Array processors in molecular graphics

David N J White*

Chemistry Department, The University, Glasgow G12 8QQ, UK

The use of array processors in molecular graphics is a relatively recent phenomenon, and the utility of these devices is not yet widely appreciated. The paper describes the architecture of array processors and the kind of scientific projects in which their use is appropriate and beneficial. There are two major areas of molecular graphics where the use of a programmable array processor either leads to a considerable increase in the speed of a particular operation, or makes a hitherto uneconomical calculation feasible in a reasonable timescale. These areas are the real-time transformation of images on a graphics screen and/or the completion of large-scale molecular mechanics, molecular dynamics, or molecular orbital calculations. The problems associated with the use of array processors in these applications are discussed at length.

Keywords: computer hardware, array processors, molecular graphics, computer architectures

received 13 May 1985, accepted 20 May 1985

INTRODUCTION

Molecular graphics is a fusion of the techniques of interactive computer graphics and computational procedures such as molecular mechanics or molecular orbital calculations. The purpose of molecular graphics is either the 3D design of usually organic molecules prior to synthesis, or the investigation and rationalization of experimentally derived structures in terms of energy.

In the past, work in the molecular graphics field has been biased either towards the purely pictorial aspects or towards the area of computation. It is largely the prospect of effective computer-aided drug design (CADD) that has provided the impetus for the fusion of interactive graphics and computational chemistry into the new discipline of molecular graphics. However, in order to discuss the impact of array processors on molecular graphics, the graphics and computational chemistry contributions must be considered separately.

During the late 1960s and early 1970s, chemists and crystallographers began to make use of computer graphics devices for the interpretation and display of experimental results. The cost factor meant that most of the devices in use were of the storage tube variety, linked to a remote mainframe computer. These devices were ideal for generating high-resolution (4 096 × 4 096 resolvable points in the display area), static, monochrome pictures. The devices incorporated no local intelligence, so that all the computations involved in picture generation and manipulation (e.g. rotating the molecule so as to view it from a different direction) were easily performed in the host computer. Storage tube devices do not enable the user to selectively erase a portion of the displayed picture, and redrawing a complex picture can take a second or two, so that moving pictures are not possible.

From 1972 onwards, calligraphic displays became sufficiently cheap, relatively speaking, for laboratory use, and these devices, which were available from several manufacturers, proliferated. Early calligraphic displays could draw pictures composed of up to 10 000 short vectors (<0.5in) at a rate of about 20 times per second. This opened up the possibility of moving pictures whereby, for example, a view of the current molecule is incrementally rotated from one 0.03s frame to the next. This kind of operation gives rise to the kinetic depth effect, which is particularly marked if parts of the molecule are selectively dimmed as they move away from the viewer. This deceives the viewer into cognition of the image as a 3D rather than 2D object. The calculation of moving pictures to generate the kinetic depth effect involves the repeated multiplication of a 4×4 matrix by a 4-long vector for every atom in the molecule at very high speed. These high-speed matrix operations with subsequent transmission of results to the display electronics, now driven by a dedicated digital processor called the picture processor, are beyond the capability of most multiuser host computers. The display manufacturers therefore attached 'black boxes' (picture transform processors) to the picture processor in order to perform the matrix calculations at high speed and place the results directly into the picture memory also at very high speed. As will be shown later, the picture transform processor is also a vestigial array processor (this term frequently means a processor of numerical arrays such as matrices and vectors rather than an array of digital processors, although the latter are becoming increasingly common and will soon be the architecture of choice). The drawback of most currently available picture transform processors is that they operate on limited precision 16-bit integers and are not user programmable, although this situation is changing.

The calligraphic display has probably reached the zenith of commercial development and is rapidly being superseded by the raster scan display, which can be produced very cheaply using integrated microelectronics and produces moving, colour pictures of wire-frame and solid objects at high resolution. The principal weakness,

*At present on sabbatical leave at: R-1046.109, Ciba-Geigy, CH-4002, Basel, Switzerland

 0263-7855/85/040136-09 $03.00

apart from manufacturing cost, of calligraphic displays is the necessity to represent solid objects by transparent wire frame models because the display electronics cannot area-fill parts of the picture with vectors quickly enough to generate a stable solid image. In the latest raster scan displays, the picture transform processor performs hardware polygon fill, in user-specified colours, 3D polygon clipping, and hidden line removal. This requires even higher arithmetic performance from the picture transform processor. Unfortunately, the user is still restricted to hardware assistance with picture transforms that the display manufacturer thinks desirable, rather than being able to program transforms as wished.

The requirement for high-speed arithmetic processing in computational chemistry is much more obvious and must be floating point rather than integer. Techniques such as molecular orbital, molecular mechanics and molecular dynamics calculations can use several tens of hours of CPU time on a conventional time-shared minicomputer, and a fair proportion of that time would be occupied in performing floating point arithmetic calculations. These calculations can be executed faster by ever more powerful (and expensive) conventional computers, but the current upper limit, even if cost is no object, still involves lengthy run times for calculations on large molecules. Fortunately, most of the calculations involved in molecular-x (MX) procedures, either involve algorithms operating on very large matrices and/or vectors, or they may be split into a number of noninterdependent processes which may be executed concurrently ('loop unrolling'). Special purpose processors have been devised which utilize the regularity of matrix–vector calculations to achieve performances one to three orders of magnitude in advance of conventional 32-bit minicomputers. More recently, the availability of low cost, high performance, floating point microprocessors has made the loop unrolling approach the more attractive proposition, particularly as the software does not need to be hand or machine 'vectorized' (see below).

ARRAY PROCESSOR ARCHITECTURES FOR PICTURE TRANSFORMS

In the early days of interactive computer graphics, only calligraphic displays were sufficiently fast to generate moving pictures and the picture transform processor was implemented in software on a dedicated host minicomputer. While this kind of arrangement did not result in spectacular animation, it did give the user total control over the displayed picture, and the calculations could be performed in floating-point arithmetic, if required. With the advent of cheap high-speed TTL (transistor transistor logic) chips in the 1960s, designers of computer graphics systems began to build dedicated electronic devices to perform the matrix arithmetic operations necessary for real-time picture animation into the graphics display[1]. For example, a typical transformation might be represented by the equations:

$$\begin{bmatrix} x_1 & y_1 & z_1 & h_1 \\ x_2 & y_2 & z_2 & h_2 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ x_A & y_A & z_A & h_A \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x_1' & y_1' & z_1' & h_1' \\ x_2' & y_2' & z_2' & h_2' \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ x_A & y_A & z_A & h_A \end{bmatrix} \quad (1)$$

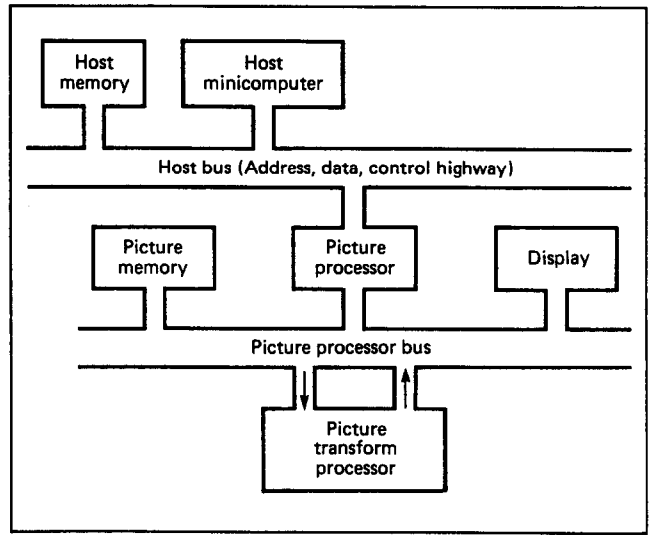


Figure 1. Diagrammatic representation of a high performance 3D graphics display system

A block diagram of the hardware required to perform these operations is shown in Figure 1, and a typical sequence of events might be as follows. The user draws pictures by making calls to a FORTRAN subroutine library supplied by the display vendor. The subroutine library then translates these calls into instructions and data for the picture processor, and these are stored in a host memory buffer. At the appropriate time, the code and data are transmitted to picture memory by a direct memory access transaction. The picture processor continuously loops around the set of instructions and operates on the data once every 0.03s or so. Some of the instructions in the code sent from the host will be for the picture transform processor, rather than the picture processor, and the latter's instruction decoder will generate an 'illegal instruction trap' when it finds an instruction for the transform processor. The trap handler will direct the transform processor to operate on the picture data, obtaining operands from picture memory by direct memory access and returning results by the same route. In order not to slow down the picture processor, this transformation should proceed in parallel with picture processor operations and should be complete by the time the picture processor cycles back to operate on the (hopefully) transformed data. Obviously, the transform processor needs to complete a (large) number of 4×4 matrix times 4-vector operations in well under 0.03s. Ideally, these matrix operations should be performed on data that is in floating-point format in order to preserve accuracy and dynamic range as well as allowing the user to work in world (i.e. application level) coordinates. In practice, however, the arithmetic is usually integer, for reasons that will become clear below.

The picture transform processor will now be examined a little more closely. The basic arithmetic operations involved in the transform described above are multiplication and addition, of which multiplication is the most time consuming. The numbers involved are usually 10 (for a 1024×1024 display) or 12 (for a 4096×4096 display) bit integers, and to preserve accuracy, around 4 guard bits are necessary. The most convenient word length to use is therefore 16 bits. The integer multiplication in the transform processors of most current displays is performed by VLSI multiplier chips from companies

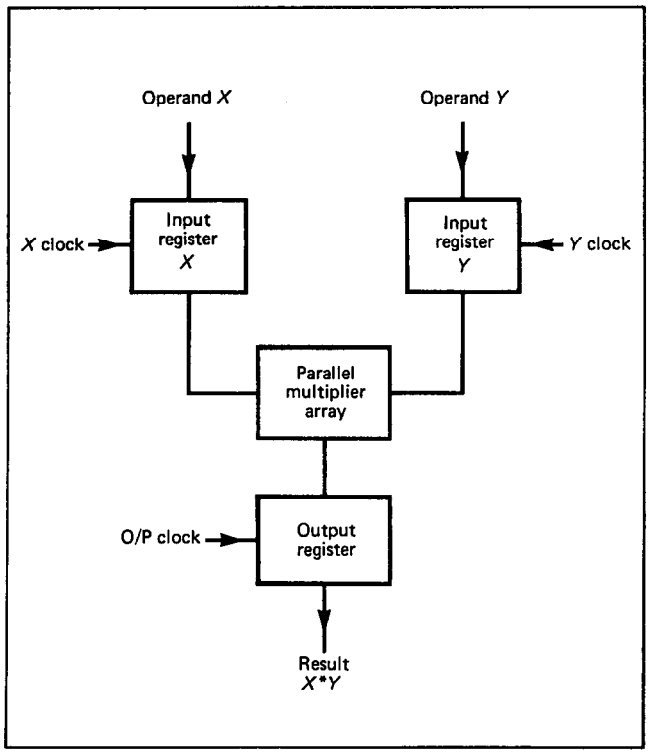


Figure 2. Block diagram of a VLSI integer multiplier chip

such as TRW[2], AMD, Analog Devices, or Monolithic Memories, which can perform a 16-bit by 16-bit integer multiplication in 50–100 ns. The optimum performance is squeezed from these chips by a technique called pipelining.

Hardware and software pipelining is central to the efficient use of array processors, and the multiplier provides a simple illustrative example of pipelining. Figure 2 shows a block diagram of a bipolar multiplier chip, and the clock is an electronic 'tick' which synchronizes the operations of all elements of the interactive graphics system. A multiplication might then proceed as follows: clock-tick 1 latches the operand X into its input register; clock-tick 2 latches the operand Y into its input register, and, after some delay, usually one clock-tick (clock-tick 3), a valid result appears at the output of the multiplier array; clock-tick 4 latches the product X^*Y into the output or product register, and the operation is finished. One multiplication therefore takes four clock-ticks or 400 ns if the operation is run with a 10 MHz clock. This can easily be reduced to three clock-ticks per multiplication by latching X and Y into their respective registers simultaneously, but we cannot latch the output register at the same time and obtain a meaningful result because the product X^*Y takes one clock-tick to appear at the output of the multiplier array. The approach has moved from a sequence of serial operations to a degree of parallelism, but pipelining has not been achieved. This is done as follows:

- Clock-tick 1: latch X_1 and Y_1 into the input registers.
- Clock-tick 2: latch X_2 and Y_2 into the input registers; the $X_1{}^*Y_1$ product is propagating its way through the multiplier logic array and can be temporarily forgotten.
- Clock-tick 3: latch X_3 and Y_3 into the input registers; $X_2{}^*Y_2$ is propagating through the multiplier logic; latch $X_1{}^*Y_1$, which has now arrived at the output of the multiplier logic, into the output register. The 'pipeline' is now full.
- Clock-tick 4: latch X_4 and Y_4 into the input registers; $X_3{}^*Y_3$ is propagating; latch $X_2{}^*Y_2$ into the output register.
- Clock-tick 5: latch X_5 and Y_5 into the input registers . . . etc.

A result is now appearing at the output register, ready for dispatch to memory, at every clock-tick or every 100 ns. The analogy with a pipeline is obvious: each new pair of operands pushed in at one end of the pipe pushes a result out the other.

The process described above may be speeded up even further by using four 8-bit multiplier chips rather than one 16-bit chip. This is because the 8-bit chip multiplies in, typically, 50 ns rather than 100 ns, and it is possible to pipeline the composite (4 × 8) 16-bit multiplier system to produce a result every 50 ns. Figure 3 shows schematically how this can be achieved. Notice, however, that the 'length' of the pipeline has gone from three to five stages. It is generally true that a pipelined process may be speeded up by increasing the length of the pipe and making each step perform a simpler function in a shorter time. However, a diminishing returns' situation sets in very quickly if the amount of data to be operated on is small, although a long pipeline will always be advantageous for a very large amount of data. The reason for this behaviour is that the pipeline takes a finite time to 'fill' (the setup time) before any usable results emerge; in this example, the fill time is 5 × 50 = 250 ns, after which

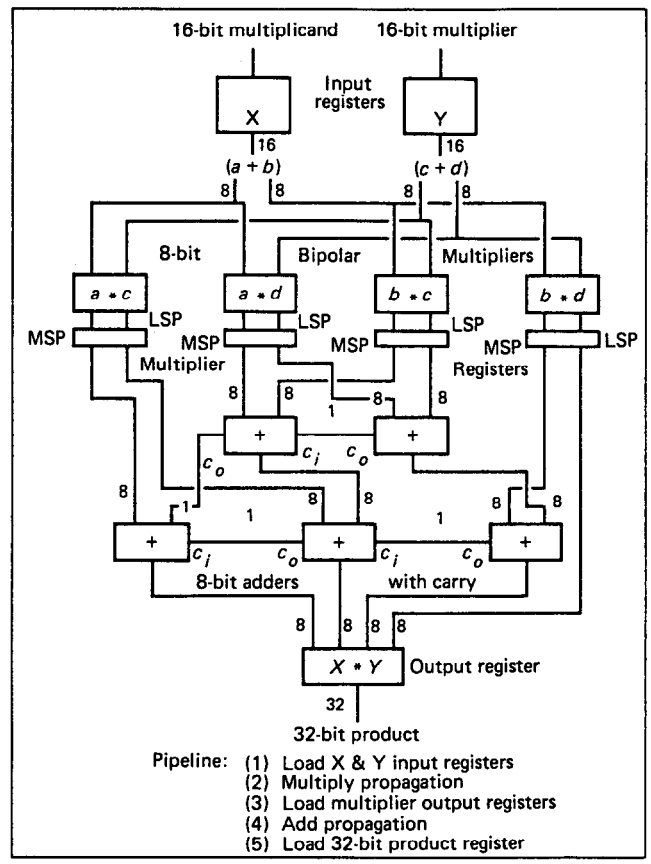


Figure 3. Schematic representation of a pipelined 16 × 16 bit multiplier subsystem built from four 8 × 8 bit multiplier chips

results emerge every 50 ns. With very long pipelines, the setup time can become longer than the time required to execute an entire calculation on, say, two or three sets of data with a very short pipeline which is quickly filled. Furthermore, a pipeline must be 'flushed' in order to complete a sequence of calculations. In the example of Figure 3, four dummy sets of data must be pushed into the pipeline in order to flush the last set of genuine data loaded into X and Y to the end of the pipeline.

In practice, most picture transform processors in state-of-the-art calligraphic displays might use four multipliers of the type shown in Figure 3 operating in parallel and in conjunction with three adders to produce an element of the result matrix of equation (1) every 60 ns or so (50 ns for the four parallel multipliers plus 10 ns for the adds), which means a complete end point transformation every 240 ns.

This simple nonprogrammable kind of picture transform processor is quite adequate for the standard picture manipulations, which are readily expressed in short sequences of matrix products, although most manufacturers do not allow the user access to the coordinates transformed by the display electronics (Evans & Sutherland is an exception). But what happens if one wants to perform 'on-the-fly' hidden line removal calculations on a rotating 3D picture; or calculate distances in real-time as one part of a model rotates with respect to another? These calculations must be performed on the host and the results, sent to the picture processor for appropriate action — a very time consuming process involving many unnecessary data transfers. It would be better if the picture transform processor were a programmable, floating point array processor rather than a nonprogrammable, integer processor. In this case, the standard picture manipulations of rotating, translating, scaling and clipping could be 'built-in', leaving the user to program parameters like distance, angle and torsion angle monitoring, for example. The results of these transformations/operations would be available to the host computer program, if required. Indeed, given a fast enough array processor with enough memory, a large part, or all, of a subroutine that required fast access to the display for the results of arithmetic intensive operations could be executed in the 'picture transform processor'.

What emerges from the above discussion is the requirement for a high resolution (at least 1024×1024) colour (a minimum of 256) display very tightly coupled (i.e. picture memory is directly accessible by the array processor) to a programmable 32-bit array processor with a large address space (at least 4 Mbyte, because picture memory and application memory are common, and multiple picture storage might be required at the same time that a large applications subroutine is executing). Ideally, such a machine should be relatively inexpensive (around £50 000) and easily programmable in standard high level languages.

Recent developments in microelectronics mean that the above list of requirements is not impossible, and displays approaching this specification are already available. This is largely due to the recent availability of one or two chip 32-bit and 64-bit floating point processors operating at 1–10 Mflop (millions of floating point operations per second) from manufacturers such as Weitek, AMD and Analog Devices for around £1 000 in single units. The commercial devices are either array processors with graphics grafted on by the array processor manufacturer, or graphics displays with the array processor grafted on by the display manufacturer.

The array processors with graphics grafted on came first from manufacturers such as CD&A or Bear systems. The CD&A product consists of a 32-bit, 5 Mflop, 2Mbyte memory array processor, user programmable via microcode or a FORTRAN callable subroutine library plus a 1024×1024 resolution, 256 colour raster scan display. The entire package is driven by either a PDP-11 or VAX under any DEC operating system. The Bear Systems machine consists of a graphics add-on for an FPS-100, AP-120B, AP-180V or Series 5000 array processor from FPS. The display resolution is somewhat low for molecular modelling work at 512×512, although there are 2^{12} simultaneously available colours, and the array processors run up to 60 Mflop. Neither of these devices has graphics capabilities appropriate to really high performance molecular modelling, particularly on macromolecules, and there is a dearth of available software (this latter is not an entirely valid criticism as users of any array or vector processor, large or small, at the present time have to develop their own software: at the moment there are no commercially available packages).

As might be expected, the machines from the display manufacturers have excellent graphics but tend to be weak on the array processor side, particularly with respect to programmability. Manufacturers such as Chromatics, Ramtek, Silicon Graphics and Raster Graphics have $>1024 \times 1024$ resolution, >256 colour, high performance raster scan displays, which incorporate >5 Mflop floating point array processors as picture transform processors. Unfortunately, none of these transform processors is programmable; they only perform the standard rotate, translate, scale, clip and hidden line removal operations. The only display suitable for high performance molecular modelling which has full local programmability comes from Megatek. Unfortunately, the programmable processor is not the picture transform processor but a 16-bit microprocessor, which indirectly controls (among other things) the relatively slow, fixed instruction set transform processor. The programming language is FORTH, relatively unknown in the molecular modelling community.

In conclusion, then, although we have seen that floating point array processors are beginning to appear as useful, integrated adjuncts to high performance graphics displays, greater developments might be expected in the future.

ARRAY PROCESSOR ARCHITECTURES FOR 'NUMBER CRUNCHING'

The volume of floating point computation involved in picture transforms is modest compared with that involved in large scale MX calculations, even if the hidden line removal is included as a picture transform. A 'number crunching' array processor should, therefore, have a floating point performance at least one or two orders of magnitude superior to a picture transform array processor, and should preferably operate on 64-bit floating point numbers, whereas 32-bit precision is sufficient for a picture transform processor.

The early days of array processors for computation-

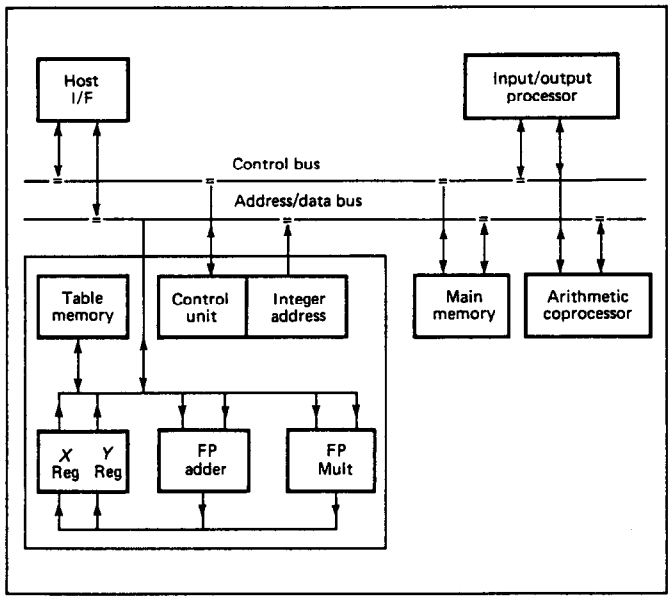
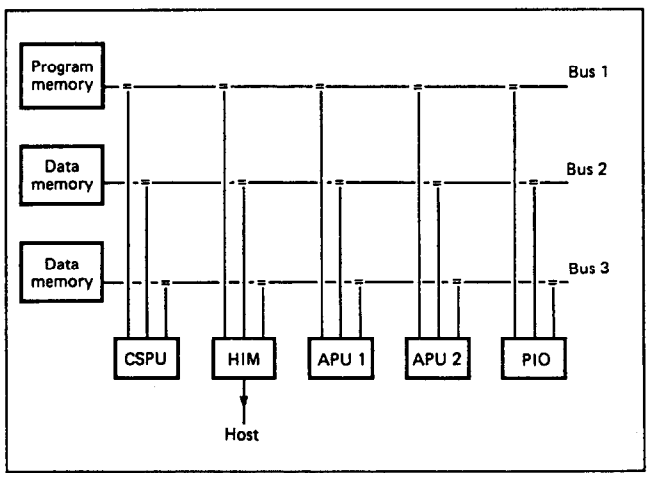


Figure 4. The main functional units of the FPS range of array processors

ally intensive scientific calculations were dominated by two companies, Floating Point Systems and CSPI, who alone produced machines of sufficient power to be useful in a general scientific context rather than providing high-speed processing in limited areas such as fast Fourier transforms or signal processing[3]. The architectures of the two ranges of machines are quite different and are illustrated in Figures 4 and 5.

The FPS machines are based on an architecture developed in the late 1960s; early versions of the genre used 38-bit words and no arithmetic coprocessor, with performance around 10 Mflop. The floating point adder and multiplier consist of three stage pipelines with all units driven in synchronous 'lock-step' by the control unit. The integer address unit is responsible for generating addresses for access to the main data memory, which



CSPU - 16-bit control processor
HIM -- Host interface module (DMA, format conversion)
APU - Arithmetic processor (+, −, x)
PIO - Peripheral I/O (discs, displays etc..)

Figure 5. Architecture of the CSPI array processors; CSPU: 16-bit control processor, HIM: host interface module (DMA, format conversion), APU: arithmetic processor (+, −, ×), PIO: peripheral I/O (discs, displays, etc.)

was initially restricted to 64 kword. All of these units are arranged so that any one can send or receive operands from any other via a 'crossbar' switch (this is not strictly true, but close enough for practical purposes) so that under favourable, and unfortunately rare circumstances, the whole machine operates very effectively (provided one has enough data) as one long pipeline. The host interface manages DMA data transfers between the host minicomputer (16- or 32-bit) and the array processor as well as performing 'on the fly' conversion between the minicomputer and array processor floating point formats. The IOP allows discs, displays, analogue to digital converters, etc. to be interfaced directly to the array processor. The floating-point add and subtract units produce results every 167 ns, and the only parallelism in the early machines came from pipelining. Subsequently, up to three arithmetic coprocessors, each containing one floating point multiplier and two floating point adders, were included (in 1984). These extra units can run in parallel with the units in the primary processor, so that both pipelining and concurrent processing may take place. Sixty four bit variants of this architecture are now available with the possibility of additional memory-mapped floating-point units (FPUs). Performance can run up to 60 Mflop (theoretical) or 300 Mflop (very theoretical) with the memory mapped FPUs. Software for early machines consisted of a library of host callable subroutines, which executed in the array processor, and a rudimentary FORTRAN cross-compiler, which needed an IBM host.

In contrast, the CSPI machines used both pipelining and concurrent processing from the very beginning, but their functional units were autonomous and did not run synchronously. Each array-processor unit (APU) contains two 480 ns floating-point multipliers and two 240 ns floating-point adders which can all operate simultaneously. Maximum theoretical throughput is 24 Mflop, and a 64-bit version of the machine is available. Because of the asynchronous nature of this array processor, it should theoretically perform better on less structured problems than the FPS machines. Data transfer to and from the host is again by DMA transactions. Software support for early machines was similar to that offered by FPS.

The abbreviated descriptions above relate to the state of affairs in 1980/81, when array processors were not significantly used in molecular modelling. £30 000 processors were available with 10 Mflop, which is very good value for money. There were two major difficulties with array processors up to 1980: they were difficult to program, and getting data in and out of the array processor quickly was a problem.

Conventional minicomputers are designed to operate on one or two items of data at a time and to produce one result at a time. Array processors, on the other hand, are designed to operate on multiple items of data (vectors or arrays) simultaneously producing multiple results; and the subroutine libraries supplied with array processors require the user to operate on vectors or arrays of data. In other words, the program flow of the user's FORTRAN code has to be recast in terms of vector and matrix algebra as far as possible (a process known as 'vectorizing')[4]. This is a considerable exercise for programs of any size. The following example illustrates what is meant by vectorizing.

FORTRAN CODE

```
      subroutine blen(nbond)
      common/bond/xo(3,60),idxi(180),idxj(180),
      vcomps(180),bl(60)

      do 10 ib3 = 1,3*nbond
        iat1 = ((idxi(ib3) - 1)/3) + 1
        iat2 = ((idxj(ib3) - 1)/3) + 1
        ic = idxi(ib3) - ((iat1 - 1)*3)
        dc = xo(ic,iat1) - xo(ic,iat2)
        vcomps(ib3) = dc*dc
10    continue
      ij = 1
      do 100 i = 1,nbond
        bl(i) = 0.0
        do 20 j = 1,3
          bl(i) = bl(i) + vcomps(ij)
          ij = ij + 1
20    continue
      bl(i) = sqrt(bl(i))
100   continue
      return
      end
```

SAME CODE VECTORIZED FOR AP

```
      subroutine blen(nbond)
      common/bond/xo(3,60),idxi(180),idxj(180),
      vcomps(180),bl(60)

      call vsusqi(xo,idxi,xo,idxj,vcomps,3*nbond)
      ij = 1
      do 100 i = 1,nbond
        call sve(vcomps(ij),bl(i),3)
        ij = ij + 3
100   continue
      call vsqrt(bl,bl,nbond)
      return
      end
```

The two subroutines listed above calculate the distances between atoms (bl) from the square root of the sum of the squares of the vector components in the x, y, and z directions (vcomps). The arrays idxi and idxj point to the elements of the coordinate array xo between which the difference is desired. The code to calculate iat1, iat2 and ic is redundant but serves to illustrate the fact that xo may be treated as either a 1 or 2D array. The mnemonics for the array processor routines vsusqi, sve, and vsqrt are derived from 'vector subtract and square indexed', 'sum of vector elements', and 'vector square root' respectively (these subroutines come from the VPLIB library used in conjunction with a Glasgow University constructed array processor[5]). However, the important point about the above examples is the dissimilarity of the two blocks of code. This is common to the maths subroutine libraries of all array processors.

Even although a lot of work might be involved in vectorizing, say, a molecular mechanics program, one might expect that the user will see a massive reduction in run-time when compared with a conventional minicomputer. However, this is not the case. The reason for this is that every time a maths library call is invoked, the data to be operated upon is transferred from host to array processor memory and the results transferred from array processor to host memory; and as the maths library routines are more efficient with arrays containing hundreds or thousands of elements, it pays to use data arrays that are as large as possible. The I/O traffic resulting from these multiple, large, back and forth data transfers is both enormous and time consuming. In fact most of the gain from fast floating point arithmetic is lost in the I/O bottleneck.

The obvious way to alleviate the I/O problem is to perform as much of the calculation in the array processor as possible and minimize the host involvement. This is impossible if the processor is programmed via the maths library of vector and matrix function calls. Array-processor manufacturers provided some help by providing 'vector function chainer' software whereby no host involvement was required if the output of one array processor subroutine fed directly into the input of the next with no host FORTRAN in between. For molecular graphics, this means that sometimes up to seven or eight array-processor calls could be strung together, resulting in decreased run-times, but the situation was far from ideal. Until fairly recently, the only satisfactory solution to the I/O problem was to write as much of the program as possible in AP Assembler. This did mean that, for instance, a very large part of a molecular dynamics calculation could be array-processor resident and the host/processor I/O traffic reduced to almost insignificance. The problem is that coding a molecular dynamics program in Assembler takes several man years of work. However, despite these difficulties, Wilson's group at UCLA[6] and Scheraga's group at Cornell[7] coded molecular dynamics and molecular mechanics programs respectively in Assembly language for the FPS AP120B.

For this reason, array processors have not been widely used by the molecular graphics community in the past. But the situation was changed by the availability of host minicomputer resident FORTRAN compilers, which produced fairly effective code for the array processors. The first of these was the Toast compiler produced for the FPS range of machines by a British spin-off of FPS called System Software Factors. The major part of a molecular mechanics program was coded in Toast FORTRAN and PDP-11/FPS-100 hardware produced very creditable run-times when compared with the original minicomputer implementation. The Toast compiler is now available on other array processors. In order to get the best from AP FORTRAN compilers, it is still necessary to arrange the code in an order such that the compiler can spot structures that will execute efficiently on the processor hardware (in a similar fashion to the way that one orders FORTRAN for the Cray FORTRAN compiler). Even when this has been done, it is not unusual to find that the FORTRAN code takes 5–10 times longer to execute than the same program hand-coded in AP Assembler.

There is another way in which the I/O bottleneck can be avoided entirely, but this approach involves hardware rather than software and is used on some CSPI machines. It is wasteful for both the array processor and host minicomputer to have memory that is used to store the same data — why not have only one set of memory accessible by both the host and the array processor? Instead of acting as a peripheral, the processor sits directly on the host bus and uses host memory so that no host to peripheral and back data transfer

is required at all. The host writes a few bytes of data to the array processor telling it what operation is required, the base addresses of the input data arrays and a base address for the output data array. The array processor then takes over and performs the operation(s), and the results appear in a position automatically known to the host FORTRAN program. This means that it is now extremely efficient to use array processor maths library routines which are usually hand-coded by the manufacturer in AP Assembler. It must be noted that, in order to use this approach most effectively, the host and array processor must have identical floating point number representations.

This was the situation that prevailed early in 1982, the point at which the array processor manufacturer explosion occurred. Advances in microelectronics made possible the advent of cheap array processors for systems such as the IBM PC and other more powerful microcomputers, mid-range array processors competitive with the CSPI and FPS offerings, and top-of-the-range minisupercomputers*.

Systolic Systems produce a 1 Mflop array processor for the IBM PC, programmable via a subroutine library for about $4000. Probably the best known small array processors are those produced by Sky Computers. These are shared memory machines with performance in the 1–15 Mflop range, and they interface to the following buses: Q, Multi, S-100, Versa, and VME, supported by the following operating systems, RT-11, RSX-11M, TSX-Plus, Unix, CP/M-86 and iRMX-86. Programming is via a vector/matrix subroutine library or optional Micro-assembler. The top-of-the-range machine costs approximately $15 000. There are also a number of block floating point (integer) array processors produced by various manufacturers, but in general these are not suitable for molecular modelling work.

The most crowded area of the array processor marketplace is for machines in the $20–100 000 price range. The novel FPS and CSPI machines have already been discussed. The Analogic AP500 is broadly similar to the FPS machines and performs at around 10 Mflop. It has a greater arithmetic precision than most nominally 32-bit machines and a large data address space of 1 Mbyte. It functions as an I/O peripheral and has no FORTRAN compiler. The cost is around $40 000.

At the top of the 32-bit array processor mid-range comes the Numerix MARS-432. The basic machine performs at around 30 Mflop, has a 64 Mbyte address space, a FORTRAN compiler, a subroutine library, a macrocode Assembler (the user writes code in a minicomputer-like Assembly language and the Macro-assembler converts it to microcode) and a microcode Assembler. A 0.25 Mbyte machine costs around $80 000. Numerix appear to have removed most of the problem areas typical of array processors and taken trouble to make the machine easy to program and use. The only minor problem is that the machine is still an I/O peripheral, but, with a data address space of 64 Mbyte, most molecular modelling programs can be squeezed entirely into the processor, making it virtually standalone.

*The author wishes to point out that the machines discussed both prior and subsequent to this paragraph are merely those with which the author is acquainted – they may or may not be the best in their class, and there may be very good machines that are not mentioned.

One of the most recent events in the array processor world is the availability of minisupercomputers with performance in the 60–150 Mflop range, with prices around $500 000. This is very cost effective compared with Cray or CDC supercomputers. Two or three machines are already available with at least one more imminent.

The first of these machines is the ST-100 from Star Technologies. This array processor has an FPS-like 32-bit architecture and a claimed peak calculation rate of 100 Mflop. Unlike older machines, the ST-100 is implemented using very high speed ECL (emitter coupled logic) gate arrays and has a hardware divide and square root module. (All of the processors mentioned so far implement floating point divide using a variant of Newton's approximation which requires only multiply and add, a look up table being used to provide the initial approximation. This makes divide 5–6 times slower than multiply.) It is attached to the host as an I/O peripheral and is programmable via a subroutine library and/or a FORTRAN IV subset-like control language. Good performance appears to be possible although the machine is difficult to program, in a manner reminiscent of early FPS and CSPI machines.

The second machine is very different. The Convex Computer Corporation's C1 is a 64-bit standalone vector/array processor with a Cray-1 like architecture. It runs the Unix operating system with C and a FORTRAN 77 compiler which accepts VAX FORTRAN code. It can have up to 128 Mbyte of physical memory, and, uniquely for a nonmainframe array processor, it has a virtual memory operating system. Also included are hardware add, subtract, multiply and divide with a peak performance of 60 Mflop. The machine is almost ideal for molecular graphics work, having the necessary 64-bit precision to deal with all molecular orbital calculations as well as molecular mechanics and molecular dynamics calculations. There are only two drawbacks: the price of around $500 000, and the fact that the vectorizing FORTRAN compiler is almost certain to waste at least 50% of the raw power of the machine.

Few of the machines discussed above (probably only the Sky, Systolic and Convex machines) can be used in conjunction with a real-time, interactive molecular graphics program, but must instead be used as batch machines. There is also the apparently conflicting requirement for ease of programming via a standard FORTRAN compiler without the loss of power that vectorization invariably entails. A partial solution to both of these problems entails the use of a processor architecture radically different from those discussed previously. Commercial products embodying this architecture are just starting to appear, the first of which are the Intel iPSC[8] and the Myrias machine[9].

Figure 6 illustrates the new architecture. Each processing element (prel) typically consists of a 16- or 32-bit microprocessor, 128 kbyte–2 Mbyte of memory, a 0.1–10 Mflop scalar floating point processor chip, together with I/O channels to receive commands and data, and transmit data. The array of processing elements is controlled by the master processor (identical to a prel but with more memory, discs and terminal I/O) which loads the appropriate code and data into each prel. The interconnect array can be a simple star, a systolic loop, a four dimensional hypercube, etc.

In order to see how this architecture is of use, let

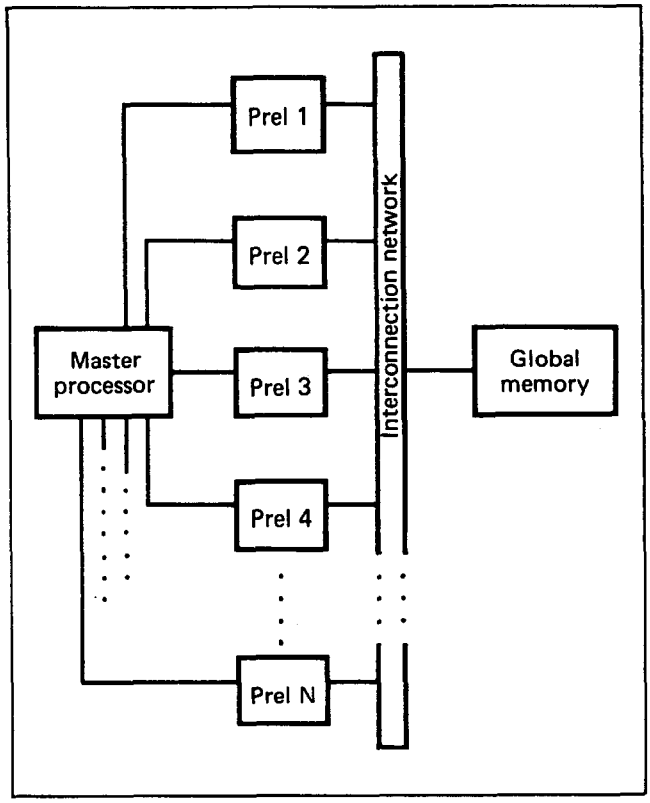


Figure 6. Architecture of the multiprocessor parallel processing-type machines

us consider an energy minimization program. Regardless of which minimization algorithm is used, the central problem is to calculate a shift vector for each atom so that the steric energy decreases from iteration to iteration. Within each iteration, provided that each atom sees the total force-field generated by all of the others, the shift calculations for each atom are computationally independent.

A possible strategy for molecular mechanics on a machine similar to that shown in Figure 6 might be as follows (alternative strategies are also possible). Write a standard molecular mechanics program that optimizes the position of only one designated atom at a time, using a standard FORTRAN compiler on the master processor. Then load a copy of this program into each prel (which uses an identical processor to the master) and a copy of the molecule's atomic coordinates into the global memory. Instruct each prel to optimize the position of a different atom whose coordinates are in the global memory and store the result in temporary local memory; then give each prel another atomic position to optimize (still using the force-field due to the initial coordinates). The result is again stored locally. The process is repeated until a new set of coordinates exists in prel local memory. When all the positions have been optimized, the locally stored coordinates from each prel are sent to the global memory, so that each prel now has access to a complete set of updated coordinates. The second and subsequent iterations are a repeat of the above process until the master processor determines that convergence has occurred. the advantage of this approach is that one obtains a high degree of parallelism not only for the floating point arithmetic but for the normal scalar instructions as well. Because the task that each prel executes is fairly considerable in terms of run time, the degree of parallel processing can approach 100%; the time required to shift the small amounts of data involved is negligible.

All of the procedures for which the use of an array processor would be considered in molecular graphics can easily be divided into a number of concurrent tasks as illustrated above. As these parallel processors contain from 32–64 kprels, their performance is very impressive and the achievable computational rate is close to the theoretical maximum, unlike pipelined array processors of the types previously discussed, which do well to sustain 10% of their peak rate.

The only minor drawback to multiprocessor systems of this kind is that the user has to take care of the segmentation of the algorithm and the loading and synchronization of the prels. The vendor usually provides a small subroutine library to simplify this task, and it has been estimated that writing code for prel arrays takes about 20% longer than writing code for a standard minicomputer. This is a small price to pay when the gains are considered.

CONCLUSIONS

This paper has discussed the principles of array processors as applied to computer graphics and molecular-x calculations. The use of array processors in molecular graphics has also been discussed.

It has been shown that the picture transform processors in high performance colour graphics displays would benefit from operation in 32-bit floating point, rather than 16-bit integer, mode and also that user programmability would be a desirable feature.

In the discussion above it has been demonstrated that array processors may be used advantageously for batch mode molecular graphics calculations and also for interactive calculations, provided that a shared memory machine is utilized. Software development is now fairly easy, even if the code generated is somewhat inefficient.

Multiprocessor parallel machines, on the other hand, are not only generally useful for batch and interactive calculations but are also very efficient. The price paid for this efficiency is a marginal overhead in the ease of software development.

REFERENCES

1 **Foley, J D and Van Dam, A** 'Advanced display architecture' in *Fundamentals of interactive computer graphics* Addison Wesley, USA (1983) pp 391–429
2 *TRW LSI multipliers* HJ Series. Data Sheet No 107A-11/78. TRW Inc, USA (1978) pp 1–20
3 **Ostlund, N S** 'Attached scientific processors for chemical computations: a report to the chemistry community' NRCC, University of California, USA Doc. LBL-10409 UC-32 (1980) pp 20–31 (Available from: National Technical Information Service, US Department of Commerce, 5285 Port Royal Road, Springfield VA 22161, USA)
4 *Infotech state of the art report: supercomputers Vol 1* Infotech International, UK (1979) pp 107–115
5 **White, D N J** 'A micro vector processor for molecular

mechanics calculations' Supercomputers and Chemistry, ACS Symposium Series, No 173, American Chemical Society, USA (1981) pp 193–236
6 **Berens, P H and Wilson, K R** 'Molecular mechanics with an array processor' *J. Comp. Chem* Vol 4 No 3 (1983) pp 313–332
7 **Pottle, C, Pottle, M S, Tuttle, R W, Kinch, R J and Scheraga, H A** 'Conformational analysis of proteins: algorithms and data structures for array processing' *J. Comp. Chem* Vol 1 No 1 (1980) pp 46–58
8 **Rosenberg, R** 'Super cube' *Electron. Week* Vol 58 No 6 (February 1985) pp 15–17
9 **Hollenberg, J** 'The Myrias 4000 parallel computer system' *Supercomputer* No 5 (1985) pp 8–9