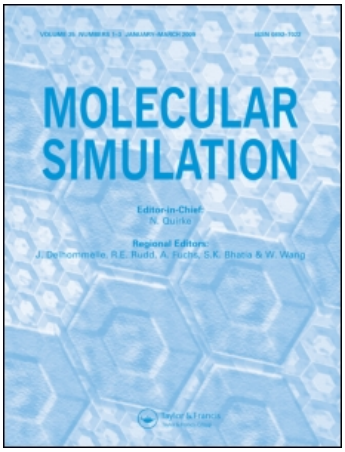

Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

Multicomputer Molecular Dynamics Simulation using Distributed Neighbour Lists

D. Fincham[ab]; P. J. Mitchell[c]
[a] Computer Centre and Physics Department, University of Keele, Staffordshire, U.K. [b] SERC Daresbury Laboratory, Warrington, U.K. [c] Computer Centre, University of Keele, Staffordshire, UK

To cite this Article Fincham, D. and Mitchell, P. J.(1991) 'Multicomputer Molecular Dynamics Simulation using Distributed Neighbour Lists', Molecular Simulation, 7: 3, 135 — 153
To link to this Article: DOI: 10.1080/08927029108022149
URL: http://dx.doi.org/10.1080/08927029108022149

Molecular Simulation, 1991, Vol. 7, pp. 135–153
Reprints available directly from the publisher
Photocopying permitted by license only

MULTICOMPUTER MOLECULAR DYNAMICS SIMULATION USING DISTRIBUTED NEIGHBOUR LISTS

D. FINCHAM

Computer Centre and Physics Department, University of Keele, Staffordshire ST5 5BG, U.K.
and
SERC Daresbury Laboratory, Warrington WA4 4AD, U.K.

P.J. MITCHELL

Computer Centre, University of Keele, Staffordshire ST5 5BG, UK

(*Received November 1990, accepted January 1991*)

A systolic loop method for parallel molecular dynamics, previously written in Occam, is reimplemented in Fortran using a general message-passing library. The method is extended to include a conventional neighbour list, fully distributed over the processors. A modification of the systolic loop method, called systolic replication, permits a parallel implementation of the fastest known sequential algorithm, in which a link-cell spatial decomposition is used to form the distributed neighbour list. The performance of each method, for a liquid argon simulation, is measured on a Transputer system and analysed in terms of the concept of scaleability. Some measurements are also made using i860 processors. The best performance achieved is 5000 particle moves per second, using 28 Transputers.

KEY WORDS: systolic loop method, distributed neighbour lists, transputer, molecular dynamics.

1. INTRODUCTION

A multicomputer is a parallel computer in which the individual processors have their own memory and run separate processes from that memory. They are thus computers in their own right, but are able to cooperate to solve a problem by passing data amongst themselves over some kind of network. Such communication events provide the only form of (loose) synchronisation between the processors. For many problems multicomputers are the most effective way of providing very high performance computing at modest cost. They are particularly suitable for molecular dynamics simulation which requires vast amounts of processor time but not usually the very large amounts of memory and very fast discs provided by more conventional super computers. Their low cost means they are affordable by individual research groups, and can therefore be easily integrated into a local computational environment with convenient access to facilities such as file store and workstation graphics.

In molecular dynamics simulation the main part of the computational task is the evaluation of interactions between pairs of particles. Normally the interaction poten-

tial has limited range, and locating interacting pairs can itself be a substantial problem except in the case of a solid material where spatial relationships between particles do not change. We use the word "particle" as a short-hand. It can represent a monatomic molecule or a small rigid molecule with a number of interaction sites or it can be an atom or interaction site within a large flexible molecule. We use the simple Lennard-Jones fluid as our prime example in this paper, but the principles apply equally in more complex cases.

There are, broadly, three ways in which multicomputers can be used for molecular dynamics simulation. They can run several independent conventional sequential simulation programs at the same time, and this can be extremely useful: for example, when there are a range of simulation projects being tackled at the same time, or when a particular project needs runs at a range of state points or with a range of potential parameters. (There may also be circumstances in which several independent simulations at a given state point can provide more efficient phase space sampling than a single long simulation). However, not all work is of this nature, and more often it is desirable to minimise the turnaround time for an individual simulation by distributing it over a number of processors. An obvious way to do this is to adopt a master/slave decomposition in which the master processor farms out the major computational tasks to a number of slave processors, and then gathers together their results. In the case of molecular dynamics the master, at the beginning of each time step, sends out the particle coordinates to all the slaves. These evaluate a subset of the pair interactions, accumulating them into force arrays. At the end of the time step the master gathers and sums these force arrays and then integrates the motion of all the particles. This is a very easy strategy if the main aim is to parallelise an existing sequential program as quickly as possible, since the master program is essentially the existing program minus its force routine, and the slave program is essentially the force routine. All that is required is to program in the coordinate distribution and force collection. However, there are two problems with this approach. First, the motion integration becomes the rate limiting step, since, however much the execution time for the force evaluation is reduced, the motion integration remains on a single processor. Second, and more seriously, communication of data into and out of the master processor becomes a bottleneck as the number of processors is increased. Any application which involves running a simulation on a large number of processors needs an approach which distributes the complete calculation, motion integration as well as force evaluation, evenly over the processors, and also minimises communication delays.

Such fully distributed parallel strategies for molecular dynamics fall into two classes. One approach is to adopt a problem decomposition which distributes the particles over the processors [1]. Every particle has a "home" processor which is responsible for its motion integration throughout the simulation. However, during a time step particles circulate round the processors in such a way that every particle meets every other particle so that their interaction can be found (if their separation is less than the range of the potential). These methods are usually described as "systolic loops". The other approach is to adopt a decomposition in which different processors control different regions of space, rather than different sets of particles [2]. In the case of a liquid simulation this means that a particular particle will move from processor to processor as it diffuses in space during the course of the simulation. Such a spatial decomposition is more suitable for systems of very large numbers of particles where the size of the system is many times the range of the potential, and the evaluation of interactions can be achieved by exchanging data between "nearby"

processors, rather than circulating round all the processors. Our own interests lie in the modelling of molecular liquids and ionic materials, where very large particle numbers are rarely required, and we concentrate on systolic loop methods. A previous paper by Raine, Smith and Fincham [3] analysed these methods in some detail, particularly from the point of view of load balancing and communication overheads. Testing was carried out on a Transputer system using the Occam programming language. The first aim of this paper, which is the subject of Section 2, is to discuss reimplementation in terms of a standard sequential language (Fortran) allied with a library of general message-passing routines. Again our testing is on a Transputer system, and we give performance measurements. The advantage of this approach, apart from the familiarity of Fortran, is that most multicomputers provide such a message-passing facility whereas Occam is specific to Transputers

In molecular dynamics simulation particles do not move very far during one time step, and it is not essential to locate neighbours afresh on every step. In Section 3 we show how it is possible to incorporate a conventional neighbour list within the systolic loop framework. Pair interactions are only evaluated for pairs appearing within the list, leading to a considerable performance improvement. An important point is that the memory-consuming neighbour list is fully distributed over the processors.

In Section 3 we also investigate an alternative method of constructing the neighbour list, based on the conventional link-cell algorithm. This algorithm is central to spatial decomposition methods, but here we use it within the systolic loop framework. To do this we need to provide every processor with a copy of all the particle coordinates, and show that this can be done in a method we call systolic replication which involves exactly the same amount of communication as the standard systolic loop method. The neighbour list itself is again fully distributed.

In our discussion, Section 4, we compare our systolic loop neighbour lists with the alternative spatial decomposition methods, from the point of view of performance and practicality. We also make some remarks about the incorporation of many-body forces.

2. FORTRAN IMPLEMENTATION OF SYSTOLIC LOOP METHODS

2.1 Hardware and software issues

Our programs are implemented in Fortran and run on our Meiko in-Sun Computing Surface with 28 Transputers using the "CSTools" software. This software has a number of interesting features which make its use quite different to that of Occam. It provides a set of utilities, running on the hosting Sun workstation under the Unix operating system, for compiling and loading Transputer programs. A user task can request any number of processors from one upwards, and is allocated an individual domain with the required number of processors: the rest of the processors are free to be allocated by the system to other tasks. The task is described by a configuration file which specifies the process to be run on each processor of the domain and the topology of the connections between the processors which are to be wired up before the task is loaded. Each process is a standard sequential language program. Usually the same program is loaded on each processor, though this is not essential. The process when it is running can call a library routine to enquire on which processor of the domain it is running, so that it can behave appropriately. Processes in the task can

communicate with each other by means of the Computing Surface Network (CSN). This is achieved by calling routines from a message-passing library. Individual processes can also read and write files, with all the passing of data through the Transputer network to the Sun being handled automatically. (It is particularly helpful for debugging purposes to be able to insert print statements in any process of the task, and a major advantage over Occam). There is also a symbolic debugger.

In Occam, direct communication between processes on different processors is only possible if they are directly connected together. If data is to be passed between processors which are not directly connected then the messages must be explicitly passed on by processes running on the intervening processors. As well as complicating the programming this makes the program specific to a particular hardware topology. Use of the CSN message-passing routines is much simpler, as a single call will send a message to any processor in the network, with data being passed across the network completely transparently as far as the application program is concerned. Communication is via "transports". A process can create transports which it can use to send or receive messages. If it wishes to receive messages on a particular transport it gives it a name which is registered centrally. The sending process then looks up this name and addresses messages to it. This ensures that the behaviour of a program is independent of the way in which the processors are wired together, which can affect only its performance.

A number of different types of communication are possible using the CSN. Communication may either be synchronous or asynchronous. A synchronous communication is not regarded as having been completed until the message has been received at its destination, and the sending process only knows this when an acknowledgement has been received. In asynchronous communication the sending process simply commits the message to the network and there is no acknowledgement. Clearly synchronous communication will be slower, but it is safer in the sense that it is not possible to saturate the network with messages that cannot be received. Thus synchronous communications may be used during program development, and asynchronous communications in production runs.

Communications can be initiated by two types of call to CSN routines, blocking and non-blocking. On a blocking call, control returns to the calling routine only when the data has been safely copied out of the program array (on a send) or copied into it (on a receive). During program development one would normally use synchronous communications in blocking form. However, for maximum performance it is desirable to use asynchronous communications and there can then be an advantage in the use of non-blocking communication calls. A non-blocking call returns as soon as the communication has been initiated, allowing the program to continue with other useful work while the communication is in progress. While a non-blocking communication is in progress it is of course not safe for the process to write to the data arrays in the case of a send, or read from them in the case of a receive. Wait routines are thus provided which can be called at the point in the program where it becomes essential to use the arrays, and these calls will not return until communication has completed. In the case of asynchronous sends the performance gain of non-blocking over blocking calls may not be very great, since a blocking call will return as soon as the program data has been copied out into system buffers. However, a blocking receive will not return until data has been received from the network, and the receiving process will be idle during this wait. Thus it is desirable, if the algorithm can be arranged suitably, to issue a non-blocking receive as early as possible, to then carry on with useful

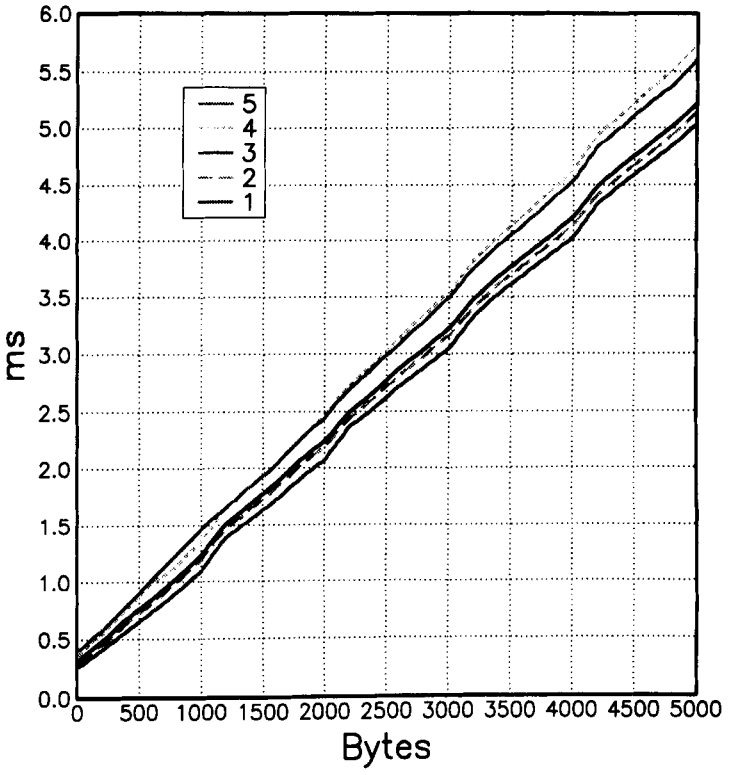

Figure 1 Communication times as a function of message length along a chain of Transputers. The labels indicate the number of links traversed.

calculations (or other communications), and issue a wait only when the data being received is required.

We have measured the performance of message-passing on our system and Figure 1 shows the time taken for messages of different lengths to be transmitted different number of steps along a chain of processors. The figure reveals some interesting features of the CSN message-passing system. Each graph is very linear, with a slope of about 1 Mbyte/s. However, the intercept on the vertical axis shows that there is a "start-up" time for a message between directly-connected processors of around 250 μs. This means that messages must be much longer than 250 bytes for the overall speed of transmission to approach the asymptotic rate of 1 Mbyte/s. Another interesting feature revealed by Figure 1 is the performance of message-passing between remotely connected processors. The asymptotic rate shown by the slope of the graphs is not noticeably less than that between directly connected processors, and the increase in start-up time is also not very great. This shows that the performance of parallel algorithms is unlikely to be very sensitive to the exact wiring of the hardware topology. The next generation of Transputer, and rival parallel processing systems, will have hardware support for general message-routing and such machines can effectively be regarded as fully-connected networks. The CSN system already emulates this in software. We have also been able to time communication rates on a Meiko

Computing Surface using i860 processors, and also on an Intel iPSC/860, which also uses i860 processors and a message-passing library. On the Meiko, communication rates are very similar to those given above: not surprising since this machine uses Transputers as communication engines. (However, this speed will increase in the near future as Meiko provide software to connect i860 processors by several Transputer links working in parallel). On the Intel, communication rates are currently approximately twice those on the Meiko machines, with a similar start-up time. Since the i860 is a much faster processor than the Transputer (about eight times on our programs), the machines based on it have a worse ratio of communication to calculation speeds, and applications are more likely to be communications bound. It should be mentioned that our performance tests have all been carried out with the processors doing nothing but communication. In principle the Transputer hardware allows communications to be carried out by the link "engines" almost completely independently of the main processor, and communication and calculation should therefore be able to continue simultaneously without either seriously affecting the performance of the other. However we have not tested how far this is true in practice using the CSN, except indirectly in some of the simulation methods described below.

The use of the CSN message-passing system does have an associated performance penalty. Between directly-connected processors using Occam we previously measured an asymptotic transmission rate of about 1.3 Mbyte/s with a very low start-up time of 15 μs. We have been able to match these figures in Fortran using routines provided by Meiko for direct data transmission over inter-Transputer links. To be able to use these, one must write one's own loading utility (a C program running on the host Sun) using a facility called "CSBuild". Since systolic circulation involves only nearest-neighbour communication it would be possible to use these routines in our programs to improve performance. However, we prefer to sacrifice this performance gain for the benefits of programming ease, portability and topology-independence provided by the CSN.

2.2 The SLS-G method

The methods we use throughout this paper are based on the SLS-G (Systolic Loop Single-Grouped) method of [3]. There is a single copy of the data for each particle which at the beginning of a time step is in the particle's home processor. During the time step groups of particles circulate around the processors in a series of "pulses" in the manner shown in Figure 2. The data circulating include particle coordinates and force accumulators, and possibly other information such as type indices, but not velocities. The "Head" processor has one group, and on each pulse it evaluates the intragroup interactions between particles in its current group. Each "Body" processor has two groups, and it evaluates the inter-group interactions between pairs of particles from different groups. At the end of the timestep each particle returns to its home processor but now its force accumulator contains the total force acting on it, so its motion can be integrated, ready for the next time step. This SLS pattern of data movement is the only one which ensures that each pair of particles meets up only once during the systolic circulation, and that each particle returns directly to its home processor after all the interactions have been evaluated. The computational work is distributed over all the processors except that Head has a special role: it has less work to do than the Body processors and so does not produce a computational bottleneck, and its spare capacity can be useful: for example, for output and graphics.

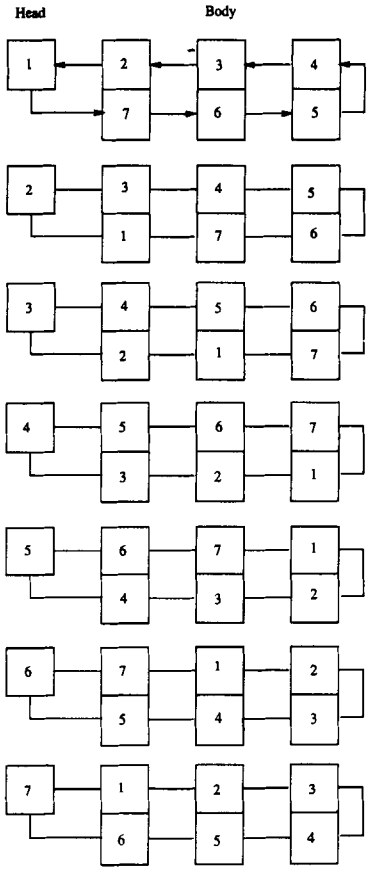

Figure 2 The pattern of data movement in the SLS parallel molecular dynamics methods.

The implementation of a Fortran program for the Lennard-Jones fluid proved to be straightforward using CSN routines. In the SLS-G algorithm each pulse begins with the evaluation of pair interactions: this is followed by the circulation of the coordinate and force data. There is no possibility of overlapping the calculation with the communication since the force arrays may not be passed on until evaluation of interactions is complete. However, non-blocking calls to the communication routines are still useful to enable the four communications required on each processor (send and receives on the "upper" and "lower" groups, see Figure 2) to be initiated as soon as possible and to proceed simultaneously. To perform the systolic circulation processes are connected in a chain and need to communicate with their two nearest neighbours, and so register two transport names to receive messages from these neighbours. These names are conveniently made up to incorporate processor numbers in character form. Processors allocated to a task are numbered sequentially from zero by the system, and we identify processor zero with Head. The same processor numbers are used in the configuration file which controls the wiring of the Transputer links. Thus we are able to physically wire the processors in a chain which matches the logical chain

of processes connected by transports. In principle, since each Transputer has four physical links it should be possible to use two links between neighbouring processors, but this is not possible with the current wiring hardware and software.

In molecular dynamics a small amount of information must be accumulated from all the processors (potential and kinetic energy etc.) and the relevant data is passed to Head. In the previous Occam programs this was laboriously sent along the chain of processors step by step, but here each process can address the data directly to Head. With a singly connected chain it is possible to use the "spare" links to reduce the physical distance from each processor to Head, and we have made use of this in another context [4], but here the amount of data involved is so small that this is hardly worthwhile.

Compared with our Occam programs the Fortran implementation is more flexible, with the possibility to have groups of different sizes, and for the program to run without change on any number of processors. We have also incorporated an option to evaluate radial distribution functions. This is easy since each processor can maintain its own histogram of pair separations, and these can be combined at the end of the run.

2.3 Scaleability analysis

In [3] the performance of various systolic loop methods was discussed in detail and timing formulae were given. We can summarise the discussion in general terms as follows. The first point concerns load balancing. If all the pair interactions in the system were to be evaluated the load balancing would be perfect, since the evaluation of interactions and the integration of particle motions is distributed evenly over the processors. (In this discussion we ignore the special role of the Head processor in the SLS methods: the other methods described do not have this complication). However, normally a spherical cut off is applied. Assuming that particles assigned to particular processors are randomly distributed in space, there will not be any systematic imbalance of load, but there will be statistical fluctuations in the number of in-range pairs on each processor on each pulse. These fluctuations will slow the calculation down since the processors are synchronised by message-passing at the end of each pulse, and other processors will have to waid for the one with the largest number of in-range interactions to evaluate. Analysis shows that this will have a negligible effect on performance providing the number of particles in a group is greater than about 10. A second point concerns the start-up time for communications. To overcome the effects of this messages must be sufficiently long, implying that groups must contain a sufficient number of particles. We have already seen that, using CSN routines, messages must be much longer than 250 bytes to approach asymptotic transmission rates. With 24 bytes to represent a particle (3 components of position and force, with 4 byte words) this requires groups to be much larger than 10 particles. Providing groups contain sufficient particles to mask the effects of statistical fluctuations and communication start-up times, the timing formulae in [3] have the general form of an execution time per time step, for N particles on p processors, proportional to

$$(\alpha N^2 + \beta N)/p + N + \gamma p \tag{1}$$

The first term represents the calculation time which is inversely proportional to p since the methods have been designed to distribute the calculation over the processors. The N^2 component arises from the need to test the separation of every pair before applying

the cut off, and the N component from the evaluation of in-range interactions and the integration of particle motions. The second term represents the communication time of the systolic circulation which is easily seen to be proportional to N and independent of p, since there are g pulses, where g is the number of groups ($g = 2p - 1$ for the SLS case), and each pulse involves the simultaneous transmission of groups each containing N/g particles. The third term represents the time to pass thermodynamic data from the other processors to Head, and accumulate it, and is proportional to the length of the chain, i.e. to p.

We would like to discuss this formula in terms of the concept of *scaleability*, which is often used in parallel computing. Ideally we would like the speed of execution, for fixed problem size, N, to be proportional to the number of processors, p. We call such a method *strongly scaleable*. This can never be achieved in practice for all values of N and p, since clearly if the number of processors exceeds the size of the problem some of them will be idle. We have seen in our case, and it is generally true, that performance will suffer if the ratio of problem size to processor number becomes too small. One can then introduce the concept of *weak scaleability*, which requires only that execution speed should be constant for a fixed ratio N/p. If a method is not scaleable, one can distinguish two cases. If the speed, for fixed N, tends to a constant as p increases, we call this *mildly non-scaleable*. If the speed tends to zero we call this *seriously non-scaleable*.

Turning to equation (1) we see that the first term alone would result in scaleable performance, since the calculation is distributed over the processors. However the second, communication, term will eventually dominate over the calculation term as the number of processors is increased. This will lead to a constant execution speed for fixed N, and so to a mildly non-scaleable regime. The third term is proportional to p and in theory will dominate the calculation for very large p, leading to a seriously non scaleable situation. However, the amount of data involved is very small and the term is entirely negligible for any reasonable number of processors, but it does illustrate an important point. The most serious problem for parallel computing with a large number of processors arises when there is a need to gather data from all processors at one point in the network, either for global accumulation or for output. Similarly, all algorithms based on a master/slave decomposition are liable to be seriously non-scaleable and hence very inefficient with more than a handful of processors.

2.4 The SLS-GO method

Can we avoid the communication term to give a more scaleable method? It was shown in [3] that it is possible to rearrange the systolic loop methods so that calculation and communication can be performed simultaneously. The SLS version of this method is called SLS-GO ("O" for "overlapped"). Briefly: while a processor is evaluating pair forces into a temporary array; it is also passing on the coordinates it is currently using and the forces from the *previous* pulse; at the same time it is receiving the coordinates it will require on the *next* pulse, and the force accumulators to which it will add the temporary forces when it has completed the current pulse. In Occam a PAR statement is used to initiate three processes running in parallel (calculation, and communications in each direction). We have implemented the same algorithm in Fortran/CSN using non-blocking communication routines. The communication calls are made at the beginning of the pulse and the process then continues with the calculation of interactions while the data is being transmitted. At the end of the calculation wait

routines are called so that the process will not continue unless and until the data transmissions are complete. Providing the total communication time is less than the total calculation time, one would expect to be able with this method to simply drop the communication term from equation (1), giving a scaleable algorithm.

2.5 Performance of the methods

The performance of the SLS-G and SLS-GO algorithms in terms of execution speed as a function of processor number is illustrated in Figure 3 for a 1000 particle system, and shows the expected features. For a small number of processors both algorithms execute at the same speed and with linear speed-up in the number of processors, showing that communication time is negligible in comparison with calculation time in this region. For large numbers of processors the SLS-G curve flattens off as the communication time remains constant while the calculation time decreases. On the other hand the SLS-GO algorithm speed continues to increase with processor number. However, there is some falling off compared with a linear speed-up and this is probably attributable to the effects of start-up time as the number of particles per processor becomes rather small.

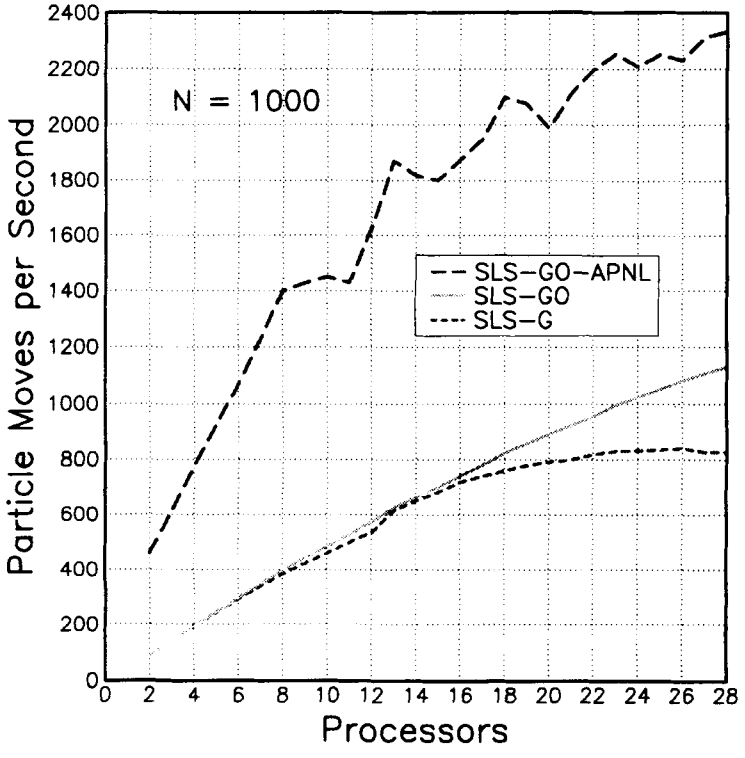

Figure 3 Performance of three algorithms as a function of number of Transputers. The test system is liquid argon with 1000 particles at a reduced state point of $T^* = 1.5$ and $p^* = 0.59$, with an interaction cut off of 2.5σ and a time step of 10 fs.

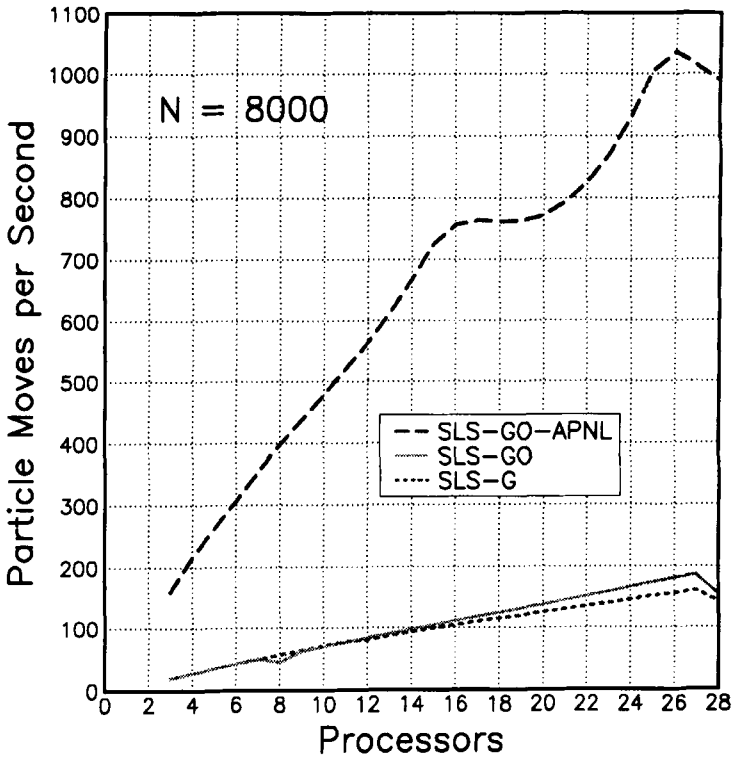

Figure 4 As figure 3 but 8000 particles. Note the change in vertical scale.

Figure 4 shows results for a system of 8000 particles. Because of the N^2 component in the calculation time communication delays have less relative effect in this case, and the number of particles per processor is also greater. The performance is accordingly more scaleable.

Figure 5 shows results for the SLS-G algorithm on a Meiko Computing Surface using i860 processors. The results for 1000 particles show non-scaleable behaviour, even with a small number of processors. This is to be expected because the processor speed is much greater without a corresponding increase in the communication rate. Results with 8000 particles are much better. The present preliminary software on this machine does not provide non-blocking communications so we have not been able to implement the SLS-GO algorithm.

3. NEIGHBOUR-LIST METHODS

3.1 All-pairs neighbour list

In a simple molecular dynamics program such as the one described in the previous section every pair of particles is examined on every timestep to see whether its separation is within the cut off distance representing the range of the interaction. It is not however necessary to make this test on every step, since particles do not move very far during a time step, and do not change many of their neighbours. Instead, one can make a list showing which particle pairs are separated by less than an outer cutoff distance somewhat larger than the interaction cutoff. In evaluating particle inter-

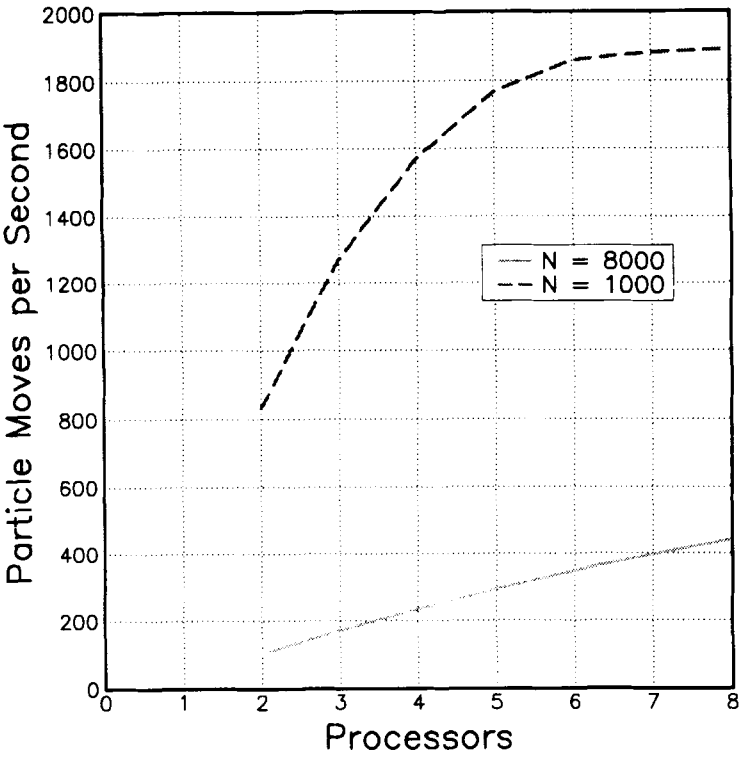

Figure 5 Performance of the SLS-G algorithm using i860 processors for 1000 and 8000 particles. Other parameters as in Figure 3.

actions it is then only necessary to loop over pairs taken from this list, discarding those separated by more than the interaction cutoff. The list must be revised periodically. To avoid missing any interactions this should be done when any pair separation may have changed by more than the difference between the outer and interaction cutoffs. A necessary condition for this is that the distance moved by at least one particle since the last list update should be greater than half the distance between the outer and interaction cutoffs. If the neighbour list is formed by examining all-pairs in the system we call this an *all-pairs neighbour list* (APNL).

We have implemented a parallel version of the APNL method within the systolic loop framework. Each processor deals with the same subset of pair interactions on each timestep. Thus during a list-forming time step it can make a list showing which of its own subset of pairs is within the outer cutoff. In this way the complete neighbour list is fully distributed over the processors. On subsequent time-steps each processor receives the coordinates of its pairs during the systolic circulation in the normal way, but only considers pairs if they appear within its list, and evaluates their interaction if their separation is within the interaction cutoff. The evaluation of interactions can be overlapped with the circulation of data as in the standard SLS-GO method. During the motion integration phase the processor monitors the total displacement of each of its home particles, accumulated since the previous list update. It determines the

number of particle displacements exceeding the critical value described above, and sends this number to Head, which sums the values. If the total is greater than zero, i.e. if any particle displacement in the system exceeds the critical value, Head signals to all processors that they should reform their neighbour lists on the next time step.

3.2 Performance of the method

The frequency of the list updates, and the overall speed of the method, depend on the choice of the outer cutoff. A larger value leads to less frequent updating, but more pairs listed that are outside the range of the interaction. Ideally one should experiment by varying the outer cutoff until the average speed of the program is maximised, but this is a tedious business which would need to be done for each choice of density, temperature, system size and timestep. We have adopted the choice of 3.0 σ for the outer cutoff (where σ is the Lennard-Jones diameter: the interaction cutoff is 2.5σ) and in our case this leads to an interval between list updates of around 10 time steps. The performance of this algorithm (SLS-GO-APNL) is also shown in Figure 3, for 1000 particles. It is faster than SLS-GO by a considerable factor, but less scaleable with processor number. This we attribute to communication delays: the use of the neighbour list has speeded up the interaction calculation on each pulse so much that it now takes less time than the systolic data circulation, and the overlapping is no longer complete. The "wiggles" in the graph are also a sign that communication effects are present: note the abrupt change in slope after eight processors, which is the number of processors on a single board in our machine; evidently communicating between boards introduces additional delays. With the 8000 particle system, Figure 4, the graph shows scaleable behaviour over a larger number of processors.

3.3 Link-cell neighbour-list method

On conventional computers simulations with very large numbers of particles (thousands to millions) almost invariably utilise the link-cell method [5]. The computational region is divided into cells. At the beginning of the timestep the coordinates of each particle are examined so that it can be assigned to a particular cell. A list is formed indicating the contents of each cell, usually in the form of a linked list: hence the name "link cell". Two cells which have no points separated by less than the interaction cut off cannot contain interacting particles. Thus, when looking for interacting neighbours of a particular particle, it is only necessary to search its own cell and a subset of other nearby cells. At no point are all pairs of particles considered, and the execution time is of order N in the particle number, with no N^2 component.

This does not mean, however, that the method locates neighbours with maximum efficiency. The conventional implementations, with a cubic box and cells, use cells whose side is equal to or greater than the interaction cut off. Neighbours of a particle are then looked for in its own cell and the 26 touching cells. (Of course, steps are taken to avoid double counting by only considering each pair of cells once.) The ratio between the number of potential neighbours found by this scheme, and the actual number of in-range neighbours, is at best $27R_c^3/\frac{4}{3}\pi R_c^3 \approx 7$. This situation can be improved by using smaller cells, and considering more of them, so that the region of space in which one looks for potential neighbours more nearly approximates the cut-off sphere. However, other inefficiences are introduced when the occupancy of the cells becomes low, which can only be partly compensated for by loop re-ordering. In

tests using a single Transputer we found that the optimal speed of a link-cell program was obtained when the side of a cell was half the interaction cut off. In this case the number of cells to be searched for neighbours is 125, and the ratio of potential to actual neighbours found is still as large as four. At a typical liquid density the average number of particles per cell is around two.

Because a link-cell program, even when optimised as described, still picks up many out-of-range neighbours whose separation needs to be tested before they can be discarded, the fastest simulation method for large systems is to use link-cell neighbour location to set up a neighbour list [6] (providing sufficient memory is available). We call this the *link-cell neighbour list* (LCNL) method. We have been able to develop a parallel implementation of this method, within the systolic loop framework.

It turns out that this method requires each processor to have a copy of all the particle coordinates, and we first describe a modification of the systolic loop method, which we call *systolic replication*, which enables this to be achieved. It is essentially the method intorduced by Craven and Pawley [7]. In the systolic loop methods the coordinates and force accumulators circulate together in a series of pulses in the course of the time step. In systolic replication the coordinates circulate first, at the beginning of the time step. As coordinates pass through, the processors retain a copy, so that at the end of the circulation each contains a complete copy of all particle coordinates. Similarly, each processor has a complete force array, which it initialises to zero. Processors then evaluate a subset of pair interactions, and accumulate forces into the fixed force arrays. After all interactions have been evaluated, force accumulators are circulated. As the accumulators for a particular group of particles pass through a processor, the processor adds on the relevant forces from its own force array. The accumulators eventually reach their home processors carrying total forces, ready for motion integration. The allocation of particles to processors is exactly as in the corresponding systolic loop method, as is the pattern of data movement. The essential difference is the separation of the force circulation from the coordinate circulation, and the replication of data across the processors. Pair interactions may be assigned to the processors in the same way as in the corresponding systolic loop method, or, since each processor has all the coordinates, in *any* convenient manner. We make use of this fact in our LCNL method described below. The simple systolic replication method has some minor advantages and disadvantages compared with a systolic loop method. In SRS, the systolic replication version of SLS, there is no possibility of overlapping communication with calculation, since they occupy separate phases of the time step. (The method of Craven and Pawley [7] is similar to the SLD method of [3] in which intra-group interactions are not evaluated on a separate processor, but on the home processor. In this case the intra-group interactions can be evaluated while the coordinates are being distributed or the forces are being collected.) Another disadvantage is the extra memory required to store the particle coordinates, and there are also extra start-up times involved in the communication as a result of circulating coordinates and forces separately rather than together. Neither of these presents a significant problem. An advantage of systolic replication is better load balancing since processors work completely independently during the evaluation of interactions. A further point in favour of systolic replication is that many-body forces may be incorporated, and distributed over the processors in any convenient way. (See Section 4 for a further discussion of many-body forces.)

Systolic replication enables us to use the link-cell method of neighbour location in the following way. Coordinates are distributed, forces collected and motions inte-

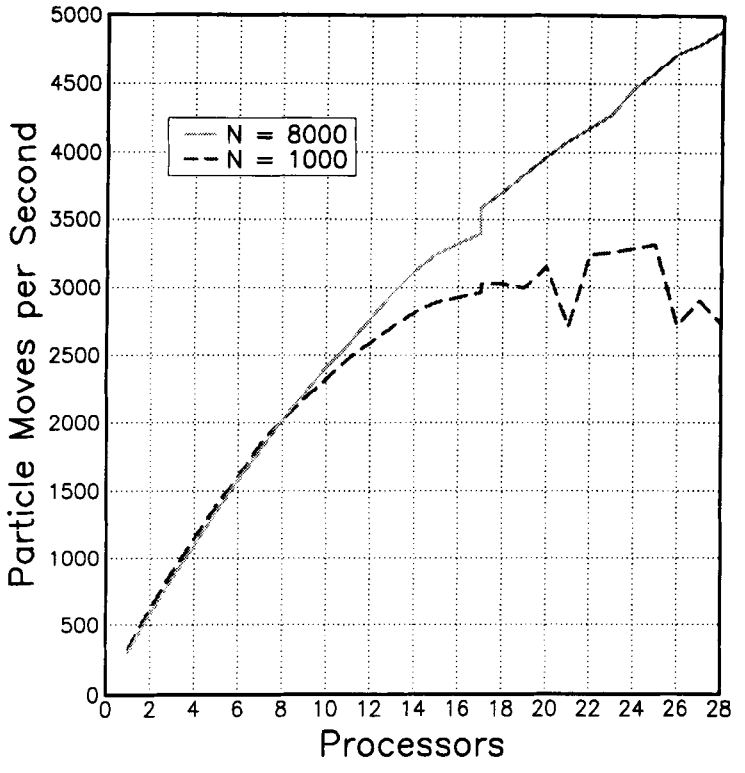

Figure 6 Performance of the SRS-LCNL algorithm for 1000 and 8000 particles.

grated as just described, using the SRS pattern of data movement. However, the evaluation of interactions is distributed over the processors in a different way, by dividing the link cells between the processors. Each processor, because it has all the coordinates, is able to set up a complete link list for the system. It is able then to evaluate interactions between particles within its own link cells, and between particles in its own link cells and particles in those nearby cells which may possibly contain neighbours. In this way each processor evaluates a subset of all the pair interactions, determined by a spatial decomposition of link cells over processors. The setting up of the linked list in each processor is an order N calculation which adds another mildly non-scaleable term, similar to the communication term, to the execution time. This is an extra argument over those given earlier in this section, for not setting up the link list every step, but rather to use it to set up a neighbour list and revise this only when necessary. Since each processor uses the link list to select a subset of all pair interactions to evaluate, there is no problem in setting up a neighbour list and using this on subsequent steps. Again we end up with a fully distributed neighbour list. We call this method SRS-LCNL. The criterion for selecting which nearby cells need to be searched for potential neighbours must now, of course, depend on the outer rather than interaction cut off. We use a link-cell side of around half the cut off, as suggested by our sequential tests.

Since completing this work we have received a preprint by Brugè [8] who has also implemented a link-cell method (but without a neighbour list) in a systolic loop framework. In this method each processor sets up a link-list for its home particles only, and this list is circulated along with the group of particles so it can be employed to speed up the interaction calculation. In other respects the method is a conventional systolic loop method and there is no replication of data.

3.4 Performance of the method

The performance of the SRS-LCNL method is shown in Figure 6. It is indeed faster than the APNL method (very much faster in the 8000 particle case), though somewhat less scaleable. Because it is an order N method the performance in terms of particle moves per second is very similar for 1000 and 8000 particles until the number of processors becomes large. The 1000 particle performance then levels off as the number of particles per processor becomes small.

4. DISCUSSION

We have implemented, in Fortran augmented by a general message-passing scheme, parallel versions of three molecular dynamics neighbour location algorithms; the simple all-pairs method with sperical cut off; a traditional neighbour list; and the link-cell neighbour list, which is the fastest known sequential algorithm. This has been done within the systolic loop framework, where the communication time is constant for a fixed number of particles, independent of the number of processors. This results in mildly non-scaleable behaviour: the graph of performance against number of processors tends to level off, rather than steadily increase, for large numbers of processors. The more efficient the neighbour location algorithm, the more noticeable this effect becomes. There are extra inefficiencies when the number of particles per processor becomes small, due to communication start-up times and imperfect load-balancing. Nevertheless our fastest algorithm achieves the remarkable speed of 5000 particle moves per second on 28 Transputers, and this speed is still increasing with processor number. This can be compared with a speed of 42000 moves per second achieved by a highly vectorised program using a similar link cell neighbour list method [6] on a Cray XMP supercomputer costing perhaps 100 times as much as our Meiko system. It is also important to point out that our benchmark system, the Lennard-Jones fluid, provides a *very severe test of any parallel algorithm* because of the extreme simplicity of the interaction. If rigid multi-atom molecules are being studied, the calculation time increases as the square of the number of sites, whereas the communication time is only doubled (because of the need to circulate orientation and torques as well as coordinates and forces). Communication delays will then have little effect for any reasonable number of processors.

We would like to compare the systolic loop approach with the alternative methods based on a spatial decomposition of the problem [2]. In these methods particular regions of the computational space (let us assume it is a cube) are assigned to particular processors, rather than there being a fixed assignment of particles to processors. For example, with a ring of processors the space is divided into slabs. Within the slab assigned to a particular processor a link cell decomposition is usually made. The ring connection facilitates the application of periodic boundaries in the

appropriate direction. The calculation of interactions between particles in neighbouring processors involves the exchange of data between these processors. Coordinates are circulated in one direction, and forces return in the opposite direction. Such a method resembles a systolic loop but with only one pulse (or perhaps a few [9]) rather than a complete circulation round the ring, because of the spatial ordering. If particles at the end of a time step move out of the region of space controlled by their current processor then all their data must be moved to the appropriate neighbouring processor. These ideas extend easily to a column decomposition of space over a toroidally-connected mesh of processors, or to a sub-cube decomposition over a three-dimensional processor grid, again with wrap-around connections.

The most important point to make about spatial decomposition methods is the following. Suppose the number of particles and number of processors are increased in proportion, so that each processor still deals with the same volume and average number of particles. Then the calculation time remains constant, as does the amount of data to be communicated between neighbouring processors. This means that the method is scaleable (weakly) and hence *it is the only practical method for really large numbers of processors.* Since there is still the requirement to have a reasonably large number of particles per processor, say 50 or more, this implies a very large numbers of particles. Set against this, however, must be the fact that, as mentioned in Section 3.3, the basic spatial decomposition i.e. link-cell method is not very efficient at locating neighbours, and ideally should be supplemented by the use of neighbour lists. We have been able to do this within the systolic loop framework, but it is difficult to combine lists with a spatial decomposition as processors deal with different pairs on a step-by-step basis as the particles move around in space.

For reasonable numbers of processors (tens rather than hundreds) systolic loop methods are competitive with spatial decomposition methods, especially if used in conjunction with neighbour lists, and particularly with more complicated molecular interactions. They also have several practical advantages. Since each particle has a home processor sufficient of its history can be stored there to enable time-correlation functions to be found. With spatial decomposition it would be necessary for a particle to take along its whole history as it moved between processors. In the systolic loop all the particles pass, in order, through the Head processor so that a trajectory file can be written. With spatial decomposition the writing of such files would require the sorting into order of particles from different processors in order to make it possible to follow individual motions. Another point in favour of the systolic loop method is that it will work with an arbitrary number of processors, in our case without any change to the program itself. We find this particularly useful in a multi-user situation where one can simply check how many processors are free and then grab them. With spatial decomposition the number of processors must be chosen to fit the requirement that each hold an integral number of link cells, and the number must be a perfect square (or at least a product) in the case of a 2D mesh of processors, and a cube (or triple product) in the 3D case. Furthermore the fact that the number of particles per processor fluctuates from step to step in this method involves some tedious book-keeping in the program.

We would like to conclude this discussion section with some remarks about the inclusion of three-body forces, to which we have given some thought but of which we have, as yet no experience. Consider first the case of bonded three-body forces (i.e. the angle bending forces used in macromolecular simulation: similar remarks apply to torsional, four-body, forces). In the case of many polymer systems it is possible to

divide the atoms into groups in such a way that these interactions never involve particles from more than two groups [10]. They can then be evaluated straightforwardly during systolic circulation. We do not know to what extent such a decomposition is possible for more complicated systems such as covalently bonded lattices. An alternative approach would be to use systolic replication, when the list of bonded forces can be distributed over the processors in any convenient way. Another very general method is for each particle to carry along a list of other particles to which it is bonded when it moves from processor to processor [11] during evaluation of the pair forces. When it meets up with its partners, either in its own processor, or during systolic circulation, or on a visit to a neighbouring processor in the case of spatial decomposition, it picks up their current coordinates. Back in its own processor its bonded interactions can then be evaluated. This means these interactions will be evaluated more than once, at least if they involve particles based in different processors.

Non-bonded three-body interactions are believed to be important in molecular liquids. Is it possible to derive a pattern of data circulation, a sort of double systolic loop, in which every triple of particles meets up once so that the three-body interaction can be evaluated? We have found solutions for 7 and 11 processors, but not in the general case. While this is an intriguing mathematical problem, it probably has little relevance to real simulation. Such three-body interactions are short in range, and probably a more effective technique is to adopt either a spatial decomposition over processors, or else our systolic-replication link-cell method, in which cases nearby interacting triples would easily be found.

Acknowledgements

We are grateful to ICI Chemicals and Polymers for the award of a Postdoctoral Fellowship to PJM, and to the SERC Computational Science Initiative and the Computer Board Initiative on High Performance Distributed Computing for provision of computing equipment at Keele. We would like to thank Prof. S.L. Fornili, Dr. W.C. Mackrodt, Mr. S. Miller, Dr. A.R.C. Raine and Dr. W. Smith for their continued interest in our work.

References

[1] N.S. Ostland and R.A. Whiteside, "A machine architecture for molecular dynamics: the systolic loop", *Annals N.Y. Acad. Sci.*, **439**, 195–208 (1985).
G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D.W. Walker, *Solving problems on concurrent processors. Volume I: General techniques and regular problems*, Prentice-Hall, New Jersey, 1988, ch. 9.
D. Fincham, "Parallel computers and molecular simulation", *Molecular Simulation*, **1**, 1–45 (1987).
F. Brugè, V. Martorana and S.L. Fornili, "Concurrent molecular dynamics simulationof ST2 water on a Transputer array", *Molecular Simulation*, **1**, 309–320 (1988).

[2] D.C. Rapaport, "Large-scale molecular dynamics simulation using vector and parallel computers", *Comput. Phys. Reports*, **9**, 1–53 (1988).
G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon and D.W. Walker, *Solving problems on concurrent processors. Volume I: General techniques and regular problems*, Prentice-Hall, New Jersey, 1988, ch. 16.
W. Smith, "Molecular dynamics on hypercube parallel computers", Daresbury Laboratory preprint DL/SCI/P687T.

[3] A.R.C. Raine, D. Fincham and W. Smith, "Systolic loop methods for molecular dynamics simulation using multiple Transputers", *Comput. Phys. Commun.*, **55**, 13–30 (1989).

[4] S. Miller, D. Fincham, R.A. Jackson and P.J. Mitchell, "Transputer molecular dynamics with electrostatic forces", in *Applications of Transputers 2,* D.J. Pritchard and C.J. Scott, eds., IOS Press, Amsterdam, 1990.

[5] R.W. Hockney and J.W. Eastwood, *Computer simulation using particles,* McGraw-Hill, New York, 1981, ch. 8.

[6] G.S. Grest, B. Dünweg and K. Kremer, "Vectorised link-cell Fortran code for molecular dynamics simulations for a large number of particles", *Comput. Phys. Commun.,* **55,** 269–286 (1989).

[7] C.J. Craven and G.S. Pawley, "Molecular dynamics of rigid polyatomic molecules on transputer arrays", preprint from Department of Physics, University of Edinburgh, to appear in *Comput. Phys. Commun.*

[8] F. Brugè, "Systolic calculation of pair interactions using the cell linked-lists method on multiprocessor systems", preprint from Department of Physics, University of Palermo.

[9] H.G. Petersen and J.W. Perram, "Molecular dynamics on transputer arrays. I. Algorithm design, programming issues, timing experiments and scaling projections" *Molec. Phys.,* **67,** 849–860 (1989).

[10] A.R.C. Raine, "Molecular dynamics simulation of proteins on an array of Transputers", in *Applications of Transputers 2,* D.J. Pritchard and C.J. Scott, eds., IOS Press, Amsterdam, 1990.

[11] S. Chynoweth, U.C. Klomp and L.E. Scales, "Simulation of organic liquids using pseudo-pairwise interatomic forces on a toroidal Transputer array", preprint from Shell Thornton Research Centre.