Theor Chim Acta (1993) 84: 385–398

Theoretica Chimica Acta

Molecular dynamics on distributed memory (MIMD) parallel computers

W. Smith

Theory and Computational Science Division, S.E.R.C. Daresbury Laboratory, Daresbury, Warrington WA4 4AD, UK

Received October 1, 1991/Accepted November 27, 1991

Summary. Several algorithms are described that have been applied to molecular dynamics simulations and their merits discussed. The subject matter is confined to distributed (MIMD) algorithms. A simple mathematical model is used to illustrate the performance characteristics of a parallel MD algorithm.

Key words: Molecular dynamics – Molecular simulation – Parallel computing

1. Introduction

At Daresbury Laboratory we have been developing parallel algorithms for scientific applications for several years, and molecular dynamics has been a particularly fruitful area of work. Three basic kinds of parallel molecular dynamics algorithm, which are described below, have been developed and studied with our collaborators. These we refer to as the Replicated Data [1], Systolic Loop [2] and Link-Cell [3, 4] (or Domain Decomposition) algorithms. These algorithms are, in some respects, prototypes, from which more sophisticated methods have subsequently been developed.

Our experience ranges over several models of parallel computer. The first was the Meiko M10 Transputer system with 11 T414 nodes, including a mass store board (8 Mbytes), a graphics node and a host board. The programming language was Occam 2. With this system David Fincham (Keele University and Daresbury Laboratory), Andrew Raine (University of York) and the author explored the use of systolic loops in molecular dynamics simulations and gained some insight into fundamental questions of efficiency and load balancing [2].

The FPS T20 computer, for which the T414 Transputer provided the communications engine, was used in early studies of the Replicated Data algorithm, primarily because its "hypercube" connectivity suggested minimal communications overheads.

The Intel iPSC/2, with 32 nodes, including enhanced scalar (SX) performance, vector (VX) processing and the Direct Connect^{TM} communications hardware, was used to further developments of both Replicated Data and Systolic Loop algorithms. On this same machine (later with 64 nodes), two independent groups worked to develop the parallel Link-Cell algorithm for different applications: Dominic Tildesley and Mark Pinches (Southampton University) were primarily concerned with Lennard-Jones systems for studies of surfaces [3]; Dennis Rapaport [4] (Bar-Ilan University) was interested in simulations of polymers and microscale hydrodynamics.

The 32 node Intel iPSC/860 currently at Daresbury is now being used for real scientific research, employing algorithms developed on the earlier machines. Mark Pinches and Dominic Tildesley have used their link-cell programs to study the adsorption on surfaces (e.g. N_2 on graphite [5]). In 1990 Dennis Rapaport and the author began a collaboration on polymer simulations [6]. The Replicated Data algorithm, incorporating the Ewald technique for long ranged forces [7], has been used by the author in a study of superionic conductors (specifically sodium β'' alumina [8]).

Meanwhile an 8 (i860)-node Meiko Computing Surface at Daresbury has been exploited by David Fincham and his students to embed the Ewald sum within a systolic loop framework [9]. The incorporation of Verlet neighbour list within the systolic loop method has also been studied [10]. Andrew Raine has written a complete biopolymer simulation code in Occam to run on Transputer systems and in so doing has addressed some awkward questions about load balancing and the incorporation of bonded interactions [11].

It is clear from these developments that parallelism is making a huge impact on molecular dynamics at Daresbury, and through various agencies (e.g. CCP5 [12]) and publications is making them available to the UK and wider academic communities.

2. Molecular dynamics

Simply stated, molecular dynamics (MD) is a method for solving the many-particle equations of motion for a molecular system [13]. The solution is obtained numerically, since the classical many-body problem is intractable, which means that the trajectories of the molecules are obtained as a sequence of instantaneous positions and velocities at discrete intervals in time (called timesteps). These trajectories are subsequently analysed to provide structural (thermodynamic) data and time dependent data such as correlation functions or transport properties (e.g. diffusion, thermal conductivity etc.). A typical MD simulation begins with a starting configuration (positions and velocities) of the atoms and calculates the forces between them. From these forces, and Newton's laws, the new positions and velocities of the molecules, one timestep later, can be calculated. The new positions are used to calculate new forces, and so the process continues for however many timesteps are deemed necessary to describe accurately the phenomenon under investigation.

The most computationally expensive parts of an MD simulation are the calculation of forces, and the integration of the equations of motion, since these operations are done many thousands of times in a typical simulation. Since in many cases it is assumed that the forces are derived from pair potentials (i.e. the interaction between pairs of atoms), the forces calculations are intrinsically of the order N^2 in cost, where N is the number of atoms, while the integration of the equations of motion is only of order N. However, it is important to note that a great deal of know-how has gone into devising algorithms where the forces calculations scale more like the order $\sim N$ [13]. The best parallel MD programs ensure that these two operations are efficiently parallelised.

3. Replicated data algorithms

The Replicated Data (RD) strategy represents perhaps the simplest strategy for parallel MD. It is applicable to systems in which all the possible pair interactions in the system must be considered; this being the case even if a spherical cutoff is applied to the potential energy functions. In other words: the effective range of the forces is of the same order as the linear dimension of the system. The essential ingredients of the simplest RD algorithm are as follows:

1. Each processor has a complete copy of the coordinates and velocities of all the atoms in the system.

2. Each node of the parallel computer is assigned the task of calculating a specific subset of all the possible pair forces. (i.e. if there are P processors and $\sim N(N-1)/2$ pair interactions, each node calculates a specific $N(N-1)/2P$ of them.)

3. The total force on an individual atom is obtained as a global sum, over all nodes, of all the pair forces associated with the atom. (This step results in every node having a complete replica of the computed force arrays.)

4. The equations of motion are integrated independently on each node, each node dealing with every atom in the system.

In this algorithm, the fact that all the nodes have a copy of the configuration data implies that this strategy is expensive in terms of memory. However, in practice, this does not preclude large simulations ($\sim$10,000 atoms) on machines like the Intel iPSC/860, which has 8 Mbyte per node. The sharing out of the pair forces calculations can be accomplished by any number of strategies, but an important requirement is that Newton's third law is exploited to reduce the computing cost. This can be tricky in parallel MD, but is simple in this case – the author generally favours a variant of the Brode–Ahlrichs scheme [14].

An important feature of this algorithm is that the global sum required to complete the forces calculations imposes a strict limitation on the efficiency of the algorithm when the number of nodes used is relatively large. This gives rise to poor scaling of the algorithm under these circumstances. For this reason the algorithm is better suited to computers with hypercube connectivity, where the communication cost of the global sum is more acceptable. Also it will be noticed that the integration of the equations of motion does not exploit the inherent parallelism in this case. This is because the algorithm remains of order N^2 in terms of the pair force calculation (global sum aside) and for most practical applications the cost of this stage is only a small percent of the total cost per timestep. Parallelising this stage only adds more communication cost to the algorithm, which it could well do without! Users anxious to get the maximum out of the algorithm may however find there is still something to gain from parallelising this operation also.

It is relatively easy to think of ways of improving this basic RD algorithm, and two deserve special mention. Firstly, the luxury of having all the configuration information on each node allows one to use classic MD strategies such as the Verlet neighbour list or the link cell algorithm to improve the node performance. Recently, the author implemented the Verlet neighbour list within the RD strategy to obtain a very acceptable algorithm suitable for many thousands of particles and giving XMP-like performance on 4–8 nodes of an

Intel iPSC/860 [7]. Secondly, it is possible to devise a version of the RD algorithm in which the communication cost is minimised by performing the communication concurrently with the forces computation. This technique exploits some of the features of a systolic loop approach, and has therefore been described as the Systolic Replication algorithm [16, 9]. It is outlined below (Sect. 4).

The main point of stress about the RD method is that despite its theoretical limitations it appears to be a versatile and powerful technique for hypercube parallel computers with up to (say) 32 nodes. It is simple to implement, even within existing serial MD programs, and the fact that its communications are entirely bound into the global summation, means that reducing the program to run on one node is trivially simple. This of course makes the program very portable. A further advantage is the ease with which methods such as the Ewald sum for Coulombic systems can be incorporated [7].

Some results of applying this algorithm [7] on a massively parallel computer (the Caltech Intel Delta) are shown in Table 1. The performance for large systems (~22,000 atoms) is impressive, but it is clearly difficult to obtain linear scaling with the number of nodes used, on account of the global summing. Scaling is especially poor with smaller number of atoms. The lesson here, as with other algorithms, is that efficient exploitation of large parallel computers implies large simulations.

4. Systolic loop algorithms

The systolic loop algorithms are amongst the most efficient MD algorithms for systems of order N^2, though their use is not confined to systems of this order. It is possible to formulate many different varieties of algorithm using systolic loops and a systematic naming scheme for them is desirable. This paper uses our earlier notation [2]. There are (at least) three basic types of systolic loop algorithm, each of which has sub-variants. These are: SLS-G (Systolic Loop Single-Group), SLD-G (Systolic Loop Double-Group) and SLB-G (Systolic Loop Bidirectional Groups). All of these algorithms are fully distributed and load balanced and therefore less memory intensive than the RD method. Also, they are all based on

Table 1. Performance of a replicated data algorithm on a massively parallel computer

N	$P=4$	$P=16$	$P=64$	$P=128$	$P=256$
4096	11.80	3.37	1.68	1.51	1.65
5832	15.80	4.40	2.13	2.11	1.90
8000	20.49	5.67	2.70	2.26	2.21
10648	26.40	7.22	3.37	3.04	2.60
13824		13.11	4.15	3.33	3.07
17576		16.30	6.07	4.35	3.21
21952			7.40	5.19	3.60

All times in seconds per timestep
System: Sodium chloride, with cutoff 5 Å
N: number of ions; P: number of nodes

the idea of circulating data "packets" between nodes, which is the origin of the term "systolic". The data packet contains the configuration data pertaining to a subset of atoms (i.e. coordinates, velocities and forces accumulators of the atoms, though not necessarily all of these).

The simplest of these to implement (and describe) is the SLD-G algorithm. In this algorithm an *odd* number of processors, connected in a ring topology, is required. The initial configuration data are shared equally amongst the nodes, so each node has one group of atoms. (The associated force accumulators are zeroed at this stage.) The group data packet, containing the coordinate arrays and force accumulators, is then duplicated on each node. One of these packets will remain "fixed" on the "home" node, while the other is passed between nodes.

The pair forces that exist within the home group (the intra-group forces) are calculated and added to the home force accumulators. Next, the duplicate packets are sent to the next node on the ring in (say) a clockwise direction. Each node then has the coordinates of two groups of atoms and their force accumulators, which allows the inter-group forces to be calculated and added to the appropriate accumulators. The duplicate data packets are then passed, in the same direction, to the next processor to permit the calculation of another batch of inter-group forces. If there are P nodes in the processor ring, the duplicated packets must be passed $(P-1)/2$ times to guarantee the calculation of all possible pair forces. (This requirement explains why P must be an odd number!) At the end of this stage, the duplicated packets must be returned to their home nodes, which means passing the packets back in the opposite direction $(P-1)/2$ times, and the force accumulators added to the home force accumulators. Finally, the equations of motion for each group of atoms on each node are integrated.

The pattern of data packet movements, for a 5-node, 2×5 data packet algorithm is given in Table 2.

The good load balancing of this algorithm is manifest and the memory requirements per node are equal. However the need to return the duplicate packets to their home nodes (the "rewind") represents a wasteful step. The SLB-G algorithm attempts to diminish this. Once again it requires an odd number of nodes and a duplication of the data packets. It functions similarly to

Table 2. The data packet movements in the SLD-G algorithm

p_0	p_1	p_2	p_3	p_4
1	2	3	4	5
1	2	3	4	5
1	2	3	4	5
5	1	2	3	4
1	2	3	4	5
4	5	1	2	3
Rewind:				
1	2	3	4	5
1	2	3	4	5

Table 3. The data packet movements in the SLB-G algorithm

p_0	p_1	p_2	p_3	p_4
1	2	3	4	5
1	2	3	4	5
5	1	2	3	4
2	3	4	5	1
4	5	1	2	3
3	4	5	1	2
3	4	5	1	2
3	4	5	1	2

the SLD-G algorithm, but here the duplicated data packets are sent in opposite direction around the ring of nodes. The result of this pattern of movement is that, at the end of $(P - 1)/2$ data passes, the duplicated data packets are within one data pass of each other, so the rewind step is much shorter than in SLD-G. However, because both packets are circulated, the concept of a home node is no longer strictly valid, which means that in SLB-G it is necessary to include the velocities of the atoms in the data packets, so the equations of motion can be integrated when the forces calculations are complete. The data movements for a 5-node, 2×5 data packets algorithm are shown in Table 3.

In the SLS-G algorithm, each node is initially assigned two data packets, which represent *different* groups of atoms. In this case the nodes can be either even or odd in number and need only be connected in a line. The algorithm also differs from SLD-G and SLB-G in that the nodes are no longer entirely equivalent; there is a "head" and a "tail" node at the start, and end respectively, of the chain of nodes. In terms of load balancing, all nodes perform the same workload and the concept of a "home" node for the data packets is again valid, as for SLD-G.

In the first stage of the SLS-G algorithm, the intra-group forces are calculated for the pair of packets on each node. In the next stage the inter-group forces, between the atoms in different packets on the same node, are calculated. This is followed by an exchange of data packets between nodes, prior to calculating the next batch of inter-group forces. The movement of data packets is different according to the location of the node in the chain, and is best described with reference to an example, e.g. a 4-node, 8 data packet algorithm (Table 4). Nodes other than the head or tail nodes (i.e. "worker" nodes), send

Table 4. The data packet movements in the SLS-G algorithms

p_0	p_1	p_2	p_3
1	2	3	4
8	7	6	5
7	1	2	3
8	6	5	4
6	7	1	2
8	5	4	3
5	6	7	1
8	4	3	2
4	5	6	7
8	3	2	1
3	4	5	6
8	2	1	7
2	3	4	5
8	1	7	6
1	2	3	4
8	7	6	5

the first data packet to the right and simultaneously receive one from the left to replace it. Then each node sends the second data packet to the left while receiving one from the right. (If the communication links between nodes are bidirectional, these send/receive operations can occur simultaneously.) The tail node (which we may assume is on the far right of the chain) sends one data packet to the left, which is replaced by the second data packet on the same node, and receives one from the left. The head node (far left) keeps one data packet *permanently fixed* while it sends a data packet to the right and receives one from the right. In Table 4 this is seen to be a movement of the data packets in a cyclic manner, except that one packet is fixed (packet 8). Note that the number of data sends in this algorithm is $2P - 1$, to return the packets to their home nodes with completed force accumulators. Otherwise the algorithm strongly resembles the SLD-G and SLB-G algorithms. The equations of motion for the home groups are integrated straightforwardly.

There is little to chose between these three algorithms as described. However, in terms of general applicability, we have found the SLS-G algorithm to be the most versatile.

A useful enhancement of the systolic loop algorithms is the overlapping of the communications with the computation of the forces. In principle, most of the communications costs could be removed by this. For example in the SLS-G algorithm, a scheme for this operates as follows. On each worker node:

1. Initiate first send-receive of data packet of coordinates only.
2. Initiate second send-receive of data packet of coordinates only.
3. Calculate inter-group forces.
4. Complete first send-receive.
5. Complete second send-receive.
6. Initiate first send-receive of data packet of coordinates and force accumulators.
7. Initiate second send-receive of data packet of coordinates and force accumulators.
8. Calculate inter-group forces.
9. Complete first send-receive.
10. Complete second send-receive.
11. Add inter-group forces to force accumulators.
12. Repeat steps 6 to 11 $2(P - 1)$ times.
13. Initiate first send-receive of data packet of force accumulators only.
14. Initiate second send-receive of data packet of force accumulators only.
15. Calculate intra-group forces.
16. Complete first send-receive.
17. Complete second send-receive.
18. Add intra-group forces to force accumulators.
19. Integrate equations of motion.

The schemes operating on the head and tail nodes are similar. (Note: It is important, in this scheme, to realise that the coordinate arrays are always one move ahead of the corresponding force accumulators.) This scheme is very elegantly handled in Occam, which supports explicit parallel constructs, but it is somewhat less elegant in FORTRAN. It must also be said that, in practice, the entire removal of the communications cost does not occur [15]. Initialising the communication is inevitably a serial event, which adds time to the overall execution, and also the transfer of data from memory often delays arithmetic operations which may be using the same memory bank. The performance of the systolic loop algorithms on Transputer networks has been studied in detail in [2].

The technique of overlapping communication and computation has also been used in the Replicated Data framework by Craven and Pawley [16]. In their algorithm, the initial data are shared amongst the nodes in groups as in the systolic loop algorithms. The first operation is to calculate the intra-group forces, while copies of the group coordinate arrays are sent around the ring of processors and consolidated into a complete array of all coordinates on each node. At this point the calculation of intra-group forces is interrupted and the inter-group forces are calculated. Care is taken to prevent duplicating pair force calculations on different nodes. In the third stage, the calculation of intra-group forces is continued, while the incomplete force arrays are circulated around the ring for global summation. The incomplete force arrays may be shortened in the case of molecular systems, by sending *molecular* force and torque arrays in place of the full atomic arrays. Finally, when all the force arrays have been completed, the equations of motion can be integrated for the home group of atoms on each node. The term "systolic replication" seems appropriate to describe this algorithm. Like the basic Replicated Data algorithm above, it is useful for simulations using the Ewald sum [9].

5. Link-cells (domain decomposition)

Domain decomposition is a universal strategy adopted in many areas of mathematical modelling. In molecular dynamics it generally goes by the name of the link-cell method. It is appropriate for systems in which the potential cut-off is very short in relation to the size of the system being simulated, and hence has applications in such diverse subjects as: polymers, microscale hydrodynamics, phase transitions, surfaces, micelles etc. Parallel adaptations of the link-cell method are particularly powerful as they enable extremely large systems to be simulated very cost-effectively.

The serial form of the algorithm is well-documented [17]. It suffices here to outline only the basics. According to the basic strategy, the molecular dynamics simulation cell is divided into sub-cells, with width slightly greater than the radius of the cut-off. A simple N dependent algorithm assigns each atom to its appropriate sub-cell and a *linked list* is used to construct a logical chain identifying common cell members. A subsidiary *header* list identifies the first member of the chain. These allow all the atoms in a cell to be located quickly. The calculation of the forces is treated as the sum of the interactions between sub-cells, in the cource of which all the pair forces are calculated. Allowance for periodic boundaries is easily made. The algorithm performs well on serial machines because it greatly reduces the time spent in locating interacting particles.

Parallel versions of this algorithm are easily constructed [3, 4]. The MD cell is divided into equal regions and each region is allocated to a node. The mapping of the regions on the array of nodes is a non-trivial problem in general, though specific solutions for machines like hypercubes are much easier to obtain. The important criterion is to ensure that neighbouring nodes on the network handle neighbouring regions of the MD cell (the "contiguity condition"). The region on each node is further subdivided into sub-cells as in the serial algorithm. The coordinates of the atoms in sub-cells on the boundaries of each region are exchanged with the neighbouring nodes sharing the boundary. After which each node may proceed to calculate all the pair forces in its region *independently*. No further communciation between the nodes is necessary until after the equations of motion have been integrated: particles which have moved out of their node region must be reallocated to a new node (though this can be done while the regions exchange boundary data at the start of the next timestep). Overall the algorithm is simple, powerful and flexible.

One subtle point needs to be mentioned regarding the exchange of boundary data. Lest it be imagined that a simple one-step exchange of data in all directions (north, south, east, west, up and down in 3D) is sufficient to satisfy the contiguity condition, it must be said that this is not so. Data can only be passed in complementary directions (north-south, east-west, up-down) at any given instant and in between exchanges, the exchanged data must be resorted before the next exchange. This is necessary to ensure the corner and edge sub-cell data are correctly exchanged between regions sharing edges and corners, rather than faces.

The ease with which this algorithm is parallelised, and the equal sharing of the data between nodes means that it is appropriate for simulations of very large systems. Consequently the algorithm is finding applications in many areas of molecular dynamics [5, 18, 6] where system size is important. The performance and scaling properties of the parallel link-cell algorithm have been studied in detail in [3].

An interesting point about the parallel link-cell algorithm is that, although specifically designed for systems with short-ranged forces, it can be used for systems with Coulombic forces, which are extremely long ranged. Work is in hand to parallelise the PPPM (Particle-Particle, Particle-Mesh) algorithm of Hockney and Eastwood [17] in which the traditional Ewald sum is optimised by two strategies. The first is to calculate the *real*-space component using the link-cell method outlined above. The second is to replace the standard *reciprocal*-space sum by a fine-grained mesh on which the Gaussian charges are replaced by finite charges on mesh points (i.e. a charge apportioning scheme is employed). The mesh permits the use of *fast Fourier transforms* in the calculation of the sum, at great saving in the computational cost. Both of these strategies are within the compass of distributed parallel computing, provided a distributed FFT algorithm is available.

6. Assessment of performance

A useful way of thinking about parallel MD algorithms is to consider a simple mathematical model. We have used such models to provide insight into the efficiency, performance and scaling properties of algorithms [2, 1]. We begin with the time T_s required to complete one timestep:

$$T_s = T_p + T_c \tag{1}$$

where T_p and T_c are respectively the processing time and communication time, per timestep, per processor (assuming the processing and communication are not overlapped – the analysis therefore presents a "worst case" picture). The main objective, in serial and vector MD algorithms, is to minimise T_s for a given size of system N. However in parallel MD it is also necessary to consider how efficiently the algorithm utilises the nodes, since the lowest T_s for a given number of nodes P, is not incompatible with a serious waste of resources.

It is convenient to write:

$$T_s = T_p(1 + R_{cp}) \tag{2}$$

where

$$R_{cp} = T_c/T_p \tag{3}$$

is the fundamental ratio.

Much can be learned from an examination of Eq. (2). Most importantly, since T_p is clearly proportional to P^{-1} (assuming scalar processing on the nodes), it follows that, when R_{cp} is constant, T_s is likewise proportional to P^{-1}. In other words, the performance of the algorithm (measured as $1/T_s$, or the number of timesteps per unit of time) scales linearly with the number of nodes P; this being true for $P > 1$. Furthermore, it is apparent that when $R_{cp} = 0$, $T_s = T_p$ and the parallel program will be maximally efficient (i.e. there is no effective loss of performance due to the communication). It follows that, in order to achieve maximum performance and linear scaling, we must strive to obtain the lowest values of R_{cp}, i.e. $R_{cp} \to 0$.

In practice, for most communicating algorithms, not only is $R_{cp} > 0$, but also it is a function of *both* N and P. In general it appears that R_{cp} *increases* with P and *decreases* with N, and we must learn which combination of these parameters makes the algorithm efficient (i.e. low R_{cp}). The importance of avoiding large R_{cp} regimes is seen in the case where $R_{cp} = 1$, since then *half* the total time T_s is spent in communications alone, effectively losing half the power of the parallel computer. Under these circumstances, since R_{cp} will usually diminish as P is reduced, the algorithm can be made more efficient on a smaller number of nodes.

It is interesting to note, with reference to Eq. (3), that apparently poorer efficiency (higher R_{cp}) can result if the value of T_p is *decreased* by (say) vectorisation of the code. This unfortunately, is true, and reflects the fact that the algorithm will spend proportionally more of its time communicating. The loss of efficiency will occur alongside a reduction in the scalability of the algorithm, which means that the increase in performance of the algorithm overall will not be in direct proportion to the increase in processing speed. The antidote is to increase N (i.e. perform a larger simulation) or decrease P, if the best use of resources is demanded.

It is therefore convenient for any given algorithm, to have some estimate of R_{cp} in any given regime of N and P, to assess the algorithm's performance. This is easily obtained as the following shows. We begin by defining some

Table 5. MD parameters on the Intel iPSC/860

Parameter	Time (ms)
α	0.0112
α'	0.00138
β	0.00277
γ	0.00863
δ	0.201

obvious parameters:

1. α – time to calculate one pair force.
2. α' – time to decide if pair is within cutoff.
3. β – time to integrate equation of motion for one atom.
4. γ – time to communicate data for one atom to a neighbouring node.
5. δ – time to initiate communication.

These parameters can be obtained by simple benchmark computations, the results for simulations with simple Lennard-Jones forces on the Intel iPSC/860 are given in Table 5. (The reader should realise that these figures represent current estimates only, since they are subject to revision with each upgrade of the compiler.)

As an example, we use the parameters in Table 5 in the following formulae pertaining to the SLS-G algorithm described above.

For the processing time T_p we write:

$$T_p = 2\left[\frac{n}{2}(fn-1)\alpha + \frac{n^2}{2}(1-f)\alpha'\right] + (2P-1)[fn^2\alpha + (1-f)n^2\alpha'] + 2n\beta \quad (4)$$

where $n = N/2P$ is the size of each group of atoms and f is a number which defines the *average* fraction of pair interactions within the defined cut-off range e.g. $f = \pi/6$ for the largest practical cut-off. (It should be noted that this analysis assumes a reasonably high e.g. liquid particle density, where number fluctuations can be regarded as small.) The first term R.H.S. represents the cost of calculating the intra-group interactions, the second the inter-group interactions and the third, the integration of the equations of motion for the groups on each node, *per timestep*.

The communications time T_c is given by:

$$T_c = (2P-1)(n\gamma + \delta) \quad (5)$$

which is clearly the time to complete the data passes for the $2P-1$ systolic "pulses".

Using $n = N/2P$, we may write:

$$R_{cp} = \frac{(2P-1)(N\gamma + 2P\delta)}{N^2(f(\alpha-\alpha') + \alpha') + N(2\beta - \alpha)} \quad (6)$$

from which the properties of R_{cp} may be easily derived. For example, when N is large (i.e. $N \gg P$), then we may write:

$$R_{cp} \approx \frac{(2P-1)\gamma}{N(f(\alpha-\alpha')+\alpha')}. \tag{7}$$

In this limit, Eq. (7) verifies the comment made above: that as P or N increases R_{cp} increases and diminishes respectively. However, in this form the statement is more quantitative. A similar dependence is obtained for the Replicated Data algorithm [1].

At the other extreme: $N \approx 2P$, we obtain:

$$R_{cp} \approx \frac{(2P-1)(\gamma+\delta)}{2P(f(\alpha-\alpha')+\alpha')+2\beta-\alpha}. \tag{8}$$

Here we see only a weak dependence on P and therefore the algorithm has approximately the same degree of inefficiency at this limit, however many nodes are being used. Most importantly however, we note the presence of the parameter δ (the communication start-up time) in the numerator. This makes a very significant (and detrimental) contribution for some parallel machines (especially the Intel hypercubes, where it is quite large – see Table 5), and as a result we can expect poor efficiency and bad scaling under these conditions. These predictions are fully borne out by experience.

Another way we can present these results is given in Fig. 1, where the log-log plot of R_{cp} *versus* N for several dimensions, D, (Note: $D = \log_2 P$) of the Intel iPSC/860. The zero ordinate in this plot represents the "break even" point in the algorithm (i.e. the point at which the communication and processing costs are equal). It is clearly apparent that increasingly larger simulations (i.e. higher N) are required to achieve the break even point as the dimension of the hypercube increases, reflecting the increasing inefficiency of the algorithm with increasing numbers of nodes. Alternatively we may say that for a given value of N, R_{cp} increases with increasing P. Between 10^3 and 10^5 atoms R_{cp} is less than 1% in all

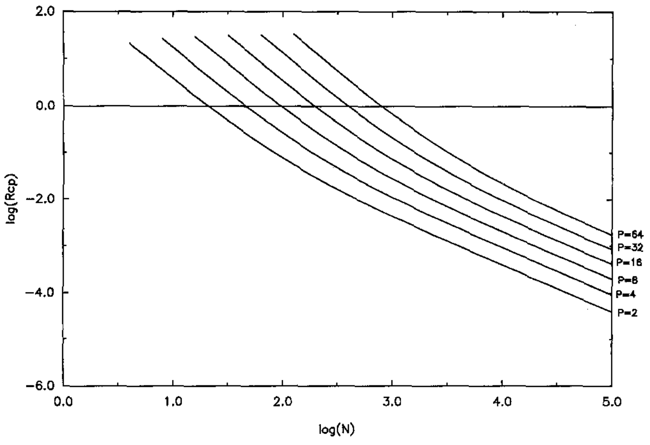

Fig. 1. Plot of $\log(R_{cp})$ versus $\log(N)$ for the SLS-G MD algorithm and a range of different hypercube dimensions ($D = \log_2 P$) of the Intel iPSC/860, based on the mathematical model described in Sect. 6

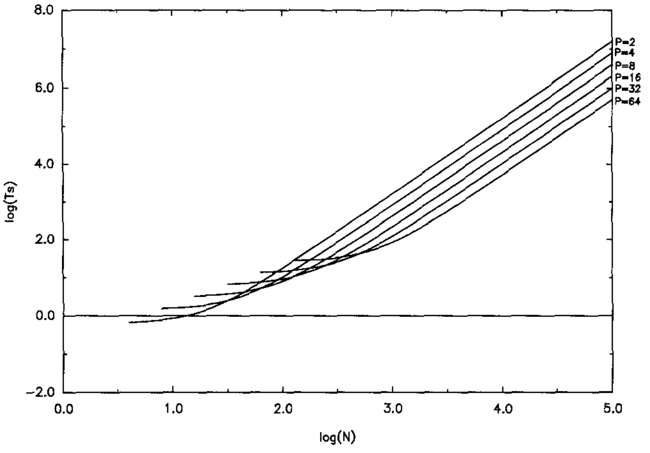

Fig. 2. Plot of $\log(T_s)$ (the time in ms) versus $\log(N)$ for the SLS-G algorithm and a range of different dimensions ($D = \log_2 P$) of the Intel iPSC/860, based on the mathematical model described in Sect. 6

cases shown, leading to the expectation of good scaling and high efficiency. Such plots are useful for defining the regime under which an algorithm is optimally efficient.

Finally, in Fig. 2, we show a log-log plot of T_s versus N for increasing hypercube dimension. At large N we see the lines for different dimensions are parallel with slope 2 (as expected for an N^2 algorithm). Since each line corresponds to a dimension of the hypercube, the fact that the lines are evenly spaced at this extreme shows that the algorithm is scaling linearly with the number of nodes. At lower N values, the line of highest dimension is the first to deviate from the ideal linear behaviour, as expected. The deviation becomes significantly worse for N approaching the break even point of Fig. 1. All dimensions show the same qualitative behaviour eventually, though it is progressively less severe as the number of nodes decreases. Once again, we are reminded of the need to choose the correct N for a given number of nodes P to ensure optimal efficiency.

7. Conclusion

In this talk we have presented the outline for three basic strategies for parallelising molecular dynamics simulations. The Replicated Data and Systolic Loop algorithms are suitable for simulations employing a relatively large cut-off in the forces calculation, and therefore scale in cost as N^2. The Systolic Loop methods are preferable in terms of storage requirements and efficiency, particularly on massively parallel computers, where the global summing of the Replicated Data strategy becomes prohibitive. However, the Replicated Data method is undoubtedly easier to program and its efficiency is not significantly poorer than the Systolic Loops on parallel computers with only a few dozen nodes.

The Link-Cell method is ideal for simulations with a short ranged cut-off, where the cost scales approximately as N. The parallel implementation of this is simple and very powerful, allowing very large simulations ($\sim 10^6$ atoms) to be simulated cost effectively.

Understanding of the efficiency and scaling properties of these algorithms is important if they are to be used to best advantage. Fortunately, mathematical models are reasonably simple to construct and provide a deep insight into these questions, even before the required MD programs have been written!

Acknowledgements. I wish to thank my collaborators with whom several of the described algorithms were developed, namely: David Fincham (Keele University), Andrew Raine (Cambridge), Dennis Rapaport (Bar-Ilan University), Dominic Tildesley (Southampton University) and Mark Pinches (Southampton University). Thanks are also due to my colleagues at Daresbury Laboratory, particularly Martyn Guest for his encouragement and enthusiasm and Bill Purvis for his advice on computing issues. Battelle Pacific Northwest Laboratory is thanked for time on the Caltech Intel Delta, which produced the results in Table 1. Lastly, I am greatly indebted to my hosts; Robert Harrison (Argonne National Laboratory) and Ray Bair (Battelle Pacific North West Laboratory) for inviting me to take part in this workshop.

References

1. Smith W (1991) Comp Phys Comm 62:229
2. Raine AC, Fincham D, Smith W (1989) Comp Phys Comm 55:13
3. Pinches MRS, Tildesley D, Smith W (1991) Mol Simulation 6:51
4. Rapaport D (1991) Comp Phys Comm 62:217
5. Pinches M, Tildesley D (1992) Unpublished work, also Pinches M (1992) Thesis, University of Southampton
6. Smith W, Rapaport D (1992) Molecular Simulation (in press)
7. Smith W (1992) Comp Phys Comm 67:392
8. Smith W, Gillan MJ (1992) J Phys: Condens Matter 4:3215
9. Miller S, Fincham D, Jackson RA (1990) in: Pritchard DJ, Scott CJ (eds) Applications of transputers 2. IOS Press, Amsterdam
10. Fincham D, Mitchell PJ (1991) Molecular Simulation 7:135
11. Raine AC (1991) Molecular Simulation 7:59
12. Smith W (1987) Molecular Graphics 5:71
13. Allen MP, Tildesley DJ (1987) Computer simulation of liquids. Oxford University Press
14. Brode S, Ahlrichs R (1986) Comp Phys Comm 42:51
15. Bomans L, Roose D (1989) Concurrency Practice and Experience 1:3
16. Craven CJ, Pawley GS (1991) Comp Phys Comm 62:169
17. Hockney RW, Eastwood JW (1981) Computer simulation using particles. McGraw-Hill, New York
18. Belak J (1991) A parallel implementation of a molecular dynamics algorithm using the PCP programming paradigm and its application to orthogonal metal cutting. CCP5 Information Quarterly, No 34, March 1992, p 23