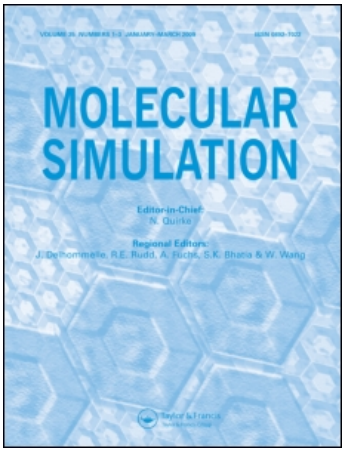


Molecular Simulation
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713644482

Parallel Computers and Molecular Simulation

David Fincham[a]
[a] Department of Chemistry, University of York, York, UK

To cite this Article Fincham, David(1987) 'Parallel Computers and Molecular Simulation', Molecular Simulation, 1: 1, 1 — 45
To link to this Article: DOI: 10.1080/08927028708080929
URL: http://dx.doi.org/10.1080/08927028708080929

Molecular Simulation, 1987, Vol. 1, pp. 1–45
Photocopying permitted by license only

PARALLEL COMPUTERS AND MOLECULAR SIMULATION†

DAVID FINCHAM

Department of Chemistry, University of York, York YO1 5DD, UK

(*Received January 1987; in final form April 1987*)

The term "molecular simulation" covers a variety of techniques that can be applied to many problems in classical statistical mechanics, ranging from lattice spin models to the dynamics of protein molecules. Two well-established techniques, Metropolis Monte Carlo and molecular dynamics, serve to illustrate the computational problems involved. The requirement for powerful and cost-effective computing has led to an interest in parallel computers in this field.

Parallel computers are defined as computers having several processors working simultaneously under the control of an application program. The processors may obey common (SIMD) or individual (MIMD) instruction streams. The ICL DAP is an example of the former; the Intel hypercube is an example of an MIMD or concurrent machine employing conventional microprocessors. The INMOS Transputer is a high-performance microprocessor designed specifically for concurrent processing. SIMD computers may be programmed in versions of Fortran extended to include operations on all elements of an array in parallel. Programming of multiprocessors in standard sequential languages is cumbersome. The language Occam is designed to support concurrent processing simply and effectively, and is the native language of the Transputer.

Problems in molecular simulation possess a rich degree of intrinsic parallelism at several different levels. There are three basic strategies in adapting them to parallel processing, involving the mapping onto processors of particles, of interactions, or of regions of space. Lattice simulations are ideally suited to the DAP, as lattice sites can be identified directly with processing elements. Fluids can be easily simulated on SIMD machines if a small system can be used and all interactions in the system evaluated in parallel. Simulation of larger systems requires the use of a neighbour list, or a cell algorithm in which different regions of space are handled in parallel. In either case, there is a "gather/scatter" overhead. On multiprocessors all-pairs and cell algorithms can be used; for less regular problems, truly asynchronous methods involving algorithmic rather than geometrical parallelism are possible. These methods are somewhat similar to those used in special purpose computers built for the Ising model and for molecular dynamics.

KEY WORDS: Parallel computers, molecular dynamics simulation, Monte Carlo simulation, Distributed Array Processor, Transputer, hypercube.

1. INTRODUCTION

The term molecular simulation can cover a variety of techniques applied to a wide range of problems, from the Ising model of spins on a lattice to the dynamics of a large protein molecule. The common feature is that the system studied is represented by an energy function U, which is expressed in terms of interactions between specific sites geometrically distributed in the system. The standard problem is the study of the equilibrium classical statistical mechanics of the model U (usually limited to pair interactions only), by means of either Metropolis Monte Carlo simulation or dynamic simulation based on Newton's laws of motion ("molecular dynamics") [1, 2]. This article reviews the application of parallel computers to these two types of simulation. Newer simulation methods (non-equilibrium molecular dynamics, stochastic dynamics, free energy calculations, optimization by simulated annealing, Monte Carlo

†An invited contribution.

renormalization group etc.) introduce no new computational problems. The essential computational task is the repeated evaluation of the function U, and its spatial derivatives in the case of dynamic simulation, for a sequence of configurations of the system. Since U may depend on thousands of coordinates, and hundreds of thousands of configurations may be required, the computational task can be immense.

The discussion is also relevant to the related problem of molecular modelling, by which we mean the direct exploration of the properties of the function U, rather than its statistical mechanics [3]. This involves observing the changes in U as the configuration of the system is changed on an interactive graphics system, together with energy minimization ("molecular mechanics"). Here the problem is not the sheer volume of calculation required, but the need for the greatest absolute speed in order to obtain a "real-time" response.

It is the possibility of speeding up the calculations and reducing their cost that has lead to the interest in parallel computers in the field of molecular simulation, following the pioneering suggestions of Wilson [4]. Parallel computers are defined, for the purpose of this review, as those having more than one processor, and with the processors working simultaneously on a common task under the control of an application program. Such designs are more cost-effective than single processor computers, whose power can be increased only through expensive technological advance. The use of novel computer architecture requires the use of novel computational algorithms, and the description of such algorithms is an important part of this review.

The plan of the review is as follows. Section 2 discusses the nature of the computational task involved in the use of simulation methods. This is very dependent on the nature of the physical problem being studied, which determines the most effective computational strategy. Section 3 introduces parallel computers, classifying different types of design and describing some important examples. To be useable, parallel computers must be programmable, and section 4 is devoted to a discussion of languages for parallel programming. We are then in a position to tackle the meat of the review in section 5, which studies in detail how molecular simulation problems may be adapted to parallel computing. In section 6 we take a slight diversion from our theme and discuss some special-purpose computers built specifically for Monte Carlo or molecular dynamics simulation. Finally, in section 7 some general questions about future developments are asked, and the reviewer offers some personal answers.

2. THE COMPUTATIONAL CHALLENGE

A number of features characterize different simulation problems from the computational point of view. The range and complexity of the interactions involved are fundamental considerations. At one extreme we have the Ising type of spin model, in which the interaction sites are fixed in position on a regular lattice. Usually interactions between sites extend to nearest neighbours only, and are simple two-valued functions depending on the relative alignment of the two spins. A somewhat more complicated situation arises in the study of liquids of small molecules. These are frequently modelled by site–site Lennard-Jones 12–6 interactions, with either fractional charges on the sites or point multipoles to represent charge distributions [5]. In order to model these fluids more accurately, and hence for simulation to be more practically useful, there is a need for more complex models, perhaps with non-spherical site cores, and with more realistic representations of the charge distribution.

All these complexities increase the computer time required. For non-polar molecules the range of interactions is relatively modest, say three molecular diameters, giving of the order of 100 neighbours for each molecule. In the case of ions or polar molecules the range is much longer, and these systems provide a greater computational challenge. Another factor of relevance is the degree of homogeneity in the interactions. An extreme case from the point of view of complexity and heterogeneity of interactions is the large macromolecule, such as a protein. There are several distinct types of interaction involved: bond-stretching, angle-bending, torsional and non-bonded. In addition there will be a number of different atom types present, and each interaction will have different parameters depending on the atoms involved.

The presence of many-body interactions introduces another order of magnitude of complexity into simulation. In the case of polarisable ions, a simple but effective technique based on the shell model was introduced many years ago [6], but not applied extensively because of its expense. With recent improvements in computer power, the time is ripe to develop and extend such calculations, and appropriate computational methods need to be studied.

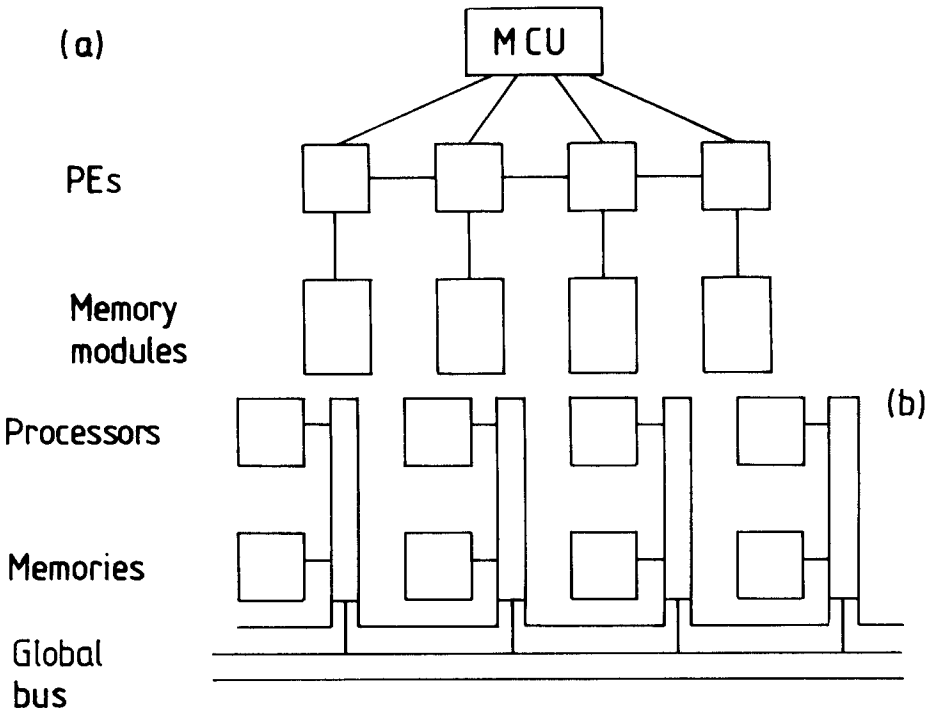
A second characteristic determining the computational nature of a simulation problem is the geometrical relationship between the sites. In the simple Ising model, sites are fixed on a regular lattice. In a glassy system, the relationship between sites is fixed but irregular. In a liquid, the molecules are in motion and so the geometrical relationships are continually changing. This can have a profound influence on the algorithms adopted since the interactions to be evaluated are those between a molecule and its *current* neighbours. The protein molecule is here an intermediate case, since it folds or unfolds only a very slow time scale, and on shorter time scales geometrical relationships change only to a limited extent.

The third important factor characterizing simulations from the computational point of view is the size of the system being studied in terms of the total number of sites or particles. This is controlled primarily by the range of correlations present in the system. Of particular relevance to computational algorithms is the relationship between the range of the correlations and the range of the interaction potentials. Study of simple liquids has benefited enormously from the fact that, at least away from the critical point, the range of correlations is no greater than that of the potential, and consequently small systems of only 100 or so molecules can be studied. However, a point to bear in mind is that such simulations are usually performed in periodic boundaries. Density fluctuations will propagate through the system at the speed of sound, returning through the periodic boundary. Dynamics in the system at longer times can therefore be unrealistic unless the effect is reduced by increasing the system size and hence the repetition time. Another reason for working with a larger system is simply to gather more statistics. The most difficult cases computationally are ones where the range of correlations is much greater than the range of the potential, so that some method of locating near neighbours is required. There is great interest in systems close to phase transitions and critical points, where there may be long wavelength fluctuations, necessitating the use of very large numbers of particles.

Another challenge is provided by systems with slow relaxation times. In the study of protein dynamics, for example, it is usual to "freeze" the rapid bond-stretching vibrations. However, this still leaves motions present on a continuous range of time scales from fractions of a picosecond to nanoseconds or longer. There seems no alternative to very long simulations. This, while not presenting any particular computational problem in itself, is a great encouragement to making the basic energy and force evaluations as efficient and cheap as possible.

3. PARALLEL COMPUTERS

The excellent book by Hockney and Jesshope [7] gives a fascinating history of the introduction of parallelism into computing. For many years computers have possessed a degree of parallelism, for example incorporating separate peripheral processors, different functional units, and bit-parallel memory access and arithmetic. Nevertheless, from the 1950s to the late 1970s this parallelism was transparent to the user, and the underlying computational model incorporated in all high level languages was of a series of expression evaluations and assignments to memory, performed strictly in temporal sequence. Indeed, the concept of a numerical algorithm is usually defined as an analysis of a problem into a series of steps performed sequentially. A vast amount of intellectual effort has been put into performing such analyses, and it is ironic to realise that this model of computation became dominant largely through historical accident. The earliest commercially successful computers used different technologies for the processors (transistor logic) and memory (magnetic cores). The memories operated more slowly than the processors and there was therefore no gain in introducing multi-processor architectures. Compare the situation before the introduction of computers when, for example, a payroll office in a large organization would employ many accounting clerks. There is nothing particularly natural about sequential computation, although those of us who have spent much of our careers writing Fortran programs tend to think there is. As early as 1958 two-dimensional processor arrays were being designed [8].



The main driving force behind the current interest in the development of parallel computers is the demand by computational scientists for ever-increasing amounts of computer power. Although there has been a steady improvement in the necessary technologies, with the basic logic speeds of computers doubling every few years over the past decades, a law of diminishing returns has now set in. There is little prospect of substantially faster single processor designs on the horizon. Even were this possible, communication with memory would be a problem, since there is an absolute limit to the speed of point-to-point communications, namely the speed of light. A very fast processor would require a very compact memory, presenting enormous technological problems.

Now that processors and memory are constructed using the same semiconductor technology, and circuits with hundreds of thousands of logic components can be made

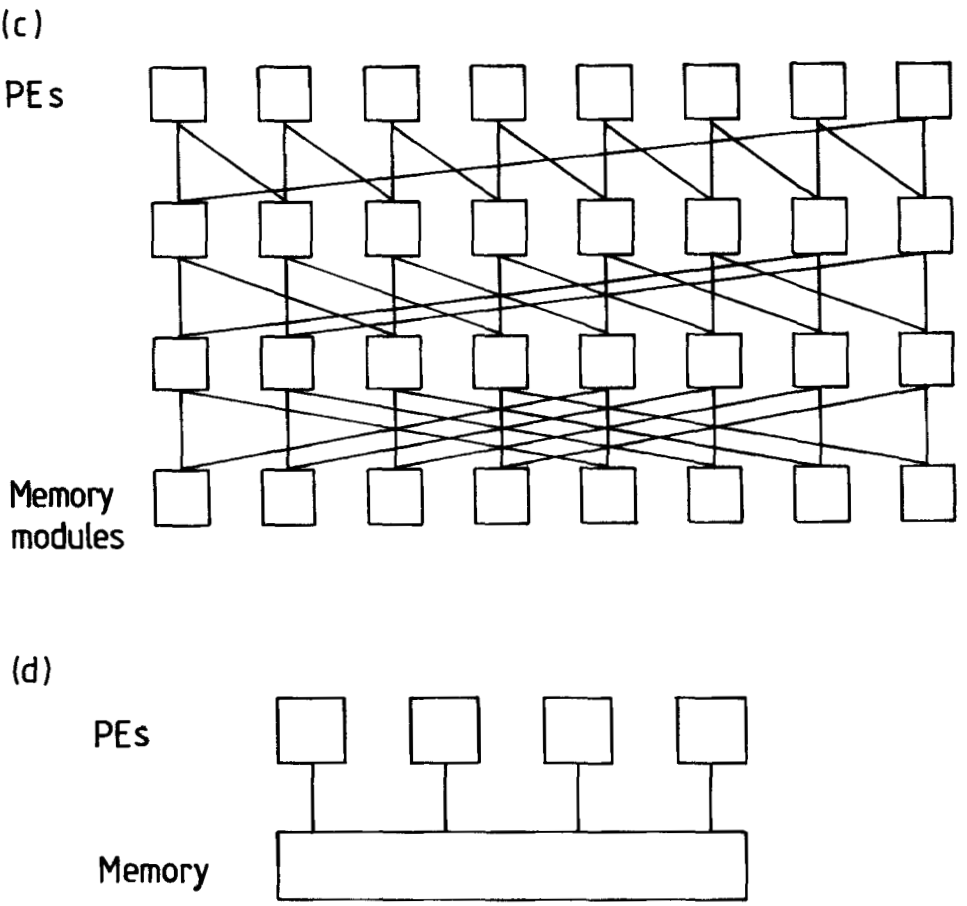


Figure 1 Different types of parallel computer. (a) The SIMD array of processing elements. (b) An MIMD multi-processor. (c) A switching network connecting processor and memory modules. (d) A shared-memory multi-processor.

on single silicon chips, there is the possibility of a much more radical approach to computer design. Individual processors, although slow in themselves, can be cheap and reliable and can be connected together in large numbers to make more powerful machines. The aim is either to produce a machine having the processing power and memory of a conventional mainframe, but at a much reduced cost, or to make machines with much greater power and memory than any ordinary computer, able to tackle the largest problems of computational science.

Parallel computer designs fall into one of two main categories. In SIMD (single instruction, multiple data) machines, each processing element (PE) acts as a slave to a master control unit (MCU) (Figure 1a). The MCU distributes an identical instruction to each PE: these act in lock-step, applying the instruction to operands in their own local memory. In any real problem there must be some communication of data between PEs, and in the figure the horizontal lines indicate data paths between neighbouring PEs. To a large extent the design of algorithms for parallel computers depends on making the data movements as local as possible.

The second category is the MIMD (multiple instruction, multiple data) computer, in which each processor executes its own instruction stream on its own data. Often each processor stores its program and data in a local memory, so that it is in itself a complete computer, perhaps a microprocessor. (The term concurrent processor, rather than parallel processor, may be used in this case.) Note that the design is more flexible than the SIMD computer, even if each processor executes an identical program, since the path through the program may be data dependent. There are local connections between processors, and usually a bus for global communications (Figure 1b). A wide variety of topologies for the local connections has been tried, including trees, pyramids, lattices, rings and hypercubes. An alternative approach to communication in an MIMD machine is the use of a switching network between processors and memory modules (Figure 1c). Each processor can access every part of the memory, but local communications involve a shorter routing and are therefore quicker. Yet another type of MIMD computer involves the use of a shared memory, directly addressable by each processor (Figure 1d). The number of processors that can be utilized in such a design is limited, because of the memory-to-processor communication problem already mentioned, and because of contention between different processors trying to access the same memory location simultaneously. In general, a second dimension of classification of parallel computer designs is provided by the distinction between machines with few fast processors and those with many, but slower processors.

The rest of this section describes in more detail some of the parallel computers that have been used in molecular simulation, or that show great promise in this field. The aim is to illustrate the different types of design by means of concrete examples, rather than to give an exhaustive catalogue of all experimental and commercial machines. In discussing specific computers we shall occasionally need to make comments about performance. It is conventional to measure computer performance in terms of Mflops (millions of floating point operations per second). While it can be useful to have a single measure like this, it should be used with extreme caution. Many problems are better formulated in terms of integer arithmetic (signal processing) or even logical arithmetic (Ising model); vector processors may suffer from start-up times; evaluation of functions may not be a matter of simple arithmetic (how many flops in a square-root?). A Mflop rate is meaningless unless it has been measured on a real problem. In particular one should beware of the peak Mflop rates often quoted by

manufacturers: real performance can easily be an order of magnitude less. In addition to Mflops, two other performance indicators are also relevant: memory bandwidth and communications bandwidth.

3.1 The ICL Distributed Array Processor

The ICL Distributed Array Processor (DAP) [7, 9] is a parallel computer that has been used extensively in statistical mechanics and molecular simulation. In the mid-1970s, its designers made a radical break with sequential tradition and decided to adopt parallelism on a massive scale. It is an SIMD machine consisting of 4096 PEs, each with 4 Kbits of local memory. The PEs have nearest-neighbour communications, being connected as a square 64 × 64 array (Figure 2). Such a topology is also appropriate for many other scientific problems, for example finite-difference mesh algorithms and image processing, and a variety of machines with broadly similar architectures has been constructed.

The DAP has several interesting features:

(a) The PEs are very simple one-bit processors, with all arithmetic being provided in software. This has various implications. The DAP has two modes of operation. In matrix mode the PEs operate in parallel on arrays of 4096 elements stored "vertically" (see Figure 2). In vector mode the PEs, using their nearest-neighbour connections, can act together to process vectors of 64 elements, each of up to 64 bits, and stored

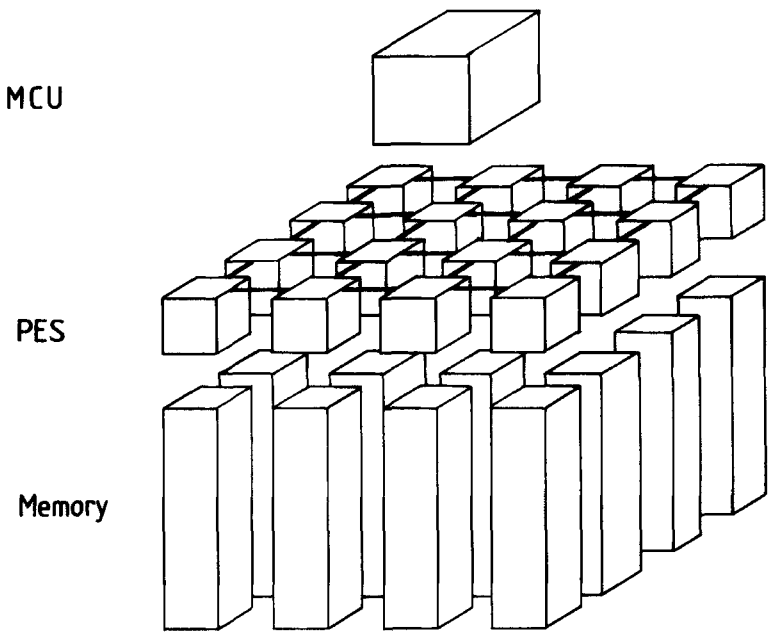


Figure 2 The ICL Distributed Array Processor (DAP). An idealized 4 × 4 DAP is shown; the real machine has a 64 × 64 array of processing elements (PEs). Each PE has direction connections to its four nearest neighbours. Not shown are row and column highways to registers in the Master Control Unit (MCU). Conventionally, one thinks of a "horizontal" plane of PEs accessing data in parallel from a store plane. A DAP Fortran variable of matrix mode is stored with the successive bits of each element stored "vertically" below the corresponding PE.

"horizontally". In matrix mode the arithmetic is bit-serial in nature, which means that arithmetic of any desired precision can be implemented, and execution speed is strongly dependent on word length. Bit-serial algorithms are available for common mathematical functions, with surprising consequences: for example, square root is faster than multiply.

(b) Each PE has an activity register that controls store operations to its memory. This register is under program control, and introduces some flexibility into the SIMD approach, as PEs can effectively be switched off for any operation.

(c) Global communications are available via highways linking rows or columns of the array to 64-bit registers in the MCU. The MCU can therefore broadcast data to the array, and collect data from it. These highways can also be used to implement periodic boundary conditions at the edges of the array, and to provide "long vectors" of 4096 elements.

The original DAP is integrated into a mainframe computer, with the DAP being essentially a modified memory module that can be directly addressed by the main CPU, as well as being capable of working on its own as a DAP. It is a very cost-effective design, providing near-supercomputer performance at much smaller cost. Execution rates on real problems with 32-bit arithmetic are around 15–20 Mflops, although Mflops are particularly misleading for the DAP since it can give much higher performance on problems using short word length, fixed-point arithmetic. Having a large number of processors with their own memory gives a high total memory bandwidth.

Currently, the DAP is being re-engineered as a 32 × 32 array at a higher level of integration and with more memory [10]. In this version it will be a stand-alone device hosted by a small minicomputer or graphics workstation. A major improvement is the provision of very fast input/output (40 Mbytes/s) direct to the array. This will enable it to support fast discs or a video output, or perhaps allow several machines to be connected together to form a larger array.

3.2 Vector Supercomputers

Vector processors, of which the Cray 1 [7, 11] was the first widely available example, are by now familiar tools of the computational scientist, with machines being available from a number of American and Japanese manufacturers. They operate on vectors of operands in pipelined mode: the functional units that perform the arithmetic overlap operations on successive elements of the vectors. Although they are not true multi-processors, they appear to the user in some ways like SIMD parallel computers. A loop that processes vector elements takes full advantage of pipelined operation (is fully "vectorizable") if three conditions are satisfied. First, it must perform identical operations on each iteration, having no data-dependent branches. Secondly, operations on successive vector elements must be logically independent, so that they can be overlapped. Thirdly, the vector elements must be contiguous in memory, or, some machines, equally spaced. These features of identical, independent operations on regularly distributed data also characterise SIMD processors, and lead to the success of very similar algorithms on the two types of machine. The advantage of the pipelined approach is that vectors may be of any length, without the complication or performance overhead of distributing them over a fixed number of processors.

To assist in vectorization it is often necessary to rearrange randomly addressed

data so that it is contiguous in memory, and many vector processors have hardware support for such data movements. Together with their usually good scalar (sequential) performance, this makes them effective general purpose computers. The main disadvantage is one of cost: they are supercomputers in price as well as performance. Several manufacturers have tried to overcome this problem by offering cheaper versions of the same basic architecture ("Crayettes").

In order to increase performance, supercomputer manufacturers are now developing multi-vector-processor shared memory designs. For example, the Cray XMP offers two or four processors. ETA Systems (a CDC spin-off) has developed the ETA 10, incorporating up to 8 vector-processing CPUs, and with shared memory. Remarkable progress has been made in the use of VLSI to reduce the size of the machine and hence increase performance: each CPU is a single circuit board, and a megaword of memory occupies a four-inch cube. The speed can be doubled by cooling to liquid nitrogen temperatures.

Peak performances of vector processors varies from around 40 Mflops for a Crayette to several Gigaflops for an ETA 10.

3.3 Hypercubes

A series of machines has been built at Caltech using standard microprocessors connected with a hypercubic topology [12, 13]. A d-dimensional hypercube contains 2^d nodes each with d connections (Figure 3a). The hypercube is a popular architecture because it limits the route length between two arbitrary nodes (maximum d links) and because many other topologies are subsets of it. In particular, it can be used to implement a lower-dimensional grid with periodic boundaries (Figure 3b). The disadvantages are that size increases must be by a factor of two, and that it may be difficult to produce hypercubes with a large number of nodes because of the large bandwidth required to support the requisite number of channels.

The original hypercubes use Intel 8086 chips at each node, with 8087 floating point coprocessors, and 0.25 or 0.5 Mbytes of memory. Each node has six point-to-point bi-directional communication channels, enabling hypercubes of up to 64 nodes to be constructed. The channels pass eight-byte packets between buffers in each node, which can be read from or written to by the node memory. There are two basic modes of operation of the channels, a "polling" mode and an "interrupt" mode. More is said about these in section 4.3, when we discuss the programming of concurrent machines. The hypercube has a separate processor as "cube-manager", which loads and controls the nodes via a communications bus. A commercial version is now available, the Intel iPSC [14], with more powerful chips and other improvements. The performance of the hypercube in terms of Megaflops is modest: about 4 Mflops with a 64-node iPSC.

The IPSC/VX series is a development of the iPSC in which each processor node is enhanced with a vector coprocessor. A 16-node system has a peak computational rate of 100 Mflops: 32- and 64-node versions are also available. The Floating Point Systems T-series [15] is a similar design, based on Transputers (see next section) as the nodes, each with a vector coprocessor. The four links of each Transputer are multiplexed up to 16 channels, and a hypercube topology is adopted. The system is modular, building up from $2^3 = 8$ nodes to a theoretical maximum of $2^{14} = 16\,384$ (which would fill a very large building)!

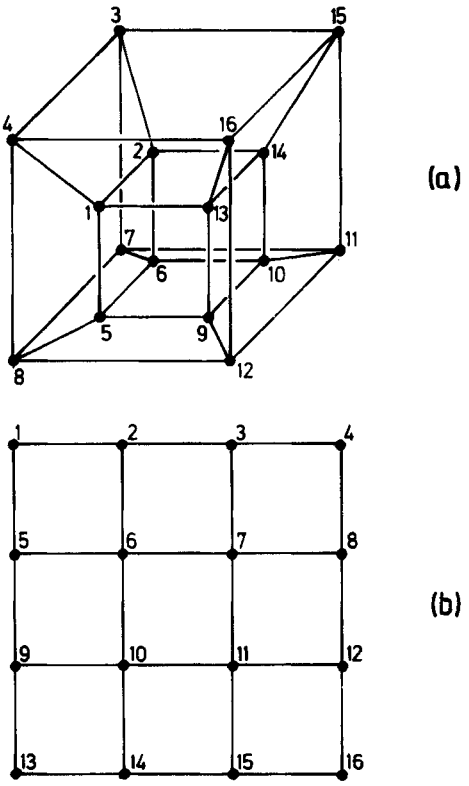


Figure 3 (a) A four-dimensional hypercube topology. (b) A two-dimensional grid with periodic boundaries is isomorphic to the hypercube of (a). For example, node 4 is connected to nodes 3, 8, 1 and 16.

3.4 The INMOS Transputer

The Transputer [16, 17] is, first, a very powerful 32-bit microprocessor, with performance exceeding that of many conventional minicomputers. Secondly, it has been designed from the start to support concurrent processing. Each Transputer has four "links" or communication channels, which enable Transputers to be connected together to form arrays of various topologies. (The links are in addition to the conventional memory interface, which can also support memory-mapped peripherals.) The links are bidirectional and transmit single-byte packets, and can run in parallel with the processor. The Transputer hardware directly implements the Occam model of concurrency, discussed in section 4.4; it is therefore easy to program, and programs for arrays can be tested on a single machine. Also, since the links work asynchronously, building Transputer arrays involves only low technology: literally a matter of connecting the wires.

At the time of writing the Transputer is available with fixed point hardware only, floating point arithmetic being provided in software or by coprocessor chips. However, a floating-point Transputer has been announced that will have a performance exceeding 1 Mflop. The links each operate at about 2 Mbytes/s. INMOS supply a Transputer evaluation module with up to 40 Transputers that can be connected in

arbitrary topology. Various manufacturers are beginning to announce other Transputer-based products, such as the Meiko Computing Surface [18].

A group at Southampton University is developing ARTS, the Array of Reconfigurable Transputer Supernodes [19]. This design is based on supernodes containing about 16 Transputers with their interconnections controlled by electronic switches. The idea is that the supernodes will be configured to match the algorithm in use. The supernodes will be connected together in a way that matches the geometry of the problem being considered.

3.5 Some Other Developments

Finally, in this section we mention briefly some other relevant developments.

Among many microprocessor arrays constructed, the Waterloop [20] is of particular interest, since it was specifically designed for Metropolis Monte Carlo simulation using an algorithm described in section 5.3. It connects processors in a ring, the architecture being described as a systolic loop. The term systolic indicates that data flows synchronously around the ring, undergoing some processing at each processor on the loop. The Waterloop is constructed very simply by connecting together commercially available single-board computers, and currently has 64 processors.

A major problem in multi-processor designs is the provision of efficient global as well as local communications. Several attempts have been made to overcome this problem by means of a hierarchical structure. The SUPRENUM project [21] aims to develop a multiprocessor supercomputer based on nodes each containing a MC68020 chip with floating-point coprocessor. Up to 16 nodes are connected by a bus to form a supernode; supernodes are then connected to form a square array using very fast row and column buses ("Upperbus"). This architecture is thought to be particularly suitable for multigrid solution of partial differential equations, but also to be applicable to problems in statistical physics. A more regular hierarchical structure is provided by the EGPA design [22], which connects processors in square planar arrays and then stacks the planes in a pyramidal manner (Figure 4).

A quite different approach to communications is provided by the Loosely Coupled Array of Processors (LCAP). In the original version of this system, two IBM 4300 minicomputers host 10 complete FPS 164 array processors (APs) [23]. There are no direct connections between the APs, which communicate only with the hosts. A virtue of this system is that standard hardware and software is used. A parallel task consists

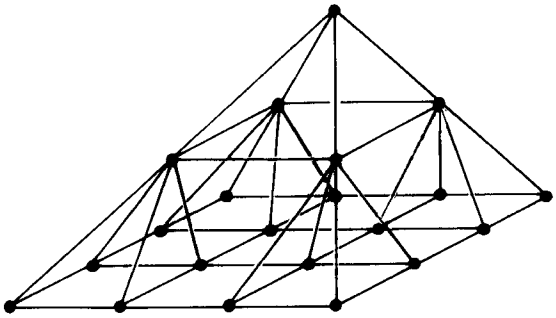


Figure 4 The EGPA hierarchical processor array.

of several Fortran programs each running in a virtual machine on the host, and controlling an AP on which further code runs. Overall performance is similar to that of Cray 1. More recently the machine has been upgraded with a second cluster of APs (FPS 264s rather than 164s), extra memory modules shared by several APs, and a direct communications bus connecting the APs.

A different architecture again is used in the GF11, a powerful parallel supercomputer being built at IBM Yorktown Heights [24]. This is an SIMD design with 576 processing elements. Each of these incorporates a 20-Mflops floating point scalar processor with 2 Mbytes of memory, giving a combined peak performance of 11.5 Gigaflops. Rather than having local connections as in other SIMD designs, the PEs are connected to a full 576 × 576 switch, so that any PE can communicate directly with any other PE.

4. LANGUAGES FOR PARALLEL COMPUTERS

To be useable, parallel computers must be programmable. Since conventional high-level languages implement the sequential model of computation, some thought must be given to appropriate language design.

4.1 Vectorizing Compilers and Library Routines

The earliest approach, adopted by the manufacturers of vector computers, was to accepted standard sequential Fortran but automatically to vectorize inner loops where successive iterations accessed vector elements in a regular and data-independent manner. For example,

```
      DO 1 J=1,N
1     A(J) = A(J) + B(J)*C(J)
```

However, it was also necessary to provide library routines to deal with more complicated loops. Consider the simple addition of the elements of a vector. A sequential programmer would automatically write

```
      SUM = A(1)
      DO 1 J=2,N
1     SUM = SUM + A(J)
```

without realizing that this is only one of many possible algorithms for performing a summation, and is non-vectorizable since the scalar quantity "SUM" must be written to at each iteration of the loop. Parallel evaluation of such a recursive sum can be done in a cascade manner (Figure 5), and accordingly a subroutine to do this is provided by the manufacturers. In the cascade method the degree of parallelism is continually reducing and such routines do not give the full performance of true vector arithmetic. Current versions of the Cray compiler can recognize recursive summations in a sequential loop and automatically insert the appropriate code. Another example is provided by the matrix product, where there are two possible parallel algorithms in addition to the sequential algorithm (Figure 6). While it is clearly advantageous to be able to move sequential code straight on to a vector processor and achieve some degree of vectorization, it is unrealistic to expect compilers to develop very much

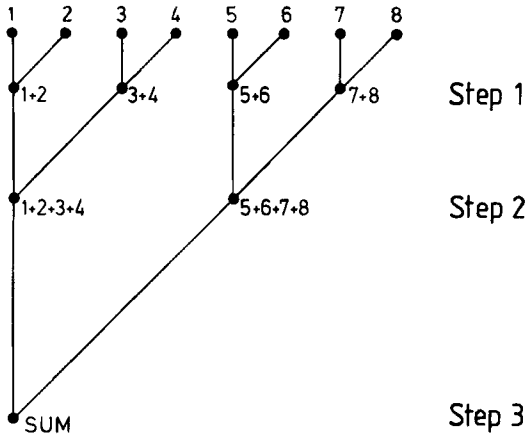


Figure 5 The cascade method of forming a sum, in this case of n elements where n is a power of 2. The total number of addition operations is $n - 1$, as in the sequential algorithms. There are $\log_{2n}$ steps, and the degree of parallelism halves at each step.


```
(a)

      DO 1 I=1,N
      DO 1 J=1,N
C the following loop is a scalar summation
      DO 1 K=1,N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
   1 CONTINUE

(b)

      DO 1 J=1,N
      DO 1 K=1,N
C the following loop is vectorisable
      DO 1 I=1,N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
   1 CONTINUE

(c)

      DO 1 K=1,N
C all iterations of the following pair of loops
C could be performed in parallel
      DO 1 I=1,N
      DO 1 J=1,N
         C(I,J) = C(I,J) + A(I,K)*B(K,J)
   1 CONTINUE

(d)

      DO 1 K=1,N
         C = C + MATC(A(,K))*MATR(B(K,))
   1 CONTINUE

(e)
      DO 1 K=1,N
         C = C + SPREAD(A(:,K),1,N)*SPREAD(B(K,:),2,N)
   1 CONTINUE
```

Figure 6 Matrix multiply algorithms: (a) inner product; (b) middle product; (c) outer product. In each case the matrix C has been initialized to zero. Algorithm (a) is sequential, since it has the form of a scalar sum. Algorithm (b) has a degree of parallelism n, since the inner loop vectorizes. Algorithm (c) has degree of parallelism n^2, since all operations in the two inner loops are independent. The DAP Fortran version of (c) is given in (d), and (e) is the Fortran 8X version.

facility in substituting one algorithm for another. One needs to be able to express problems at a higher level of abstraction, without decomposing them into sequential algorithms. Subroutine libraries are the first crude step in this direction.

4.2 DAP Fortran and Fortran 8X

DAP Fortran [7, 9, 25] contains simple extensions to Fortran to handle parallel operations on matrices and vectors. The declaration

```
REAL A(,),B(,),C(,)
```

declares A, B, and C to be 64 × 64 matrices, and the statement

```
C = A + B
```

adds the two matrices A and B on an element-by-element basis. Matrices and vectors can be defined with a wide range of precisions. Logical matrices occupy a single store plane, i.e. each element of the matrix is a single bit. Integer matrices can be defined with lengths of 1–8 bytes, and reals with lengths of 3–8 bytes. Logical matrix variables or expressions are used to select subsets of the 64 × 64 matrix space, such expressions taking the place of subscripts in sequential codes. For example

```
A(A.GT.2.0) = 2.0
```

selects the elements of A that are greater than 2.0 (i.e. those elements where the logical expression evaluates to .TRUE.) and sets them equal to 2.0, leaving the other elements unchanged. This corresponds to a hardware feature we have already mentioned: the activity registers in the PEs that can effectively switch them off for a given operation.

Sets of matrices (or vectors) can be defined. For example

```
REAL A(,,10)
```

is a set of 10 matrix variables. The 10 members of this set are stored below each other in the DAP store. A set of logical matrices occupies a stack of successive planes. It is possible to equivalence, say, a real variable to a logical set

```
REAL*4 A(,)
LOGICAL LA(,,32)
EQUIVALENCE (A,LA)
```

to give bit-level access to matrix A. The first element of the set LA corresponds to the most significant (sign) bit of A, and so on. Logical sets can also be used to write arithmetic routines of any required precision. Thus, DAP Fortran is both a lower-level language than conventional Fortran, and a higher-level language.

The language also has facilities for broadcasting data:

```
REAL A(,),B( )
.
.
A = MATC(B)
```

sets each column of the matrix A equal to the vector B. The intrinsic function MATC is an example of a rank-increasing function, since it takes a vector argument and returns a matrix result. There are also rank-reducing functions such as MAXV,

which finds the maximum value of its vector or matrix argument. Particularly important rank-reducing functions are those devoted to summation. For example

$$B = SUMC(A)$$

sets the vector B equal to the sum of the columns of A (i.e. an element of B is the sum of all the elements of the corresponding row of A). We have already mentioned parallel algorithms for summation; it is a feature of the DAP that it is able to use its PEs in a combination of matrix and vector modes in order to perform summations very efficiently. Figure 6d shows the DAP Fortran code for the outer-product matrix multiply algorithm.

The array-processing extensions in the proposed Fortran 8X standard [26] are similar to those in DAP Fortran, with generalizations to arrays of arbitrary size, shape and rank. Element-by-element operations on whole arrays can be performed by a single statement. The WHERE statement gives a similar facility to the use of logical subscript expressions in DAP Fortran:

```
WHERE(A.GT.2.0) A = 2.0
```

but there is also a block form. There are intrinsic rank-reducing and rank-increasing functions, but they are capable of being applied to any dimension of the array. As an example, the matrix outer product is expressed in terms of these functions in Figure 6e.

4.3 Programming MIMD Computers

The programming of MIMD computers introduces a new range of problems, which we will illustrate by reference to the Cray XMP, a shared-memory multi-processor super-computer, and the Caltech hypercube, a multi-microprocessor with communication channels between processors. In each case the processors will be running different sequential codes (tasks or processes) simultaneously, and these can be written in a standard high-level language. However, there need to be language extensions to control the interactions between processors.

The Cray XMP runs a time-sharing operating system so that separate jobs may occupy the different processors. It is not necessary for a program to use more than one processor, unless it requires very large amounts of memory or processor time. The techniques used in this case have been described by Hicks and Lynch [27]. Parallel tasks are created by a call to a library routine, which takes as an argument the name of the subroutine that is to constitute the task. The calling and called routine proceed in parallel with each other, i.e. run simultaneously and independently. The operating system automatically assigns the created task to a processor, and the task will normally run to completion. The calling routine is also able to call a wait routine, which suspends it until the called parallel task has completed. This will be necessary if the calling routine at some point needs to employ the results from the created parallel task. Special techniques must be used if parallel tasks need to coordinate their access to shared variables.

On the hypercube each node has its own memory and runs a separate process, and there is no question of shared variables. However, data need to be passed from one node to another over the connecting channels. There are two sets of support routines that may be called to achieve such communication, corresponding to the two modes of operation of the channels mentioned in section 3.3 above.

In "polling", mode communications are synchronized. A processor wishing to transmit data checks the transmit buffer and, if it is full, waits until it has emptied before copying the data from memory. A processor wishing to receive data will check the receive buffer. If it is empty, it waits until it has been filled before copying the data to memory. Thus, either of the processors wishing to communicate is able to wait if necessary, and the communication only proceeds when both are ready. Each communication is explicitly programmed in both processors, and the programmer will wish to minimize the waiting time. The processors will then run in a loose lockstep. This type of communication is most suitable for problems with a regular structure, and for this reason the appropriate set of support routines is called the Crystalline Operating System.

In the "interrupt" mode of communications, the channel hardware sends a signal to the node processor when data arrive. The processor suspends its current task and transfers control to a special routine called an interrupt handler, which copies the data to or from memory before proceeding with its original task. The set of support routines for this type of communication is called the Interrupt Driven Operating System and is quite sophisticated: for example, a data packet can have a header indicating its destination and automatically be passed from one node to another.

The attempt to run multi-processors by using extensions to standard sequential languages can lead to very complex programming. In the next section we describe a language designed from the start to support concurrent programming, which makes things much simpler. Another point relevant to programming should be mentioned here. An asynchronous multi-processor can give non-reproducible results. This does not necessarily mean they are wrong, as we will see when discussing Monte Carlo simulations. However, it does mean that debugging incorrect programs can be very difficult, and there seems to be no easy solution to this problem.

4.4 Occam

Occam [28, 17] is a language designed specifically to support concurrent computing and is the native language of the Transputer. The basic concepts in the language are the *process* and the *channel*. There are three primitive processes; input from a channel

channel ? variable

assignment to a variable

variable := expression

and output to a channel

channel ! expression

Other processes are built up from these three primitive processes in a hierarchical manner, using *constructors*. A constructor with its component processes is itself a process. Two important constructors are SEQ and PAR, whose component processes are executed sequentially and in parallel, respectively. Parallel processes communicate only by means of channels; one processes outputs over the channel and the other process inputs from the channel. Either process will wait until the other is ready before the data transfer takes place. (This is very similar to the Crystalline Operating System of the Caltech hypercube.) Parallel processes may not read from or write to the same channel. Nor are there shared variables between parallel processes (not even different

```
CHAN comms:
PAR
  WHILE TRUE
    INT x:
    SEQ
      chan1 ? x
      comms ! x*x
  WHILE TRUE
    INT y:
    SEQ
      comms ? y
      chan2 ! y*y
```

Figure 7 Occam code for a pipeline that takes in a value x and squares it in the first stage, and squares it again in the second stage. The two stages are processes running in parallel and communicating over the channel "comms". In the code indentation denotes components of a constructor; thus, the PAR has two components corresponding to the two stages of the pipeline. The WHILE constructor creates a loop of its components, and each iteration has an input and output performed in sequence. Declarations are attached to processes by means of the colon: note that channels are declared in the same way as ordinary variables.

```
WHILE TRUE
  CHAN input1, input2, output:
  REAL x:
  ALT
    input1 ? x
      output ! x
    input2 ? x
      output ! x
```

Figure 8 Occam code for a demultiplexor that simply passes on the first input it receives.

elements of the same array); if a process changes a variable by assignment or input, it is illegal for a parallel process to refer to it. Instead, variables are declared locally to the process referring to them, and values are transmitted between the processes over channels. This carefully controlled interaction between parallel processes makes Occam programming much simpler and more reliable than alternatives based on existing sequential languages. Figure 7 gives a simple example of Occam programming.

Parallel processes can run on separate Transputers, when the channels are implemented over the links, each link being able to support one channel in each direction. Parallel processes can also execute on a single processor, when channels are implemented by memory locations, and the processes are timesliced. Thus, a processor can continue with other work while it is waiting to transmit or receive data over a link. Also, programs for a multi-Transputer network operate in a logically identical manner on a single processor: this aids the program development process.

Another important constructor is ALT, which has a number of inputs associated with it and executes the component process that *first* receives an input (Figure 8). This constructor takes the place of the concept of an interrupt on a conventional processor, and provides a much more regular and controlled way of handling asynchronous communications.

5. SIMULATION ALGORITHMS

The design of algorithms for parallel computers involves analysing the problem into elements that are logically independent and so capable of being calculated in parallel, and the mapping of these problem elements onto the processing elements of the

computer. An examination of molecular simulation from this point of view reveals several sources of parallelism. Let us take as our first simple example the two-dimensional Ising model. A configuration of the system consists of a set of spins arranged on a square lattice, each of which is either "up" or "down". Conventionally, the spins are examined one by one (either sequentially or at random). The energy change that would result from "flipping" the chosen spin is evaluated so the Metropolis test can be applied and the new configuration accepted or rejected.

The calculation of the trial energy change involves comparing the direction of the chosen spin with its four nearest neighbours. Here there is possible parallelism, since there is no reason why the four comparisons may not be made simultaneously. Are there higher levels of parallelism? Yes, because it is possible in principle to form a new trial configuration by flipping more than one spin simultaneously. However, this tends not to be done in practice on the grounds that larger fluctuations of this nature are less likely to be accepted under the Metropolis criterion, and the overall efficiency of the simulation will decrease. (But see the remarks below about Monte Carlo liquid-state simulations.)

It is more effective in the case of the Ising model to create many *different* configurations simultaneously. Imagine colouring the spins "black" or "white" in a chessboard pattern. A black spin interacts only with its nearest-neighbour white spins. Thus, any number of black spins can be flipped simultaneously to form a set of trial configurations whose acceptance or rejection are independent. This level of parallelism is possible because of the short-range nature of the interactions in this model. Finally, it should be remembered that our purpose in performing simulations is to obtain ensemble averages. There is no objection to using more than one Markov chain for the purpose, and so it is possible to run several simulations simultaneously and average their results.

The above example illustrates that problems in molecular simulation can possess a high degree of intrinsic parallelism. In the rest of this section we describe implementations of specific algorithms on specific parallel computers, and will see that there are three basic strategies that may be adopted. We may map onto the processing elements particles, pair interactions or regions of space.

5.1 Lattice Simulations on the DAP

The DAP is the parallel computer *par excellence* for performing lattice simulations. Interaction sites can be mapped directly onto the 4096 processing elements, and the nearest-neighbour connections used in evaluating the site–site interactions. We will examine in detail the implementation of the solid-on-solid (SOS) model [29], which provides a good example of the type of algorithm that is adopted and its programming in a parallel language. The essentials of the simplest program for this problem are shown in Figure 9.

The SOS model represents a solid surface as a set of columns of "atoms" of various heights above or below a reference level. Each column can change its height by loosing an atom or gaining an atom. The model exhibits a Roughening Transition at a temperature above which the free energy of surface features is zero. The properties of the model and this transition can be investigated by Monte Carlo simulation. In the version of the model represented by the program the columns are arranged on a square 64×64 lattice and the energy of a column depends on the sum of the absolute differences in height between it and its four nearest neighbours. In the program UP

```
      INTEGER*2 HEIGHT(,),THEIGHT(,),ENERGY(,),TENERGY(,)
      INTEGER*2 SOUTH(,),NORTH(,),WEST(,),EAST(,)
      REAL*4 RAND(,)
      LOGICAL BLACK(,),UP(,),ACCEPT(,)
      .
      .
C initialise
      .
      .
C loop over iterations
      DO 100 JIT=1,NITS
      RAND = RANDOMOFREALS
      UP = RANDOMOFLOGICALS
C trial heights
      THEIGHT(UP) = HEIGHT+1
      THEIGHT(.NOT.UP) = HEIGHT-1
C loop over black and white columns
      DO 101 JB=1,2
C calculate trial energies
      TENERGY = ABS(THEIGHT-SOUTH)
     1         + ABS(THEIGHT-NORTH)
     2         + ABS(THEIGHT-WEST )
     3         + ABS(THEIGHT-EAST )
C apply Metropolis criterion
      ACCEPT = EXP(-(TENERGY-ENERGY)*BETA).GE.RAND
      ACCEPT = ACCEPT.AND.BLACK
      HEIGHT(ACCEPT) = THEIGHT
C calculate energies using new heights
      SOUTH = SHNC(HEIGHT)
      NORTH = SHSC(HEIGHT)
      WEST  = SHEC(HEIGHT)
      EAST  = SHWC(HEIGHT)
      ENERGY = ABS(HEIGHT-SOUTH)
     1       + ABS(HEIGHT-NORTH)
     2       + ABS(HEIGHT-WEST )
     3       + ABS(HEIGHT-EAST )
C interchange black and white
      BLACK = .NOT.BLACK
  101 CONTINUE
C accumulate energies
      SUMENERGY = SUMENERGY+0.5*SUM(ENERGY)
  100 CONTINUE
```

Figure 9 Essentials of a simple DAP Fortran program for the solid-on-solid model.

is a plane of random bits, used as a subscript mask to set trial moves in which the height of each column is increased or decreased by one. To compare the heights of columns with their neighbours, intrinsic "shift" functions are used. For example, the matrix SOUTH is obtained by shifting the matrix of heights one place to the north. The matrix expression HEIGHT-SOUTH then gives, for each column in parallel, the difference in height between it and its southern neighbour. Thus, the energies TENERGY of the columns that would result from the trial moves are obtained. The Metropolis criterion is applied and a logical mask ACCEPT set up for acceptable moves. This is within a loop over the black and then white columns of the chessboard ordering.

In this algorithm, periodic boundary conditions are automatically included because the shift functions perform cyclic shifts (non-cyclic shift functions are also available). The code is much more compact and clear than in a sequential version, partly for this reason, partly because of the use of array-valued variables and expressions, and partly because logical masks are used to identify subscript sets by name. The black–white ordering means that the efficiency of utilization of the processors is only 50%. Although we may not like the idea of running a computer at half power, it is perhaps worth remembering that many scientific computers perform all arithmetic at 64-bit precision, even though many problems could manage with much lower precision. In fact, even with this inefficient algorithm the DAP is providing cost-effective

computing with impressive performance, about 250 complete iterations (i.e. 250 times 4096 equals 10^6 column updates) per second. The algorithm has made use of the parallel creation of trial configurations, but no other level of parallelism.

The most direct and obvious mapping for the SOS model, of a 64 × 64 problem directly onto the DAP, only achieved half the maximum possible efficiency. This point emphasizes that the design of algorithms for processor arrays is essentially a matter of finding the best data structure. The goal is to utilize the processors fully, and to minimize data communications. In the case of two-dimensional lattices with black–white ordering, this can be achieved by simulating a 128 × 64 problem, using a type of mapping known as "crinkling" (Figure 10), in which neighbouring elements appear in successive DAP planes, and each plane consists of all black or all white sites.

The DAP has been used extensively to perform various three-dimensional Ising calculations [30]. In one 64 × 64 × 64 calculation [31], a black–white ordering was adopted, but full efficiency of the processors was achieved by mapping two successive horizontal planes onto the plane of processors, so that all the black sites on the two planes could be updated simultaneously (Figure 11). All the black sites in the cube were updated before all the white sites. Another calculation was of a 128 × 128 × 144 lattice [32]. In this case it was possible to map a 2 × 2 × 2 sub-cube of the problem onto each processing element. This calculation used assembly language programming, and table look-up for transition probabilities, and achieved very high performance: 218 million spin updates per second. In each of these cases the mapping was arranged so that the DAP Fortran cyclic shift functions, utilizing the hardwired cyclic shifts of the machine, were able correctly to implement the periodic boundary conditions of the problem. Also, full efficiency was achieved by arranging for each plane to consist of all black or all white sites. Another simulation studied an Ising model for the real ferrimagnetic lithium ferrite, involving 4320 atoms [33]. In this more complex type of problem, it is necessary to use some masking-out of processors, and extra code to handle some of the boundary conditions.

More complicated Monte Carlo simulations than Ising models have been performed. Allen [34] considered orientational motions of linear molecules on a two-dimensional triangular lattice, the molecules being confined to the plane. There were quadrupole–quadrupole interactions between nearest neighbours only. Use of a rhomboidal 128 × 128 system allows a crinkled mapping of four "colours" (Figure

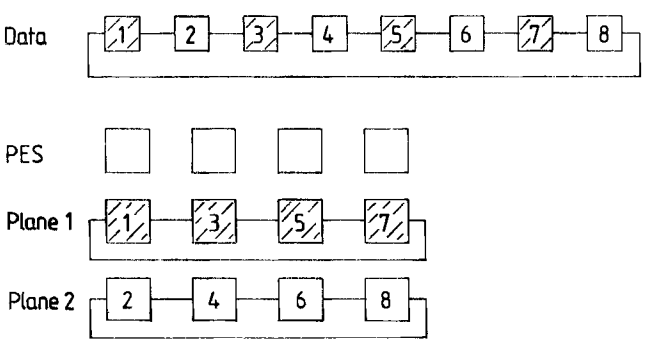


Figure 10 The "crinkled" method of mapping a 128 × 64 problem with black–white ordering onto the DAP. For clarity, the case of a linear DAP of eight PEs is shown. To find, for example, the right-hand neighbour of each black spin we look below it. To find the left-hand neighbour of each black spin, we look below and to the left, taking into account the cyclic boundary conditions.

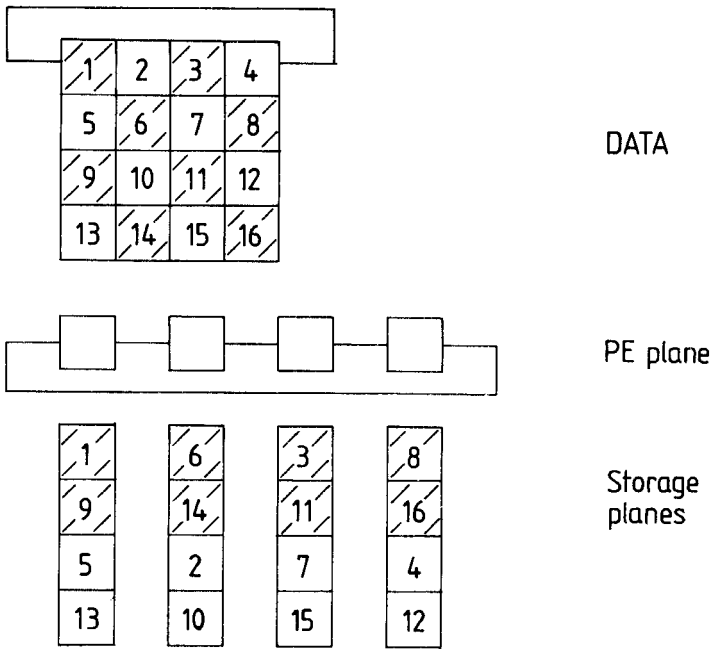


Figure 11 Type of mapping adopted for a 64 × 64 × 64 simulation on the DAP, here illustrated for the case of a four-PE linear DAP. To find, for example, the right-hand neighbour of each black square we look one place to the right and two places downwards. Periodic boundaries in the horizontal direction are handled automatically by using the hardware cyclic boundaries of the DAP. Periodic boundaries in the vertical direction must be handled explicitly by the program.

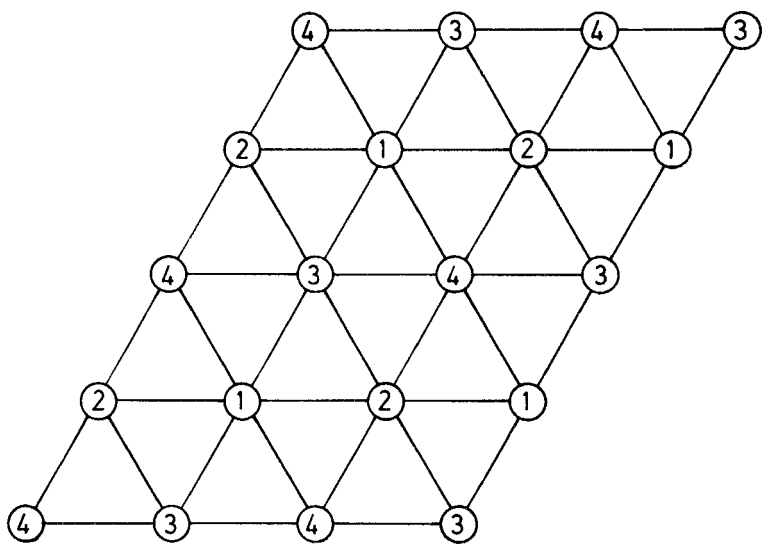


Figure 12 A crinkled mapping of a triangular 128 × 128 lattice onto the DAP. Sites have four "colours", here numbered 1–4. Sites of the same colour are updated simultaneously.

12). Since there is no interaction between sites of the same colour, trial configurations can be obtained by moving all 4096 sites of the same colour simultaneously. In this work, part of the analysis involved searching for domains of molecules having one of the three possible herring-bone structures. Such searches can be performed on the DAP by moving out from a single site in a series of steps in which the frontier moves in all directions simultaneously, until the boundary of the domain is encountered. Only logical arithmetic is involved, and the algorithm is therefore very fast.

An important consideration in all Monte Carlo simulations is the random-number generator. It must be efficient, have a repeat length greater than any feasible calculation, and satisfy various criteria for quality. The DAP benefits from a fast random-number generator of the shift-register type, which produces very high quality random numbers [35].

Molecular dynamics of lattice systems has also been performed on the DAP, in the series of calculations by Pawley, Dove and others [36, 37]. In molecular dynamics the total force on each molecule is accumulated, and then the molecules are moved. In the type of algorithm used in these calculations one molecule is assigned to each processing element. For each molecule in parallel it is then possible to evaluate the interaction with a particular neighbour, using the shift functions. The loop over neighbours can extend out to as many neighbours as the range of the potential requires. Newton's Third Law means that each pair of neighbours is examined only once, the pair force being added to the force accumulator for one particle and subtracted from the accumulator for the other particle. Once the total force on the molecules has been obtained, their equations of motion can be integrated over the timestep, in parallel. If rigid non-spherical molecules are modelled, torques as well as forces must be obtained. Both translational and rotational motion can be studied, but the translational motions must not destroy the lattice structure, since this is assumed in the search for neighbours.

These molecular dynamics calculations mapped a three-dimensional lattice of 4096 molecules onto the two-dimensional DAP. The nature of such mappings is discussed

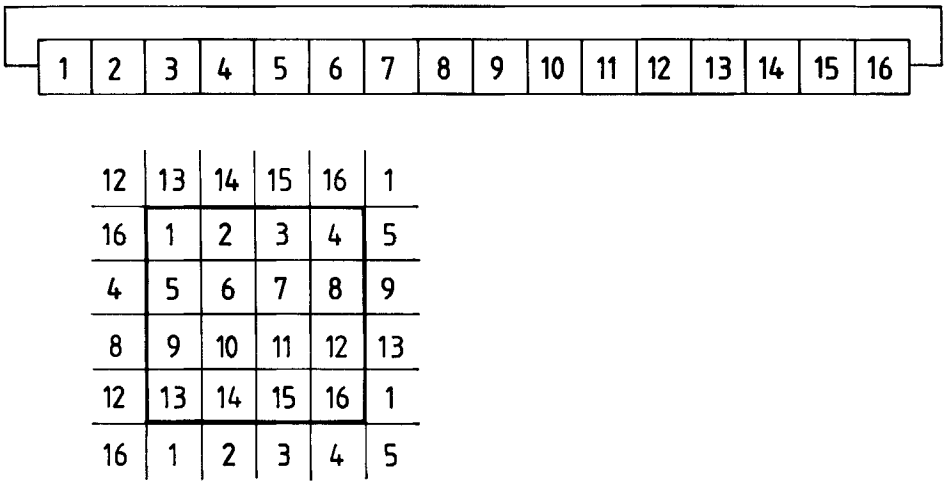


Figure 13 Mapping a lattice simulation problem onto a processor array of lower dimension, illustrated for the case of a two-dimensional system on a one-dimensional array. The periodic boundaries of the array lead to skewed periodic boundaries for the physical system being simulated.

in detail by Pawley and Thomas [38]. Mapping a lattice simulation onto a processor array of lower dimension means that the hardwired periodic boundaries of the array lead to skew periodic boundaries in the simulated system (Figure 13). This skewness is not usually a problem since the periodic boundaries are, anyway, an artefact incorporated in the attempt to simulate an infinite system with a finite number of particles without introducing surface effects.

5.2 Simulation of Fluids on SIMD Computers

The simulation of fluids differs from that of lattice problems because the neighbours of a given molecules are always changing, and because structural correlations are generally of short range. Here we are concerned for simplicity with fluids consisting of spherical molecules (particles), and will consider first molecular dynamics simulation. There is no difference in principle in extending the arguments to small rigid molecules modelled by site–site interactions.

Often, such simulations are performed with a small number of molecules, 108 and 256 being popular choices, and the simplest technique is to treat every molecule as a neighbour of every other molecule. It is instructive to consider a conventional sequential algorithm (Figure 14) and then analyse for parallelism. A double loop examines each pair of molecules, and the vector separating them is found. Periodic boundaries are applied by means of the nearest-image convention (Figure 15). Then a spherical cut-off is applied, the pair force being evaluated only for pairs now separated by less than a distance, the cut-off radius, which can be regarded as the range of the potential. The pair force is added to the force accumulator for one molecule and subtracted from that of the other (Newton's Third Law). When all the forces have been found, the equation of motion of each molecule is integrated over the timestep.

The obvious parallelisms here are: first, in the evaluation of the pair forces, since

```
      DO 1 I=1,NUMBER-1
      DO 1 J=I+1,NUMBER
         XIJ = X(I)-X(J)
         YIJ = Y(I)-Y(J)
         ZIJ = Z(I)-Z(J)
         CALL PERIODIC(XIJ,YIJ,ZIJ)
         RSQ = XIJ**2+YIJ**2+ZIJ**2
         IF(RSQ.LE.RCUTSQ) THEN
            FDIVR = FORCE(RSQ)
            FXP = FDIVR*XIJ
            FYP = FDIVR*YIJ
            FZP = FDIVR*ZIJ
            FX(I) = FX(I)+FXP
            FY(I) = FY(I)+FYP
            FZ(I) = FZ(I)+FZP
            FX(J) = FX(J)-FXP
            FY(J) = FY(J)-FYP
            FZ(J) = FZ(J)-FZP
         ENDIF
    1 CONTINUE
      DO 2 I=1,NUMBER
         VX(I) = VX(I)+FACTOR*FX(I)
         VY(I) = VY(I)+FACTOR*FY(I)
         VZ(I) = VZ(I)+FACTOR*FZ(I)
         X(I) = X(I)+VX(I)
         Y(I) = Y(I)+VY(I)
         Z(I) = Z(I)+VZ(I)
         CALL PERIODIC(X(I),Y(I),Z(I))
    2 CONTINUE
```

Figure 14 Outline of force and dynamics loops for a sequential molecular dynamics program for a simple fluid. In practice boundaries and force evaluation would be performed by in-line code. This example is for spherical molecules, with the equations of motion being integrated by a leap-frog algorithm. There is no difference in principle in extending the algorithm to rigid polyatomic molecules.

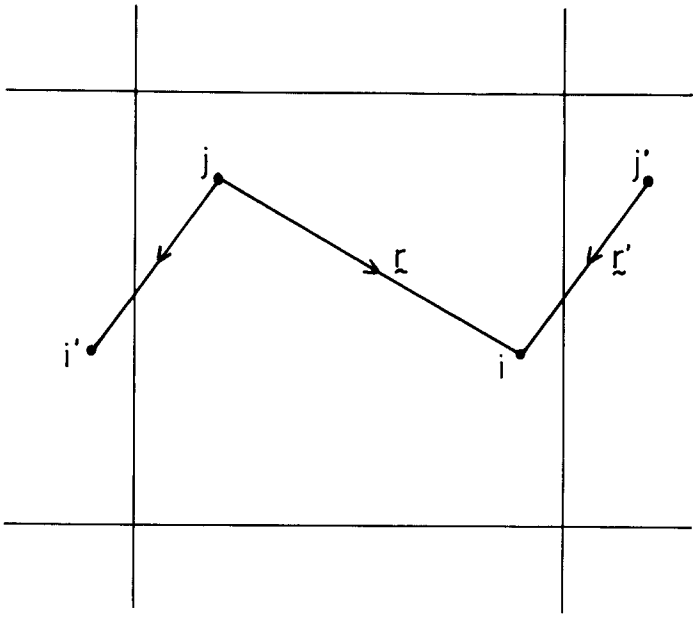


Figure 15 The nearest-image convention. The vector **r** is replaced by the vector **r**′ if the latter is shorter. This convention is consistent with the spherical cut-off, providing the cut-off radius is no greater than half the box length (for a cubic box).

```
SEQ
  PAR
    xij := x[i] - x[j]
    yij := y[i] - y[j]
    zij := z[i] - z[j]
  periodic (xij,yij,zij)
  rsq := xij*xij + yij*yij + zij*zij
  IF
    rsq <= rcutsq
      SEQ
        rsqr := 1.0/rsq
        ratio6 := rsqr**3
        ratio12 := ratio6*ratio6
        difference := ratio12 - ratio6
        PAR
          energy := energy + difference
          rf := ratio12 + difference
        PAR
          virial := virial - rf
          g := rf*rsqr
        PAR
          fxij := g*xij
          fyij := g*yij
          fzij := g*zij
        PAR
          fx[i] := fx[i] + fxij
          fy[i] := fy[i] + fyij
          fz[i] := fz[i] + fzij
        PAR
          fx[j] := fx[j] - fxij
          fy[j] := fy[j] - fyij
          fz[j] := fz[j] - fzij
```

Figure 16 Pseudo-Occam code showing the extent of possible parallelism within the evaluation of the Lennard-Jones pair interaction.

all $N(N-1)/2$ pair forces are logically independent; and secondly, in the integration of the equations of motion, where the N molecules can be moved independently. Before discussing the exploitation of these parallelisms, it is worth mentioning other levels of parallelism in the algorithm. There is scope for parallel evaluation within each pair force, particularly over the three Cartesian coordinates (Figure 16). The only exploitation of this kind of parallelism to date is in special purpose computers (see section 6.2). Also, it is possible to update the position of each "i" particle as soon as its inner "j" loop is complete (Figure 17), so in principle the integration of the equations of motion could proceed partly in parallel with the force evaluation. However, this has never been done.

The vectorization of the inner loop in the above simple algorithm was discussed several years ago [39]. The IF statement inhibits vectorization, since pipelining requires all loop iterations to be treated identically, and so it is removed. Vectorization is thus achieved at the expense of evaluating all pair interactions in the system, instead of employing a cut-off. (Interactions outside the cut-off radius are often set to zero so that simple long-range corrections for thermodynamic quantities can still be applied). The accumulation of $FX(J)$ etc. does not affect vectorization but that of $FX(I)$ does, since $FX(I)$ is a scalar as far as the inner loop is concerned. This can be handled by storing the pair forces appropriate to each "i" particle in an array and summing them outside the inner loop using the library summation routine. Nowadays on the Cray the loop can be left alone as the compiler will automatically adopt this strategy. The inner vectorized loop has a length that decreases as the outer "i" index increases. This is no particular disadvantage on the Cray, which gives good performance on short loops.

An improved method of vectorizing such an "all-pairs" algorithm has been given by Brode and Ahrlichs [40], which is particularly suitable for machines, such as the Cyber 205, that like longer vector loops. By changing the order in which pair interactions are evaluated, it also avoids the use of a scalar summation for the forces. It works as follows. We assume there are N molecules and, for the sake of simplicity, take N to be odd. The coordinate arrays are doubled up; for example, the X coordinates are stored in a vector of length $2N$, with $X(N+1) \ldots X(2N)$ being a copy of $X(1) \ldots X(N)$. All the $N(N-1)/2$ pair separations can then be calculated in the way illustrated in Figure 18, which involves $(N-1)/2$ vectorizable loops, each of length N. The loops are vectorizable because there is a constant increment between the pairs in each loop; 1 in the first loop, 2 in the second, and so on. Once the pair separations have been obtained and stored in a vector, they can be used to evaluate the pair forces in a vectorizable loop, in principle of length $N(N-1)/2$. (If there is not enough memory for arrays of this length to store pair separations and forces, it is possible to break the calculation into blocks of length N or greater.) The total forces are then

```
      DO 1 I=1,NUMBER
         DO 2 J=I+1,NUMBER
            XIJ=X(I)-X(J)
            .
            .
            FZ(J)=FZ(J)-FZP
2        CONTINUE
         VX(I)=VX(I)+FACTOR*FX(I)
         .
         .
         CALL PERIODIC(X(I),Y(I),Z(I))
1     CONTINUE
```

Figure 17 A rewrite of the algorithm of Figure 14 in which the position of moleucle "i" is updated as soon as the force on it has been found.

```
(a)

      OFFSET = 0
      DO 1 IL=1,(N-1)/2
         DO 2 I=1,N
            XIJ(OFFSET+I) = X(I) - X(IL+I)
  2      CONTINUE
         OFFSET = OFFSET + N
  1   CONTINUE

(b)

      OFFSET = 0
      DO 3 IL=1,(N-1)/2
         DO 4 I=1,N
            FX(I) = FX(I) + FXIJ(OFSET+I)
            FX(IL+I) = FX(IL+I) - FXIJ(OFFSET+I)
  4      CONTINUE
         OFFSET = OFFSET + N
  3   CONTINUE
      DO 5 I=1,N
         FX(I) = FX(I) + FX(I+N)
  5   CONTINUE
```

(c)

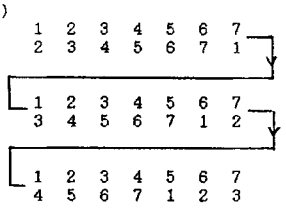


Figure 18 A fully vectorizable method of evaluating and accumulating pair forces for molecular dynamics in an "all-pairs" strategy. (a) Evaluation of the pair separations. (b) Accumulation of pair forces. (c) The ordering of pairs illustrated for the case $N = 7$; there are three vectorizable loops, each of length 7. The coordinate arrays are doubled-up, so that, for example, $X(8)$ contains a copy of $X(1)$.

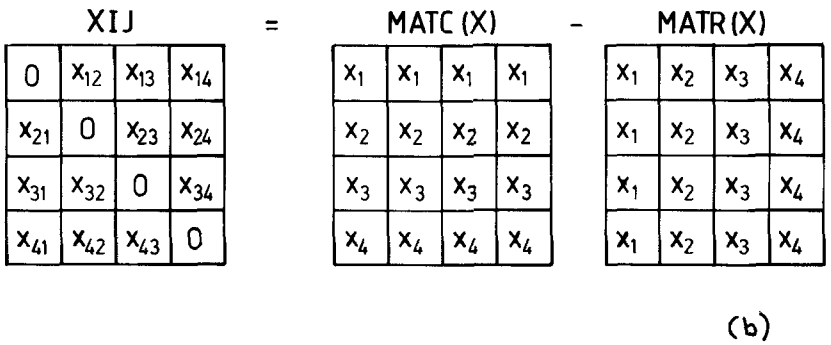
accumulated from the pair forces, again involving $(N - 1)/2$ vectorizable loops of length N (Figure 18b). The force vectors, like the coordinate vectors, are of double length, and the final stage sums the two halves.

The "all-pairs" strategy also works well on the DAP, emphasizing the similarity between vector processors and SIMD parallel computers. In principle, the algorithm of the last paragraph adapts immediately to parallel processing with the vectorizable loops of length N being replaced by parallel evaluation of N pair forces; cyclic boundaries in long-vector mode would be used instead of doubled-up arrays. However, there is little reason to use an all-pairs method with as many as 4096 molecules. Instead, an alternative approach is used that is more suitable for smaller numbers of molecules. Figure 19 shows DAP Fortran code for the force evaluation for the case of 64 molecules, where all 4096 pair forces can be evaluated simultaneously. However, the efficiency is only 50%, since no use is made of Newton's Third Law. Real problems require the use of larger numbers of particles, when the pair interactions are evaluated in 64×64 blocks. Only the diagonal and upper triangular blocks then need to be considered, so the efficiency rises. This technique has been used extensively, in studies of molecular liquid mixtures [41] and in determining the dielectric constants of liquids of polar molecules [42].

Some programs on sequential computers use look-up tables for the pair energies and forces, usually with linear interpolation between grid points, rather than direct evaluation of the appropriate expressions. This can be advantageous when the

(a)

```
REAL X(),Y(),Z()
REAL XIJ(,),YIJ(,),ZIJ(,),RF(,)
LOGICAL OFFDIAGONAL(,)
.
.
XIJ = MATC(X)-MATR(X)
YIJ = MATC(Y)-MATR(Y)
ZIJ = MATC(Z)-MATR(Z)
CALL PERIODIC(XIJ,YIJ,ZIJ)
RSQ = XIJ*XIJ+YIJ*YIJ+ZIJ*ZIJ
RSQR = 0.0
RSQR(OFFDIAGONAL.AND.RSQ.LE.RCUTSQ) = 1./RSQ
FDIVR = FORCE(RSQR)
FXIJ = FDIVR*XIJ
FYIJ = FDIVR*YIJ
FZIJ = FDIVR*ZIJ
FX = SUMC(FXIJ)
FY = SUMC(FYIJ)
FZ = SUMC(FZIJ)
```



(b)

Figure 19 (a) Outline of DAP Fortrain molecular dynamics force evaluation for a 64-molecule system. (b) Diagram showin „ for a 4 × 4 DAP, how built-in functions are used to convert from a vector of coordinates to a napping of pair interactions onto the DAP plane.

potential model has a complicated form or is derived in tabular form by *ab initio* calculations. Table look-up is an example of the random addressing that inhibits vectorization, and on the Cray-1 it is best not to use it, even for quite complicated potentials [43]. On the DAP, table look-up is likely to be inefficient, but in compensation the DAP is more efficient than other computers at evaluating complicated potentials involving exponentials etc., since bit-level algorithms make these almost as quick as ordinary arithmetic operations.

Liquid simulations involve the use of periodic boundaries. The maximum range to which correlations can be measured is the radius of a sphere inscribed in the computational cell. With an "all-pairs" algorithm it is important to achieve computational efficiency by maximizing the range for a given number of particles (i.e. for a given cell volume), or conversely by minimizing the number of particles for a given range. This requires the use of a cell that is as nearly spherical as possible, so that there are few "wasted" interactions in the "corners" of the nearest-image cell. Of the regular space-filling solids, the truncated octahedron is the most suitable, since as well as being reasonably spherical its periodic boundaries are coded easily and efficiently [44, 45]. On the DAP both cubic and truncated octahedral boundaries can be very efficiently applied using single-bit operations [46] (see Figure 20). We expect truncated-octahedral boundaries to become standard for vector and parallel computers.

One further point should be made specific to the DAP. The degree of parallelism in the integration of the equations of motion is equal to the number of molecules,

```
REAL X(,),Y(,),Z(,),ONE(,),TWO(,)
LOGICAL XSIGN(,),YSIGN(,),ZSIGN(,),TWOSIGN(,)
EQUIVALENCE (X,XSIGN),(Y,YSIGN),(Z,ZSIGN)
EQUIVALENCE (TWO,TWOSIGN)
LOGICAL CORNERS(,)
ONE = 1.0
TWO = 2.0
.
.
TWOSIGN = .NOT.XSIGN
X(ABS(X).GE.1.0) = X + TWO
TWOSIGN = .NOT.YSIGN
Y(ABS(Y).GE.1.0) = Y + TWO
TWOSIGN = .NOT.ZSIGN
Z(ABS(Z).GE.1.0) = Z + TWO
CORNERS = ABS(X)+ABS(Y)+ABS(Z).GE.1.5
ONESIGN = .NOT.XSIGN
X(CORNERS) = X + ONE
ONESIGN = .NOT.YSIGN
Y(CORNERS) = Y + ONE
ONESIGN = .NOT.ZSIGN
Z(CORNERS) = Z + ONE
```

Figure 20 DAP Fortran code for periodic truncated octahedral boundaries. The truncated octahedron is formed from a cube by cutting off the corners until half the volume remains. The internal units are such that all coordinates lie between -1 and $+1$. The equivalences set logical variables corresponding to the sign bit of real variables. For example, in the first step if X is greater than one, two is subtracted from it. The first transformations on X, Y and Z correspond to cubic boundaries: the extra transformations complete the application of truncated-octahedral boundaries.

typically a couple of hundred, and so much less than the number of processors. This part of the problem is therefore solved inefficiently, and the time spent on it can become a significant proportion of the whole, unlike in the case of a vector processor. It is therefore sensible to make the code for this as simple and as efficient as possible. (The possibility of doing the three Cartesian components in parallel should also be mentioned, although this has not been used). Fortunately, the most stable algorithm for centre-of-mass motion is the leapfrog [47], which is very simple. For rotational motion of rigid molecules simple algorithms are also available, either a constraint algorithm for linear molecules [48] or an explicit two-step (modified leapfrog) algorithm for the general case [49, 2].

The algorithms we have classified as "all-pairs" are simple and very suitable for SIMD processing. However, since the early work of Verlet [50], there has been a school of though in the fluid simulation field that has preferred to use rather larger numbers of particles, and this of course is essential if there are long-wavelength correlations in the fluid. In an all-pairs algorithm the amount of work increases with the square of the number of particles, and this quickly becomes very inefficient, since we really need to evaluate only the interactions between particles separated by less than the cut-off radius for the potential. Another complication is that in a fluid particles do not remain in a fixed relationship with each other.

On sequential computers two methods have been designed to cope with this problem [51]. The "neighbour-list" method [52, 53] maintains a complete list of pairs separated (taking into account the periodic boundaries) by less than a specified distance which is somewhat larger than the cut-off. In evaluating the forces only pairs in this list are considered. The list is completely re-formed at intervals, say every 10 time-steps, by examining all pairs. In the "cell" method [54, 55] the computational box is divided into a number of sub-boxes or cells. At each step of the calculation every particle is assigned to a cell by examining its coordinates. Then only interactions between particles in the same cell, or in nearby cells separated by less than the cut-off,

need to be considered at all. Usually the cells are cubic with side approximately equal to the cutoff, so that only adjacent (nearest and next-nearest) cell pairs need be considered. The method is advantageous only if there are at least 4^3 cells, since otherwise every cell would be a neighbour of every other cell when the periodic boundaries are taken into account. This means the method is suitable for numbers of particles of around 2000 or more, with its execution time then scaling as N, the number of particles. The neighbour-list method is suitable for rather smaller numbers of particles, both because of the storage required for the list, and because the computational work still has a component proportional to the square of N, due to the periodic reformation of the list, at least in the standard method. It is possible to use pre-sorting of particles according to their Cartesian coordinates to speed up the creation of the neighbour list [56], giving the method something in common with the cell technique.

Neither of these methods vectorizes particularly well, since in each case accessing particle coordinate arrays according to indexes in the pair list or cell list is an example of random addressing. It is necessary to call library "GATHER" routines, which rearrange the required sets of particle coordinates into contiguous regions of memory so that the force evaluation can proceed in vector mode. Then "SCATTER" routines distribute the forces back into the appropriate places in the force arrays. On the Cray-1 it was found [39] that with 256 particles there was hardly any gain in execution speed because of the lack of hardware support for GATHER/SCATTER. Later machines with such hardware support obtain better gains [56, 57].

An alternative approach to determining neighbours in large systems has been described by Boris [58]. Like the cell method this has an execution time increasing less quickly than N^2, but unlike the cell method it is fully vectorizable. The data pertaining to each particle (position, velocity, mass etc.) are stored in arrays indexed by (i, j, k) such that the X (or Y or Z) positions of the particles increase monotonically with index i (or j or k, respectively). The coordinate arrays define a Monotonic Logical Grid (MLG), satisfying

$$\begin{aligned} X(i, j, k) &< X(i + 1, j, k) \quad \text{for } 1 \leqslant i \leqslant NX - 1 \\ Y(i, j, k) &< Y(i, j + 1, k) \quad \text{for } 1 \leqslant j \leqslant NY - 1 \\ Z(i, j, k) &< Z(i, j, k + 1) \quad \text{for } 1 \leqslant k \leqslant NZ - 1 \end{aligned}$$

where the dimensions NX, NY, NZ of the arrays are fixed, usually equal to each other, and obey $N = NX \times NY \times NZ$. Thus, the indices (i, j, k) map each particle onto one of the nodes of a distorted cubic grid. The initial assignment of particles to this grid can be done by standard vectorizable sorting routines in times that scale as $N \log N$. The updating of the grid after each time-step, to take account of particles that have changed relative positions in any dimension, can be achieved by an iterative algorithm in which vectorizable loops swap data between neighbouring positions in the MLG arrays. The geometric sorting of the particle indices means that the neighbours of a given particle are found in a contiguous region of memory, and the interactions can be calculated in a vectorizable loop over nearby nodes of the grid. However, given the approximate cubic nature of the grid, around 50% of these neighbouring nodes will be outside the spherical cut-off radius: just as with the "all-pairs" algorithm, vectorization is achieved only by evaluating extra interactions. If, as is usually the case with molecular dynamics, one needs to be able to follow the trajectories of individual particles, it is necessary to include the constant absolute

index of each particle as one of the data items stored in the variable (i, j, k) position of the MLG. A GATHER type operation is then required to sort particle coordinates step by step into memory or disk files for analysis.

On the DAP a neighbour-list method has been implemented [59]. Neighbours were marked in a bit-mask set up for all pairs, and a compression technique used to gather the in-range pairs into 64 × 64 blocks for force evaluation. The cell technique as conventionally implemented is not suitable for the DAP, and liquid state simulations with large numbers of particles have not been performed. One technique might be to use a cell method with one cell per processor. Because of the incompressibility of molecules it would be possible to arrange that a cell held either zero or one molecules, but not more. Then a true SIMD algorithm could be used in which one looped over neighbouring cells exactly as one loops over neighbouring processors in the three-dimensional lattice simulations discussed above. A degree of inefficiency would result from considering interactions with empty cells, and the proportion of these would depend on the density. The inefficiency would be reduced by making the cells as nearly spherical in shape as possible. Alternatively, it might be possible to use a monotonic grid method to assign molecules uniquely to processors.

Fluids with long-range electrostatic forces, due to the presence of ions or dipolar molecules, do not present any particularly difficult problems. The most common method of dealing with such a system is by way of the Ewald sum, which breaks the sum over images into two rapidly converging sums, one in real space and one in reciprocal space. The real-space sum is handled along with the other interactions using the methods just described. The reciprocal space part of the Ewald sum vectorizes easily [43].

An alternative method of handling the reciproal space contribution is by a particle-mesh technique [54, 60, 61] using fast Fourier transforms. These are also very suitable for vectorization or parallel processing. Yet another method, applied recently on the DAP [62], involves the use of effective short-range pair potential approximations to the true Ewald sum. In this method the effective potentials must be applied between all pairs in the computational box, without use of a spherical cut-off. This makes the all-pairs algorithm on parallel computers particularly appropriate. The method is also much simpler to use than the exact Ewald sum, and we expect it to become standard for molecular systems.

So far we have discussed how to do the simulations, but said nothing about the analysis of results. One important type of analysis is the study of structural correlations by means of the radial distribution function. This is essentially a histogram of pair separations in the fluid, and since these separations are determined when finding the forces it is usual on sequential computers to accumulate the histogram in the main forces loop. On a vector computer the indirect addressing involved in forming a histogram would prevent vectorization. Thus, the pair separations are usually stored in an array, and then histogrammed in a separate, non-vectorizable loop.

On the DAP a different approach is used [63]. The code for this is shown in Figure 21. One loops over bins of the histogram, rather than over particles. The matrix of pair separations is examined in a parallel operation to determine how many separations fall inside the current bin. The code relies entirely on logical arithmetic and comparisons. The random addressing problem has gone away because we are accessing the DAP store by content rather than by addressing in the usual sense. This is possible because the DAP can rapidly search through its memory locations in parallel on a bit-by-bit basis. On both the vector processor and the DAP, incorporation of the

```
      REAL VALUES(,),X(NUMBIN)
      INTEGER NUMBER(NUMBIN)
      LOGICAL BIN(,),ALREADY(,)
      .
      .
      ALREADY = .FALSE.
      DO 1 JBIN=1,NUMBIN
         BIN = (VALUES.LE.X(JBIN)).AND.(.NOT.ALREADY)
         NUMBER(JBIN) = SUM(BIN)
         ALREADY = ALREADY.OR.BIN
    1 CONTINUE
```

Figure 21 DAP Fortran code for forming a histogram. X(JBIN) is the upper limit of bin JBIN of the histogram.

radial distribution function will slow things down. However, this is not serious because there is no requirement to accumulate at every step of the simulation. Structural correlations persist over several time-steps and the accumulation can be done, say, every 10 steps without loss of statistical efficiency. The other important type of analysis in molecular dynamics is the determination of time-correlation functions. There are no problems in doing these in vector or parallel mode, and they do not contribute significantly to the total computer time.

Monte Carlo simulation of fluids is often regarded as being difficult to implement on SIMD computers. This is because, conventionally, only one molecule is moved per step and so the degree of parallelism in the energy evaluation is of order N rather than N^2. This can be a problem on processors that like long vectors, and an even bigger problem on the DAP. In the latter case, Adams [64] has shown how to use the DAP effectively by running a number of Markov chains simultaneously, making use of a higher level of parallelism which we mentioned above. There is a certain loss of efficiency involved in equilibrating many rather than one system. Recently it has been shown [65] that Monte Carlo fluid simulations can in fact work efficiently if trial configurations are created by moving several particles simultaneously (at least up to 32). This method could be used to increase the degree of parallelism and hence the effectiveness of Monte Carlo. In a system with a large number of molecules it would be possible to create several configurations simultaneously by moving particles separated by more than the range of the interaction, in a way similar in principle to the use of black–white ordering in the Ising type of problem. We describe a multi-processor version of this technique in the next section.

Much of what has been said above in discussing the simulation of fluids of small molecules is also relevant to the simulation of flexible macromolecules, if the word "molecule" or "particle" is replaced by "atom". The main difference is that the types of interaction are more heterogeneous, with bond-stretching, angle-bending, and torsional interactions as well as non-bonded interactions of the van der Waals and electrostatic types familiar from small-molecule simulations. Most programs use lists of atoms involved in each type of interaction. For non-bonded pairs the list is similar to the ordinary neighbour list, except that bonded pairs are excluded. The other lists are more complex: for example, the angle-bending list must index the three sites involved. In most cases the lists will also carry an extra index indicating the type of interaction and parameters to be employed. Such programs only vectorize partially because of the random addressing involved in the use of lists, and would be extremely difficult to implement on true SIMD computers such as the DAP. The best chance of progress for this type of problem lies in the use of MIMD multi-processors.

5.3 Multi-processor Algorithms

Much less experience has been gained with multi-processor MIMD algorithms than with SIMD algorithms. The simplest approach is to adapt an SIMD algorithm, which can easily be done for the "all-pairs" method. Figure 18 is a good starting point since it represents a fully vectorized (i.e. true SIMD) all-pairs algorithm for molecular dynamics. One simply replaces the vectorized loops of length N, where N is the (odd) number of molecules, by N processes working in parallel. Rather than examine this algorithm in detail, we will consider a simpler version [66], in which there are $(N - 1)/2$ parallel processes. The discussion is in terms of processes rather than processors, since it is always possible in principle to implement such algorithms with more than one process per processor, even it in reality the processes on a given processor run in sequence. On Transputer arrays it makes little difference whether the parallel processes run on the same or different processors, and the algorithm can utilise from one to $(N - 1)/2$ processors.

It works as follows (Figure 22). The processes are connected by channels in such a way that the coordinates of the molecules can circulate: the force accumulator for each molecule circulates with its coordinates. At each stage two molecules are present in each process, which evaluates the relevant pair force, adding it to and subtracting it from the force accumulators of the two molecules. After N steps every molecule has met every other molecule in a process, and the total forces have therefore been obtained. The parallel processes can now update the velocities and coordinates of their two current molecules, and are then ready for the next molecular dynamics step. Since molecules circulate round the processes, undergoing some processing at each stage, we call this algorithm "Pass the Parcel" [67]. This is a systolic algorithm, since the data movements are, at least approximately, in step.

An alternative systolic all-pairs algorithm is the "Tractor-tread", which was developed specifically for the Waterloop [68]. It uses $N/2$ processors (N even) connected in a loop by unidirectional channels, and is best understood by considering an example, Figure 23. Each processor contains an "upper" molecule and a "lower" molecule. The lower molecules remain fixed (the roadway), while the upper molecules circulate cyclically (the tractor tread). At each alternate step one processor is not active in calculating a pair interaction: instead, it maintains the tractor tread by exchanging upper and lower molecules. (Conceptually, the tractor moves over a roadway that forms a vertical circle; this makes the analogy rather hard to follow.) All pairs have been considered after $(N - 1)/2$ steps.

This algorithm can be used for Metropolis Monte Carlo as well as for molecular dynamics: indeed it was initially introduced specifically for this purpose [69]. As can be seen from the figure, any one molecule interacts with the other molecules strictly in ascending numerical (modulo N) order. For example, before the first step molecule 1 can make a trial move, and then its interactions with molecules 2–6 are evaluated on this and subsequent steps, so that after step 5 the move can be accepted or rejected. A trial move is made for molecule 2 before step 3, and its interactions with the other molecules evaluated on this and subsequent steps, and so on. This procedure correctly creates a chain of moves in which the individual molecule positions are updated independently in numerical order. A complete Monte Carlo sweep, in which each molecule is moved once, occupies two cycles of inter-processor moves.

A good example of a true asynchronous multi-processor application is provided by the Monte Carlo simulation in which a two-dimensional Lennard-Jones fluid of 1024

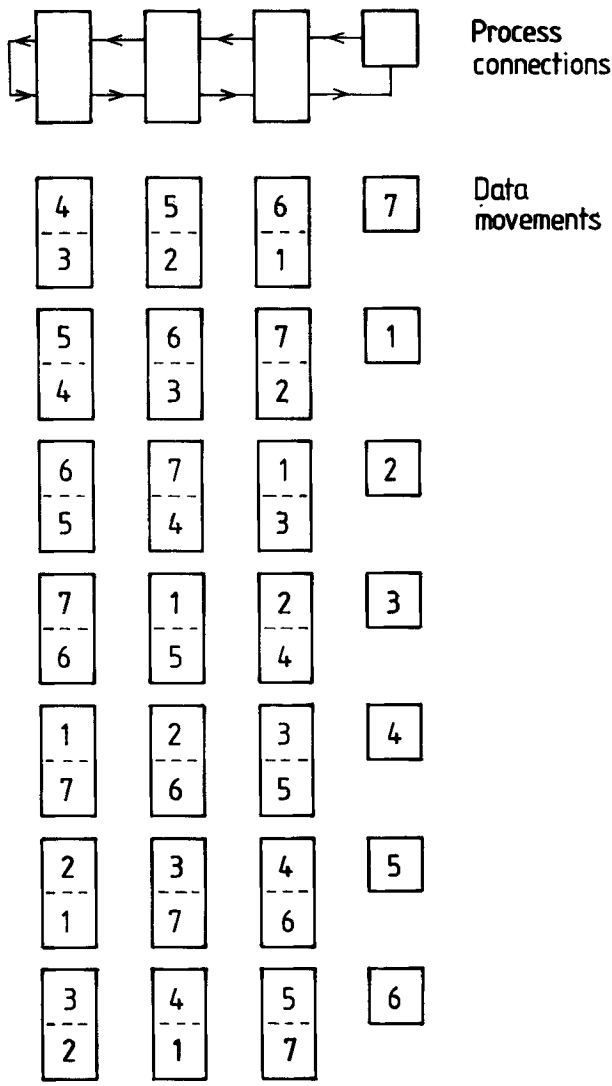


Figure 22 "Pass the Parcel" algorithm for molecular dynamics on multi-processors, illustrated for the case of seven molecules and three parallel processes. The molecule coordinates and force accumulators circulate, with the processes evaluating pair interactions at each step. After seven steps all the pair forces have been evaluated.

particles was simulated on a 64-node Caltech hypercube [70]. In this type of calculation a cell algorithm is used, each processor node being assigned to one cell of an 8×8 square grid with periodic boundaries. Thus, the average number of particles per cell is 16. The density and the range of the interaction are such that only particles in the same or adjacent cells can interact. Because of the irregular nature of a fluid simulation, with varying numbers of particles per cell, the Interrupt Driven Operating System is used.

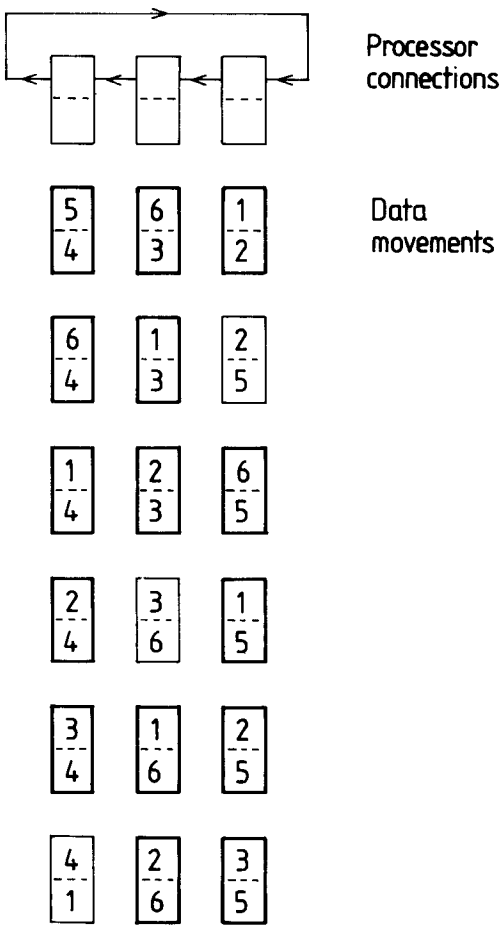


Figure 23 The tractor-tread algorithm for molecular dynamics or Monte Carlo on a processor ring, illustrate for the case of six molecules and three processors. The upper molecules circulate right to left. The outlined processors evaluate pair interactions. On alternate steps one processor does not evaluate an interaction but maintains the tractor-tread motion by interchanging upper and lower molecules; the positions after this interchange are shown. After six steps all 15 pair interactions have been evaluated, completing one molecular dynamics step: a complete Monte Carlo sweep requires twice as many steps.

The processors work simultaneously and asynchronously. Each processor creates new configurations by moving its particles one by one. Non-adjacent processors can work completely independently. When a processor makes a trial move, it first determines which of the adjacent cells may be in range and sends the old and new coordinates of the moved particle to the adjacent cells. The adjacent processor works out the resulting energy change due to interactions with its in-range particles, and returns the result. While waiting for a response the first processor calculates the energy change of interaction of the moved particle with the other particles in the same cell. Conflicts arise when a processor receives a request concerning a particle it is currently updating. In this case one processor waits until the other has completed its current update. To decide which update proceeds, update requests are labelled with the time the update commences, and the one timed first proceeds first. Assignment of particles

to cells is made at the end of each sweep, i.e. when every particle has been updated once.

The achievement of reasonable efficiency with a multi-processor algorithm requires approximate balancing of loads between the processors, so they are rarely idle waiting for results from a neighbour, and little loss of time in communications between processors. The algorithm just described has an efficiency of around 80%, which is excellent considering the hypercube is built from standard microprocessors.

One interesting feature of Monte Carlo runs on such an asynchronous multi-processor is that the sequence of configurations may be irreproducible. Slight differences in clocking of the chips on a repeat run may interchange the temporal order of two updates. From this point the two Markov chains will diverge. Physical results, which are averages over the chains, will, of course, be unchanged.

Molecular dynamics simulations have been performed with very large numbers of soft-sphere particles (around 160 000) on a coupled set of four FPS 264 processors [71]. The aim was to simulate, in two dimensions, fluid flow around an object, and to compare the results with those from continuum hydrodynamics. A region of the computational space is assigned to each processor. Because of the large size of the space compared with the range of the potential, only particles close to the boundary of the regions need to be exchanged between processors.

The Southampton group has used Transputers to study the generalized XY lattice model by Monte Carlo simulation [72]. A 64 $\times$ 64 lattice was simulated with 16 Transputers, plus a controlling Transputer, each one handling a "vertical" 64 $\times$ 4 strip. The model has spins able to rotate continuously in the plane, with nearest-neighbour interactions only. Horizontal communications between strips are required when updating spins at the edges of the strips, and then synchronization is required to ensure that different configurations are created by independent moves.

The latter three examples illustrate geometric decomposition of the problem onto the processors. Algorithm decomposition are also possible, in which different stages in the algorithm are assigned to different processors. For example, the author is programming in Occam an algorithm for performing molecular dynamics of flexible macromolecules. In such calculations there may be many heterogeneous interactions and it is necessary to maintain lists of various interacting atoms. To be specific we will concentrate on the way in which pair interactions may be evaluated. A process "Organiser" maintains coordinates of the atoms and the lists of interacting atoms. A series of identical processes, "Evaluator", are connected with Organiser in a ring (Figure 24). The ring is divided into outer and inner parts. To evaluate pair interactions, Organiser sends out pair separation vectors over the outer ring. These circulate until they find an Evaluator that is free. The job of Evaluator is to take a pair separation and evaluate from it the corresponding pair force. The pair forces then circulate around the inner ring. The Organiser can take pair forces one by one from the inner ring and accumulate them until the total force on each atom is obtained. Both pair separations and pair forces need to be labelled by the indices of the particles involved in the interaction, since there is no specified order in which they are evaluated. As a data packet circulates until it finds a free process to utilize it, we call this algorithm "Musical Chairs" [73]. It illustrates nicely one of the ways in which processors running completely asynchronously can be programmed in Occam to cooperate successfully. Some of the Occam code involved is shown in Figure 25. Further algorithmic decomposition of this problem would be possible; for example, evaluation of the pair forces could be broken down into stages in a pipelined mode,

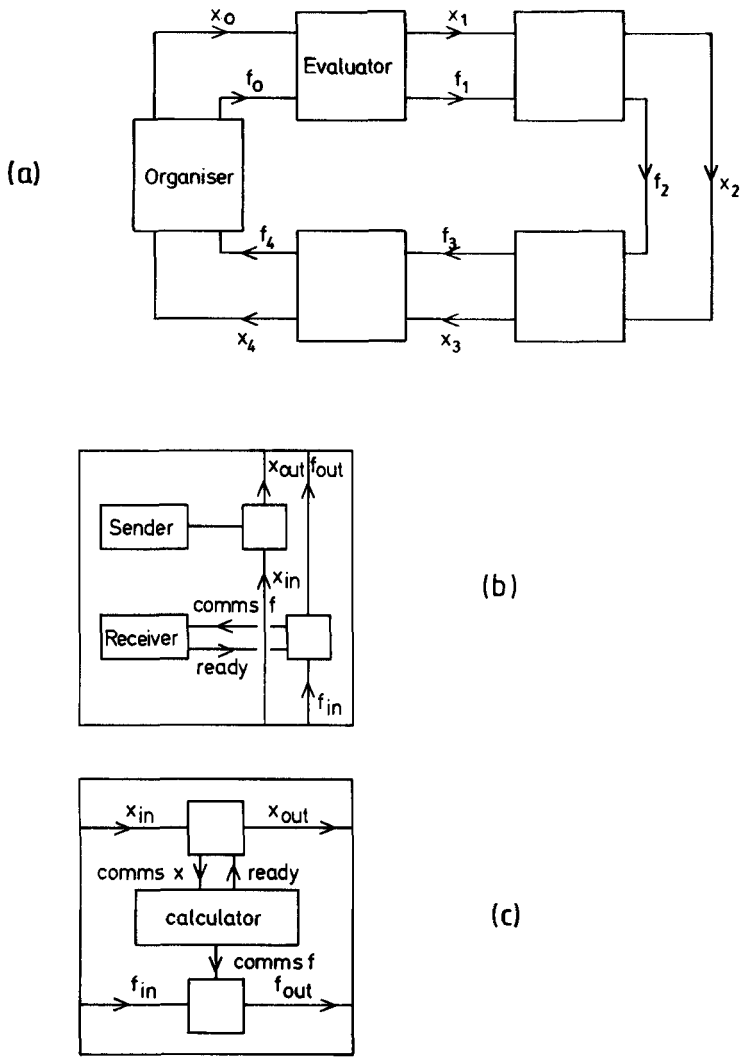


Figure 24 (a) A parallel asynchronous algorithm. "Musical Chairs", for the force evaluations in a polymer dynamics program. The Evaluator processes calculate pair forces, while Organiser sends out pair separations to the outer ring, and collects and accumulates the pair forces from the inner ring. (b) The sub-proceses and communication channels within Organiser (c) The sub-processes and communication channels within Evaluator.

some of them involving parallel operations on the x, y, and z components, as in Figure 16.

Some of the relative advantages and disadvantages of geometric and algorithmic parallelism have been discussed [74]. Geometric parallelism, in fairly regular problems, leads to reasonable load balancing. Algorithmic parallelism needs careful design to balance loads between, say, different stages of a pipeline. Also, with processors handling a small part of the computational task, communication overheads can

```
PROC Evaluator(CHAN xin,xout,fin,fout)=
  -- this procedure is replicated around the ring
  CHAN commsx,commsf,ready:
  PAR
    --  x switch
    WHILE TRUE
      VAR ii,jj,x:
      ALT                    -- if force calculator is ready to accept
        ready ? ANY          -- coordinates pass them on. otherwise
          SEQ                -- pass them round the ring
            xin ? ii;jj;x
            commsx ! ii;jj;x
        xin ? ii;jj;x
          xout ! ii;jj;x
    --  force calculator
    WHILE TRUE
      VAR ii,jj,x,f:
      SEQ
        ready ! ANY
        commsx ? ii;jj;x
        f:=force(x)
        commsf ! ii;jj;f
    --  f switch
    WHILE TRUE
      VAR ii,jj,f:
      ALT
        commsf ? ii;jj;f    -- if force evaluation is ready to provide
          fout ! ii;jj;f    -- forces pass them on. otherwise keep
        fin ? ii;jj;f       -- forces circulating
          fout ! ii;jj;f:



--  organiser
PROC Organiser (CHAN xin,xout,fin,fout)=
  CHAN commsx,ready,commsf:
  PAR
    -- sender
    SEQ il=[1 FOR number.list]
      commsx ! ilist[il];jlist[il];x[ilist]-x[jlist]
    -- x switch
    WHILE TRUE
      VAR ii,jj,x:
      ALT
        commsx ? ii;jj;x    -- if sender is outputting coords pass them
          xout ! ii;jj;x    -- round the ring. otherwise keep the ring
        xin ? ii;jj;x       -- circulating
          xout ! ii;jj;x
    --  f switch
    WHILE TRUE
      VAR ii,jj,f:
      ALT
        ready ? ANY      -- if receiver signals ready send forces to it
          SEQ            -- otherwise just keep them circulating
            fin ? ii;jj;f
            commsf ! ii;jj;f
        fin ? ii;jj;f
          fout ! ii;jj;f
    --  receiver
    WHILE TRUE
    VAR ii,jj,f:
      SEQ
        ready ! ANY
        commsf ? ii;jj;f:    -- receive forces
--------------------------------------------------
--  body of program
CHAN x[5],f[5]:
PAR
  Organiser(x[4],x[0],f[4],f[0])
  PAR i=[1 FOR 4]
    Evaluator(x[i-1],x[i],f[i-1],f[i])
```

Figure 25 Some of the Occam code corresponding to Figure 24. In reality the pair separations and pair forces are three-dimensional vectors.

become more dominant. A compensating advantage is that the processors may have very small memory requirements – in the case of the Transputer perhaps using on-chip memory only – and this can lead to greater speed and cost effectiveness. Algorithmic parallelism is more easily adaptable to problems of varying size, and in general to less regular problems. Incorporation of algorithmic parallelism has much in common with the design of special purpose computers, which is the subject of the next section.

6. SPECIAL PURPOSE COMPUTERS

In this section we describe some special purpose computers, for the Ising model and for molecular dynamics. While not strictly parallel computers according to our criterion, they have some parallel features, and are interesting as an alternative approach to obtaining cost-effective computing for particular problems. Special purpose computers can provide very high-performance computing at modest cost, because they can match their design to the algorithm, using pipelining and parallelism where appropriate; they can work at the most appropriate word length, and provide exactly the right amount of high speed memory.

6.1 Ising Model Computers

Ising model processors have been built at Delft [75], Santa Barbara [76] and Murray Hill [77, 78]. Figure 26 illustrates the Santa Barbara machine. The spin memory of 2 Mbit is implemented by means of 16-bit RAMS connected in the form of a large ring shift register. The spins circulate around this ring at 1.5 MHz. The six nearest-neighbour values of simple cubic lattice can be tapped from the ring at six fixed locations in parallel and fed to the spin update processor. The updated central spin is fed back to the ring. In fact, the update processor is so fast that it can accept multiplexed data from 16 non-overlapping parts of the ring. Just as with the DAP, mapping the physical data structure onto a hardware structure of lower dimension necessitates the use of skewed periodic boundary conditions. The machine achieves 10 times Cray 1 performance at a component cost of $2000; however, it is relatively inflexible in its applications. The Delft processor, in contrast, can handle a wide variety of lattice

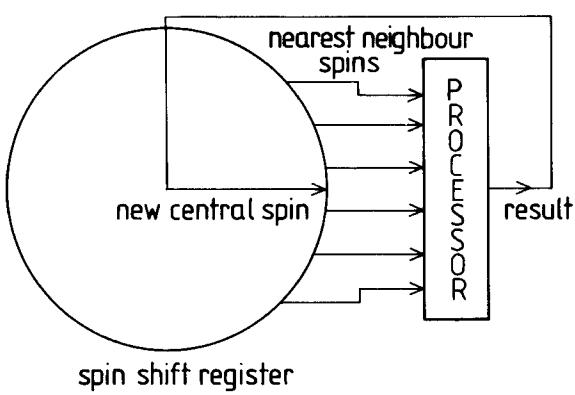


Figure 26 The Santa Barbara Ising processor.

types and interactions, and the Murray Hill machine can, in addition, treat random interactions. The Santa Barbara and Delft machines fetch all neighbours of a central spin in parallel, whereas the Murray Hill machine processes 32 spins in parallel, but takes their neighbours one by one.

6.2 Molecular Dynamics Computers

The Delft molecular dynamics computer [79] is designed for simulation of systems of spherical molecules. Initially, it was applied to two-dimensional problems [80, 81], later being adapted to three-dimensional problems and to the study of mixtures. It can simulate systems of up to 16000 particles.

The cell technique is used to determine interacting pairs of particles. The particle memory contains position and momentum information about each particle in a 188-bit word, and a linked list giving the addresses of the particles in each cell. Cells are taken two at a time and the linked list used to load copies of the relevant particle coordinates into two register files in the momentum-update section of the hardware. The momenta of particles from the two cells are also loaded into two register files. In the momentum-update hardware the forces between each pair of particles in the register files are calculated in a pipeline. Separate hardware elements are devoted to each elementary stage of the pipeline: calculation of particle separation vector; squared distance between particles; force evaluation by table look-up; force scaling to give momentum increments; accumulation of these in the momentum register files. Where possible x, y and z components are treated in three parallel hardware elements. All arithmetic is fixed point. Finally, the momentum registers are copied back into particle memory. When all necessary pairs of cells have been considered, a separate position-update section of the hardware increments the position of all particles in a simple pipeline. Overall performance is similar to that of the Cray 1.

A molecular dynamics/mechanics computing system has been developed at Columbia University for macromolecular simulations [82]. It consists of a VAX host, a commercially available array processor, the STAR ST100, and a special purpose device, FASTRUN. The latter is devoted entirely to the calculation of pair forces using a neighbour list. The ST100 handles all other forces (angles, torsions etc.), the momentum and position updates, and periodic revision of the neighbour list. FASTRUN consists of a pipeline that takes the neighbour list and, using it to index coordinates in the coordinate memory, evaluates pair forces by table look-up, and then accumulates them into the force memory. As in the Delft device, parallel evaluation of the three Cartesian components is used within the pipeline where possible. It differs from the Delft machine in using commercially available floating-point chips, with most arithmetic in 32-bit precision, but energy and force accumulation in 64 bits. Peak performance is about 600 Mflops.

7. QUESTIONS FOR THE FUTURE

7.1 Is Parallelism Really Necessary?

In this review I have concentrated on architectures and algorithms and have not had space to discuss the scientific results obtained. However, study of the references will show that much scientific work has been carried out on parallel computers that would have been impossible on conventional machines. Nevertheless, I cannot hide the fact

that this work has been restricted almost entirely to the academic environment, and it has been notable how many innovatory parallel computers have failed to gain any commercial success. The reasons have been conservatism among users, a desire to protect investment in existing software, sometimes poor design, and sometimes uninspired marketing.

The technical arguments for the use of parallel architectures as the only way to produce computers of greater power and cost-effectiveness are very strong, and the signs are that they are now beginning to bear fruit in terms of commercial products. In my opinion, the manufacturers have often overestimated the conservatism of users, particularly scientific users, who form a commercial market in their own right. My experience as an adviser of potential users of the DAP was that they fell rather sharply into two classes: some conservatives were unwilling to change a line of their existing Fortran; but the more adventurous welcomed their introduction to parallel computing, not only because it enabled them to solve their scientific problems, but also because they discovered that they positively enjoyed thinking about them in a new way.

7.2 What about Portability?

Most of this review has been about mapping particular problems onto particular hardware, and every programming example has been machine-specific. Will we ever be able to simply write a program that expresses what we want to do, and let the compiler sort out how to do it, on whatever machine we happen to be using? This is what computer scientists are trying to achieve with dataflow architectures, single assignment languages and the like, and physical scientists await the results with interest. In the meantime, we can make some progress on our own.

One thing is certain: sequential Fortran can never be the universal language that captures the essence of our problem. We definitely need to express algorithms at a higher level of abstraction. The array extensions in Fortran 8X will be a step forward for the vector processors and perhaps SIMD computers, but the MIMD multiprocessors are a much tougher nut to crack. Occam provides a very general model of concurrency that could be applicable across a range of multi-processor types, but it is still a very low-level language. It will probably always be the case that scientists wishing to work at the forefronts of computational power will have to be aware of how to exploit the specific hardware being used: to some of us, beating the computer is part of the fun.

7.3 SIMD or MIMD?

The MIMD type of design is "obviously" more flexible, and some articles discussing parallel computing make no reference at all to SIMD architectures. However, we have seen that the DAP has made a very important contribution to molecular simulation, as it has in other fields. Also, we have emphasised that the vector supercomputers, now the workhorses of scientific computing, are essentially SIMD devices. There is no doubt that the programming of SIMD computers is easier than that of MIMD machines, at least for the regular problems for which they are most suitable. The question must be regarded as an open one. MIMD design at the moment seems to mean connecting together micro-processors, which are individually fairly conventional sequential computers. Another approach to exploiting VLSI is the more radical

one of having many simple processors integrated on a single chip, probably working under central control. This may well be the preferred route for certain applications, such as image processing. What does seem to be happening among some major manufacturers is an approach that may be classed as globally MIMD and locally SIMD. The supercomputer manufacturers found that they could only increase the power of their machines by adding on further vector processors. Intel found that it needed to increase the power of its hypercube and added vector processors to the nodes, and FPS have produced a similar design. In contrast to this American approach, the Europeans, with ARTS and SUPRENUM, are putting an MIMD cluster at the nodes of their regular network. It will be interesting to see which approach is more successful, scientifically and commercially.

7.4 What Topology?

We have described a number of ways in which processors can be connected together. In most cases, these connections have been chosen to match the geometry of some typical class of problem, although algorithmic parallelism can also be used on less regular problems. A square array is an obvious choice, well adapted to image processing, grid methods in field problems, and molecular simulation of lattice problems, or fluids using the cell method. Three-dimensional problems can be mapped onto it if skewed periodic boundaries are acceptable. A hypercube provides more than just local connections if it has higher dimensionality than the problem. This is at the expense of more connections per processor, and there may be problems in supporting a sufficient communication rate, especially if the nodes are enhanced with vector processors. Much of the success of the DAP has been due to its very judicious mix of local and global connections. In general, one suspects that many interesting physical problems have correlations present at a variety of length scales. This means either mapping a smallish grid of processors successively onto the problem with different geometrical scaling, or having different scales present in the processor array. In this respect the pyramidal structure of the Erlangen computer seems particularly interesting.

7.5 Special or General Purpose?

No doubt occasionally scientists will build computers specifically to study a single problem that could not be tackled with reasonable cost in any other way. However, a trend seems to have been established to make special purpose computers as flexible as possible so that they can handle a whole class of related problems. There is an interesting comparison here with the way scientific software evolves. When a new field opens up, programs are written on a problem-by-problem basis. Later, when techniques are established and begin to be applied on a routine basis, they are built in to standard widely used packages. For example, CHARMM [83] is very flexible program for molecular dynamics and molecular mechanics that can be applied to almost any system. Special purpose computers may develop in the same way, and indeed we may see combined hardware and software packages being offered.

7.6 Big or Small?

If we assume that by the use of parallel architectures we can make computers one or two orders of magnitude more cost-effectively, various possibilities open up. One is

to build computers many times more powerful than present supercomputers, at similar cost. Such computers may be necessary in some cases, and there is even a certain glamour attached to them, but we should be clear that their primary advantage is the large *size* of problem to which they can be applied due to their large high-speed memory, fast disks etc. Their construction is not justified on the grounds of speed alone, for the simple reason that, due to their cost, they must be shared among hundreds of users. To an individual user, a 10 Gigaflop computer may give only 10 Megaflops performance. Many problems in molecular simulation do not have large memory or disk requirements, but do need large amounts of CPU time. For such problems the alternative approach of building high-performance computers at a cost that can be afforded by an individual research group, and so devoted to a single problem at a time, is more attractive. A particular advantage is that laboratory-based computers can be much more easily used in an interactive way in conjunction with a graphical form of output, than can machines that, by their nature, must remain remote to most of their users.

Acknowledgements

I am grateful to all those who sent me information about their work prior to publication. Graham Doggett and Dennis Rapaport made suggestions that helped to improve the manuscript. The work has been funded in part by International Computers Ltd and by the Science and Engineering Research Council.

References

[1] J.P. Hansen and I.R. McDonald, *Theory of Simple Liquids*, Academic Press, London, 1976.
[2] D. Fincham and D.M. Heyes, "Recent advances in molecular dynamics computer simulation," in *Dynamical Processes in Condensed Matter*, M.W. Evans, ed., (*Advances in Chemical Physics*, volume LXIII), John Wiley Inter-science, New York, 1985.
[3] D.N.J. White, "Computer methods for molecular design," *Phil. Trans. Roy. Soc. Lon.*, **A317**, 359–369 (1986).
[4] K.R. Wilson, "Multiprocessor molecular mechanics," in *Computer Networking and Chemistry*, ACS Symposium Series no. 19, 17–52 (1975).
[5] D.J. Tildesley, "Towards a more complete simulation of small polyatomic molecules," in *Molecular Liquids – Dynamics and Interactions*, A.J. Barnes, W.J. Orville-Thomas and J. Yarwood, eds., (NATO Advanced Studies Institute Series C Volume 135) Reidel, Dordrecht, 1984.
[6] G. Jaccuci, I.R. McDonald and K. Singer, "Introduction of the shell model of ionic polarisability into molecular dynamics calculations," *Phys. Lett.*, **50A**, 141–143 (1974).
[7] R.W. Hockney and C.R. Jesshope, *Parallel Computers*, Adam Hilger, Bristol, 1981.
[8] S.H. Unger, "A computer oriented towards spatial problems," *Proc. Inst. Radio Eng. USA*, **46**, 1744–1750 (1958).
[9] D. Parkinson, "The Distributed Array Processor," *Comput. Phys. Commun.*, **28**, 325–336 (1983).
[10] Manufacturer's information from Active Memory Technology Ltd, 65 Suttons Park Avenue, Reading RG6 1AZ, UK.
[11] R.M. Russell, "The Cray 1 computer system," *Commun. ACM*, **21**, 63–72 (1978).
[12] E. Brooks *et al.*, *Nucl. Phys.*, **B220**, 383 (1983).
[13] Preprint CALT-68-1112 from California Institute of Technology, Pasadena, CA, USA.
[14] Manufacturer's information from Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, Oregon 97006, USA.
[15] Manufacturer's information from Floating Point Systems Ltd, Apex House, London Road, Bracknell RG12 2TE, UK.
[16] Product information from INMOS Ltd, PO Box 424, Bristol BS99 7DD, England, UK.
[17] "Transputer" and "Occam" are commonly written with lower-case initial letters, since this is how the trade marks are registered. We prefer to maintain standard English usage for proper names.

[18] Manufacturer's information from Meiko Ltd, Lewins Mead, Bristol BS1 2NT, UK.
[19] A.J.G. Hey, C.R. Jesshope and D.A. Nicole, "High performance simulation of lattice physics using enhanced Transputer arrays," in *Computing in High Energy Physics*, L.O. Hertzberger and W. Hoogland, eds, Elsevier, Amsterdam, 1986.
[20] N.S. Ostland, "Waterloop V2/64: a highly parallel machine for numerical computation," *Comput. Phys. Commun.*, **37**, 109–117 (1985).
[21] Information from SUPRENUM GmbH, D-5300 Bonn 1, FRG.
[22] W. Handler *et al.*, "A tightly coupled and hierarchical multiprocessor architecture," *Comput. Phys. Commun.*, **37**, 87–93 (1985).
[23] E. Clementi, G. Corongiu and J.H. Detrich, "Parallelism in computations in quantum and statistical mechanics," *Comput. Phys. Commun.*, **37**, 287–294 (1985).
[24] J. Beetem, M. Denneau and D. Weingarten, "GF11", *J. Stat. Phys.*, **43**, 1171–1183 (1986).
[25] Reference manual *DAP: Fortran Language*, from ICL, 60 Portman Road, Reading RG3 1NR, UK.
[26] J.K. Reid and A. Wilson, "The array features in Fortran 8X with examples of their use," *Comput. Phys. Commun.*, **37**, 125–132 (1985).
[27] H.R. Hicks and V.E. Lynch, "An introduction to programming multiple-processor computers," *J. Comput. Phys.*, **140**, 140–156 (1986).
[28] INMOS Ltd, *Occam Programming Manual*, Prentice Hall, London, 1984.
[29] D. Fincham and N. Quirke, "Parallel programming for lattice problems on the DAP," *Information Quarterly for MD and MC Simulations* (The CCP5 Newsletter), **10**, 13 (1983). This is an informal publication available on request from SERC Daresbury Laboratory, Warrington WA4 4AD, UK. It now has a new name: *Information Quarterly for Simulation of Condensed Phases.*
[30] G.S. Pawley, K.C. Bowler, R.D. Kenway and D.J. Wallace, "Concurrency and parallelism in MC and MD simulations in physics," *Comput. Phys. Commun.*, **37**, 251–260 (1985).
[31] G.S. Pawley, R.H. Swendsen, D.J. Wallace and K.G. Wilson, "Monte Carlo renormalisation group calculations of critical behaviour in the simple cubic Ising model," *Phys. Rev.*, **B29**, 4030–4046 (1984).
[32] S.F. Reddaway, D.M. Scott and K.A. Smith, "A very high speed Monte Carlo simulation on the DAP," *Comput. Phys. Commun.*, **37**, 351–356 (1985).
[33] C.J. Tinsley, "A Monte Carlo investigation of the dependence of spontaneous magnetization on temperature in Ising models of lithium ferrite," preprint, Department of Mathematics and Statistics, Portsmouth Polytechnic, England, UK, submitted to *Phil. Mag.*
[34] M.P. Allen, "A Monte Carlo simulation study of orientational domain clusters in the planar quadrupole model," unpublished report, Department of Physics, Bristol University, England, UK.
[35] K.A. Smith, S.F. Reddaway and D.M. Scott, "Very high performance pseudo-random number generation on the DAP," *Comput. Phys. Commun.*, **37**, 239–244 (1985).
[36] M.T. Dove *et al.*, "Collective excitations in an orientationally frustrated solid: neutron scattering and computer simulation studies of SF_6," *Molec. Phys.*, **57**, 865–870 (1986).
[37] M.T. Dove, "A simulation study of the disordered phase of CBr_4," *J. Phys. C: Solid State Phys.*, **19**, 3325–3341 (1986).
[38] G.S. Pawley and G.W. Thomas, "The implementation of lattice calculations on the DAP," *J. Computat. Phys.*, **47**, 165–178 (1982).
[39] D. Fincham and B.J. Ralston, "Molecular dynamics simulation using the Cray-1 vector processing computer," *Comput. Phys. Commun.*, **23**, 127–134 (1981).
[40] S. Brode and R. Ahlrichs, "An optimized MD program for the vector computer Cyber 205," *Comput. Phys. Commun.*, **42**, 51–57 (1986).
[41] D. Fincham, N. Quirke and D.J. Tildesley, "Computer simulation of molecular liquid mixtures," *J. Chem. Phys.*, **84**, 4535–4546 (1986).
[42] M. Neumann, O. Steinhauser and G.S. Pawley, "Consistant calculation of the static and frequency dependent dielectric constant in computer simulations," *Molec. Phys.*, **52**, 97–113 (1984).
[43] N. Anastasiou and D. Fincham, "Programs for the dynamic simulation of liquids and solids," *Comput. Phys. Commun.*, **25**, 159–176 and 177–179 (1982).
[44] D.J. Adams, "Alternatives to the periodic cube in computer simulation," *Information Quarterly for MD and MC Simulations*, **10**, 30 (1983).
[45] W. Smith, "The periodic boundary condition in non-cubic MD cells," *Information Quarterly for MD and MC Simulations*, **10**, 37 (1983).
[46] D. Fincham, "Parallel computers and non-cubic boundary conditions," *Information Quarterly for MD and MC Simulations*, **12**, 43 (1984).
[47] D. Potter, *Computational Physics*, John Wiley, London, 1973.
[48] D. Fincham, "Rotational motion of linear molecules," *Information Quarterly for MD and MC Simulations*, **10**, 43 (1983).

[49] D. Fincham, "An algorithm for rotational motion of rigid molecules," *Information Quarterly for MD and MC Simulations*, **2**, 6 (1981).
[50] L. Verlet, "Computer experiments on classical fluids," *Phys. Rev.*, **159**, 98–103 (1967).
[51] W.F. van Gunsteren, H.J.C. Berendsen, F. Colonna, D. Perahia, J.P. Hollenberg and D. Lellouch, "On searching neighbours in computer simulations of macromolecular systems," *J. Computat. Chem.*, **5**, 272–279 (1984).
[52] S.M. Thompson, "Use of neighbour lists in molecular dynamics," *Information Quarterly for MD and MC Simulations*, **8**, 20–28 (1983).
[53] D.J. Adams, "More on neighbourhood tables," *Information Quarterly for MD and MC Simulations*, **3**, 32–33 (1981).
[54] R.W. Hockney and J.W. Eastwood, *Computer Simulation Using Particles*, McGraw Hill, New York, 1981.
[55] W. Smith, "Fortran code for the link-cell method," *Information Quarterly for Computer Simulation of Condensed Phases*, **20**, 52–58 (1986).
[56] F. Sullivan, R.D. Mountain and J. O'Connell, "Molecular dynamics on vector computers," *J. Computat. Phys.*, **61**, 138–153 (1985).
[57] R. Vogelsang, M. Schoen and C. Hoheisel, "Vectorisation of molecular dynamics Fortran programs using the Cyber 205 vector processing computer," submitted to *Comput. Phys. Commun.*
[58] J. Boris, "A vectorised near neighbour algorithm of order N using a monotonic logical grid," *J. Computat. Phys.*, **66**, 1–20 (1986).
[59] H.J.C. Berendsen, W.F. van Gunsteren and J.P.M. Postma, "Molecular dynamics on Cray, Cyber and DAP," report from Groningen University, The Netherlands.
[60] J.W. Eastwood, "Optimal P^3M algorithms for molecular dynamics simulations," in *Computational Methods in Classical and Quantum Physics*, M.B. Hooper, ed., Advance Publications, London 1976.
[61] J. Eastwood, R. Hockney and D. Lawrence, "PM3DP – the three dimensional periodic particle–particle/particle–mesh program," *Comput. Phys. Commun.*, **19**, 215–261 (1977).
[62] D.J. Adams and G.S. Dubey, "Taming the Ewald sum in the computer simulation of charged systems," preprint from Department of Chemistry, University of Southampton, England, UK, submitted to *J. Computat. Phys.*
[63] D. Fincham, "RDF on the DAP: easier than you think!" *Information Quarterly for MD and MC Simulations*, **8**, 45 (1983).
[64] D.J. Adams, "The implementation of fluid phase Monte Carlo on the DAP," preprint from Department of Chemistry, University of Southampton, England, submitted to *J. Computat. Phys.*
[65] W. Chapman and N. Quirke, "Metropolis Monte Carlo simulation of fluids with multiparticle moves," *Physica*, **131B**, 34–40 (1985).
[66] D. Fincham, to be published.
[67] "Pass the Parcel" is a children's game in which a parcel is passed from hand to hand, one layer of wrapping being removed by each child.
[68] 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).
[69] R.A. Whiteside, P.G. Hibbard and N.S. Ostland, "A systolic algorithm for Monte Carlo simulations," in *Proc. Third Intern. Conf. Distrib. Sys.*, IEEE Computer Society, pp. 800–804.
[70] M.A. Johnson, *Ph.D. Thesis*, document no. 268 from Concurrent Computation Project, California Institute of Technology, Pasadena, USA.
[71] D.C. Rapaport and E. Clementi, "Eddy formation in obstructed fluid flow: a molecular dynamics study," *Phys. Rev. Lett.*, **57**, 695–698 (1986).
[72] C.R. Askew *et al.*, "Simulation of statistical mechanical systems on Transputer arrays," *Comput. Phys. Commun.*, **42**, 27 (1986).
[73] Musical Chairs is a game in which children circulate until the music stops, when they must sit down on the nearest available chair.
[74] C.R. Askew *et al.*, "Monte Carlo simulation on Transputer arrays," preprint SHEP 85/86-26 Department of Physics, University of Southampton, England, UK, submitted to *Journal of Parallel Computing*.
[75] A. Hoogland, J. Spaa, B. Selman and A. Compagner, "A special purpose processor for the Monte Carlo simulation of Ising spin systems," *J. Computat. Phys.*, **51**, 250 (1983).
[76] R.B. Pearson, J. Richardson and D. Toussaint, "A fast processor for Monte Carlo simulation," *J. Computat. Phys.*, **51**, 241–249 (1983).
[77] A.T. Ogielski and I. Morgenstern, "Critical behaviour of three-dimensional Ising spin-glass model," *Phys. Rev. Lett.*, **54**, 928–931 (1985).

[78] J.H. Condon and A.T. Ogielski, "A fast special purpose computer for Monte Carlo simulations in statistical physics," *Rev. Sci. Instr.*, **56**, 1691–1696 (1985).
[79] A.F. Bakker, C. Bruin, F. van Dieren and H.J. Hilhorst, "Molecular dynamics of 16 000 Lennard-Jones particles," *Phys. Lett.*, **93A**, 67 (1982).
[80] A.F. Bakker, C. Bruin and H.J. Hilhorst, "Orientational order at the two-dimensional melting transition" *Phys. Rev. Lett.*, **52**, 449–452 (1984).
[81] J.H. Sikkenk, H.J. Hilhorst and A.F. Bakker, "A molecular dynamics simulation of liquid–vapour interfaces in two dimensions," *Physica*, **131A**, 587–598 (1985).
[82] C. Levinthal, R. Fine and G. Dimmler, "FASTRUN – A special purpose hard-wired computing device for molecular mechanics," report from Department of Biological Sciences, Columbia University, New York, USA.
[83] B.R. Brooks *et al.*, "CHARMM: a program for macromolecular energy, minimisation and dynamics calculations," *J. Computat. Chem.*, **4**, 187–217 (1983).