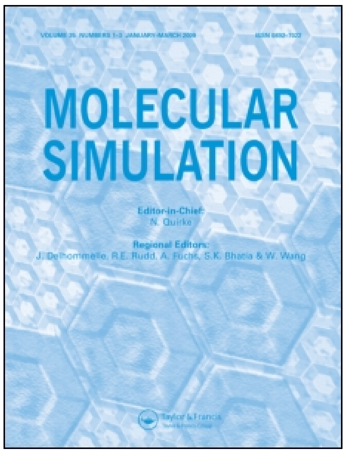


Molecular Simulation

Large Scale Molecular Dynamics on Parallel Computers using the Link-cell Algorithm

M. R. S. Pinches[a]; D. J. Tildesley[a]; W. Smith[b]
[a] Chemistry Department, University of Southampton, Southampton, United Kingdom [b] Theory and Computer Science Division, SErc Daresbury Laboratory, Warrington, United Kingdom

To cite this Article Pinches, M. R. S. , Tildesley, D. J. and Smith, W.(1991) 'Large Scale Molecular Dynamics on Parallel Computers using the Link-cell Algorithm', Molecular Simulation, 6: 1, 51 — 87
To link to this Article: DOI: 10.1080/08927029108022139
URL: http://dx.doi.org/10.1080/08927029108022139

Molecular Simulation, 1991, Vol. 6, pp. 51–87
Reprints available directly from the publisher
Photocopying permitted by license only

LARGE SCALE MOLECULAR DYNAMICS ON PARALLEL COMPUTERS USING THE LINK-CELL ALGORITHM

M.R.S. PINCHES AND D.J. TILDESLEY

Chemistry Department, University of Southampton, Highfield, Southampton, Hants SO1 5NH, United Kingdom

W. SMITH

Theory and Computer Science Division, SErc Daresbury Laboratory, Warrington WA4 4AD, United Kingdom

(*Received April 1990, accepted May 1990*)

Parallel computers offer a more cost-effective route to high performance computing than traditional single processor machines. Software for such machines is still in its infancy and they are often much more difficult to program than sequential machines. In addition many of the algorithms which are successful with sequential and vector processors are no longer appropriate. Both the force calculation and integration steps of molecular dynamics are parallel in nature and for that reason we have developed a parallel algorithm based on the link cell technique. This method is particularly efficient when the range of intermolecular potential is much smaller than the dimensions of the simulation box. The details of the algorithm are presented for systems of atoms in two and three dimensions using a number of decompositions into sub-units. The algorithm has been tested on an Intel iPSC/2 and a Cray X-MP/416 and the results are presented for simulations of up to $2 \cdot 10^6$ atoms.

KEY WORDS: Large scale molecular dynamics, parallel computers, link-cell algorithm.

1. INTRODUCTION

Since the first computer simulations of liquids were performed by Metropolis *et al.* [1] the power of computers has increased by as much as seven orders of magnitude. Early computers executed instructions sequentially, on only one piece of data at a time. The cost of memory was high and the speed at which a processor could fetch data from memory, or communicate, was slow relative to its processing power. This meant that there was little to be gained from building a multiprocessor machine. Over the last decade advances in silicon technology have effectively reversed this situation and the coupling of many processors together to share the work load has become an attractive option. A computer with several small processors is now often much cheaper than a uniprocessor machine which can offer the same processing power. The application of parallel machines to molecular simulation has been recently reviewed [2], however a few comments need to be made about the different architectures available. These can be distinguished by considering the way the memory is used, either shared by all the processors or distributed between them, and how the processors execute their instructions. If all the processors execute the same instruction on a different piece of data they are described as single instruction multiple data (SIMD) machines. If each processor is free to execute instructions independently of any other the machine is described as multiple instruction multiple data (MIMD).

An example of a shared memory machine is the Cray X-MP, which has up to four processors each with vector processing capability. It is a MIMD computer with inter-processor communications effected via the shared memory. If more than one processor is allowed to modify the same memory location, or variable, they will, in general, attempt this in a random order, one may even interfere with another during such an operation. Such a situation is known as memory conflict and the unpredictable results that it produces must be avoided. The problems that this causes means that most machines of this type have relatively few processors and memory speeds at the limits of present technology. Avoiding memory conflict is the major consideration when developing an algorithm which will parallelise on this type of machine. The vector processors which are available on the Cray X-MP can be viewed as SIMD. They can, with certain constraints, apply the same operation to vectors using a method known as pipelining. This means that the vector processor behaves like a production line where the machine code instructions that make up the required operation are executing simultaneously on each of the elements of the vector. After each instruction is performed the data is pushed along the pipeline to the next instruction. Vectorisation of a program often offers performance improvements with little modification to code and should always be considered before parallelisation. These machines are normally used to process more than one job at the same time rather than a single program truly running in parallel. The application of parallelism on such machines is coarse grained, meaning that parallelisation is best performed at the program and subroutine level coding.

The Intel iPSC/2 is an example of a distributed memory MIMD machine on which each processor has its own programs and data to process. If one processor requires data which resides in another processors memory, it must be sent to that processor by the processor which controls that piece of data. Communications between processors is often a bottle-neck in such machines and a good algorithm needs to minimise both their volume and number. The distributed memory MIMD architecture is most versatile, since it is able to emulate all the others. As a result, however, distributed memory MIMD machines are probably the most difficult the program efficiently.

Transputer arrays are a more common example of a machine with this type of architecture. The Transputer is a microprocessor that has been designed such that many of them can be used in parallel. Each processor has a floating point co-processor, some memory, and communications hardware incorporated onto the same microchip. The communications hardware consists of two, or four, links which allow transfer of data between processors. Data communication can proceed in parallel with the execution of instructions by the processor. The physical connections between processors in these arrays are therefore limited to rectangular grids or rings. Other connection schemes can be simulated by software but often requires processors to act as relays for messages between processors that are not directly connected.

On all parallel machines the best algorithms must have good load balancing. It is not cost effective to distribute the computing such that one processor still does most of the work, or that processors are left waiting for others to perform a particular action. An equally important feature is that the algorithm scales efficiently. For example, if we double the number of processors the job should be completed in half the time, although we recognize that there will be a lower bound when the overheads of distributing the work exceeds the gains to be had from further distribution. In general this is only possible if the work over processors is totally independent i.e. there is no data sharing. A way of reducing data sharing is to calculate key parts on each processor. The compromise between communicating data and introducing redundant calculations needs careful consideration.

More often than not the same program needs to be run several times. When this is the case it is often a better alternative to run the program in parallel rather than attempt to parallelise the code where possible. This option is most attractive on machines which have coarse grained parallelism and powerful processors such as the Cray X-MP. Problems which require large amounts of memory and CPU time are better candidates for parallelisation. On a shared memory machine, use of all the memory prevents other jobs from being loaded and executed on idle processors. This is an example of bad load balancing and common charging algorithms penalise such jobs which make no use of this untapped processing power. Distributed memory machines often do not possess enough memory on a processing node to hold all the data for a large job and one has no choice but to parallelise the program.

2. OUTLINE OF THE PAPER

In the following section we introduce the salient features of the molecular dynamics simulation technique. The link-cell algorithm is introduced as a method of reducing the computational requirements of a molecular dynamics simulation. The conditions under which this technique offers optimum performance are presented. Potential problems which may cause poor performance are discussed and possible remedies are suggested.

Section 4 describes the modifications we have made to the link-cell algorithm so tht it can be applied to shared memory and distributed memory MIMD architecture parallel machines. The additional constraints required to ensure optimum performance are discussed. In the subsequent section we explain the structure of the programs we have developed for the Cray X-MP and Intel iPSC/2. We use a two dimensional simulation of Lennard-Jones atoms as an example to describe the steps that the programs perform during a complete timestep. Only the steps which differ from a sequential version of the program are described in detail.

The timings obtained using these programs on a Cray X-MP are discussed in section 6.1. Results for differing numbers of both two and three dimensional Lennard-Jones atoms are presented when using four processors. The performance of different versions of the programs on the Intel iPSC/2 is compared in section 6.2. How these timings scale with numbers of processors and number of atoms are discussed for simulations of two and three dimensional systems of Lennard-Jones atoms. We offer our conclusions in section 7 and provide some general rules to help the user obtain optimum performance from these programs.

3. MOLECULAR DYNAMICS

Molecular dynamics is the integration of the equations of motion of a set of N atoms, which may be associated as molecules. These atoms often interact with each other through an intermolecular potential and there may also be an external field representing a surface. Generally the intermolecular potential is a function of all the molecular coordinates, translational and orientational. The true potential includes many body terms which require direct evaluations of triplets, quartets etc. and this is a time consuming calculation. Fortunately the total intermolecular interaction can often be accurately approximated as a sum of effective pair potentials, $v(r_{ij})$, such as the Lennard-Jones potential

$$v(r_{ij}) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \qquad (1)$$

where ε and σ are energy and length parameters. The force on each atom is required at each timestep, and is just the sum of the force due to its interaction with each of the other $N - 1$ atoms and the force due to any external field. The intermolecular force on atom i is conveniently expressed as

$$F_i = \sum_{i \neq j} f_{ij} = - \sum_{i \neq j} \nabla_{r_{ij}} v(r_{ij}) \qquad (2)$$

This calculation of all forces occurs in a double loop which takes 90–100% of the time in most molecular dynamics simulations. Use of Newton's third law, $f_{ij} = - f_{ij}$, reduces the number of pairs which are explicitly considered to $N(N - 1)/2$. As N gets large such a calculation becomes prohibitively expensive. for this reason simulations are usually limited to a few thousand atoms.

When the range of the forces between atoms is of order of the size of the simulation box all $N(N - 1)/2$ pair interactions will have to be calculated. Systolic loop algorithms have already been published with efficiently parallelise the all pairs calculation in such a case [3].

For short ranged forces one can apply a spherical cutoff, with radius r_c and set the potential $v(r_{ij})$ to zero for $r_{ij} \geqslant r_c$. For a three dimensional system of 1000 Lennard-Jones atoms at $\rho\sigma^3 = 0.75$ a cutoff of $r_c = 2.5\sigma$ means each atom interacts with approximately 50 atoms. For a system with large N and a relatively small value of r_c one can use several methods which consider only those pairs of atoms which are within $r_c + r_{\text{skin}}$ of each other. The region of space between r_c and r_{skin} contains atoms which, although they do not interact with the atom under consideration, must be considered due to some feature of the algorithm. This may be due to the algorithm being inefficient in identifying the interactions which we wish to consider, or, for example when using a Verlet neighbour list r_{skin} can be used to minimise the number of periodic updates of the neighbour list.

The linked cell algorithm [4] is one such method. It involves dividing the simulation box into M^D sub-cells each with sides of length $l = L/M$, where D is the dimensionality of the system and l is as close to, but always greater than, r_c. Figure 1 shows a two dimensional system, where the atoms within the cutoff distance can be found by considering each cell and its 8 nearest neighbours (26 in three dimensions). The use of Newton's third law allows us to consider only half of these neighbouring cells, which are hatched in the figure.

We now consider how the amount of computing time required to perform each timestep scales with N, the total number of atoms. There will always be fixed overheads such as initialising scalar variables, outputting intermediate results, and the time taken to start a loop. The integration steps, and the creation of the linked list in the linked cell method, are simple loops of length N and so the time to execute them scales linearly with N. When using an all pairs algorithm which considers $N(N - 1)/2$ pairs, the time to perform the force calculation, and hence the overall program

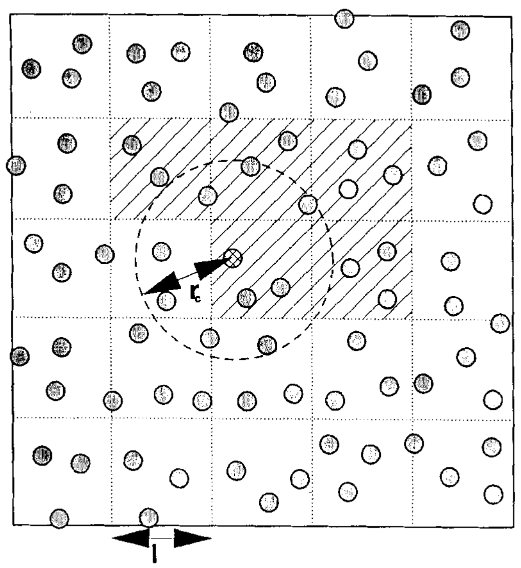


Figure 1 Schematic diagram of link-cell algorithm for a system of two dimensional system of Lennard-Jones discs.

performance, scales as N^2. When using the link cell algorithm in three dimensions for each atom we consider as potential interacting pairs, on average, half the occupants of the central cell containing it and the occupants of 13 nearest neighbour cells. If each cell has n_c occupants then we consider $13.5\,Nn_c$ pairs and so the time required, to a first approximation, scales linearly with N. However this relationship is not strictly linear, n_c is proportional to l^3, and so, between points where the box dimensions are an integer multiple of the cutoff, the time taken follows a cubic curve. This gives the saw-tooth shaped curve shown in Figure 2, using $\rho^* = 0.75$ and $r_c = 2^{1/6}\sigma$ for a three dimensional system, where we plot $13.5\,Nn_c$ against N. It is often the case that the box dimensions are not a multiple of r_c and, so, l can be significantly larger than the cutoff. In such a situation it is possible that increasing the box dimensions, by adding more atoms at constant density, so that l becomes closer to r_c will increase the speed of the program. In general these teeth are polynomials of the same order as the dimensionality of the system.

There is the extra overhead of creating the linked list, and so, depending on the cutoff, the density, and even the type of computer, there is a cross-over point where the linked cell method becomes more efficient than the Verlet neighbour list [4] and the straight forward all pairs calculation. The method will not improve the speed of the calculation until $M > 3$ when the linked cell method considers less than all the pair interactions. The memory requirements of the Verlet neighbour list also increases faster, with the number of atoms being simulated, than the link-cell algorithm. In its simplest form this method is non-vectorisible due the use of a linked list to indirectly address the coordinate arrays. Several workers [5], [6] have modified this algorithm to produce code that vectorises. The link-cell technique has also been adapted for a MIMD shared memory architecture when small system sizes are being used [7]. The simulation space is divided into slabs and distributed over a ring of processors. The

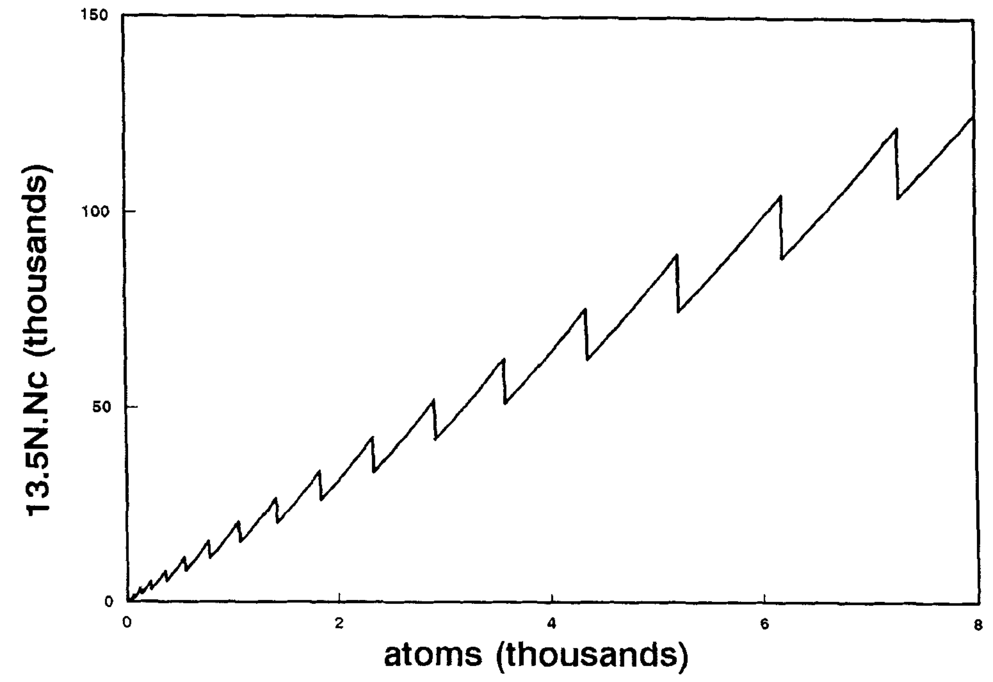


Figure 2 Average number of pair interactions considered when using link cell algorithm in a simulation of Lennard-Jones atoms in three dimensions against number of atoms when $\rho^* = 0.75$, $r_c = 2^{1/6}\sigma$.

slabs have widths that are fractions of the cell length. This means that to complete the force calculation there must be exchange of configurational data between processors which contain the remaining information of a cell and its nearest neighbours. This is handled in a manner similar to the systolic loop algorithm of reference 3.

In three dimensions if one has cells of length r_c then the volume considered by the algorithm is $13.5\,r_c^3$ while the interacting atoms we wish to consider will be within an hemisphere of radius r_c. This means that in three dimensions 84%, and in two dimensions 65%, of the space we are searching need not be considered. One remedy is to make the link cell sizes much smaller and consider a set of neighbours which is more hemispherical. This increases memory requirements as there are now many more cells most of which will be empty. Some kind of memory management such as a bit map is therefore required to stop this data structure from getting too unwieldy. This modification to the algorithm was not used for the programs we have developed for reasons of simplicity.

4. PARALLEL LINKED CELL ALGORITHM

The geometrical breakdown of the simulation space in the linked cell method is an obvious candidate for parallelisation on all three types of machine discussed in the introduction. This domain decomposition can be performed in many ways. The simplest division use equally spaced planes parallel to one, or more, of the faces of the

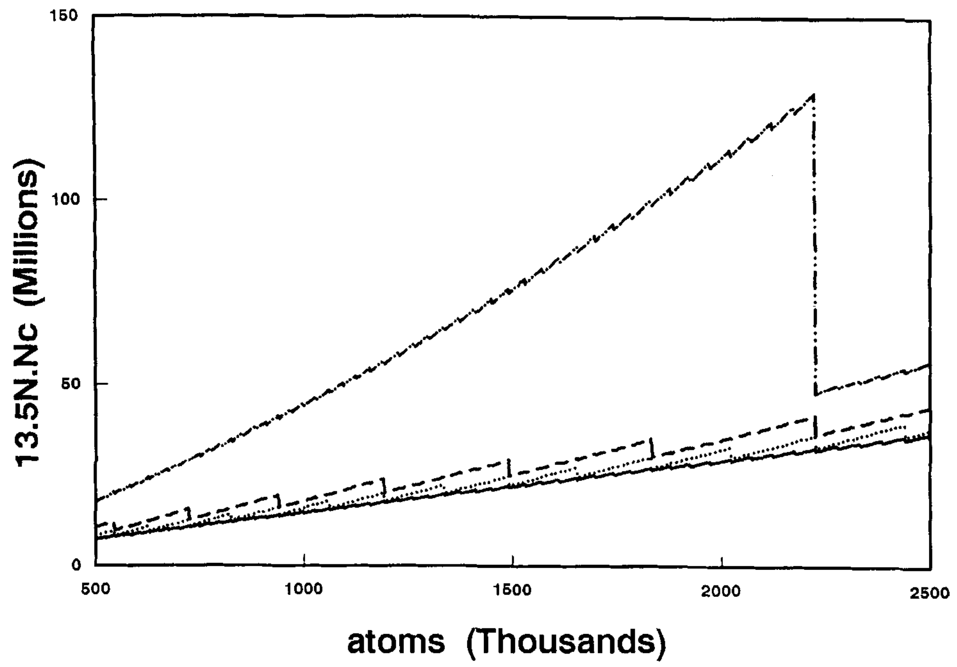


Figure 3 Average number of pair interactions considered when using distributed memory link cell algorithm on 64 processors in a simulation of Lennard-Jones atoms in three dimensions against number of atoms when $\rho^* = 0.75$, $r_c = 2^{1/6}\sigma$. Curves represent decompositions into 64 slabs (dot-dot-dash line), 8 by 8 square prisms (dash line), 4 by 4 by 4 sub-cubes (doted line), and on one processor (solid line).

simulation box. This leads to D possible domain decompositions in a D dimensional simulation space, for example, in three dimensions one can cut the box into slabs, square prisms, or cubes. Each of these domains is then sub-divided into link-cells. For maximum efficiency the length of the cell must be the same size as the cutoff and the total number of cells in any direction must be an integral number of the number of processors assigned to that direction. To keep the work equally distributed between the processors we require the same number of cells in each direction on each processor. Failure to meet these requirements accentuates the teeth of the saw-tooth curve in Figure 2.

To illustrate this point more fully in Figure 3 we have plotted the average number of pair interactions considered in the simulation of a fluid in three dimensions as a function of the total number of atoms. The curves are for the distributed memory version of the algorithm described in section 4.2. The potential is a Lennard-Jones interaction cut at $2^{1/6}\sigma$ and shifted upwards by ε. The force is a continuous function of r. The density ($\rho^* = \rho\sigma^D$) is 0.75 and the temperature ($T^* = kT/\varepsilon$) is 1.0. The average number of pairs is $13.5\,Nn_c$, where N is the total number of atoms and n_c is the average number of atoms per link-cell. In the plot the solid line is for a simulation on one processor and shows a weak saw-tooth. Note that the extra work that the saw-tooth represents gets less, in proportion to the total amount of work we have to perform, as the system gets larger. With a system size of $5 \cdot 10^5$ atoms on a single processor it has become hardly noticeable. If we distribute the simulation over 64 processors we have the additional constraint that we have an equal number of cells on each processor in a particular direction. The saw-tooth effects becomes more pronounced and is strongest for the domain decomposition into slabs in which the number of cells in one direction has to be a multiple of 64 and weaker for the decomposition into cubes since the number of cells only has to be a multiple of 4. When using slabs and prisms there are directions along which we have only periodic boundaries and not a decomposition over processors. Along these directions the only constraint that need be applied is that we have a minimum cell size i.e. greater than the cutoff. This means that we have smaller teeth due to this effect superimposed on the large teeth due to the distribution over processors in other directions. Note that because we duplicate some of the force calculations on a distributed memory machine we always calculate more interactions than for the same system on one processor on a shared memory machine. Changing the cutoff to 2.5σ for the full Lennard-Jones potential magnifies this effect. The teeth are functions of the volume of each linked cell, which now have to get larger before an extra cell can be accommodated, and hence the teeth are larger for a larger cutoff.

The ideas are equally applicable to SIMD, MIMD, and shared memory architecutres. We will now discuss the implementation of the program on the different architectures.

4.1. Linked cells on a shared memory machine

When using a shared memory machine almost no modification is required. Each processor works on a section of the linked list corresponding to a slab of space. Except

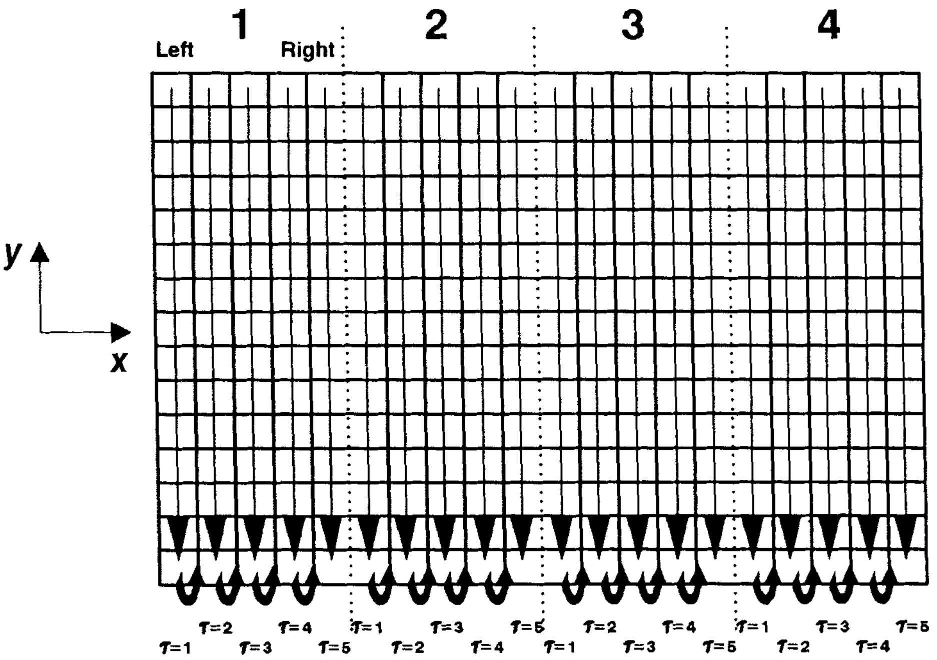


Figure 4 The order in which we search the link cells in two dimensions when using the shared memory algorithm. Dotted lines are domain boundaries, solid lines are link-cell boundaries. Domain 1 has the columns on the left and right domain boundaries marked.

when searching cells at the edge of its slab, a proessor can work independently of all the others. As can be seen from Figure 4, in a two dimensional system, if the searching of cells is done in columns from one side of the slab to the other, memory conflicts can be minimised. When processor 2 begins at $\tau = 1$, processor 1 is working on a section of its domain physically removed from that required by processor 2, processor 1 can have access to any nearest neighbour cell that it requires. The last calculations by processor 2, at $\tau = 5$, requires data from cells controlled by processor 3, but this processor should have finished its calculations using these cells, and, in principle, they should also be available to processor 2 immediately. If we use a higher dimensional domain decomposition there is less potential work between cells in the domains where one encounters possible memory conflict. This method has been used on a Cray X-MP using the locks feature of the multitasking software. Locks are flags which can be used to prevent a processor from executing whilst a memory conflict could arise. When a processor enters a critical region of code it attempts to set a lock. If the lock is not set already it is set and the code continues executing, clearing the lock when exiting the region of the program that requires protection. When the lock is already set the processor suspends its execution until the lock is cleared by the process which set it. The processor then sets the lock and continues execution as described above. Although locks are required to ensure that only one processor has access to a cell at any time, it has been observed that, if there is sufficient work to be performed between the two edges of the slabs, they almost never cause one processor to wait for another. On a machine executing a large number of different jobs, it is likely that the multitasked processes will execute on fewer than the required number of processors for a significant proportion of the time, unless the program occupies all the memory. On a machine executing many jobs synchronization between the processors may be lost and in this situation there may be considerable memory conflict. Multitasked programs use more memory than equivalent single processor versions and if there is appreciable memory conflict this can cause the multitasked program to be more expensive than a single processor version. The conclusion is that on machines such as the Cray X-MP there is little to be gained from this method unless the machine is dedicated to your program, that is there are no other jobs running on the machine, or you have used all the memory preventing other jobs from executing at the same time. Few researchers have exclusive use of a Cray X-MP and so this method will only be of general use for large system sizes. The structure of the program is described in detail in section 5.1.

4.2. Linked cells on a distributed memory machine

On a distributed memory MIMD machine each processor runs a linked cell program corresponding to a particular domain of the simulation box. If planes in N_c orthogonal directions were used to divide the simulation space into domains, each processor has $2N_c$ processors sharing its faces, with periodic boundary conditions in the remaining D - N_c directions. To complete the force calculation for atoms in cells at the interface between processors each processor needs to know the coordinates of the atoms in the adjacent cells which will be found on a neighbouring processor. To handle this problem we construct an extra layer of cells on each processor at the interface between processors. At each timestep we swap information across the interface between adjacent processors to describe the atomic configuration in these edge cells. At this point each processor has all the information required for its own force calculation. After the force calculation the atoms are moved and they may move into domains controlled by other processors. In this case the information about this atom must be moved between processors. Both these exchanges of data can be achieved by one set of communications between the processors. Naively, this implies that at each timestep a processor needs to communicate once with all its $3^{N_c} - 1$

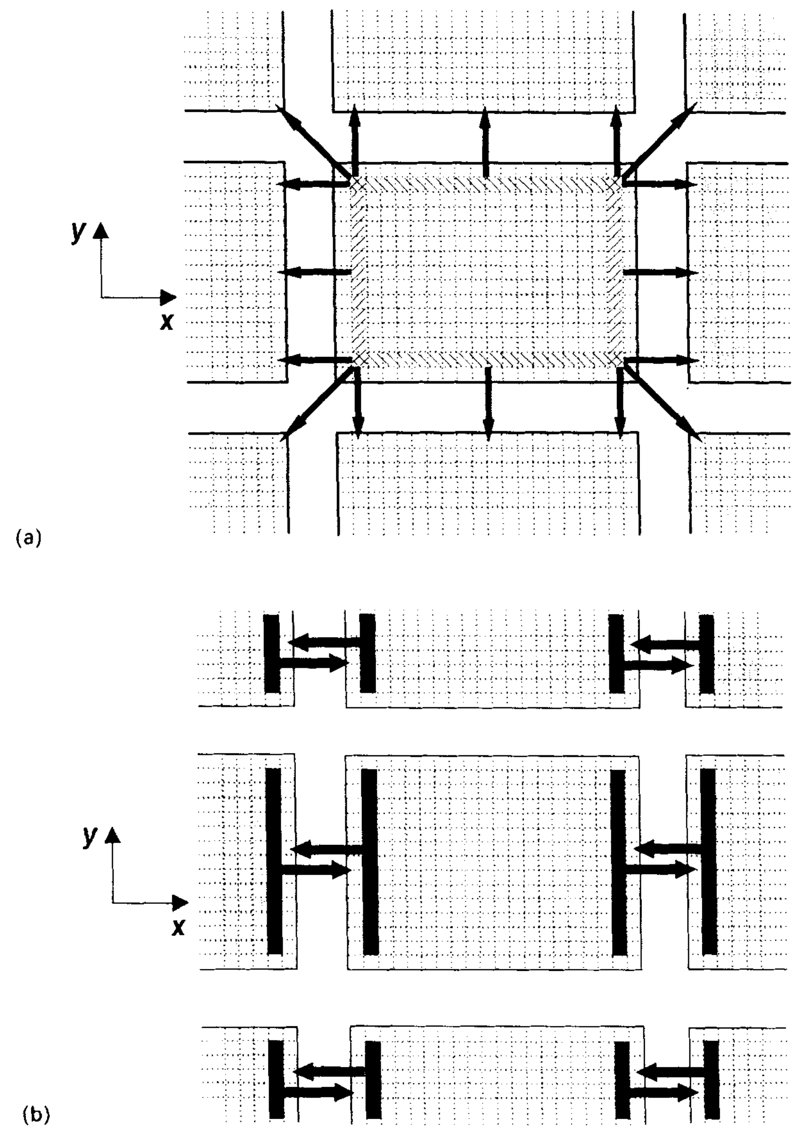


Figure 5 (a) Where the contents of the hatched link cells needs to be sent to complete the edge cells controlled by each processor when using the distributed memory algorithm in two dimensions and a decomposition into squares. Which cells are passed (filled in) and where in the first (b) and second (c) edge swapping phase. Shaped and hatched regions already contain necessary configurational information.

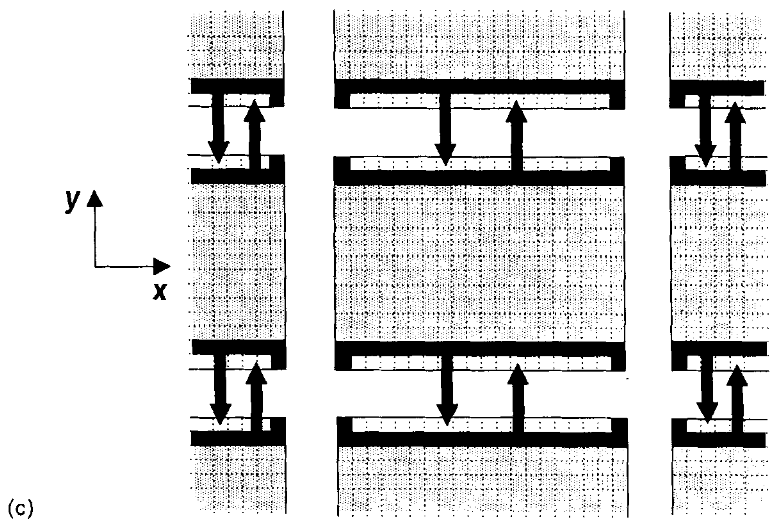


Figure 5 (continued)

nearest neighbours, that is the processors with which it has domain boundaries in common. If one organises the communications so that data is passed simultaneously in opposite coordinate directions, and processed before passing along other directions, the number of communications is reduced to $2N_c$ at each timestep. As an example we consider a processor and its neighbours if $N_c = D = 2$ in Figure 5. The bold arrows in Figure 5(a) shows the directions in which we have to pass the configurational information contained in cells at the interface between processors. Dotted lines indicate the boundaries of the link-cells in each domain, shading denotes cells which contain the configurational information corresponding to the domain controlled by the processor. We see that the information contained in cells at corners of the processor's domain (cross hatched in the figure) is required on three other processors, while the information contained in edges (hatched in the figure) is needed by only one other processor. Figure 5(b) and 5(c) show the first and second phases of the exchange of configurational information. Shaded areas already contain the correct information, filled areas denote cells from which information is being passed, and bold arrows indicate the destination of this information. Note that we have to pass the contents of the corner edge cells in the second phase of data exchange. This is to ensure that we create all the images of an atom that has moved, from the corner link-cell of a domain, into a corner edge cell during the last integration step.

Figure 6 shows the edges of two adjacent processors A and B and focuses on three atoms 1,2, and 3. Each atom is in a different cell. Atoms 1 and 2 are in the domain controlled by processor B and we describe them as real atoms. They are stored on processor A as edge atoms 1′ and 2′. Real atom 3 is controlled by processor A and has an image 3′ which is an edge atom on processor B. The calculation of the force between atoms 2 and 3′, and 2′ and 3, is a duplication of the force calculation between the real atoms 2 and 3. We only store the components of the force on the real atoms 2 and 3 and not on the edge atoms 2′ and 3′. The force between real atoms 1 and 2 occurs on processor B, no force is calculated between 1′ and 2′ on processor A. This

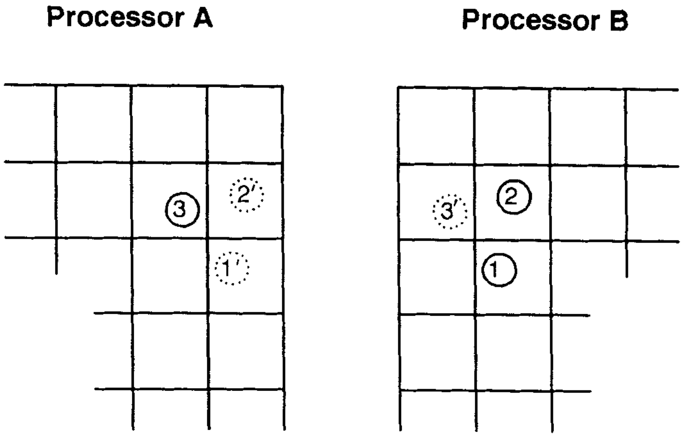


Figure 6 Duplication of force calculation of edge atoms. Prime denotes an edge atom.

small redundancy in the force calculation seems preferable to having a extra phase of data exchange between processors. This would involve the forces, which would necessitate an additional synchronization point in each timstep where processors could be left waiting for others to complete a task. In all our programs there is only one synchronization point at each timestep. This is a nearest neighbour synchronization and does not extend over the whole processor network. The redundancy in the force calculation is minimised by keeping N_p/N, the ratio of the average number of atoms passed to create the edge regions to the number of atoms controlled by each processor, small. Note that if we have a uniform density distribution this is equivalent to the ratio of the number of edge link-cells to the number of link-cells in the domain on each processor.

There is one case where we could see a reason for calculating all the fores only once, when N_p/N is large and we are doing a sizable number of duplicated force calculations. An extra data exchange means the edge coordinates need only be passed in one of the two opposite coordinate directions. This saves on memory associated with the edge coordinated i.e. the linked list on each processor can be slightly smaller, and requires less buffer space for passing information. Having no duplicated force calculations would also aid the scaling properties of the programs with number of processors.

We have described a number of domain decompositions. The efficiency of a particular decomposition will be machine dependent. The smaller the value of N_c the fewer the number of communications required. However N_p/N, which is a measure of the amount of information for the whole set of communications at each timestep, decreases significantly in going from $N_c = 1$ to 2 (i.e. from slabs to prisms) and less significantly for $N_c = 2$ to 3 (i.e. for prisms to cubes). The three ratios R_1, R_2 and R_3, corresponding to the three possible decompositions are discussed in appendix A. Important factors are the relative speeds of communication of large and small data packets, the start up time to initiate a communication and the ratio of processing to communications speed. The communications hardware on the Intel hypercube has a relatively slow start up time when compared to a Transputer. This means that for practical purposes there is almost no difference in speed between synchronous and asynchronous communications on the hypercube. It also suggests that the passing of one large data packet rather than several smaller ones will result in a more significant improvement in speed on the hypercube than on a transputer. Another consideration is the way in which the processors are connected. As Transputers have four communications links a common connection scheme is a square grid, although it is more accurately described as a torus. In this case a three dimensional simulation would only involve exchange of information between directly connected processors if $N_c \leqslant 2$. If N_c is three, then some kind of message passing, using intermediate processors, is required. This may be significantly slower than passing between adjacent processors.

5. PROGRAM STRUCTURE

All the programs used were developed from the link cell program in fiche 20 of reference 4. The structure of the original linked list program is as follows.

```
READ IN INITIAL DATA e.g. Temperature, density.
READ IN CONFIGURATIONAL DATA i.e. coordinated, velocities.
SET UP MAP TO FIND NEIGHBOURING CELLS
DO FOR NUMBER OF TIMESTEPS
  FIRST PART OF VELOCITY VERLET ALGORITHM:
        $R_i(t) = R_i(t - \delta t) + \delta t V_i(t) + \frac{(\delta t)^2}{2} F_i(t - \delta t)$
        $V_i\left(t + \frac{\delta t}{2}\right) = V_i(t) + \frac{\delta t}{2} F_i(t - \delta t)$
  CREATE LINKED LIST
  CALCULATE FORCES: $F_i(t)$
  SECOND PART OF VELOCITY VERLET ALGORITHM:
        $V_i(t + \delta t) = V_i\left(t + \frac{\delta t}{2}\right) + \frac{\delta t}{2} F_i(t)$
  OUTPUT INTERMEDIATE RESULTS e.g. total energy, temperature
WRITE OUT FINAL AVERAGES AND FLUCTUATIONS
SAVE CONFIGURATION
```

This program performs a molecular dynamics simulation of Lennard-Jones atoms in three dimensions and is easily modified to simulate systems which are not cubic and which have different dimensionality.

5.1 Parallel linked cells on the Cray X-MP

Little modification was required to develop of a parallel version of the linked cell code for use on the Cray X-MP. The subroutine FORCE was called in parallel using the TSKSTART multitasking routine. The single loop over cells in this routine is replaced by a double loop of the following form for a two dimensional simulation.

```
DO FOR $I_y = M_y \cdot (P_{id} - 1)/P_t$ TO $M_y \cdot P_{id}/P_t - 1$
    DO FOR $I_x = 1$ TO $M_x$
        ICELL $= I_y \cdot M_x + I_x$
```

Where P_t is the number of multitasked subroutines, P_{id} is a process identification number between 1 and P_t, M_x, and M_y are the number of cells in the x, and y directions, and ICELL is the index of the cell currently being considered.

P_t locks are defined which prevent memory conflict in searching of the cells. One lock is defined for each of the right hand column of cells in each of the strips that each multitasked subroutine controls (see Figure 4). When the multitasked subroutine is entered an extra lock is used to allow the copying of global scalar variables into local ones. The searching of cells starts from the left hand side of each strip and each task sets the lock corresponding to the strip of cells controlled by the processor to its left. When the second iteration of the outer loop is completed this lock is cleared. Prior to execution of the penultimate iteration of the outside loop the lock corresponding to the tasks own strip is set. This lock is cleared on exiting the force loop and the extra lock is used to accumulate the local scalar variables into the corresponding global ones.

Multitasking the integration routines i.e. those which perform the velocity Verlet algorithm, are much simpler in that one only has to divide the loop over N atoms into P_t loops of length N/P_t. It should be noted that there may not be enough work involved in these steps, that is the overheads in multitasking these routines may exceed the reduction in real time that it takes to execute them in parallel. One could then use the microtasking software which has lower overheads. This is most applicable when there is a loop where each iteration is independent of the others. Free processors execute the next iteration of the loop and ensure that loop counters are incremented appropriately. Neither of these methods were used in the programs presented here, the gain in performance they offer is expected to be small except for the largest of system sizes. A copy of this multitasked link-cell program for a two dimensional simulation has been deposited with the CCP5 program library (see Appendix B).

5.2 Parallel linked cells on the Intel iPSC/2

The program is split into two parts which we call the master and worker. The master program runs on the system resource manager which is the front end of the Intel hypercube. Its main function is to handle the transfer of information between the workers and storage media and output devices. Worker programs, which are modified link-cell programs, run on each of the nodes of the hypercube and perform the molecular dynamics of the atoms which are contained within the domain they control. They pass out local thermodynamic data to the master which can be accumulated and averaged for the complete system. Operations which involve the communication of large volumes of data between each of the workers separately must be avoided. In such a situation there may be considerable periods where only the communicating worker is doing meaningful work. This is unavoidable if we wish to save a configuration and we are careful to limit the frequency of such an operation. Global broadcasts of the same data to all the workers are less expensive operations but will still hinder performance. For example if we wish to equilibrate the system to a set temperature we scale the velocities using a local total kinetic energy within that processor. A rescale using the global total kinetic energy requires its accumulation on the master processor and if performed only rarely will improve program performance. It should be noted that the iPSC/2 has several library routines which will perform simple operations such as a global sum over processors using the minimum amount of communications and will return the sum to each processor. These routines, whilst offering a faster method of performing these operations, are machine dependent and, as an aid to portability, we have not employed them in the programs we have developed.

The master program begins by prompting the user for information about how the simulation space is to be divided and what simulation to perform. The master uses the information about the geometry of the domain decomposition required to request the correct number of processors, or nodes, from the hypercube. Worker programs are then loaded onto each of the nodes and start execution. Information such as the total number of atoms, the density, the cutoff, and the geometry and size of the processor network is used to determine whether the simulation is possible with the requested domain decomposition. These data are then passed to each worker program with information such as the timestep and the physical dimensions of each processor's domain. Each worker then calculates the dimensions of its domain and the correct number of link-cells to use and various factors to enable the correct creation of the linked list. The master then passes the configurational information to the workers which transform the coordinates so that they are in the range $-l_\alpha/2 \leqslant R_{\alpha i} < l_\alpha/2$, where l_α is the size of that processor's domain in the α direction. When using small system sizes of $N < 200000$, an equal number of atoms are sent to each processor and then they are shuffled about by the network until each atom is on its home processor. The shuffling of large numbers of atoms can deadlock the communications and we have to resort to a different procedure when loading the configurational information onto the processors. When dealing with large system sizes, the master calculates or reads in the coordinates and velocities of atoms associated with a worker and sends them to that worker. At no point are the coordinates of all the atoms stored on the master (i.e. for an eight processor system the configuration is read or created up to eight times). The physical limits on the size of a data packet, 256 K in this case, that can be sent between the master and a worker on a hypercube means that the configurational data may have to be sent out in two, or more, separate data packets.

To determine which processors control adjacent domains we identify each worker uniquely i.e. in a three dimensional processor system a worker might have the address (2,1,3) which can be combined to give a unique identification number between 1 and N_w, the total number of worker programs running. This address is created by considering each processor in the network as a cell in a link-cell structure. We then use the linked-cell algorithm to determine the addresses of a processor's neighbours. Processors (1,1,1) and (1,2,1) communicate through the Direct Connect communications hardware of the hypercube. We need to know nothing about the real space positions of the processors in the hypercube or the real path of communications between them. This makes these codes readily transportable to other MIMD machines with similarly organised communications. The program only requires a general message passing harness (such as CSTOOLS[8] provides on Meiko Transputer based systems). We note that an optimization algorithm which placed workers which control adjacent domains on processors with the shortest communications path might improve the communications speed of the program, but would certainly reduce its portability. Such optimisation should always be considered when applying these programs to a different machine as such factors may be important.

The major modifications for the worker programs are in the routine where the linked list is created. We illustrate this by considering the structure of this routine for a two dimensional simulation which uses edge passing in both x and y directions. Passing in fewer directions is a trivial simplification of this routine while to pass in more directions is a straight forward logical extension.

This routine is called after the part of the velocity Verlet integration in which the coordinates have changed. We recall that each processor has a linked cell structure of dimensions of M_x by M_y.

5.2.1 Phase 1 processing the atoms on each processor

We use following equation to define a cell index ICELL_i for atom i, note that i is an atom that the processor currently controls (i.e. we are not considering the edge atoms from the previous timestep),

$$\begin{aligned} I_{xi} &= 1 + \mathrm{INT}[(R_{xi} \cdot SF_x + 0.5) \cdot M_x] \\ I_{yi} &= 1 + \mathrm{INT}[(R_{yi} \cdot SF_y + 0.5) \cdot M_y] \\ \mathrm{ICELL}_i &= I_{xi} + [I_{yi} - 1] \cdot M_x \end{aligned}$$

remembering that the coordinates are in the range $-l_\alpha/2 \leqslant R_{\alpha i} < l_\alpha/2$, l_α is the size of that processor's domain in the α direction, and SF_α is a scale factor which transforms these coordinates such that atoms within the domain will have $I_{\alpha i}$ values in the range $2 \leqslant I_{\alpha i} \leqslant M_\alpha - 1$. Atoms with $I_{\alpha i}$ in this range are incorporated into the linked list in the normal manner. If we are not passing in a direction β then SF_β is such that we have no edge cells in this direction.

During this process we may have atoms that have moved out of the domain controlled by that processor into one of the edge cells. These can easily be identified as $I_{\alpha i}$ will be either 1 or M_α. We collect the coordinates and velocities of these edge atoms into buffer arrays to be pased to an adjacent processor. To keep the coordinate and velocity arrays tidy for the processor under consideration the atom with highest index (i.e. i = NATM) is placed in the position of the atoms in the array that has just been removed and NATM, which gives the number of atoms on the processor, is reduced by one. An important consequence of this is that we can no longer identify an atom uniquely by its index in these arrays. If we wish to keep track of individual particles we will need an extra array which contains an atoms label and this will have to be updated in the same manner as the coordinate and velocity arrays. The coordinates of atoms that have moved out of the processor's control will still be required since they will be edge atoms at the current step. They are therefore incorporated into the linked list. These coordinates are stored in the same array as those of the atoms still controlled by the processor but they are added from the end of the array downwards. Therefore if a coordinate array of size NMAX contains NATM real atoms and NEDGE edge atoms, it has the real atoms stored with indices 1 to NATM and edge atoms from NMAX-NEDGE to NMAX. This allows us to determine whether an atom is in an edge cell or not by simply testing whether its index is greater than NATM. The coordinate array must be large enough to ensure that these sets of data do not overlap. Such an overlap causes the coordinates to be corrupted and, more importantly, the linked list becomes recursive. This means that when the subroutine FORCE is next entered the program may generate an error or find itself in an infinite loop. Fortunately a simple test can detect this situation and at least prevent unnecessary wastage of CPU time. The remedy is to increase the array size NMAX.

At this point every atom which was on the processor before the integration step has been incorporated into the linked list as either real or edge atoms. All the atoms that are edge atoms have been parcelled to send towards the processors which will control them as real atoms (note that these may not find a home on the first processor to which they are passed).

5.2.2 Phase 2: the first passing of edges

The next step is to send out the coordinates of atoms in cells just inside the domain boundary which are required for the edges of other processors. These atoms can be obtained directly by using the linked list already created to search only the relevant cells. The appropriate cells to search for the first set of communications is shown in Figure 5(b). This information is added to the buffer containing the configurational information about the real atoms that became edge atoms during the last integration step which are being passed in the same direction. Two data packets are then sent in the opposite coordinate directions. The first packet, of fixed size, contains two indexes, which contains information that gives the number of potential new real atoms, and the total number of atoms being passed (real and edge). This initial data packet also allows the receiver processor to determine the size, which will be variable, of the second packet which contains the configurational data. We must perform two separate communications because both the sending and receiving processors must know the size, in bytes, of every data packet being passed. Note that we do not pass the associated velocity of an edge atom.

As a potential aid to synchronization we collect the edge atoms for passing in the other direction in between the sending and receiving of the first data packets. We also process the first data packet received in between sending and receiving the data packets passed in the second direction. This could allow each worker to perform some useful work before waiting to receive data packets from other processors. In the programs we present, we use blocked communications. That is when a processor is sending data to another processor it waits for an acknowledgement from the other processor to announce that it has received the data before continuing execution. When receiving data a processor waits for the data to arrive and posts its acknowledgement before continuing execution. This means that the program knows the state of the buffer space at all points in the code. In particular it will know when the buffer space is free for reuse. Asynchronous communication where the program does not wait for the communication means that we have to define twice as many buffers; one set for sending and the other for receiving data. A version of the program which performed asynchronous communications using a decomposition into sub-cubes in three dimensions was timed. The speed of execution of this program was not noticeably differently to that of one using blocked communications even when using 64 processors. The use of ansynchronous communications be of more potential use on a transputer based machine as they can be performed more efficiently than on the hypercube.

When both packets from a particular direction have arrived we start processing the data by unpacking the real atoms incorporating them into the link-list as described in phase 1. Since the passing has occurred in only one direction, it is still possible that some of these potential real atoms have not arrived at their proper destination and they are only edge atoms on this processor. These atoms will need to be passed on in an orthogonal direction and stored as edge atoms on this intermediate processor in their journey. The edge atoms are unpacked and incorporated into the link-list.

5.2.3 Phase 3: the second passing of edges

The next step is to complete the passing on of the real atoms in an orthogonal direction, with the real atoms which are moving for the first time in that orthogonal direction, and the coordinates required to complete the edges in the new direction. This is achieved by searching the filled cells in Figure 5(c). Note that the information

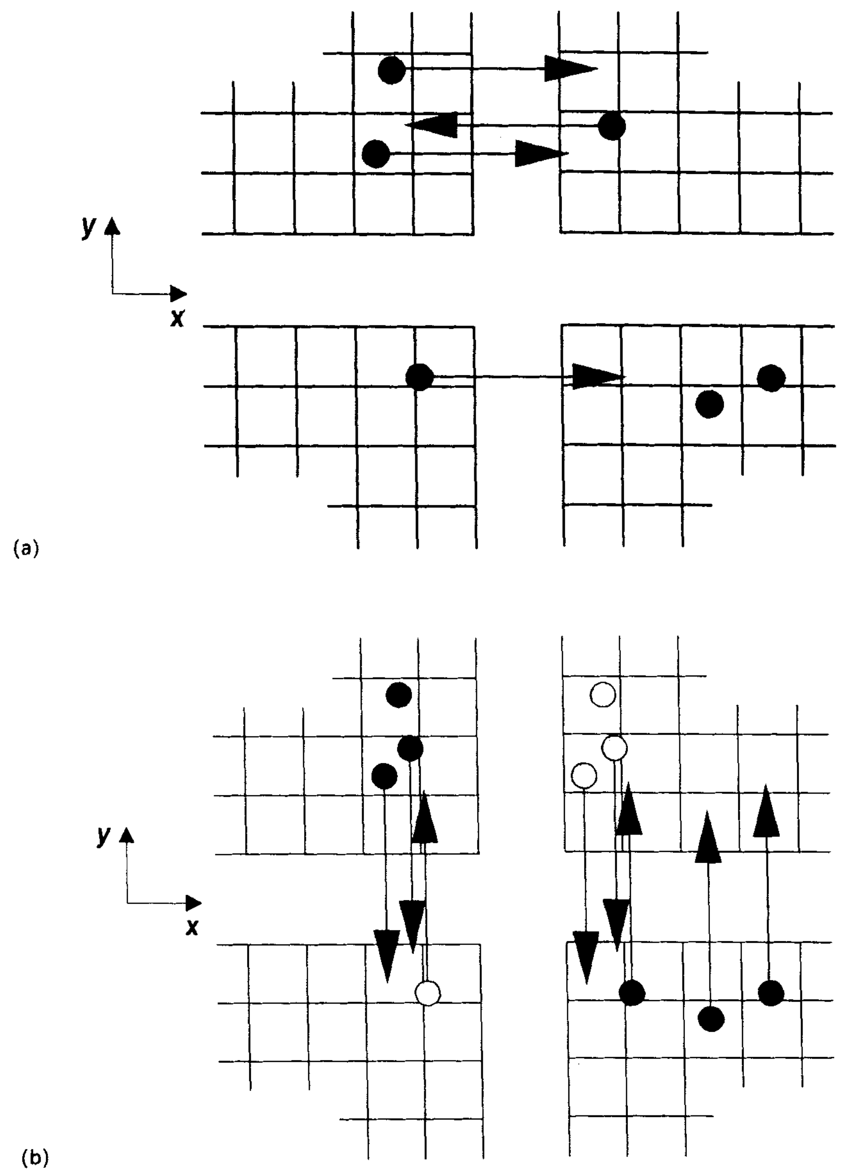


Figure 7 The movement of atoms and creation of edge atoms during passing of edges for a decomposition into squares. (a) prior to first passing phase, (b) after first passing phase, (c) after passing has been completed. Real atoms are filled, edge atoms created during first passes are shaded, those created in the second phase are hatched.

currently contained in the four corner cells needs to be passed. This is to ensure that we still create all the edge images of an atom that has moved between two diagonally adjacent processors during the last integration step. When atoms that have moved between processors arrive we must ensure that they are incorporated into the linked list in the manner that the processor which sent them expected. For example a

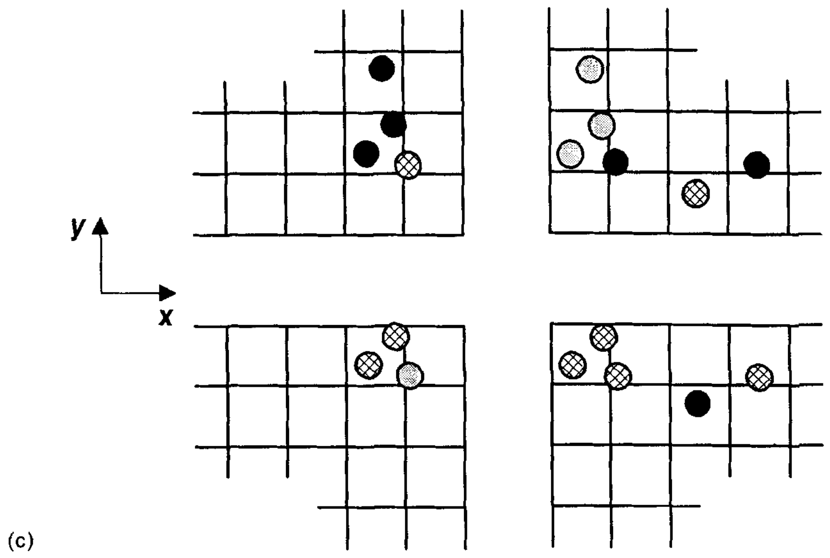


Figure 7 (continued)

problem can occur with atoms whose coordinates exactly coincide with the edge of a link cell. Rounding errors may result in these atoms not being associated with the correct link cell. In particular a real particle being moved between processors may be incorporated into an edge cell. This results in the forces on this atom not being accumulated during the next force calculation. Another possibility for such an atom is the creation of too many edge imges which will result in an attempt to calculate the force due to itself on itself. This will inevitably result in a division by zero error. This problem is resolved by using the fact that atoms that are moving between processors should be associated with cells adjacent to the edge cells which are in the interface between the sending and receiving processors. The passing of data packets is performed in the manner described in phase 2. All the real atoms will be arriving at their final destination and, therefore, just need to be added to the real part of the configurational arrays and the link-list.

This complicated series of manoeuvres is illustrated in Figure 7 for a representative number of atoms. Filled atoms represent the current position of the real atoms on the network. Shaded atoms are the edge images created during the first phase of passing along the x direction, hatched atoms are the edge images created when we pass along the y direction. Arrows denote the directions in which an atoms data is going to be passed.

Note that if we are passing in three directions the second and third passing of edge particles will have to include the edge particles in the cells along the edges of the face of the cube from which we are passing. In the second passing phase the cells along two of these edges will be empty. Thus the program only searches the cells along the other two sides and the four corners of this face.

5.2.4 Modifications to other subroutines and general comments

The subroutine FORCE is modified so that interactions within edge cells and between two edge cells are not calculated. There is an extra routine COUNT which, at each timestep, passes back the local thermodynamic data back to the master for global accumulation. To enable the global temperature rescaling during equilibration every tenth step the subroutine SCALET waits to receive the velocity scaling factor from the master program.

A useful feature is that we can use the arrays used for the forces as the buffer space to store the information to be passed to other processors, hence minimising the memory requirements of the routine.

When simulating large $N \approx 10^6$ one configuration requires 24 megabytes of data if we are using single precision arithmetic. This can take a considerable amount of time to load onto, and extract from, the processor network. It is also obvious that even a few configurations will require more storge than is usually available. Therefore any processing of configurational data must be performed as the configurations evolve during the simulation. Many systems, including the iPSC/2, offer distributed disk communications where it is possible for all the processors to read and write, asynchronously and concurrently, to a file on disk. This offers a much faster way of saving configurational data. The user also needs to be aware of any data structures that such systems impose upon the data saved in such a manner. The size of these configurations will still limit the number of configurations that can be saved. It may be sufficient to save just a representative portion of the configuration for later analysis. There will, in general, have to be some kind of compromise between the number and the completeness of the configurations that can be saved. It is of course possible to generate an initial configuration on the processor network. Care must be taken with things like random number generation in such circumstances to ensure this configuration is reproducible.

Copies of these programs are to be deposited in the CCP5 program library (see Appendix B).

6. RESULTS

We have timed these programs using a Lennard-Jones potential cut at $2^{1/6}\sigma$ and shifted by ε. The density ($\rho^* = \rho\sigma^D$) is 0.75 and the temperature ($T^* = kT/\varepsilon$) is 1.0. Results for a cutoff of 2.5σ are also presented for a two dimensional simulation using a domain decomposition into squares on the hypercube. The times presented are all attempts to give average time to complete one timestep of a molecular dynamics simulation in its production phase (i.e. integration, linked list creation, and force calculation, but not temperature scaling). we use the units of μs per atom per step to measure performance. This is equivalent to the slope of a graph of time per time-step against number of atoms. Smaller values of the slopes of these plots indicates better performance.

All the system sizes used were chosen to minimize the saw-tooth effect discussed in sections 2 and 3. The plots presented have lines drawn that are fitted to the points to stress the underlying linearity of the curves. It should be remembered that these lines do not predict the timings that one would obtain for an arbitrary number of atoms due to saw tooth effects. The starting configurations are all bcc lattices with each atom assigned velocities from a random uniform distribution. To ensure conservation of total linear momentum pairs of atoms are assigned equal, but opposite, random velocities.

The values of the total energy for similar systems agreed, within the bounds of reasonable error, between all the programs developed for this work.

6.1 The Cray X-MP programs

Two programs were timed on the Cray X-MP/416 at the Rutherford Appleton Laboratory in Oxfordshire. We performed simulations of two and three dimensional Lennard-Jones atoms using all of the four processors available. The machine has many users and usually has a substantial number of jobs executing and queued. The maximum allowed memory for a job is seven megawords out of a total of sixteen available. This situation is one where these programs would be expected to perform most poorly. However for larger system sizes the program still performs reasonably well. The multitasked routines are almost never required to wait for a lock to be cleared if we use systems containing more than 30,000 atoms. Only one in three multitasked subroutines cause the calling program to wait. Because these programs are swapped in and out of memory whilst they compete with other executing jobs the real time performance has proved impossible to measure. The nature of the jobs that the programs has to compete with can significantly affect the CPU times that are reported. However it has been observed that a moderately large three dimensional simulation program used 15 seconds of CPU time in 5 seconds of real time.

What we present in Figure 8 is one graph containing the timings obtained for both dimensionalities. We plot the total time taken, i.e. the sum of the times taken by the program and the multitasked subroutines, against the number of atoms, during a production phase of the simulation. This is obtained by performing two simulations. Firstly we equilibrate using temperature scaling for 50 time-steps. We then perform

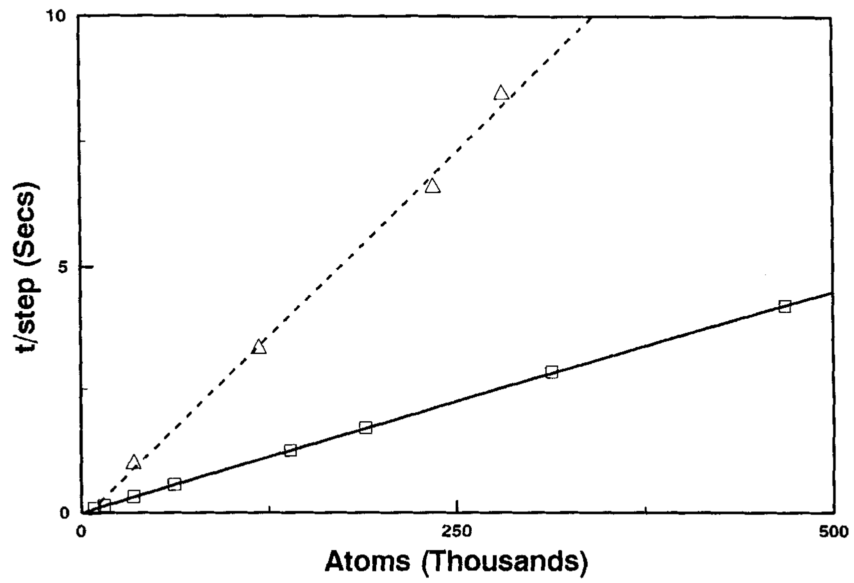


Figure 8 Total time per step against number of atoms for the X-MP multitasked programs, in two dimensions (solid line, squares), and three dimensions (dashed line, triangles).

a simulation of 50 equilibration, and then, 50 production steps. The time for 50 production steps is then estimated by difference between the two times obtained. Both of these curves are linear with a slope of 8.9 μs per atom in two dimensions, and a slope of 29.6 μs per atom in three dimensions. It can be assumed from the observation of these programs in execution that they may execute more than three times faster than this in real time. Due to the fact that we cannot vectorise the force routine these programs only report a megaflop rate, based on the total CPU time used, of approximately 14. This is substantially lower than the peak rate attainable by a single processor, when using its vector processing unit on a Cray X-MP, indicating that there is considerable room for improvement in performance if the force routine can be vectorised efficiently.

6.2 The Intel iPSC/2 programs

The Intel iPSC/2 used consisted of 64 nodes. Each node consists of an Intel 80386 processor with an 80387 maths coprocessor running at 16 MHz. In addition to this each node had a scalar accelerator known as SX, and 4 megabytes of memory. 32 of the nodes also possessed vector processors known as VX with 1 megabyte of associated memory.

Programs were timed for all possible domain decompositions in two and three dimensions. The SX hardware but not the VX hardware was used for all these tests. Timing these programs presents some problems. Each processor has a clock which we can access to determine the length of time it has spent executing our code. However these programs do not execute perfectly in parallel and there will always be some time when a processor will be idle waiting for another to complete a task. We therefore adopt the approach that the maximum of the times that each processor reports will be the best estimate of the real time that the program takes in performing the simulation. For sensible numbers of particles per processor this time seems to agree reasonably well with the elapsed real time. We equilibrate the system for 25 or more time-steps using temperature scaling prior to starting the clock. We then perform between 10 and 100 time-steps after which we stop the clock. This number of time-steps is far too short to produce any meaningful simulation results but here we are only interested in the performance. A few longer simulations were performed to test the scaling properties of the timings with number of time-steps used. These longer simulations conserved the total energy to 1 part in 10^4. All the timings presented here are for programs using single precision arithmetic (32 bit). Double precision arithmetic on the iPSC/2 is between 2 and 3 times slower. The use of single precision does not present a problem, it only requires a slightly smaller time-step to obtain energy conservation equivalent to that obtained using double precision. We note that the calculation of small fluctuations may require the use of double precision. Although there were vector processors attached to some of the processors on the hypercube these were not used. The gain in performance expected in using the vector complier with these programs will not be significant as we have not used a force routine that is readily vectorisible.

The results are presented using three different graphs. Firstly we plot CPU time per time-step against total number of atoms for each different processor configuration. For comparison the slopes of these lines are tabulated using units of μs per atom per time-step. Secondly we plot the CPU time per step against the average numbers of atoms on each processor. These curves should superimpose if the size of the processor netowrk does not affect performance. As an aid to comparison these two sets of curves are plotted using the same scales, for different decompositions of a particular dimensionality of system. Finally we plot the logarithm of the CPU time taken per step against the logarithm of the number of processors used for a constant total number of atoms. These curves should also be linear with a slope of -1 if the algorithm scales perfectly with the number of processors used. It is this measure that we use to define how well the performance of the programs scale with the number of processors used for a constant number atoms. Perfect scaling implies that doubling the numbers of processors used doubles the speed of execution of the simulation. This is not possible as the number of duplicated force calculations increases with the number of processors used. However this limiting value is approached when we have low values of N_p/N, and hence reasonably large systems sizes.

In general higher dimension domain decompositions performed much better. Some of the difference is probably due to the fact that some minimum imaging and periodic boundary calculations can be omitted from the higher dimension domain decompositions. This reduces the number of functions evaluated in the innermost parts of the force routine and hence increases its speed. It is interesting to note that all the programs developed performed better, on one processor, than the original code. This must be a result of the reduction of minimum imaging calculations. The performance trends are the same as those when the simulations are distributed over many processors, described in the following sections. The conclusion is that, when using one processor, there is much to be gained from creating a shell of particle images around the simulation box so that minimum imaging can be ignored in the force calculation. This method was employed in a vectorised link-cell algorithm presented in reference 8.

If we wish to maximise the number of atoms that we can simulate, the neighbour list which gives the indices, in the linked list, of a cell's neighbours can be omitted. It then becomes possible to simulate approximately twice as many atoms as the maximum reported in this work when using the same system parameters. A cell's neighbours then have to be calculated explicitly in the force routine. For a three dimensional system decomposed into sub cubes this increases the execution time by approximately 70% as compared to a program using the neighbour list.

6.2.1 iPSC/2 timings for 2 dimensional systems

Up to $2 \cdot 10^6$ atoms were simulated over 64 processors. The number of link-cells in each direction was large enough for us to be able to easily find numbers of atoms which minimised the saw tooth effect, even when using 64 strips. Figure 9 shows the timing curves for a decomposition into strips, whilst Figure 10 shows the curves obtained using squares. Figure 11 shows the result obtained using $r_c = 2.5\sigma$, $\rho^* = 0.75$ and $T^* = 1.0$ and a domain decomposition into squares. Table 1 shows the slopes of the curves in Figures 9(a) and 10(a). As can be seen the decomposition into squares performs significantly better than any decomposition into strips using the same number of processors. Table 2 shows the slopes of the curves in Figures 9(c) and 10(c). It can be seen that the decomposition into squares scales marginally better than the decomposition into strips. As we increase the number of processors committed to the problem we also see an increase in the minimum number of atoms we require per processor before the performance scales well. Both decompositions scale well for systems with approximately 500 atoms, or more, per processor when using the smallest processor arrays. When using 64 processors this lower bound increases to a minimum of 2,000 atoms. Below this number of atoms we also see a breakdown in the correspondence between the real time of execution and the maximum CPU time taken by the processors.

The time taken should scale as the cutoff to the power of the dimensionality of the system. Using this relationship we would expect the timings for the systems using a cutoff of 2.5σ to be a factor of $2.5^2/2^{1/3} \approx 5$. However if we consider the bcc lattice we start from we see that 12 atoms are within a cutoff of 2.5σ whilst 4 are just within a cutoff of $2^{1/6}$. The ratio of 12 to 4 is much closer to the value of ≈ 2.85 which we actually observe. We need to increase the cutoff just a little before we find the next shell of eight neighbouring atoms are within the cutoff and we would then increase the execution time by a factor of 5.

6.2.2 iPSC/2 timings for 3 dimensional systems

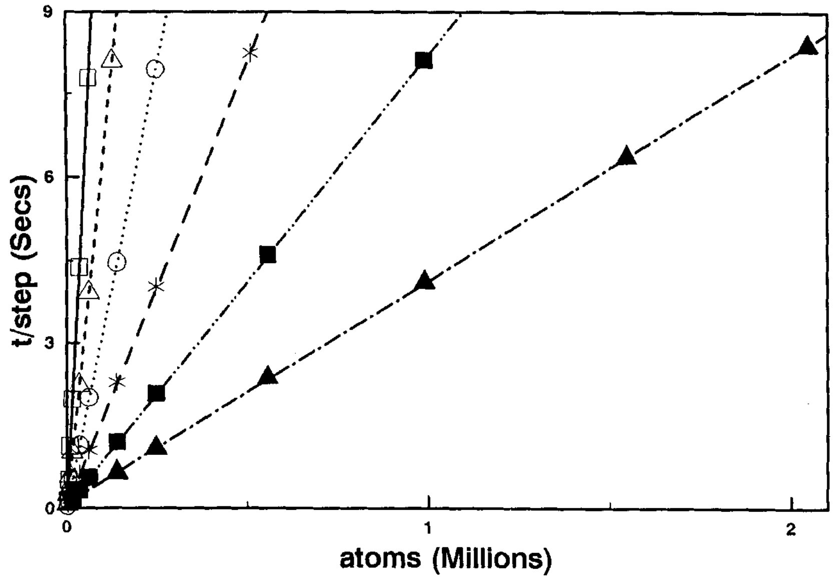
Up to 10^6 atoms were simulated over 64 processors. The maximum number of link-cells along a side of the simulation box was 100. Memory requirements meant that it was not possible to simulate large enough systems for a decomposition into 32 or 64 slabs without large saw-tooth effects or excessivley large values of R_1 (see appendix A and Figure 3). These programs performed particularly poorly and the results are not presented. The decompositions into prisms and sub cubes perform much better. Figure 12 shows the curve for the prisms decomposition and Figure 13 shows the curves for the sub cubes decomposition. Table 3 shows the slopes of the curves in Figures 12(a) and 13(a). The sub cubes program was over 40% faster than the prisms for similar simulations. Table 4 shows the slopes obtained from Figures 12(c) and 13(c). These show that the scaling for the decomposition into sub cubes only becomes better than that for a decomposition into prisms for the larger system sizes. The scaling is also poorer than that obtained for both the decompositions used in two dimensions. We also note a similar increase in the minimum number of atoms, per processor, required before the scaling is reasonably linear.

Table 1 Slopes obtained from curve fits to Figure 9(a) and 10(a).

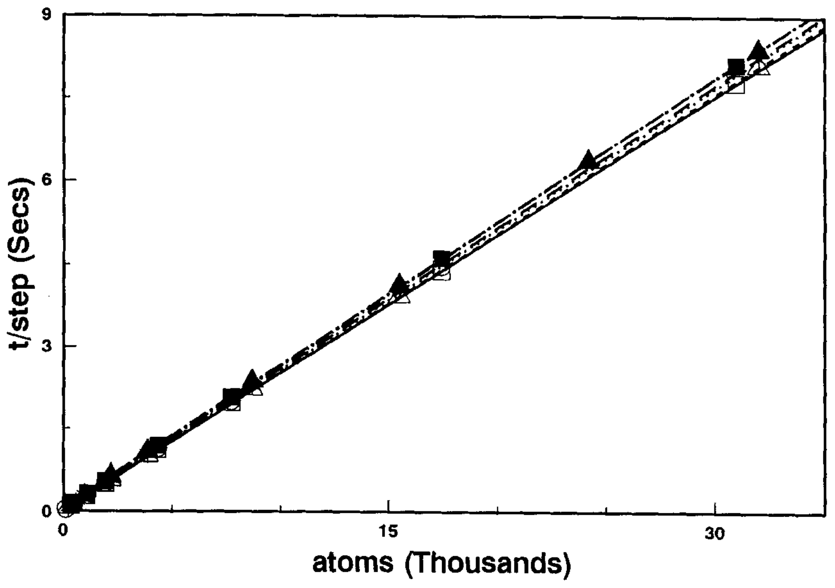
Number of Processors	*Strips (Fig. 9(a))* (/μs)	*Squares (Fig. 10(a))* (/μs)
2	125	-
4	63.0	41.3
8	31.9	20.7
16	16.0	10.3
32	8.1	5.1
64	4.0	2.5

Table 2 Slopes obtained from curve fits to Figures 9(c) and 10(c).

Number of atoms	*Strips (Fig. 9(c))*	*Squares (Fig. 10(c))*
3872	− 0.891	− 0.776
8712	− 0.931	− 0.919
15488	− 0.928	− 0.942
34848	− 0.953	-
61952	− 0.965	− 0.983
128018	-	− 0.989
247808	− 0.963	− 0.998

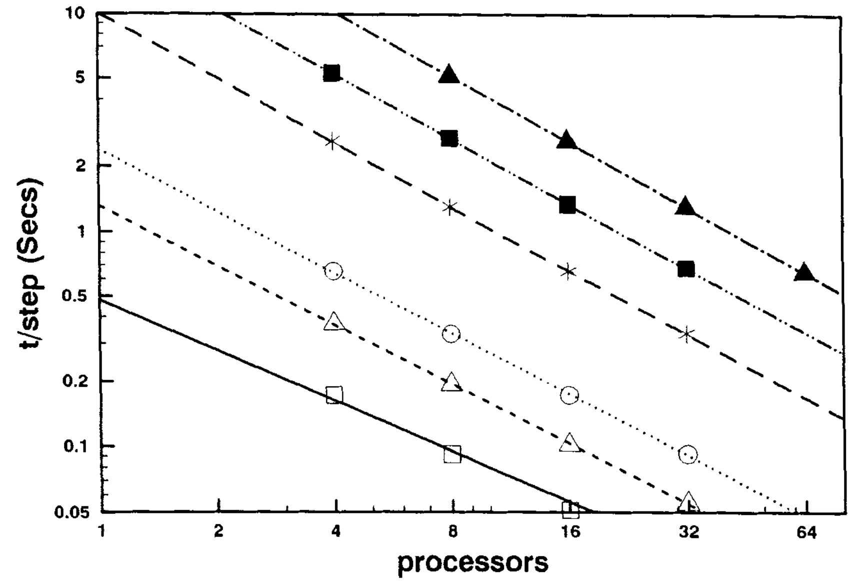


(a)



(b)

Figure 9 Timings form Intel iPSC/2 for two dimensional system decomposed into strips using 2(solid line, squares), 4(dashed line triangles), 8(dotted line, circles), 16(long dashed line, asterisks), 32(dot-dot-dashed line, solid squares) and 64(dot-dashed line, solid triangles) processors. $T^* = 1.0$, $r_c = 2^{1/6}\sigma$, and $\rho^* = 0.75$. (a) Time per step against total number of atoms. (b) Time per step against average number of atoms per processors. (c) Logarithm of time per step against logarithm of number of processors used for 3872(solid line, squares), 8172(dashed line, triangles), 15488(dotted line, circles), 34848(long dashed line, asterisks), 61952(dod-dot-dashed line, solid squares), and 247808(dot-dashed line, solid triangles) atoms.



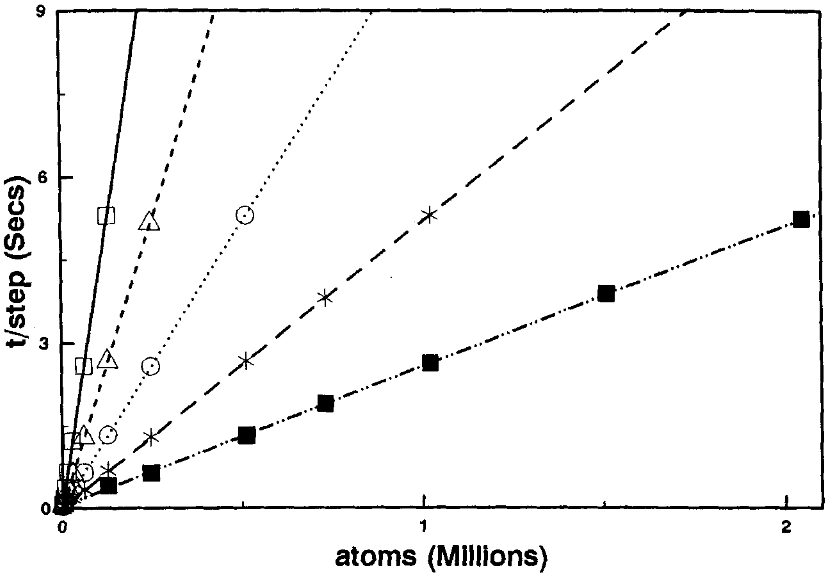
(c)

Figure 9 (continued)

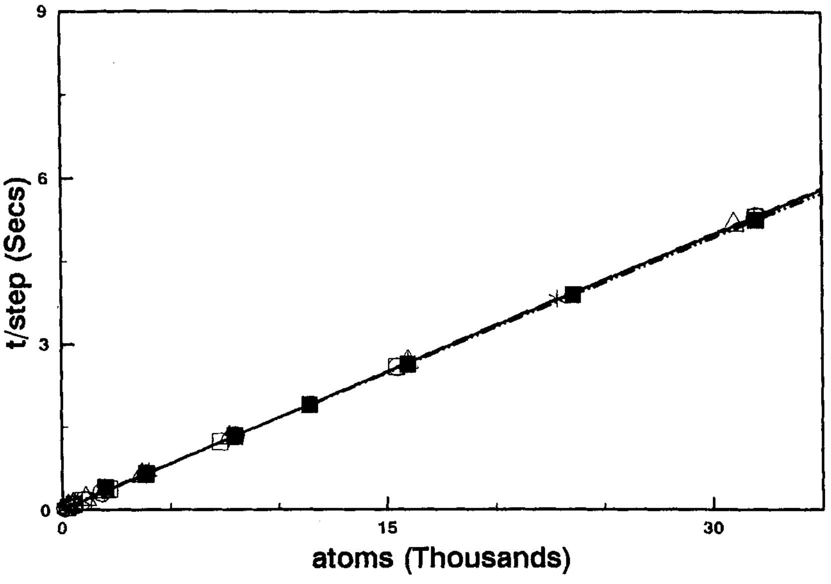
The timings for the sub-cubes decomposition using 4 by 2 by 2, and 4 by 4 by 2, processor arrays are not in agreement with those using an equal number of processors in each coordinate direction. This can be explained by calculating the value of R_3 for these systems. We see that the decompositions into non-cubic sub-cubes have slightly larger values, and hence, are doing more duplicated force calculations.

7. CONCLUSIONS

We have established that the link-cell algorithm can be used on a range of parallel machines to study the MD of molecules in two and three dimensions. The performance of this algorithm on a relatively cheap parallel machine such as the Intel iPSC/2 is comparable with that of much more expensive super computers such as the Cray X-MP. For example the performance of the three dimensional simulations on the Cray X-MP is 29.6 μs per atom per time-step. This may a factor of three, or more, better from observations of the programs whilst executing. The iPSC/2 achieves a performance of 7.5 μs per atom per time-step using sub-cubes on sixty-four processors. When using distributed memory systems such as the iPSC/2 higher dimensional decompositions have been shown to be preferable both in terms of choice of system sizes, and more efficient in terms of speed and execution. Such decompositions perform best if the domains are as close to cubic in three dimensions and as close to square in two dimensions. The algorithm performs well when the range of the force is much smaller than the size of the box. All the decompositions used perform best when there is a uniform density distribution. A non-uniform density distribution may require a more complex domain decomposition which allocates domains of differing to each processor. Care needs to be taken in the choice of system size, because of the saw tooth effect discussed in the paper, but it is easy to predict which system sizes will

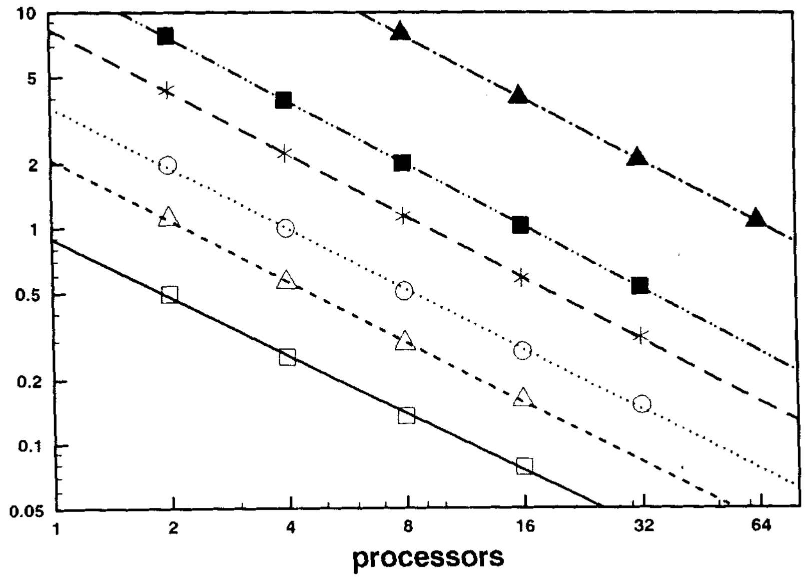


(a)



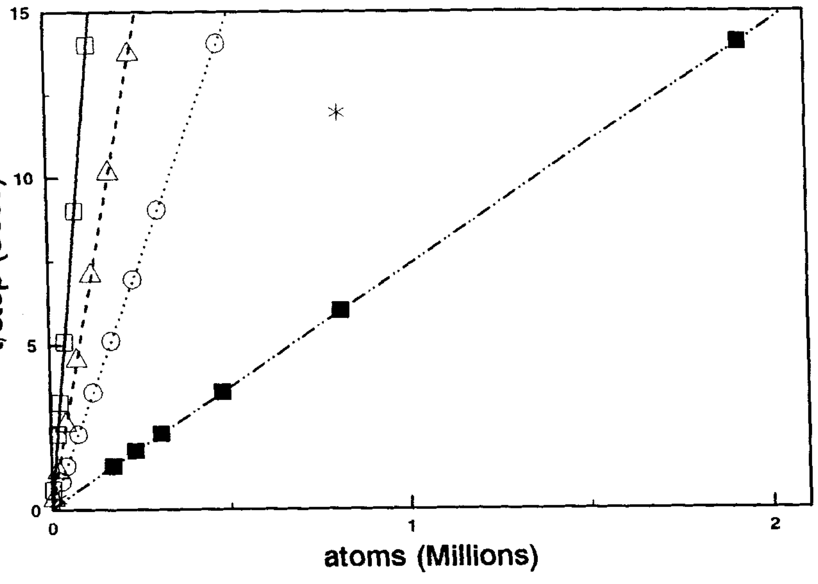
(b)

Figure 10 Timings from Intel iPSC/2 for two dimensional system decomposed into squares using 2 × 2(solid line, squares), 4 × 2(dashed line, triangles), 4 × 4(dotted line, circles), 8 × 4(long dashed line, asterisks) and 8 × 8(dot-dot-dashed line, solid squares) processors. $T^* = 1.0$, $r_c = 2^{1/6}\sigma$, and $\rho^* = 0.75$. (a) Time per step against total number of atoms. (b) Time per step against average number of atoms per processor. (c) Logarithm of time per step against logarithm of number of processors used for 3872(solid line, squares), 8172(dashed line, triangles), 15488(dotted line, circles), 61952(long dashed line, asterisks), 128018(dot-dot-dashed line, solid squares), and 247808(dot-dashed line, solid tirangles) atoms.



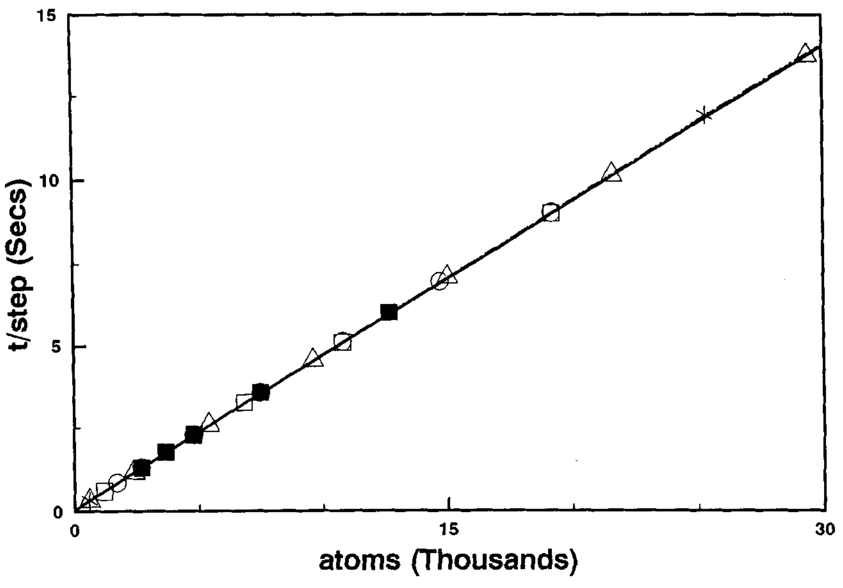
(c)

Figure 10 (continued)

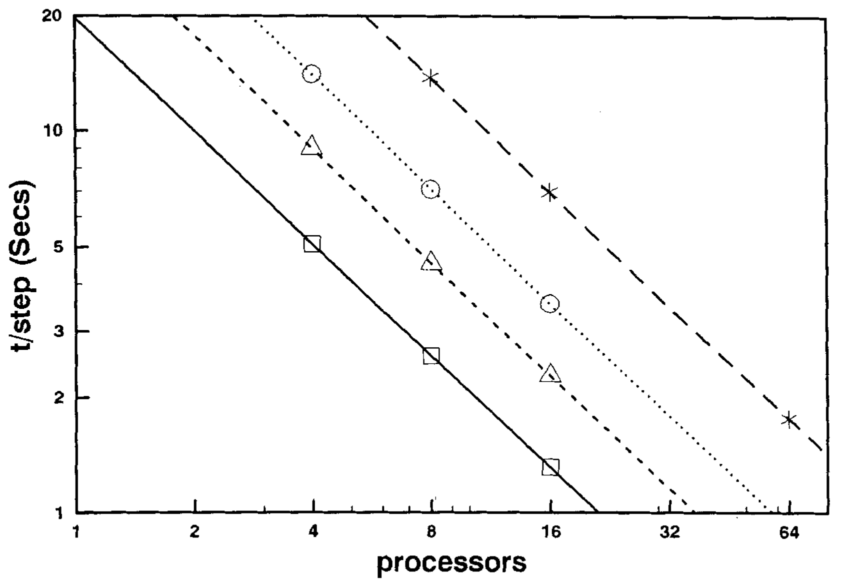


(a)

Figure 11 Timings from Intel iPSC/2 for two dimensional system decomposed into squares using 2×2(solid line, squares) 4×2(dashed line, triangles), 4×4(dotted line, circles), 8×4(asterisk) and 8×8(dot-dot-dash line, solid squares) processors. $T^* = 1.0$, $r_c = 2.5\sigma$, and $\rho^* = 0.75$. (a) Time per step against total number of atoms. (b) Time per step against average number of atoms per processor. (c) Logarithm of time per step against logarithm of number of processors used for 48218(solid line, squares), 76832(dashed line, triangles) and 120050(dotted line, circles), and 235298(long dashed line, asterisks) atoms.



(b)



(c)

Figure 11 (continued)

offer optimum performance. The algorithm only requires one synchronization point per time step, which in general is unavoidable, and this seems to be preferable to any algorithm which may require more. The programs used have several synchronisation points, however all but one could be removed by using asynchronous communications.

It has become clear that the creation of a shell of images around the simulation box

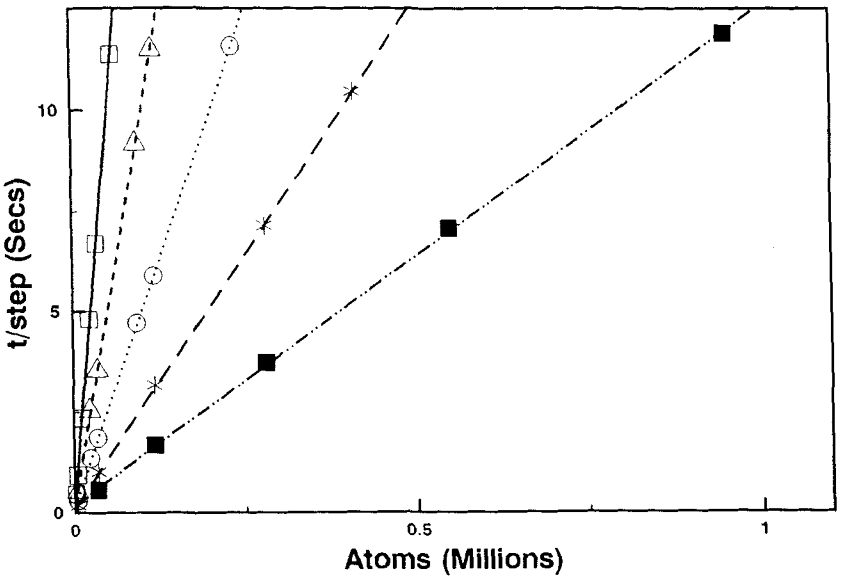
Table 3 Slopes obtained from curve fits to Figures 12(a) and 13(a).

Number of Processors	*Prisms (Fig. 12(a))* (/μs)	*Sub cubes (Fig. 13(a))* (/μs)
4	189.0	-
8	96.0	58.6
16	48.6	35.6
32	25.1	17.8
64	12.3	7.5

Table 4 Slopes obtained from curve fits to Figures 12(c) and 13(c).

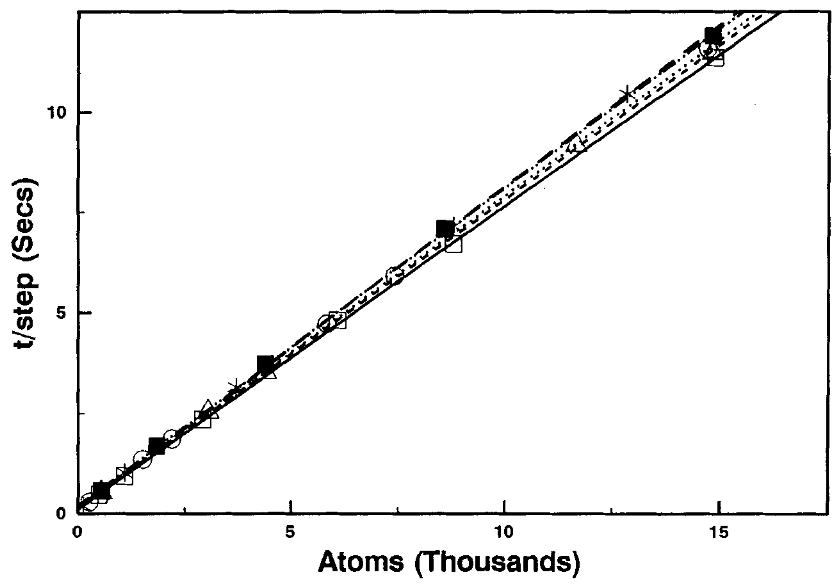
Number of atoms	*Prisms (Fig. 12(c)*	*Sub cubes (Fig. 13(c))*
4394	– 0.809	– 0.752
24334	– 0.917	-
35152	– 0.896	– 0.899
118638	– 0.925	– 0.96

can offer performance improvements even when only one processor is used. The reason for this is that the minimum imaging convention can then be ignored during the force calculation. The improvement that this offers will depend on the ratio of the time taken to perform the minimum imaging and the total amount of work performed during each iteration of a force loop. The most significant improvements will be

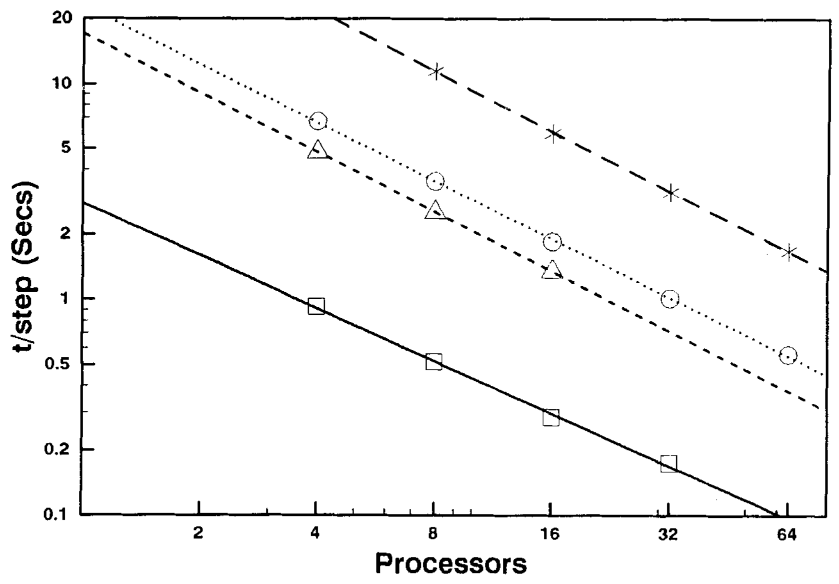


(a)

Figure 12 Timings from Intel iPSC/2 for three dimensional system decomposed into square prisms using 2 × 2(solid line, squares), 4 × 2(dashed line, triangles), 4 × 4(dotted line, circles), 8 × 4(long dashed line, asterisks) and 8 × 8(dot-dot-dashed line, solid squares) processors. $t^* = 1.0$, $r_c = 2^{1/6}\sigma$, and $\rho^* = 0.75$. (a) Time per step against total number of atoms. (b) Time per step against average number of atoms per processor. (c) Logarithm of time per step against logarithm of number of processors used for 4394(solid line, squares), 24334(dashed line, triangles) and 35152(dotted line, circles), and 118638(long dashed line, asterisks) atoms.



(b)

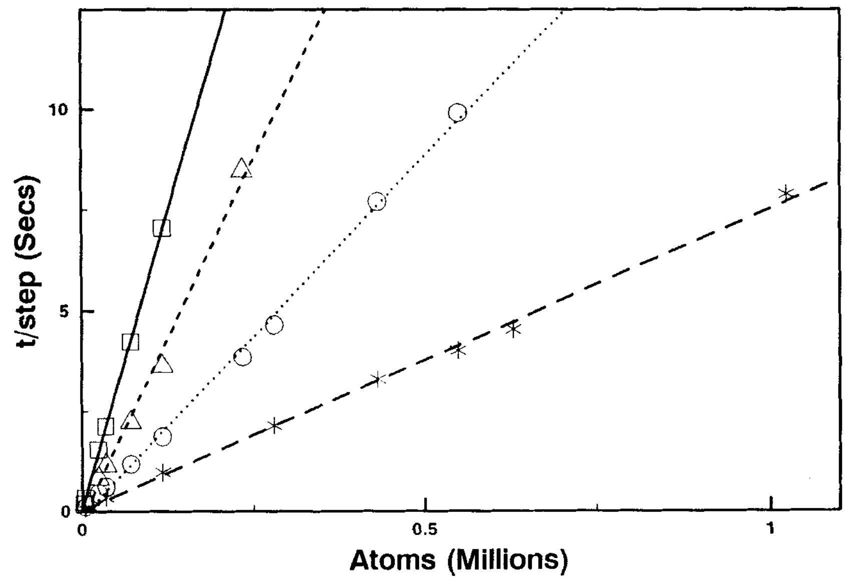


(c)

Figure 12 (continued)

obtained when using this ratio is high, i.e. when using a simple force calculation. With the omission of the array containing the map of neighbouring cells, in excess of $3 \cdot 10^6$ atoms, in three dimensions, can be simulated when using the programs developed. The cost of such an omission is a degradation in performance of 70%. Such a simulation would execute 10,000 time-steps in 110 hours or approximately five days.

(a)



(b)

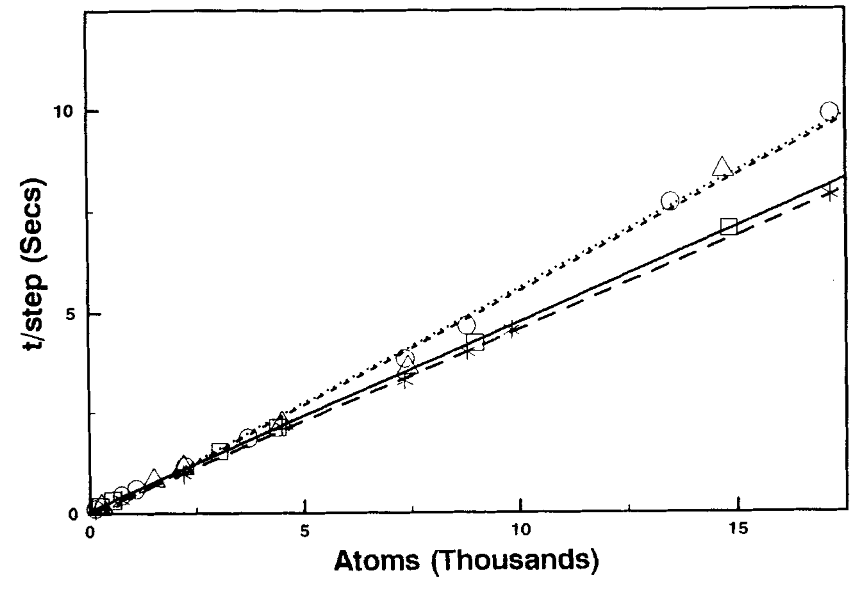
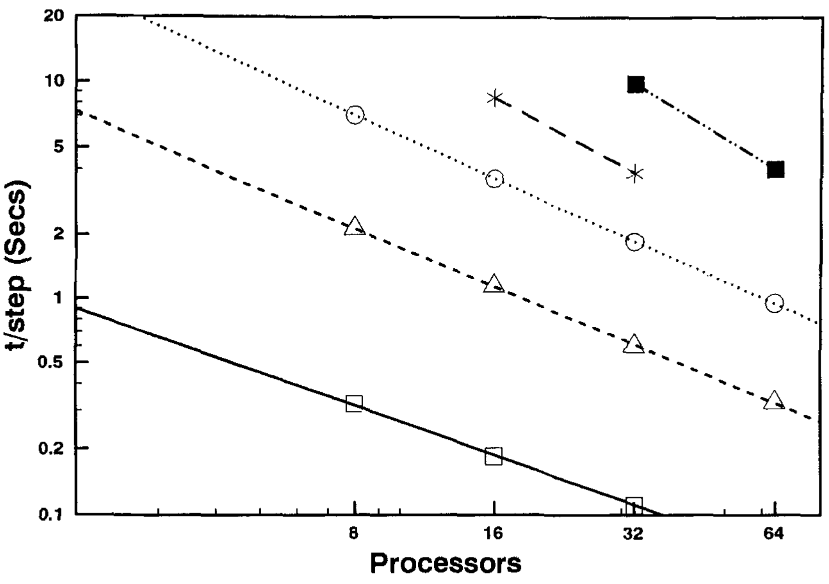


Figure 13 Timings from Intel iPSC/2 for three dimensional system decomposed into sub-cubes using $2 \times 2 \times 2$(solid line, squares), $4 \times 2 \times 2$(dashed line, triangles), $4 \times 4 \times 2$(dotted line, circles) and $4 \times 4 \times 4$(long dashed line, asterisks) processors. $T^* = 1.0$, $r_c = 2^{1/6}\sigma$, $\rho^* = 0.75$. (a) Time per step against total number of atoms. (b) Time per step against average number of atoms per processor. (c) Logarithm of time per step against logarithm of number of processors used for 4394(solid line, squares), 35152(dashed line, triangles) and 118638(dotted line, circles), 235298(long dashed line, asterisks), and 549250(dot-dot-dashed line, solid squares) atoms.



(c)

Figure 13 (continued)

When using the neighbour list a similar simulation of 10^6 atoms would take 21 hours or approximately one day. The amount of memory available is the limiting factor which decides the maximum system size that one can simulate on the iPSC/2 using these programs. Asynchronous communications may improve the performance of the programs when small numbers of atoms or large processor arrays are being used. It should also be considered when porting the program onto a different parallel machine such as one based upon Transputers. The functions of the iPSC/2 controller programs can be incorporated into the worker programs, if required, for use on computer systems where there is no host processor.

These programs make possible the simulation, in reasonable timescales, of the large system sizes required for several problems. The domain boundaries formed when a monolayer of krypton is at a density just above that of its commensurate solid structure required simulations of between 20,000 and 160,000 atoms [9]. These simulations used a two dimensional model of the krypton atoms in the monolayer. The study of fluid flow past a fixed boundary used larger system sizes of up to 270,000 atoms in two dimensions [10]. Examples of eddy formation and vortex shedding were observed in these simulations. It is a surprising result that these relatively small system sizes showed the major features of flow on a macroscopic scale. It should be noted that both these examples made use of the link-cell algorithm and some of the fluid flow simulations were performed on parallel machines. The simulation of polymer melts requires large system sizes because of the size of the molecules being simulated but also requires long simulation time due to the exceptionally slow time evolution of such systems [11]. The simplest models of such systems use only short range interactions and can easily be simulated using the algorithm presented in this paper. A simulation of 10^6 Lennard-Jones atoms has been performed in a study of crystalline formation when such a fluid is cooled [12]. Simulations of this type must be as long as possible. This is due to the fact that even the slowest cooling rates that can be simulated are several orders of magnitude faster than those that can be attained experimentally.

Recently the successor to the iPSC/2, the iPSC/860, has been announced. This new machine has the same architecture as the iPSC/2 but the processors on each node are considerably more powerful. We fully expect that the programs presented here will execute on this machine, with little modification, at even higher levels of performance.

Acknowledgements

MP would like to thank the SERC for a studentship and CCP5 for sponsoring a collaboration with Daresbury Laboratory.

APPENDIX A

Here we derive the equations from which we obtain N_p/N the ratio of number of atoms passed to the total number controlled by a processor. First we note that these ratios are independent of the dimensionality of the system, for example, similar decompositions into strips in two dimensions and slabs in three dimensions give the same ratio. Secondly a D dimensional system has D possible decompositions and a corresponding equation for N_p/N for each of them. We therefore only consider the three ratios for a three dimensional system and call them R_1, R_2, and R_3. Note however that for a D dimensional system the values of N increases as the power of D, and N_p increases as the power of D-1, with respect to the linear dimensions of the system.

We assume that the corner edge cells that we have to search in the prisms and cubes decompositions will, on average, be empty and that the distribution of atoms is uniform over the simulation box. The number of atoms passed is then proportional to the number of full cells passed. To simplify matters we also assume that we have a cubic simulation box with an equal number M of link-cells in each direction. If we have P_t processors with P'_α processors assigned to the α direction. The number of primes denotes the dimensionality of the domain decomposition that we are using i.e 1 prime means a 1 dimensional decomposition into slabs (or strips). The number of link-cells m_α on each processor in the α direction is given by

$$m_\alpha = \frac{M}{P_\alpha^D} \tag{A.3}$$

In addition we have two extra edge cells that we have to add along this direction giving a total of $m_\alpha + 2$ cells and we assume that M is an integral multiple of P_α^D. Along directions β where we have not performed a decomposition we have M cells.

If there are P_t processors then each processor controls M^3/P_t cells containing real atoms.

We now consider the decomposition into slabs along the x direction. Each processor controls a link-cell structure of dimensions $m_x + 2$ by M by M. At each timestep we pass the information from $2M^2$ cells. Therefore

$$R_1 = \frac{2 \cdot P_t}{M} \tag{A.4}$$

Next we consider the decomposition into prisms using a processor array of P_x'' by P_y'' where

$$P_t = P_x'' \cdot P_y'' \tag{A.5}$$

Each processors now controls a $m_x + 2$ by $m_y + 2$ by M link-cell structure. If we pass along the x direction first then we pass the contents of $2m_y M$ cells. The second phase of passing in the y direction involves the contents of $2M(m_x + 2)$ cells and we obtain

$$R_2 = \frac{2 \cdot m_y \cdot M + 2 \cdot M \cdot (m_x + 2)}{\frac{M^3}{P_t}} \tag{A.6}$$

$$= \frac{2 \cdot M \cdot P_x'' + 2 \cdot P_t \cdot \left(\frac{M}{P_x''} + 2\right)}{M^2}$$

We may have several choices of P_x'' and require the one that minimises N_p/N. Differentiating with respect to P_x'' gives

$$\frac{\partial R_2}{\partial P_x''} = \frac{2 \cdot \left[1 - \frac{P_t}{P_x'' \cdot P_x''}\right]}{M} \tag{A.7}$$

Setting this to zero and rearranging to give P_x''

$$P_x'' = \sqrt{P_t} \tag{A.8}$$

From this we see that P_x'' and P_y'' must be as close to each other as possible. In addition the size of the data packets passed is minimised if the first phase of data passing is along the direction with more processors assigned to it.

Finally consider a decomposition into sub-cubes with a P_x''' by P_y''' by P_z''' array of processors and

$$P_x''' \cdot P_y''' \cdot P_z''' = P_t \tag{A.9}$$

A processor's link-cell structure is now $m_x + 2$ by $m_y + 2$ by $m_z + 2$. If the order of edge passing is along x, y, and then z, the first phase of data passing involves $2\,m_y m_z$ cells. The second phase involves $2(m_x + 2)m_z$ cells and the last $2(m_x + 2)(m_y + 2)$ cells. Therefore

$$R_3 = \frac{2 \cdot m_y \cdot m_z + 2 \cdot (m_x + 2) \cdot m_z + 2 \cdot (m_x + 2) \cdot (m_y + 2)}{\frac{M^3}{P_{\text{tot}}}} \tag{A.10}$$

$$= \frac{2 \cdot [M^2 \cdot (P_x''' + P_y''' + P_z''') + 2 \cdot M \cdot (P_x''' \cdot P_y''' + P_x''' \cdot P_z''' + P_y''' \cdot P_z''') + 4 \cdot P_t]}{M^3}$$

The same rules about the relative sizes of P_x''', P_y''' and P_z''' apply as for the decomposition into prisms and the optimum value of P_α''' is $P_t^{1/3}$. For example a 4 by 4 by 4 array is better than an 8 by 4 by 2 array, which in turn is better than a 2 by 4 by 8 array.

We now consider the relative sizes of these ratios. It can be shown that

$$\frac{R_1}{R_2} = \frac{P_t}{P_x'' + P_y'' + 2 \cdot \frac{P_t}{M}} \tag{A.11}$$

$$\frac{R_1}{R_3} = \frac{P_t}{P_x''' + P_y''' + P_z''' + 2 \cdot \frac{(P_x''' \cdot P_y''' + P_x''' \cdot P_z''' + P_y''' \cdot P_z''')}{M} + 4 \cdot \frac{P_t}{M^2}} \tag{A.12}$$

$$\frac{R_2}{R_3} = \frac{P_x'' + P_y'' + 2 \cdot \frac{P_t}{M}}{P_x''' + P_y''' + P_z''' + 2 \cdot \frac{\{P_x''' \cdot P_y''' + P_x''' \cdot P_z''' + P_y''' \cdot P_z'''\}}{M} + 4 \cdot \frac{P_t}{\mathrm{M}^2}} \tag{A.13}$$

If we then use the optimum values for P_x'' and P_x''' we obtain

$$\frac{R_1}{R_2} = \frac{P_t}{2 \cdot \left(\sqrt{P_t} + \frac{P_t}{M}\right)} \tag{A.14}$$

$$\frac{R_1}{R_3} = \frac{P_t}{3 \cdot P_t^{1/3} + 6 \cdot \frac{P_t^{2/3}}{M} + 4 \cdot \frac{P_t}{M^2}} \tag{A.15}$$

$$\frac{R_2}{R_3} = \frac{2 \cdot \left\{\sqrt{P_t} + \frac{P_t}{M}\right\}}{3 \cdot P_t^{1/3} + 6 \cdot \frac{P_t^{2/3}}{M} + 4 \cdot \frac{P_t}{M^2}} \tag{A.16}$$

Taking the limit as M gets large

$$\frac{R_1}{R_2} = \frac{\sqrt{P_t}}{2} \tag{A.17}$$

$$\frac{R_1}{R_3} = \frac{P_t^{2/3}}{3} \tag{A.18}$$

$$\frac{R_2}{R_3} = \frac{2}{3} \cdot P_t^{1/6} \tag{A.19}$$

remembering that P_t must be integral. We see that higher dimensional decompositions minimise the amount of information passed as edges and hence the amount of duplicated force calculation.

Note with the larger three dimensional simulations reported here we can not assume the large M limit equations except for R_2/R_3. The largest M used was 100 with a 64 processor array. With a value of M of 100 the $1/M$ terms still contribute significantly to the ratios. In the case of R_2/R_3 these terms cancel out and for $M = 100$, R_2/R_3 is ≈ 1.33 which is still close to 4/3. However R_1/R_2 is 3.7 not 4 and R_1/R_3 is 4.93 and not 16/3.

What we have shown here is that to minimise both the number of duplicated force calculations, and the volume of communications, we must minimise the surface to volume ratio in three dimensions, or the perimeter to area ratio in two dimensions, of the domain used. This means that in three dimensions the optimum domain shape is cubic, and in two dimensions the optimum domain shape is square. This is true even when the simulation space is non cubic.

APPENDIX B

The following programs have been deposited in the CCP5 program library.

MDPLLC3SL a Cray X-MP pogram in three dimensions using Lennard-Jones atoms and a decomposition into slabs.

MDPLLI2SQ an iPSC/2 program in two dimensions using Lennard-Jones atoms and a decomposition into squares.

MDPLLI3CU an iPSC/2 program in three dimensions using Lennard-Jones atoms and a decomposition into sub-cubes.

As far as we are aware, the programs work correctly, but we can accept no responsibility for the consequences of any errors, and would be grateful to hear from you if you find any. You should always check a program or subroutine for your particular application. The programs are written in FORTRAN-77 with some additional machine specific functions and subroutines that are necessary for parallel execution of the code. There are some explanatory comments included in the code.

References

[1] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, 'Equation of state by fast computing machines', *J. Chem. Phys.*, **21**,1087, (1953).
[2] D. Fincham, *Computer modelling of fluids, polymers and solids*, (ed. C.R.A. Catlow, S.C. Parker, M.P. Allen), p. 269. NATO ASI series, Kluwer academic publishers, London.
[3] A.R.C. Raine, D. Fincham, W. Smith, 'Systolic loop methods for molecular dynamics simulation using multiple Transputers', *Comput, Phys. Commun*, **55**, 13, (1989).
[4] M.P. Allen, D.J. Tildesley, *Computer simulation of liquids*, (Clarendon Press, Oxford, 1987).
[5] G.S. Grest, B. Dünweg, K. Kremer, 'Vectorised link cell FORTRAN code for molecular dynamics simulations of large numbers of particles', submitted for publication.
[6] D.C. Rapaport, 'Large scale molecular dynamics simulation using vector and parallel computers', *Comput, Phys. Reports*, **9**, (1988).
[7] H.G. Petersen, J.W. Perram, 'Molecular dynamics on Transputer arrays. I. Algorithm design, programming issues, timing experiments and scaling projections', *Mol. Phys.*, **67**, 849, (1989).
[8] P. Garrett, 'Abstract machines for scientific processing', Supercomputer, **90**, North Holland.
[9] F.F. Abraham, W.E. Rudge, D.J. Auerbach, S.W. Koch, 'Molecular dynamics simulations of the incommensurate phase of Krypton on Graphite using more than 100,000 atoms', *Phys. Rev. Lett.*, **52**, 445, (1984).
[10] D.C. Rapaport, 'Microscale hydrodynamics: discrete-particle simulation of evolving flow patterns', *Phys. Rev. A*, **36**, 3288, (1987).
[11] G.R. Grest, K. Kremer, 'Dynamics of polymers: A molecular dynamics approach', submitted for publication.
[12] H.C. Andersen, W. Swope, preprint.