares to map these columns on. We will calculate the cost of communicating the neighbors to the processor with the sub-universe.

First, the $m * l$ cells to the left of the sub-universe have to be communicated. The distance is q processors, so the cost is $q * m * l$. Then we communicate the $m * l$ cells at the back, again at cost q. The $m * m$ cells at the bottom can be communicated with cost 1 (in the ring). Then we still have to obtain the columns at the front left and back left, and a few at the bottom and top. The distance of these cells and the sub-universe seems to be $2 * q$, but since these cells have already been sent to closer neighbors in the processor network, we obtain the information more cheaply. To be precise, we can obtain $2 * l$ remaining cells at cost q and $4 * m + 4$ at cost 1. Writing $m = nq/Q$, the total communication cost becomes

$$\frac{2n^2}{Q} + \frac{2n}{q} + \left(\frac{nq}{Q}\right)^2 + \frac{4nq}{Q} + 4. \tag{1}$$

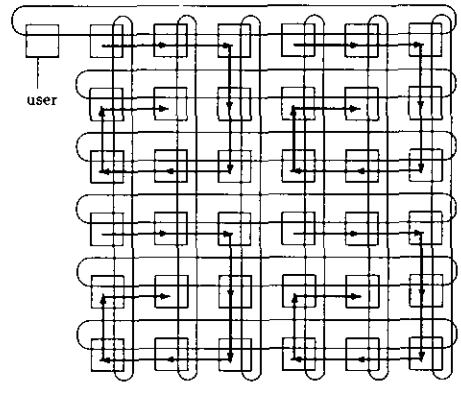

FIG. 4. A processor torus: $Q = 6$, $q = 3$.

Things are a little different when $q = 1$. In that case, the top and bottom layer of the universe reside in the same processor, so there are

$$\frac{2n^2}{Q} + 2n \tag{2}$$

neighbors, all at distance 1 in the network. Because of the overlap, the straight $q = 1$ mapping has a "natural" advantage over $q \geqslant 2$.

(There is also the special case $q = Q$. In that case, each processor has $n * n * n/Q^2$ cells and communicating the neighbors costs

$$(n+2)^2. \tag{3}$$

For cubic universes however, this is not interesting.)

With these formulae, we can calculate which q is the best, given a certain universe and processor network. Also, it is possible to investigate how a certain mapping behaves with respect to scaling of the network or the problem size. Let us first look at the (unrealistic) example of $n = Q$, in which the size of the network scales with the size of the universe. The expressions 1 and 2 then yield

$$n = \frac{q(q+2)^2}{2(q-1)}, \tag{4}$$

which gives for each q the universe size n for which $q = 1$ is equally good. For larger n, the $q \neq 1$ mapping is cheaper.

It is, however, more realistic to use a fixed Q, since usually the size of the network is not scalable. In Table II we see for different values of Q and q how the two expressions relate and what the relative gain of using expression (1) compared to expression (2) is. The n's chosen are a multiple of Q to assure a proper workload balance.

TABLE II

Comparison of Expressions (1) and (2)

Q	q	Range	n	Ex. (1)	Ex. (2)	% gain
10	2	—	10	46	40	−15
20	2	8–52	20	76	80	5
			40	236	240	2
	4	—	20	86	80	−8
32	2	6–186	32	112	128	12
			64	356	384	7
	4	5–59	32	116	128	9
			64	388	384	−1
100	2	5–2295	100	316	400	21
	4	3–834	100	286	400	28
	5	3–557	100	289	400	28
	6	3–393	100	297	400	26

We see that for $Q = 10$ it is cheaper to use straight z-mapping; for $Q = 20$ it is a little cheaper to use cluster mapping, although not for large n. If the algorithm has to scale well with n, it will be better to use the z-mapping ($q = 1$). With the technique described here, it is also possible to investigate universes of the form $a * b * c$ (e.g., beams), to be mapped on networks of size $P * Q$.

5. IMPLEMENTATION

In our laboratory, two Transputer networks are available. Both are toroidally connected; one has 36 processors and the other 400. Because of this hardware and the analysis of the previous sections, we chose to implement the $q = 1$ mapping. Furthermore, each processor has a rectangular mesh of adjacent columns of the universe. The sizes of the cells in the three dimensions are not related to the cutoff, which means that it may not be sufficient to look at the 13 neighbors of each cell only. The program actually computes the "stretch," i.e., the number of neighbors it needs to cover distance R_c. This approach also makes it possible to have only a few particles per processor; in some cases cells of size R_c^3 are already too large [12]. Our implementation is written in Transputer Pascal [8].

5.1. *Procedures*

In Fig. 5 we see the main program. The procedure **initialize** reads and distributes the parameters of the simulation. It either reads information about the particles from a file or generates a new configuration on an fcc lattice.

The procedure **integrity** checks the positions of all particles in the universe. Since each processor has its own area (columns) of the universe, particles can cross processor boundaries, and they will change processors. Upon termination of **integrity**, each particle resides in the right processor.

```
    initialize
;   while not_finished and GOON do
      begin integrity
      ;       if continue then
                begin make_verlet
                ;       for ITloop:=1 to NVERL
                          do move_it
                end
              else GOON:=false
      end
;   if GOON then
      begin integrity
      ;       store_results
      end
```

FIG. 5. The main program.

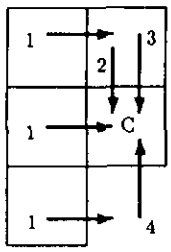

FIG. 6. One Processor ("C") with four neighbors.

The function **continue** checks the error status of all the processors. Since it is not possible for the individual processors to write error messages whenever an error occurs, errors have to be logged until they are read by **continue**.

The procedure **make_verlet** builds the Verlet neighbor lists for each particle. This is done by acquiring copies of the particle information of as many neighbors as is necessary to have all particles within R_c (see II). In the simplest case, there are four neighbors to communicate with, and with the following communication scheme, there is no danger of deadlock (see Fig. 6). First, information is sent to the east (and simultaneously received from the west, flow 1), then to the south (flow 2). Next, information from the north-west is needed, but this has just been sent to the east, so it can be found at the north processor (flow 3). Likewise, information from the south-west can be found at the south processor (flow 4). A generalization of this principle can be found in II. Once the required information is on each processor, the neighbor lists can be made. Note that it is important to perform **integrity** before **make_verlet**.

Next is a loop performing **NVERL** times the procedure **move_it**. **Move_it** first sends back the forces calculated in **make_verlet** while building the neighbor lists. The forces are communicated along the same route as the positions, but backwards. This has to be done, since the force between any two particles is only calculated once, which means that it may have to be communicated if the two particles reside in different processors. After the force updates, the new positions are calculated using a leap-frog scheme. If necessary, **move_it** rescales the temperature. Finally (if the iteration is not the last of the loop) the new positions are communicated and the new forces are computed.

Note that we have two synchronization points during one iteration: sending the positions and sending the forces. This, however, does not reduce the velocity (as suggested in [11]). The reason is that if a processor has to wait for forces, it will have to wait for positions too. Likewise, if it does not have to wait for forces, it will not be waiting for positions either. Synchronization is only harmful if waiting is a random process. Here the number of particles being dealt with by a processor is the deciding factor.

Finally, the procedure **store_results** writes position and velocity of the particles to file, together with data on some macroscopic properties of the system.

5.2. *Data Structure*

Each processor owns adjacent columns of the universe. These columns consist of cells, which have to contain pointers to the linked list of particles for each cell. Since a particle can be in more than one Verlet neighbor list, we implemented this list by a one-dimensional array in which a processor keeps track of all neighboring particles for every particle it has. The list is updated every **NVERL** iterations.

The specific choices are not new and can be found in the literature [1, 6, 11].

6. RESULTS AND CONCLUSIONS

In this section we present some timing results of simulations performed on several machines. Results on physical properties obtained with our implementation are discussed in [13].

In Table III we see timing results of simulations done on a Cray X-MP (single processor), and on 36, 100, and 400 T800 Transputers. Typical values for the variables are temperature = 1, time step = 0.005, **NVERL** = 10, Lennard–Jones cutoff = 2.5σ. We should note that the FORTRAN implementation for the Cray is fully vectorized. From these tables we make the following observations:

- As stated in Section 2, the algorithm is indeed of order N. Let ρ denote the density and P the number of processors

TABLE III

Comparison of Execution Times (Seconds per Iteration)

RHO	# particles	Cray X-MP	36 T800	Ratio
0.5	2916	0.11	0.48	4.4
0.7	4000	0.19	0.79	4.2
0.9	5324	0.32	1.47	4.6
1.0	6912	0.48	1.84	3.8
RHO	# particles	Cray X-MP	100 T800	Ratio
0.5	13500	0.53	0.78	1.47
0.7	19652	1.05	1.46	1.39
0.9	23328	1.33	2.34	1.75
1.0	27436	1.75	2.80	1.60
0.7	39304	2.05	2.85	1.39
RHO	# particles	Cray X-MP	400 T800	Ratio
0.7	19652	1.05	0.41	2.56
0.7	39304	2.05	0.86	2.38

of the Transputer network; then a formula describing the number of seconds per iteration reads:

$$0.064 + 0.011 * \rho * N/P \quad (5)$$

with mean absolute error 0.066 and standard deviation 0.09.

• For these molecular dynamics simulations a Cray X-MP single-processor is on average 4.25 as fast as the network of 36 Transputers and 1.41 as fast as the 100-Transputer network. The 400-Transputer network is over 2.5 times as fast as the Cray. Hence for this kind of calculation the price–performance ratio strongly favors the parallel processor networks.

• An additional advantage of the z-mapping is that the height of the simulation box can be varied without adapting the implementation. When, for instance, oil/water interfaces are to be investigated, then two interfaces are established due to the periodic boundary conditions. By varying the z-size of the box the effects of the distance between the two interfaces can be studied. The last row of the middle table in Table III shows the timing results for a simulation box which is twice as high as the other entry with density 0.7. As might be expected there is hardly any influence on the execution timings.

The timing results show that the z-mapping is indeed efficient. It can be deduced from Table II in Section 4, that, in the case of a toroidal network of 32×32 processors, a different kind of mapping would be even more efficient; the simplicity of the z-mapping, however, may compensate for this inefficiency. Whether or not mapping can be improved further does not alter an important conclusion that can be drawn, namely, that molecular dynamics simulations can benefit greatly from parallel computing, both in time and in cost.

ACKNOWLEDGMENT

We thank our colleague John Somers for insightful discussions about mappings and communication costs.

REFERENCES

1. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids* (Oxford Sci., London, 1987).
2. D. J. Auerbach, W. Paul, A. F. Bakker, C. Lutz, W. E. Rudge, and F. F. Abraham, *J. Phys. Chem.* **91**, 4881 (1987).
3. F. Brugè, V. Martorana, and S. L. Fornili, in *CONPAR88*, edited by C. R. Jesshope and K. D. Reinartz (Cambridge Univ. Press, Cambridge, UK, 1989).
4. K. Esselink and P. A. J. Hilbers, *J. Comput. Phys.* **105** (1) (1993).
5. D. Fincham. *Mol. Simul.* **1**, 1 (1987).
6. R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles* (McGraw-Hill, New York, 1981).
7. J. Li, D. J. Brass, D. J. Ward, and B. Robson, A study of parallel molecular dynamics algorithms for N-body simulations on a transputer system, *Parallel Comput.* **14**, 211 (1990).
8. J. J. Lukkien, Comput. Sci. Note 8912, University of Groningen, Dept. Computing Science, P.O. Box 800, 9700 AV., Groningen, The Netherlands, *1989*.
9. V. Martorana, M. Migliore, and S. L. Fornili, *OUG 7: Parallel Programming of Transputer Based Machines, Amsterdam, 1988*, edited by T. Muntean (I.O.S.).
10. H. G. Petersen and J. W. Perram, *Mol. Phys.* **67** (4), 849 (1989).
11. M. R. S. Pinches, D. J. Tildesley, and W. Smith, *Mol. Simul.* **6**, 51 (1991).
12. H. Sato, Y. Tanaka, H. Iwama, S. Kawakika, M. Saito, K. Morikami, T. Yao, and S. Tsutsumi, in *Scalable High Performance Computing Conference SHPCC'92*, (IEEE Comput. Soc., IEEE Comput. Soc. Press, 1992), pp. 113–120.
13. B. Smit, P. A. J. Hilbers, K. Esselink, L. A. M. Rupert, N. M. van Os, and A. G. Schlijper, *Nature* **348** (6302), 624 (1990).
14. L. Verlet, *Phys. Rev.* **159** (1), 98 (1967).