Multi-million particle molecular dynamics
II. Design considerations for distributed processing

D.C. Rapaport

Physics Department, Bar-Ilan University, Ramat-Gan 52100, Israel *

and

Theory and Computer Science Division, SERC Daresbury Laboratory, Warrington WA4 4AD, UK

Received 12 January 1990

This paper describes recent progress in implementing large-scale molecular dynamics simulations on distributed processor systems. The computational load is spread evenly among the processors by employing a spatial subdivision — a technique that introduces only minimal communication overheads dependent on the number of particles per processor, and which therefore lends itself to essentially unlimited expansion. Issues of interprocessor communication and overall control of the computation are discussed. Feasibility studies for two-dimensional systems with 4 million particles and three-dimensional systems with 2 million particles have been carried out on a hypercube array containing 64 processors; the performance was found to be comparable to that of the latest vector supercomputers, even though vectorization was not employed in the tests.

1. Introduction

In this, the second of two papers dealing with the implementation of extremely large-scale molecular dynamics simulations on modern supercomputers, we describe an approach designed for use in a distributed processing environment. There can be no doubt that distributed processing has an assured future, for the simple reason that no matter what processor speeds the most advanced of device technologies deliver, a multiplicity of processors will always improve performance. This represents the situation not only with high-price ultra-fast processors, but also at the low-cost end where it is possible to replicate inexpensive processing elements to an almost unlimited degree. For certain classes of computation the latter approach could well prove to be significantly more cost-effective than computer systems based on a small number of expensive processing elements.

Which of the two approaches to processor design is the preferred one depends, to a considerable extent, on the nature of the computations that are to be carried out; some kinds of calculations utilize data in a localized manner that makes them ideal candidates for a fully distributed approach using as many processors as available, each with its own private memory, while others do not allow compartmentalization of data and are therefore better suited to a smaller number of processors accessing a shared memory. Irrespective of approach, the need to be able to map algorithms onto multi-processor systems, along with the concomitant issues of communication and synchronization, must be addressed.

The focus of this paper is on fully-distributed systems in which no common memory exists, with the consequence that all data sharing requires explicit communication. An alternative approach that could be adapted for shared memory use was covered towards the end of the preceding paper [1] (hereinafter I). Vectorization is also a topic not addressed here since the discussion in I is equally applicable to one or

* Permanent address.
many vector processors; the technical details of vector and distributed processing are essentially independent of one another, although current trends in supercomputer evolution suggest that both issues will prove central to computations employing future large-scale systems.

2. Distributed processing

2.1. Processor connectivity

Distributed processing has yet to reach the maturity of vector processing. Vector processors are now taken for granted; their desirable and not so desirable features have been cataloged in detail [2], and extensive performance comparisons between various machines have been made — see e.g. ref. [3] — however, for obvious reasons, the results of such comparisons may not generalize to problems significantly different from those used in the tests. The actual vectorization of software can often be dealt with automatically by the compiler, although, as was shown in I, efficient vectorization can also necessitate changes to the algorithm. The current situation with distributed processing is one in which available hardware products differ greatly in concept and design [2,4,5], performance comparisons are harder to carry out meaningfully because of the extra dimension of communication, and there is a dearth of automated tools due to the fact that the intellectual investment for distributing a computation tends to exceed that required for mere vectorization.

If the multiprocessor version of a computation makes heavy use of interprocessor communication, then it is important that the predominant communication routes make use of the most direct communication paths in the system. Complete point-to-point connectivity becomes rapidly unfeasible, since the number of links would grow as the square of the number of processors, and the computer designer is compelled to resort to a communication network which may or may not match the user requirements. All processors can of course communicate by indirect paths, but there is an additional cost in terms of communication time as well as the possibility of bottlenecks should different communication routes share common intermediate stages. In practice, the average user has to make do with whatever facilities are available; the potential for improved performance may exist if the network structure is considered when designing the application.

A prominent feature of any distributed processor system is the manner in which processors are linked physically — in other words, the topology of the underlying communication network. One popular family of networks is the hypercube [4,5]; such a network consists of \( 2^d \) processors configured as the vertices of a \( d \)-dimensional hypercube with the direct links lying along the edges of the hypercube. Each processor is thus connected to \( d \) neighbors, and the most distant pairs of processor require communication paths that traverse \( d - 1 \) intermediate processors. Provided the processor labels \((0, \ldots, 2^d - 1)\) are suitably assigned (this occurs at the hardware level), the direct links can be immediately identified as being those between processors whose labels, expressed in binary notation, differ by only a single bit (a total of \( d \) bits are required to cover the full range of label values).

Other networks [2] include “reduced” hypercubes, where the degree of connectivity is fixed irrespective of the number of processors (such as a three-dimensional grid), with or without periodic wraparound, as well as ring and star topologies. These are examples of networks where the connectivity is hardwired. An alternative is the switched network, in which links are established according to the requirements of the application; one way of implementing a network of this type is based on crossbar switches. Another option is the use of packet transmission, possibly enhanced by dynamic routing to maintain a balanced communication load throughout the network. A technique used in recent large-scale molecular dynamics simulations with over \( 10^5 \) particles [6–8] provides an example of yet another way of constructing a network: a set of processors were organized so that communication took place through a single shared-memory unit; a hardware mechanism that allowed only a single processor to access the shared
memory at any instant simplified the synchronization issue, though in general, any approach based on a central hub is prone to congestion as the communication load increases.

2.2. Data sharing

Although the subject of interprocessor communication is an extensive one, the concern of the end user is that communication be carried out as efficiently and unobtrusively as possible. Fortunately, for molecular dynamics simulations of the kind discussed here – in which the region occupied by the system is spatially subdivided – it is not difficult to achieve this goal.

The fact that the measure of success in implementing a distributed computation is the fraction of time the processors are kept busy computing does not in any way diminish the importance of the little-known issues of communication and control – how the various parts of the computation being executed on separate processors are kept in phase and how data is transferred at the correct instants. If adequate attention is not given to this subject from the outset, the problems that can arise include wrong data being delivered to a processor, or processors becoming deadlocked while each awaits something from the other; when such erroneous situations occur their origins are not always easily determined. While the actual data transfer between processors is generally the responsibility of the operating system that accompanies the hardware, application dependent details, such as which data to transmit, the recipient of the data, and at what stage a processor should expect to receive data, are as much a part of the overall computational scheme as the numerical details themselves.

Modern multiprocessor systems tend to allow computation and communication to proceed concurrently. For problems in which the communication overheads are high, it is important that data transfer and computation overlap as much as possible, even if this requires additional design effort. The less efficient alternative is to keep the two activities separate, possibly even going to the extreme of awaiting acknowledgement of receipt from a particular destination before sending further messages; applications using this approach are simpler to develop and test, as well as more robust when changes to the program are made subsequently, but the luxury of being able to do just one thing at a time can only be tolerated if the performance penalty is minimal. Fortunately, in the molecular dynamics computations described here, this is exactly the situation, so that the simpler approach can be adopted. Once sufficient experience is gained with such problems, conversion to the more efficient method can always be made if warranted; the gain may be illusory, however, since the mechanisms the system actually uses for message passing may indeed follow the rudimentary lines just outlined.

While applications that use the processor array effectively should spend most of their time computing using locally available data, with only a minimal fraction of the overall time “wasted” in communication, the hypercube, for example, does allow certain classes of operations involving data from all processors to be carried out relatively efficiently. These operations must of course be commutative, examples including the sum and extremum of corresponding quantities in all processors (such as would be needed in computing the total energy and maximum velocity in the course of a molecular dynamics simulation); they are carried out in exactly $d$ sequential stages in parallel on all processors, and involve communications over the direct links only.

The overall control of the set of processors working together on a task requires no special mechanism beyond simply prescribing that the role of certain kinds of data transfers is to issue commands specifying the actions to be carried out, while other transfers serve to acknowledge that these actions have been completed, possibly returning status or other information required by the processor issuing the command. The question then arises as to whether the role of controller is assigned to a particularly processor (either a member of the network, or an outside processor whose role is principally supervisory), or whether the interlocked nature of the distributed application is such that no explicit control processor is necessary and mutual cooperation among processors is adequate. The latter “symmetric” approach allows identical
software to run on all processors, whereas the alternative requires a distinct program to execute on the controller (if the control processor is part of the network, then a common program might be loaded onto all processors but the supervisory functions would only be executed by the designated controller). The existence of some centralized facility that loads the software into the processors and routes output to the user is taken for granted – in terms of the overall effort it plays a very small role.

In the case of the molecular dynamics computations, the algorithm lends itself naturally to distributed control, but in order to implement an application where, for example, summary data is stored on disk for later analysis, or where checkpointing is used to preserve intermediate stages of the computation, a certain amount of centralized control is preferred. Once the symmetry has been broken there is of course no reason not to use centralized control more widely, even where the distributed approach may be equally acceptable. Examples showing how different parts of the overall implementation lend themselves to the alternative approaches appear in section 3.

2.3. Efficiency and portability

For the kinds of simulation that are described in this paper, the issue of communication efficiency is not a serious one, since the amount of data that has to be transferred between processors is relatively small in comparison to the total amount of data involved in the simulation. Nevertheless, it is worth briefly examining the actual performance level of the hardware configuration used, and we do this by citing some of the measurements made on a similar machine to the one used here [9]. While these results are obviously limited to a particular family of machines they do reveal the performance levels at which the communication overheads have, for all practical purpose, completely disappeared from the problem. More important, such measurements can not only be used to help algorithm and software developers decide where it might be worth investing effort in optimization, but also provide information as to where these efforts would not prove effective. For example, in the present work, the time spent transferring messages is extremely low, so there would be little gain from investing effort to ensure that communication and computation overlap.

On the Intel iPSC/2 hypercube the time to transfer a message between processing elements (or nodes) consists of a fixed setup period together with a value proportional to message length. Different mechanisms are used to prepare for message transfer that depend on the message length, so that setup takes one of three values depending on whether the message consists of a header but no data ("zero" length) or whether or not the length exceeds 100 bytes; the values are in the range 300–600 μs for neighboring nodes. The actual transfer times are 0.8 or 1.4 μs/byte, for short and long messages, respectively. Communication between non-neighbor nodes requires that messages be relayed through a series of intermediate nodes, with the total number of transfers involved being the number of bits that differ in the two processor labels. The setup period for messages between non-neighbors grows with the number of intermediate nodes, but the transfer itself runs at full speed. Clashes are possible of course, and if a node serves as an intermediary in several concurrent transfers it could produce congestion. The fact that a processor is busy computing does not affect its ability to act as a message relay. Further details appear in ref. [9].

Message transmission and reception may involve blocking the processor until a particular message has been sent or received (synchronous communication), or the processor can check on the status of outgoing or incoming messages from time to time while continuing to carry out other useful work (asynchronous communication); the main reason to avoid blocking is that the processor can continue to compute while data is being transferred. This issue only becomes important if the time spent passing messages amounts to a significant part of the total processing time, otherwise, as in the present simulations, it can be ignored. Finally, there is the question of communication with the hypercube control processor (the "system resource manager") that may determine how much of a role this processor should be allowed to play in the computation – communication speeds are down by a factor of 4–15 compared with node–node message
passing. Communication between the hypercube and a host computer that would be used for interacting with the hypercube in a multi-user environment (as well as a provider of data storage, graphics and so forth) can be slower still – by orders of magnitude – and the overall computational approach should be designed to allow this slowest of links to play no more than a minor role.

The iPSC/2 provides a variety of operations to support distributed processing. Only a few of the operations used in the present implementation will be mentioned by way of example. Synchronous communication is handled by the functions csend and crecv, where transmission allows associating distinct “types” with the messages, and reception can be selective as to message type desired. A processor can await the arrival of any incoming message using cprobe, and then obtain details of the message length, type and sender using the functions infocount, infotype and infonode. As pointed out earlier, synchronous communication blocks all further processing until the message in question has been dispatched or received; a corresponding set of asynchronous operation exist that allow communication and other processing to overlap, but these were not needed for the present work. Finally, when for example the sum of contributions from each processor to a quantity such as the total energy is required, the “global” function gssum can be used in carrying out the computation with a minimal amount of optimized communication between nearest neighbor processors; a variety of such functions are available.

Portability of any program using these and similar operations depends on the availability of equivalent operations on the targeted system. No significant effort has been made by way of standardization, and a switch to a system based on similar concepts, but implemented differently, could require substantial changes to the parts of the program that deal with communication.

3. Distributed molecular dynamics

3.1. Distribution schemes

The approaches to a distributed implementation fall into two classes. One class covers those methods in which it is the particles themselves that are assigned permanently to processors; this approach has been described elsewhere [10,11] and will not be treated here, except to state that the communication load is proportional to \(N\), the total number of particles. The alternative is spatial subdivision, in which subregions are permanently assigned to processors, and particles migrate between processors as they travel about in space; if the interactions are short-ranged, the communication requirements are proportional to the number of particles close to the subregion boundaries. It is the latter alternative that will be treated here.

The spatial subdivision of a two- or three-dimensional system can be carried out in various ways. One reasonable criterion for choosing a subdivision scheme might be to minimize the number of particles close to the subdivision boundaries. The optimal subdivision would then entail splitting the system into smaller squares or cubes, or shapes as close to these as possible taking into account the aspect ratio of the volume containing the system and the number of processors employed. An alternative criterion leading to a different subdivision is one that minimizes the number of interprocessor communications (though not the length of the messages); for this a one-dimensional subdivision is used, with the system cut into strips or slices (as in I) that span the system in all directions but one. In this approach [7,8], a processor has to communicate with only two neighbors rather than four or six. In view of the low overheads associated with communication, as well as a desire to minimize the computations associated with boundary-crossing interactions, the present study employed the multi-dimensional subdivision.

In the three-dimensional case, if the processors are organized to correspond to a region that is spatially subdivided into \(P = P_x \times P_y \times P_z\) subregions, then processor \(p\) \((0 \leq p \leq P - 1)\) corresponds to the subregion indexed by the triple \((p_x, p_y, p_z)\), where

\[
p_x = \left\lfloor p/(P_y P_z) \right\rfloor, \quad p_y = \left\lfloor p/P_z \right\rfloor \mod P_y, \quad p_z = p \mod P_z.
\]
An adjacent subregion in (for example) the x-direction has the x-index changed to \((p_x \pm 1) \mod P_x\); the processor responsible for this subregion is readily identified since \(p = (p_xP_y + p_y)P_z + p_z\).

Such an approach to mapping subregions onto processors ignores the physical connectivity of the processor network. If the processors are connected as a hypercube and each of the \(P_x, \ldots\) is a power of two, then it is possible to ensure that neighboring subregions inhabit directly connected processors by using (binary reflected) Gray codes \([12]\) – permutations of the integers \(\{k | 0 \leq k \leq 2^n - 1\}\) in which the binary representations of successive values differ by a single bit only (in other words, unit Hamming distance). If \(\mathcal{G}(k)\) is the operator that Gray-encodes the integer \(k\), then the processor number \(p\) should be evaluated using \(\mathcal{G}(p_x)\) (etc.) rather than \(p_x\) itself. The processors responsible for adjacent subregions are determined by, for example, changing the x-index to \(\mathcal{G}(\mathcal{G}^{-1}(p_x) \pm 1)\), where \(\mathcal{G}^{-1}\) is the inverse operator, and the \(\pm 1\) is evaluated modulo \(P_x\). This ensures that no intermediate nodes are required for the bulk of the communications used in the course of the molecular dynamics computations.

3.2. Algorithm for distributed systems

The basic concepts involved in implementing a spatially subdivided simulation are described here. The method bears a certain similarity to the partitioned approach of I, but is, in some respects, even simpler, because of the simultaneous processing of all subregions. For simplicity, as well as to emphasize the fact that overt control is not required due to the computations being self-synchronizing, no mention is made of a control processor in this treatment. As discussed elsewhere in this paper, a controller would normally be needed, for example, in order to perform the file-related operations required in a computation whose results are to be saved for subsequently analysis or display.

Two data transfer operations introduced in I are also employed here. The “copy” operation is used to transfer atom coordinates between processors responsible for the different subregions of the system: for short-range interactions only atoms close to subregion edges will be involved. While it is usually just the coordinates that are required in computing forces, other data associated with the atoms – such as indices that might be used in grouping atoms into polymers – might also be needed. Copied data is discarded once the interactions have been computed; the integration of the equations of motion for these atoms takes place in the originating processor. The “move” operation transfers all the data associated with an atom that has changed subregions to the processor responsible for the new subregion. The version of the data in the previous processor is flagged as invalid and the storage made available for reuse. In both transfers, if a periodic boundary lies between the two subregions in question, then an appropriate shift is made to the affected component of the coordinates.

The transmission and reception of messages containing data are denoted by the operations \(\rightarrow\) and \(\leftarrow\) (the long double arrows supersede the less flexible notation adopted in ref. [8]); the processors that are the sources of incoming messages or the destinations of outgoing messages are indicated by subscripts on the names of the buffers that temporarily hold the arriving or departing data – the existence of all the buffers participating in the transfers is emphasized in this discussion because they also appear explicitly in any actual implementation of the algorithm. \(\mathcal{S}\) stands for the data describing the atoms in the processor under consideration that might be of interest to the communication functions (principally coordinates and velocities); \(\mathcal{S}\) is temporarily augmented by copied coordinate data from adjacent processors during the interaction computations, and data is added and deleted as a result of move operations that transfer atoms between processors. \(p^h\) and \(p^l\) are the two adjacent processors in the direction \((x, y, z)\) of the current copy or move operation, where the superscripts \(h\) and \(l\) denote a subregion index that is one higher or lower than the subregion under consideration (with cyclic wraparound used to provide periodic boundary conditions, and processor indexing beginning from zero to simplify the arithmetic). Buffers \(\mathcal{C}^h\) and \(\mathcal{C}^l\) are used to hold the data involved in copy operations in the high and low directions, with \(\mathcal{M}^h\) and \(\mathcal{M}^l\) serving similarly for moves; \(\mathcal{C}^{in}\) and \(\mathcal{M}^{in}\) are the input buffers for receiving this data prior to its being
incorporated into \( \mathcal{S} \). The buffers are assumed to be sufficiently large to handle each transfer as a single block (for efficiency and simplicity), with a watch being kept for unforeseen overflow.

The operator \( \leftrightarrow \) (a short double arrow as distinct from the long double arrows introduced above) denotes transfer of a context-dependent set of data items within a single processor; these transfers are involved in filling and emptying the different buffers used in the interprocessor transfers. If the commonly used leapfrog integration method is used, then for a copy operation this set of data contains the coordinates of the atoms close to the particular subregion boundary, while the data for the move operation includes both coordinates and velocities of the atoms which are to be transferred to another processor. For other integration schemes, such as predictor–corrector, additional data (typically accelerations computed at earlier times) will be involved.

The copy and move operations are the responsibility of a set of functions \( \text{copy}_{hi}( ) \), \( \text{move}_{hi}( ) \), etc.; these functions select the atom data that is then placed in the appropriate buffer awaiting outbound transfer. They are slightly modified versions of similar functions introduced in I (there designed for transfers in one direction only), now extended to deal with transfers in all directions. Periodic boundary adjustments are made where relevant as determined by the value of the quantity \( \Delta_w \); \( \Delta_w \) is zero except when a copy or move spans a periodic boundary. The function \( \text{proc}_{id}( ) \) is used by each processor to establish the identity of the processor that handles the adjacent subregion in the direction of transfer; one possible indexing scheme that ensures physical adjacency of processors responsible for neighboring subregions in a hypercube processor array was described earlier.

The interaction computations and integration required to advance the subregions of the system by a single time step are carried out independently and concurrently on each processor by \( \text{process}( ) \). The data repacking function \( \text{compress}( ) \) eliminates gaps in the data arrays left by departing atoms, but if a limited number of gaps does not seriously degrade efficiency, the repacking need not be carried out at each time step. The loops over \( x, y, z \) indicate that the data transfers must be carried out for each direction separately; some atoms (those near subregion edges and corners) can participate in more than one transfer. Under most situations of interest, the possibility of an atom having a velocity that enables it to move between non-adjacent subregions in a single step can be excluded.

The following sequence of operations summarizes the basic approach just described. The algorithm is executed concurrently on all processors, with the interlocked data transfers ensuring correct synchronization.

\[
\begin{align*}
\text{for } \text{num}\_\text{steps} \text{ iterations do} \\
\quad \text{for } x, y, z \text{ do} \\
\quad \quad p^h \leftarrow (p_x + 1) \mod P_x & \quad p^l \leftarrow (p_x - 1) \mod P_x \\
\quad \quad p^h \leftarrow \text{proc}_{id}(p^h, p_y, p_z) & \quad p^l \leftarrow \text{proc}_{id}(p^l, p_y, p_z) \\
\quad \quad \Delta_w \leftarrow 0 & \quad \text{if } p_x = P_x - 1 \text{ then } \Delta_w \leftarrow L_x \\
\quad \quad \varrho^h \leftarrow \text{copy}_{hi}(\mathcal{S}, \Delta_w, x) & \\
\quad \quad \Delta_w \leftarrow 0 & \quad \text{if } p_x = 0 \text{ then } \Delta_w \leftarrow L_x \\
\quad \quad \varrho^l \leftarrow \text{copy}_{lo}(\mathcal{S}, \Delta_w, x) & \\
\quad \quad \varrho^h \leftarrow \varrho^h_{\text{in}} & \quad \varrho^l \leftarrow \varrho^l_{\text{in}} \\
\quad \quad \varrho^h \leftarrow \varrho^h_{\text{in}} & \quad \varrho^l \leftarrow \varrho^l_{\text{in}} \\
\quad \text{enddo} \\
\quad \text{process}(\mathcal{S}) \\
\quad \text{for } x, y, z \text{ do} \\
\quad \quad p^h \leftarrow (p_x + 1) \mod P_x & \quad p^l \leftarrow (p_x - 1) \mod P_x \\
\quad \quad p^h \leftarrow \text{proc}_{id}(p^h, p_y, p_z) & \quad p^l \leftarrow \text{proc}_{id}(p^l, p_y, p_z) \\
\quad \quad \Delta_w \leftarrow 0 & \quad \text{if } p_x = P_x - 1 \text{ then } \Delta_w \leftarrow L_x \\
\end{align*}
\]
Even though the processors are all treated identically, there are two occasions where it is sensible to let a particular processor play a dominating role (which might be usurped by the control processor if present). The first is during initialization of the system when random numbers are used to initialize the velocities; in order to allow reproducibility of the computations, irrespective of the number of processors that participate, a systematic procedure for distributing random-number seeds to the processors is required. The simplest method is to do this calculation serially on one processor at a time, with each processor passing the final seed value (assuming a congruential generator [13]) to the next in line. The second occasion is when a summary of some property measurement is to be output; even though an efficient mechanism exists for performing operations such as global sums, only one processor should be allowed to output the result. Both these situations are minor departures from the otherwise total equivalence of all processors participating in the computation. These and other tasks needed for a full-fledged simulation are readily incorporated; however, since a control processor is generally needed for at least some of these tasks, the details will be addressed later.

### 3.3. Performance

Performance measurements were carried out on an Intel iPSC/2 system with 64 processor nodes, each of which was equipped with a scalar accelerator. Although vector processing capability was present on part of the system, it was not used in these tests for two reasons. The first concerns the nature of the vector operations implemented in the Intel hardware, and though a portion of the molecular dynamics computations could have been carried out using the vector operations available, one crucial capability is absent. It will be recalled (see section 4 of I) that the vectorized layer implementations of the interaction computations rely heavily on either bit or index vectors; here the optional vector unit does not support bit vectors and, while it does provide a gather/scatter feature, an efficient means for generating the necessary index vectors is apparently absent. The second reason is the fact that vector processing is restricted to data contained in only a fraction of the address space, with the implication that considerable (non-vectorized) data shuffling would be required. One can reasonably expect that such shortcomings will be rectified in future products of this type.

For two-dimensional soft disk fluids, systems with up to $4 \times 10^6$ atoms were tested at the same state point used in the vector-processor measurements (see I). The time required (with the load spread over the entire system) was found to be 4.5 μs per atom step. This lies between the values obtained for the Cray XMP and YMP (I) and, as already mentioned, does not depend on vector processing. In three dimensions the tests were made on systems with up to $2.05 \times 10^6$ atoms; the time per atom step was 13.1 μs, a value comparable to what might be expected for the Cray YMP given the measured XMP performance and the relative speeds observed in the two-dimensional tests. Smaller systems running on proportionately fewer processors (down to a single processor without communication – but retaining the computational steps needed to support move and copy operations) demonstrated that performance scales perfectly, and only depends on the number of atoms per processor. It should be pointed out that the possibility of improved
performance using assembler language coding was not explored, but the same is true for the vector measurements. For serious production work this avenue merits exploration.

The typical numbers of atoms involved in the move and copy operations are readily estimated, and the numbers are born out by measurement. Even for the largest systems tested, the number of atoms that move between neighboring subregions in the course of a single time step is of order unity. The number involved in a copy operation is proportional to the area (or length) of the subdivision boundary; for a cubic subdivision in which each processor handles some $10^4$ atoms, each copy operation (assuming the very short range interactions of $I$) will involve roughly 500 atoms – each processor will be a partner in twelve such transfers (six neighbors and a transfer in either direction) per time step. The time needed to complete such a transfer, given the communication speeds discussed earlier, would be under 10 ms, and transfers in a given direction involving distinct processors will overlap; the typical time required for a copy operation should be contrasted with the almost 30 seconds needed for calculating a single time step. A simple test of the time spent in communication is based on watching the console lights on the machine, which show whether each processor is computing or communicating at every instant; the message passing time is represented by very brief flashes.

The approach based on neighbor lists (see $I$) can be used in the distributed processing environment to further enhance the performance, although a few modifications to the communication schedule described earlier are needed. The copy operation is carried out at each time step using a boundary thickness equal to the neighborhood size $r_n$, rather than the interaction cutoff $r_c$; however, the sets of atoms to be copied are established when the neighbor lists are about to be updated and are not altered until the next update is due. The move operations are not performed at each time step, but only when a neighbor list update is due to occur. Updating the neighbor lists is carried out by all processors at the same time, and determining when such an update is due requires monitoring the overall maximum velocity across all processors (this provides an upper bound to particle movement). There will of course be a price to pay in terms of storage requirements, but substantially improved performance is to be expected.

Load balancing is a subject not addressed here in any detail. It goes without saying that if processors are not allocated similar amounts of work, the imbalance will lead to reduced throughput. In the case of systems that are spatially homogeneous – with only small density fluctuations – this problem will not arise, but if strong inhomogeneities occur, some consideration must be given to altering the sizes of the subregions in order to spread the computational load more evenly. To what extent this measure will prove successful depends on the particular problem.

3.4. Centralized control

The approach based on distributed control leads to a concise symmetric formulation of the computations, but given the exigencies of real computer systems, a certain amount of centralized control must be introduced in order to run production type computations. Distributed control, with its minimal communication needs, can be retained for the main part of the computation, with centralized control kept to a minimum. A skeletal outline of an approach that incorporates a control processor is given here.

In the following, $H$ denotes the control processor and $\{ P_n | n = 0, \ldots, P - 1 \}$ the distributed processor nodes. The two communication operations introduced earlier are used: $\longrightarrow$ and $\longleftarrow$ stand for message transmission and reception, but now the message may contain either commands or data, depending on circumstances. The actual commands are shown here as "...", but in the real implementation a series of numerical codes would be substituted. In some cases it may be appropriate to have the nodes acknowledge that commands have been completed, but this is not indicated here.

The major tasks performed in a typical molecular dynamics simulation are shown, and others are easily added. The roles of each of these tasks can be summarized as follows: Carrying out an analysis involves measuring and accumulating various properties (thermodynamic variables, correlation functions, etc.) and
is likely to require additional internode communication, as will the preparation of a summary. Snapshots represent the storage of, for example, the complete coordinates of the system at a given instant; such data might be used in further analysis, or perhaps in some graphical rendition of the evolution of the simulation. Checkpointing amounts to storing the complete state of the system, both to allow the calculation to be broken into a series of shorter runs, and to minimize losses in the event of force majeure. Construction of an operational program from the outline below is straightforward, but many of the details are highly system specific.

The control processor part of the collaboration is

```
input simulation parameters, new/continued run
for n = 0 to P - 1 do  H→P_n: simulation parameters
if new run then
    for n = 0 to P - 1 do  H→P_n: "prepare initial state"
else
    for n = 0 to P - 1 do  
        H→P_n: "accept checkpoint data"
        read checkpoint data from file
        H→P_n: checkpoint data
enddo
endif
forever do
    H→P_n: "do md calculations"
    if analysis required then
        for n = 0 to P - 1 do  H→P_n: "do analysis"
    endif
    if analysis summary required then
        H→P_0: "send summary"
        H←P_0: summary data
    endif
    if configuration snapshot required then
        for n = 0 to P - 1 do
            H→P_n: "send snapshot data"
            H←P_n: snapshot data
            write snapshot data to file
        enddo
    endif
    if checkpoint required then
        for n = 0 to P - 1 do
            H→P_n: "send checkpoint data"
            H←P_n: checkpoint data
            write checkpoint data to file
        enddo
    endif
    if end of run then
        for n = 0 to P - 1 do  H→P_n: "terminate"
        terminate
    endif
enddo
```
The nodes \( P_n \) respond to instructions sent by the controller. After initialization they adopt a passive role and carry out whatever tasks are assigned to them. During actual processing, the nodes act together in the symmetric manner described earlier, without any external supervision. There is an obvious correspondence between the features of the node and control programs.

\[
P_n \leftarrow H: \text{simulation parameters} \\
\text{forever do} \\
P_n \leftarrow H: \text{command "..."} \\
\text{if "prepare initial state" then } P_n \leftarrow H: \text{checkpoint data} \\
\text{elseif "accept checkpoint data" then } P_n \leftarrow H: \text{checkpoint data} \\
\text{elseif "do md calculations" then } \text{do md calculations} \\
\text{elseif "do analysis" then } \text{do analysis} \\
\text{elseif "send summary" } (P_0 \text{ only}) \text{ then } P_0 \rightarrow H: \text{summary data} \\
\text{elseif "send snapshot data" then } P_n \rightarrow H: \text{snapshot data} \\
\text{elseif "send checkpoint data" then } P_n \rightarrow H: \text{checkpoint data} \\
\text{elseif "terminate" then } \text{terminate} \\
\text{enddo}
\]

There are other details to be taken into account when constructing a working program. One example is that transfers involving large quantities of data (such as might be required while checkpointing) are likely to require that the data be broken into smaller blocks for efficiency. In order to coordinate the transfer, the recipient must be told the number of blocks to expect; the alternative (particularly when the precise amount of data is not readily determined at the start of the transfer, as might be the case when data is being input from disk by the control processor and then transferred to the node) is to provide some form of indication that the final block of the series has arrived, for example, by sending an additional block of zero length or altering the message type of the final block in an agreed upon manner.

3.5. Extensions

The demand for extremely large molecular dynamics studies is – at least for the present – confined to several specific classes of problem. The examples mentioned in I were polymers, incommensurate phases and hydrodynamic phenomena. The most conspicuous characteristic of problems requiring modeling of this kind is that the phenomena of interest cover length scales that exceed the characteristic atomic scale by orders of magnitude and therefore require simulations of systems with several tens of thousands of particles at a bare minimum, and sometimes considerably more. It is not unreasonable to expect that useful information will be derived from simulations of million particle systems at some time in the near future, although it must be remembered that the time scales on which large-scale phenomena occur are also increased, leading to a substantial growth in the number of time steps required by the computation. The fact that the power of distributed processor systems will also grow should partly compensate for this.

The methods described here are easily extended to other kinds of system. Just one example will be included here. The case is the simulation of a polymer melt, where each polymer is a flexible, non-self-intersecting chain, constructed out of the same simple spherical atoms discussed in these papers.

Chain connectivity is preserved by the addition of attractive forces between adjacent chain atoms; how to construct such an interaction based on the repulsive interatomic interaction used in the remainder of the calculation is described elsewhere [8]. The other modifications needed are minor in nature: The criterion for determining which atoms are to be copied into adjacent processors must be changed to guarantee that intra-chain forces can be evaluated independently in each processor; instead of the repulsive interaction range \( r_c \) being the thickness of the boundary region containing atoms to be copied, the thickness must now...
be determined by the maximum separation between adjacent chain atoms. The other change is that information specifying which atoms are actually linked together in each chain is needed to determine which atoms attract one another; if all chains have the same length, this can be simply deduced from the atom identifiers (this represents the simplest case – heteropolymers will require additional information), so this data will have to be included in the copy operation.

4. Conclusion

The distributed processing environment has been shown to be ideal for molecular dynamics simulation of systems in which the interactions are short-ranged. Communication, though frequent, involves comparatively small amounts of data, and does not impede progress in any noticeable manner. Clearly the distributed approach is one definitely worth pursuing given that the additional algorithmic complexity is not excessive, and that distributed systems provide an extremely cost-effective alternative to top-of-the-line vector supercomputers.

Acknowledgements

I would like to acknowledge the hospitality of the TCS division at Daresbury where the work on the Intel system was carried out. William Smith is thanked for assistance and a great deal of helpful discussion.

Note added in proof

Measurements on the Intel iPSC/860 show approximately 6-fold (2d) and 10-fold (3d) performance gains over the iPSC/2 on a per-processor basis.

References