MASSIVELY PARALLEL COMPUTERS

R N IBBETT

Rapporteur: M J Elphick

MASSIVELY PARALLEL COMPUTERS

R.N. Ibbett
Department of Computer Science
University of Edinburgh
The King's Buildings
EH9 3JZ

Parallelism comes in essentially two varieties, *data parallelism* and *code parallelism*, and most available systems can be divided correspondingly into Flynn's SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) systems. A few systems can act as both (e.g. the Fujitsu AP-1000) and Multiple SIMD systems will doubtless appear as technology progresses.

SIMD systems have been around for a long time, and are relatively easy to program, since the code is sequential, and what is needed are language extensions (typically to FORTRAN) to allow software data structures to be mapped on to the data parallelism in the hardware. Naturally the usefulness of these systems depends strongly on there being a significant amount of data parallelism in the application, and although they represent a fairly mature technology, their use has until recently been largely confined to niche areas.

MIMD systems are taking longer to mature since they offer an additional dimension of complexity *i.e.* multiple, concurrent threads of code execution, and this calls for a great deal more human ingenuity to make effective use of such systems. The interconnection problem is also more complex, since whereas SIMD systems depend for their performance on regularity in the data and consequently require, in most cases, only simple geometric communication patterns, MIMD systems frequently involve either full or random communication patterns which cannot easily be provided in hardware.

Thus the answer to the question "What constitutes a massively parallel system?" is tens (possibly hundreds) of thousands of processors in SIMD systems, but only hundreds (possibly thousands) of processors in MIMD systems.

1 MIMD Systems

The taxonomy of MIMD systems has been an area of fruitful (fruitless?) academic endeavour for many years. The principal division is between shared memory systems and message passing systems. Shared memory systems (*e.g.* the Cray X-MP/Y-MP series) have been successful as supercomputers primarily because each processor is so powerful; the scope for parallelism is strictly limited, simply because all the memories are shared by all the processors and there has to be a bottleneck somewhere. Message passing systems provide greater scope for parallelism. Here the novelty has been primarily in devising interconnection schemes which are bounded in time and space by something less than the square of the

number of processors (*c.f.* the cross-bar switch in C.mmp, the original multiprocessor). Hypercubes and butterfly networks have found favour in the USA; in Europe, where most parallel systems are based on the Transputer, more *ad hoc* solutions involving a multiplicity of small configurable cross-bar switches have been used. These allow the user to change the topology to suit the application. If the problem seems to require a tree of processors, a square or even a hypercube, the switches can be configured accordingly. These switching mechanisms were developed as extensions to the limited functionality of the point-to-point communications protocols implemented in the Transputer. The development of efficient software routing protocols first in software [7] and in hardware in the T9000 is now obviating the need for switch configurability.

Hypercubes

The hypercube, formally known as a 'binary k-cube', connects $N = 2^k$ network nodes in the form of a cube constructed in k-dimensional space. The corners of this cube represent the nodes, and the edges represent the inter-nodal connections. More formally, if the nodes are numbered from 0 to $2^k - 1$, nodes whose binary numbering differs in exactly one position have connections between them. Figure 1 shows how binary k-cubes are constructed for k in the range 0 to 4.

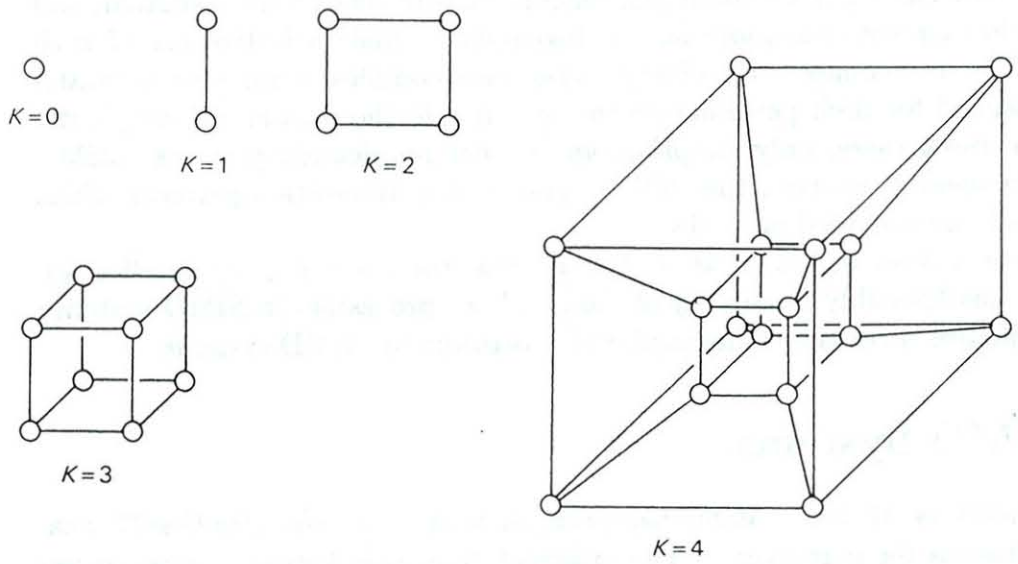

Figure 1: Constructing binary k-cubes

The binary k-cube therefore has k routing functions, C_i $\{0 \leq i \leq k-1\}$, one routing within each dimension, defined thus

$$C_i(x) = \{\epsilon_i \mid I\}(x)$$

Informally, for each dimension either an exchange permutation (ϵ_i) or an identity permutation (I) is applied to x in order to establish a route from any source

to any destination node. A route from any source to any destination label can be found by starting at the source node and then comparing each bit in the source and destination labels in turn. If the bits are the same, then the identity permutation is applied to the source label and the route is not extended. If the bits are different, then the exchange permutation is applied to the source label, and the route extends along the link connecting the current node to a new node with a label equal to ϵ_i(current label). Such a route is illustrated in figure 2. Since the maximum number of bits required to identify n processors uniquely is $k = \lceil \log_2 n \rceil$ the path length between an arbitrary pair of nodes is at most k.

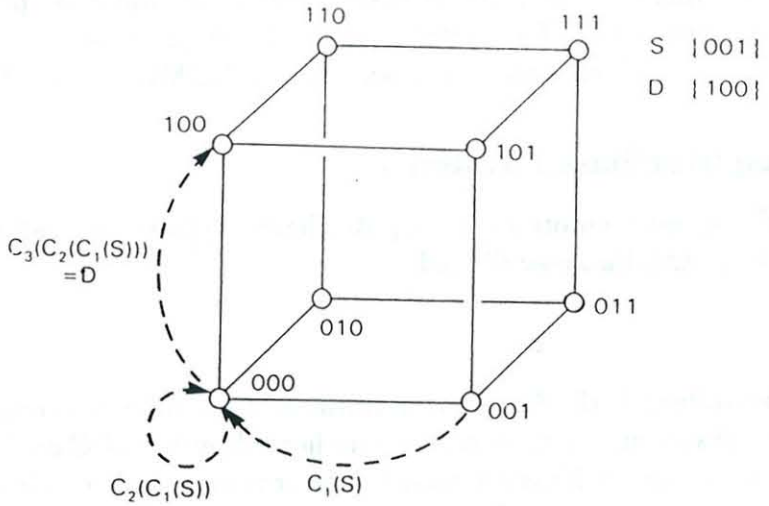

Figure 2: Routing in a binary k-cube network

It is immediately apparent that the binary k-cube has a very rich interconnection structure, with a total of $k2^{k-1}$ bidirectional connections, and k communication links per node. One possible problem, which could limit the number of nodes in an k-cube network, is the number of communication links required per node, and hence the physical complexity of the whole network. In fact, it is the length of the interconnecting wires which poses the most serious problem for networks with large values of k. This can be shown by examining the rate of growth of the volume of the network.

The rate of growth of the inter-nodal distances in a binary k-cube depends on the length of one side of the machine. Since most machines are constructed physically in three-dimensional space, one side of a machine must have length which is $\Theta(N^{1/3})$. Consequently, the time delay associated with the transmission of messages across the most significant dimension of the network will also be equal to $\Theta(N^{1/3})$. If the system is synchronous then the clock speed of the machine must decrease in proportion to this increasing delay; alternatively, if each processor runs at $O(1)$ instructions per second then the interval between each communication event must increase in proportion to the increased transmission delay. The net effect of increasing wire length is that the communication bandwidth per node

decreases as the system becomes larger. This is essentially a problem of *physical scalability.*

The Cosmic Cube [17] and Mosaic [16] experiments carried out by Seitz at Caltech are typical of the kinds of architecture that can be constructed using binary k-cube topology. The commercial derivatives of these are the Intel iPSC machines and the machines from N-Cube Corporation.

The initial Intel range of iPSC/1 machines, and the subsequent iPSC/2 and iPSC/860 machines, all have a maximum configuration size of 128 processors, and offer peak performances of 27 MFLOPS (iPSC/2) and 7.6 GFLOPS (iPSC/860). A maximum configuration N-CUBE-2 system would contain 8192 processors; the largest system delivered so far contains 1024. Each processor node contains a Vax-like 64-bit CPU which offers a performance of 7.5 MIPS/3.3 MFLOPS.

1.1 Transputer based systems

The major European vendors of Transputer-based systems are Meiko Scientific Ltd., Parsys Ltd. and Parsytec GbmH.

Meiko

Meiko's main product is the Computing Surface, a modular and expandable distributed memory system. Early versions were hosted by Vax, PC or Sun machines, but current versions are self-hosted, stand alone or networkable multi-user systems. A Computing Surface consists of:

- an arbitrary number of individual processors (which may be Transputers, but may also be SPARCs or i860s)
- an arbitrary number of special purpose boards (graphics, I/O, etc)
- an interconnection network for message passing
- a supervisor bus which provides some control functions for the system as a whole.

Interconnection between the processors is provided by means of custom VLSI crossbar switches which connect the processors to the backplane, and by the backplane routing mechanism itself, details of which remain confidential. There is no theoretical limit on the number of processor modules which can be installed in a system, but an upper limit on the length of the intermodule link wire imposes a practical limit. The largest system currently in operation is the Edinburgh Concurrent Supercomputer, which contains over 400 compute processors.

Parsys

Parsys produce the SuperNode series of machines (backplane compatible with Telmat T-Node machines) which offer a maximum possible configuration of 1024 Transputers (the largest so far delivered contains 256). The prime feature of the SuperNode is its switch architecture, which was designed as part of an ESPRIT project. A single level switch is constructed from eight 72-way crossbar switches, which can themselves be interconnected by further switches to form a hierarchy.

Parsytec

The top range machines from Parsytec, the SuperCluster series, are stand alone machines with up to hundreds of Transputers. The basic building block is a 16-Transputer cluster which contains a Network Control Unit (NCU). The NCU provides full connectivity between the 16 Transputers in the cluster and has a further 32 links with which to interconnect clusters. Systems with up to 400 Transputers have been delivered.

2 SIMD Sytems

The evolution of SIMD array processors can be traced as far back as 1958, when Unger published a paper entitled "A Computer Oriented Towards Spatial Problems" [20], from which the first array processor SOLOMON was developed [18, 10]. The SOLOMON design consisted of a two-dimensional array of 32×32 processing elements (PEs), each of which had 128 32-bit words of store and a bit-serial arithmetic unit. All PEs acted in unison, under the control of a single stream of broadcast instructions. The SOLOMON design had a major effect on the subsequent thinking of computer architects, and led to the development of several important high-performance architectures including the ILLIAC IV machine [1, 9], the Burroughs Scientific Processor [13], the Burroughs PEPE machine [8, 21], the Goodyear Aerospace MPP [5], the Goodyear Aerospace STARAN [2, 3, 4], and the ICL Distributed Array Processor (DAP) [14]. Advances in VLSI technology have had a considerable impact on the design of SIMD array processors. The reduction in minimum feature size, and the availability of high-density gate-arrays and full-custom VLSI as a means of realising a particular implementation contributed towards the construction of the Connection Machine [11] by Thinking Machines Corporation in 1985.

2.1 The DAP

The ICL DAP enjoyed moderate commercial success and extensive use by the scientific research community, particularly in the U.K. In 1986 Active Memory Technology was formed as a spin-off company to develop a VLSI version of the DAP suitable for use as an accelerator hosted by, for example, a MicroVAX or a

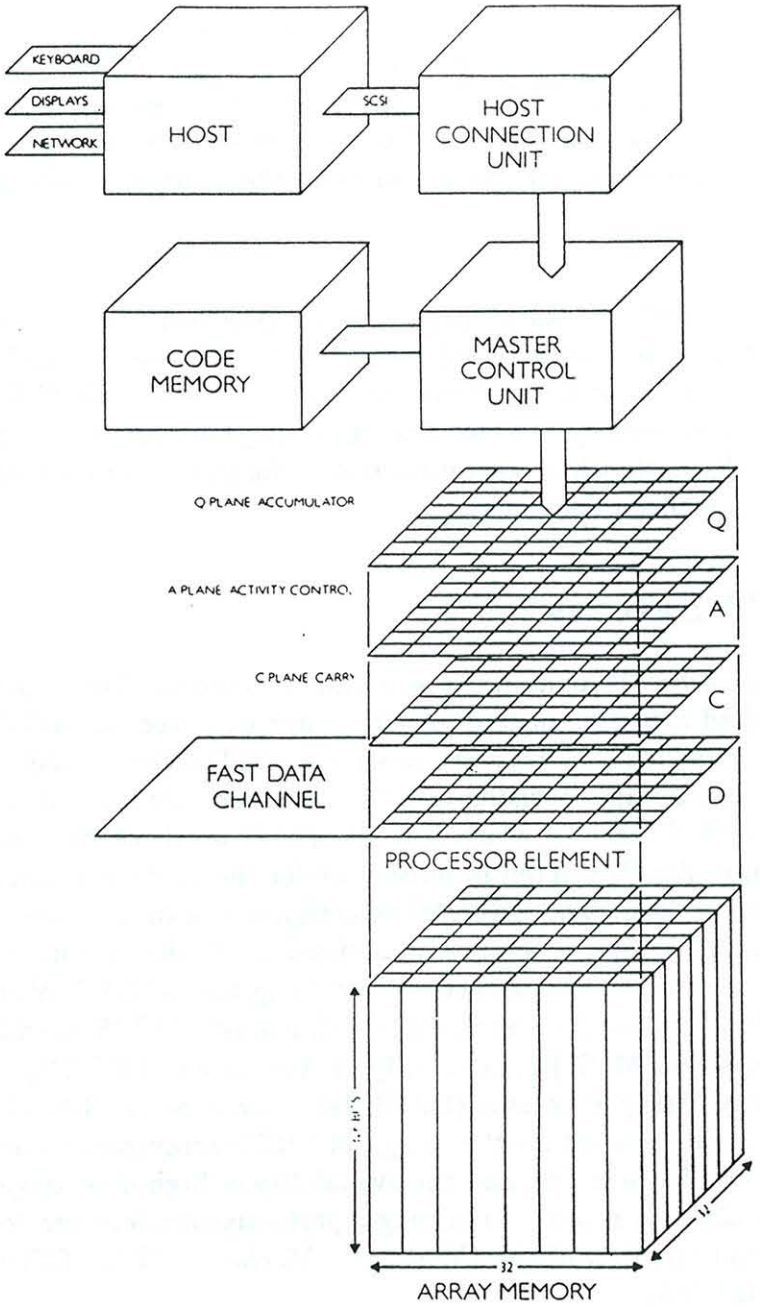

Figure 3: DAP architecture

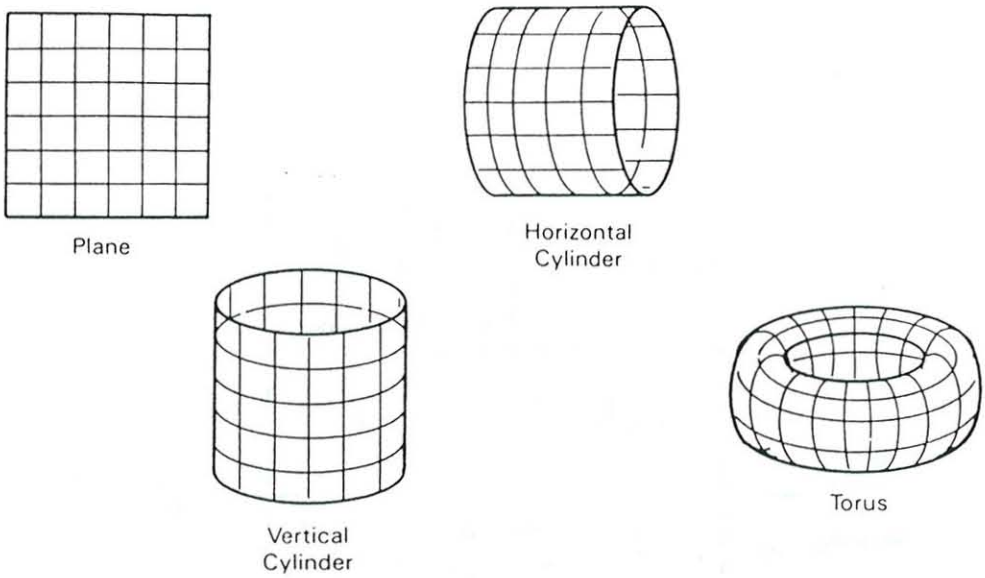

Figure 4: DAP array geometries

Sun workstation. The ICL DAP had a 64×64 processor array; the first AMT DAP (Model 510) had a 32×32 array, but is now also available in a 64×64 configuration (Model 610), running on a 10 MHz clock.

The DAP architecture is shown in figure 3. The processing elements (PEs) are arranged in a square array, and each comprises a single-bit processor (in the DAP/CP8 range announced in 1990 each PE also includes an 8-bit coprocessor). Each PE has a local memory which can range from 32 Kbits to 1 Mbits per PE.

Each PE has input connections from its four nearest neighbour processors in the North, South, East and West directions. The boundary connections at the perimeter of the array are determined by bits in the instruction. Either the boundary inputs are set to zero and boundary outputs are discarded, or else the boundary inputs are taken from the boundary outputs within the same dimension. Hence, East may be connected to West and North may be connected to South. The resulting four geometries are illustrated in figure 4.

Program control is carried out by the Master Control Unit, which takes instructions from the Code Memory, interprets then and controls the PEs, the memory and data transfers. Access to the array is via row and column data highways.

Interaction between the DAP and the host is controlled by the MC68020-based Host Connection Unit. Data can also be transferred to and from the memory via the D plane over a fast data channel (70 Mbytes/s). This is particularly useful for the attachment of high-speed graphics displays.

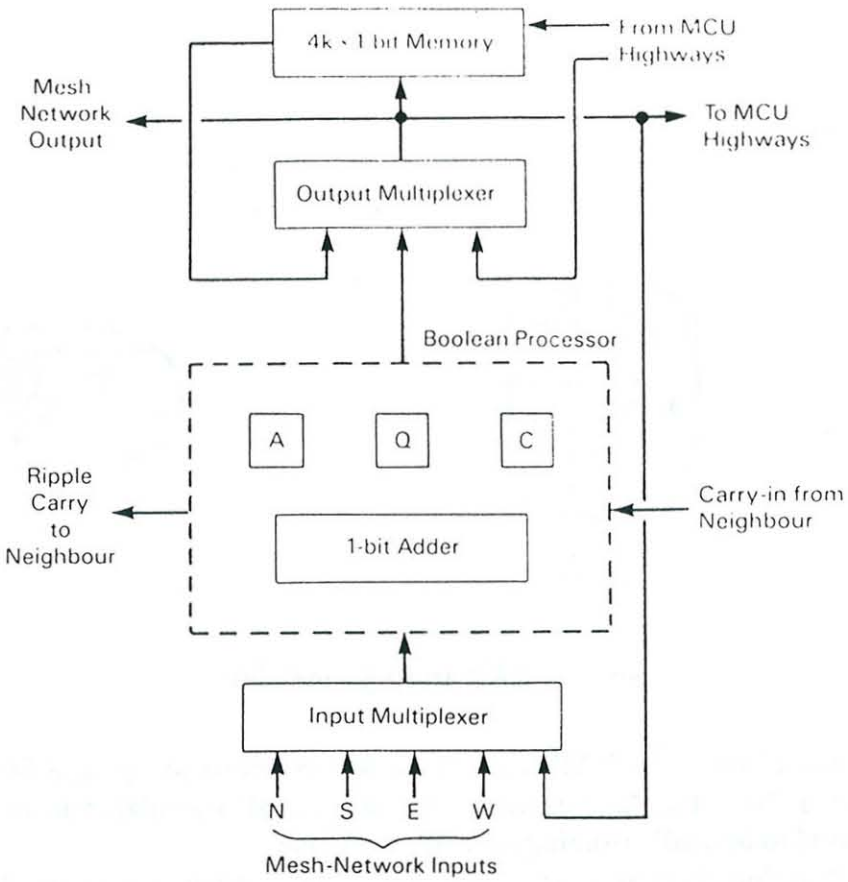

Figure 5: DAP processing element architecture

PE architecture

A simplified view of the internal architecture of a single PE is shown in figure 5. Each processing element consists of a single-bit adder, an input multiplexer, an output multiplexer and an $n \times 1$-bit store. The ALU consists of three one-bit registers, the accumulator Q, the carry register C and an *activity* bit A. The activity bit is used for local enabling or disabling of certain actions within the PEs, thus permitting a subset of the array to take part in whatever computation is in progress.

The input multiplexer selects data either from the output of one of the four nearest neighbours or from the local memory, depending on the instruction being executed. The output multiplexer selects which source of information is used when writing to the local memory. The options include the output from the local adder and the row and column highways.

The single-bit adder performs full addition of the accumulator and the selected input, with an optional carry input. The selected input may be complemented before addition, enabling subtraction and logical inversion operations to be implemented. The carry-in to the single-bit adder may come from the local carry register

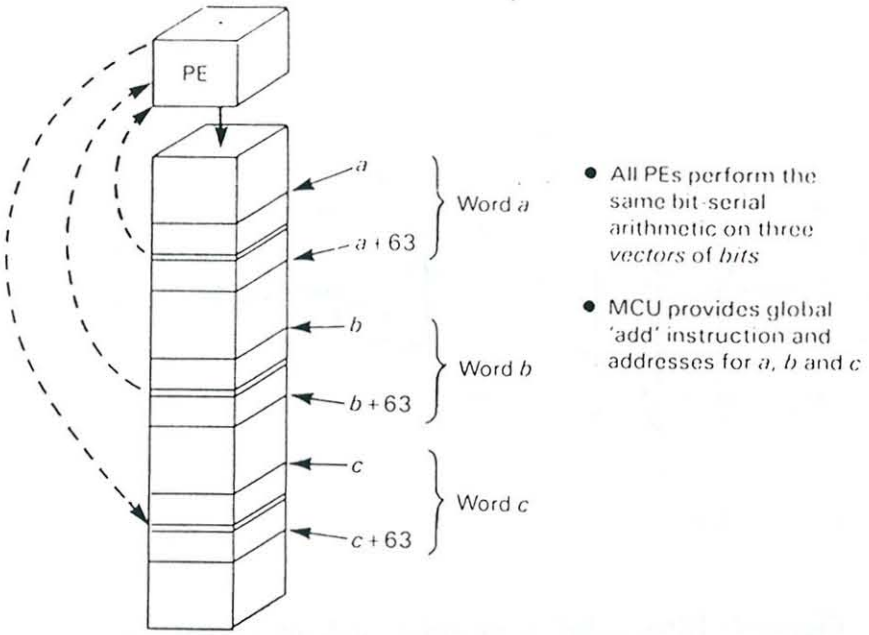

Figure 6: Bit-serial word-parallel mode of arithmetic

or the carry-out of the Eastern neighbour, depending on the operating mode of the array. This choice permits the DAP to perform word-arithmetic in two quite distinct ways, either *bit-serial (word-parallel)* or *bit-parallel (word-serial)*. These two modes of operation are illustrated in figures 6 and 7.

The normal mode of arithmetic in the DAP is bit-serial word-parallel. In this mode word values are assumed to be stored vertically as vectors of bits in the z dimension of figure 3. A full word operation is programmed out as a DO loop, consisting of n iterations, for n-bit words. As an example, consider the addition of 64-bit integers, stored as the bit-vectors represented by $\underline{a}$, $\underline{b}$ and $\underline{c}$. To perform $\underline{a} + \underline{b} \rightarrow \underline{c}$ it is necessary to index through these three bit-vectors, adding the two operands in bit-serial fashion, and storing the carry at each stage in the C register. This sequence of operations can take place in all the PEs simultaneously. Thus, although the time to perform a single Integer Add takes many clock cycles, the massive parallelism can nevertheless produce very high overall processing rates.

An alternative method of performing word arithmetic, which is supported by the DAP system software, involves configuring each row of PEs as a 64-bit ripple-carry adder. This permits words stored in the x-dimension to be operated on directly with a guaranteed carry-propagate speed of at least four bit-positions per clock period. Under this scheme the three operand addresses are scalar values, addressing a single bit in each row memory.

Although this method of processing is essentially word-serial within each row, the fact that there are 64 rows means that a moderate amount of word parallelism also occurs in this mode. Bit-serial word-parallel mode yields a much greater

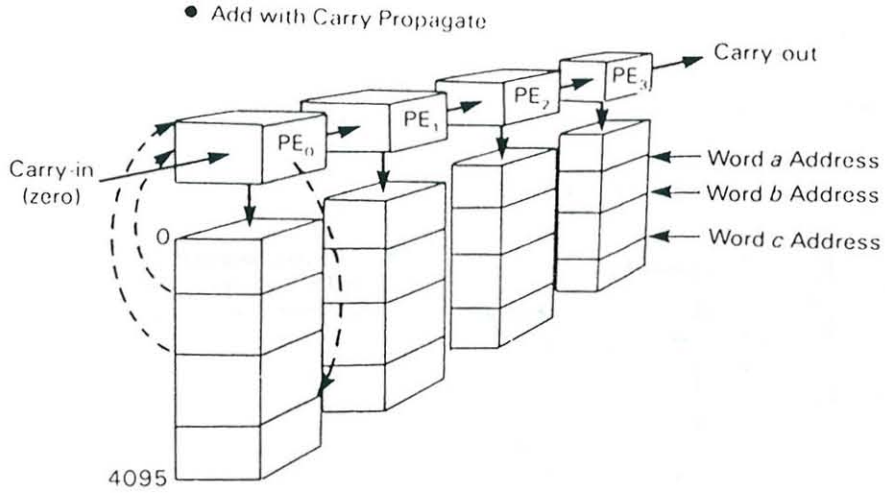

Figure 7: Bit-parallel word-serial mode of arithmetic

maximum performance level than the bit-parallel word-serial mode, however, due to the relatively slow carry-propagate speed compared with the cycle time of the carry-save technique used in bit-serial mode.

Performance

The performance of the DAP can be considered in several ways. At the simplest level we can examine the raw speed of its component parts and compare them with other high performance scientific machines. This produces a set of *peak* performance figures, but does not advance any insight into how well the machine will perform on a real problem.

It is possible to characterise the raw performance of the DAP in terms of the bandwidth of the distributed memory, the serial arithmetic rate and the rate of data manipulation through the processing element network. The clock period of the production ICL DAP was 200 ns, and in this time it was capable of performing one memory cycle in each processing element memory. Each memory operation involved one bit, and therefore the raw memory bandwidth was

$$\frac{4096}{2 \times 10^{-7}} = 20.48\,\text{Gbits/s}$$

This is four times the 80 MWord/s effective memory bandwidth of the CRAY-1, although in fairness the CRAY-1 also has a very fast set of vector registers which provide all the operands for the computational units. The CYBER 205 has a memory bandwidth of 200 MWord/s per Pipe, and hence a 2-Pipe CYBER 205 has 25% more memory bandwidth than the DAP. This is a fair comparison since the CYBER 205 architecture implements memory-to-memory vector operations.

Table 1: Instruction timings for the DAP

Operation	Time (μS)	Processing rate (MOPS)
$Z \leftarrow X$	17	241
$Z \leftarrow X * S$	40–130	32-102
$Z \leftarrow X^2$	125	33
$Z \leftarrow X + Y$	150	27
$Z \leftarrow \sqrt{X}$	170	24
$Z \leftarrow X * Y$	250	16
$Z \leftarrow X/Y$	330	12
$Z \leftarrow \mid Z \mid$	1	4096
$S \leftarrow \sum_{i=1}^{4096}$	280	175
$I \leftarrow J + K$	22	186

Arithmetic performance in the DAP is heavily dependent on the chosen word length, w; addition and subtraction requiring $O(w)$, and multiplication requiring $O(w^2)$ micro-cycles respectively. According to Reddaway [14], integer addition takes $3w + \Delta$ cycles, where Δ is a small constant value, and fractional integer multiplication takes

$$\frac{w\,(3w + 13)}{2}$$

cycles. Floating-point operations require extra cycles due to the exponent arithmetic, mantissae alignment and result normalisation. Table 1 shows the timing, and resulting processing rates for a representative sample of bit-serial fixed and floating point operations, taken from [15]. All operations are in 32-bit precision and are hand-crafted, assembler-coded system routines. The X, Y and Z values are real arrays containing 4096 elements, S is a real scalar value and I, J, and K are integer arrays containing 4096 elements.

Several points are worth noting from these figures. Firstly, because bit-serial algorithms for transcendental functions are very different from their equivalent algorithms on bit-parallel machines so that, for example, the time to compute the square root of a real number is less than the time to compute the product of two real numbers. The implementation of certain functions is trivial; for example,

computing the absolute value of 4096 real numbers takes only 1 μs thus yielding a burst processing rate of 4096 MOPS. The technique of optimising at the bit level is exemplified by the $\sum$ operation which, instead of taking $\log_2(4096) \times 150$ (i.e. 1650) cycles, takes only 280 cycles.

A major feature of the DAP architecture is the two-dimensional PE interconnection structure. This structure is capable of shifting an array of 64 $\times$ 64 bits, held in the Q registers at a rate of one shift per clock period, excluding instruction startup overheads. Hence, to move a bit of information from one memory to another takes

$$x + y + \Delta$$

clock cycles, where x and y are the relative displacements of the source and destination memories within the array and Δ is a small overhead for instruction fetch and memory read/write cycles. The grid of interconnections and the Q registers together form a parallel switch with a peak throughput of 4096 bit position transfers per clock period, or 20.48 G bit-positions/s. It is also possible to use the row and column highways to move any single row or column of 64-bits into an MCU register, or to move the contents of an MCU register into one or all of the rows or columns of the array. These data transfer operations can be carried out at a rate of one every 2.5 clock periods. This is an extremely powerful mechanism, as it permits the rows and columns to be selected, exchanged or broadcast to the whole array very rapidly.

2.2 The MasPar MP-1

The basic architecture of the MasPar MP-1 family of SIMD processors [6] is similar to that of the DAP, but uses a different interprocessor communication mechanism. It is available in five array sizes of between 1024 and 16384 processing elements configured in each case as a two-dimensional array. This array is tightly coupled to a Vax front-end host.

The processing elements are RISC-like processors grouped in clusters of 16 arranged as a 4 $\times$ 4 grid. Each PE cluster also has associated PE memories (currently 16 Kbyte DRAM) and connections to the communications network. Each PE provides operations on 1, 8, 16, 32 and 64-bit operands includes a 64-bit mantissa unit, a 16-bit exponent unit, a 4-bit ALU and a single-bit logic section. Each functional section within the PE can be active simultaneously during each microcode instruction; floating-point operations, for example, require use of the exponent, manitassa, ALU and logic units together. The internal 4-bit nature of the PE is not visible to the user, allowing future machines to be produced with larger (or smaller) ALUs without changing any user code.

Instructions are issued to the PEs by the Array Control Unit (ACU) (figure 8) which fetches and decodes instructions, computes addresses, performs scalar arithmetic, sends control signals to the PE grid and monitors the PE array status. (Its function is thus similar to that of the MCU in the DAP.) Currently it is a RISC-like

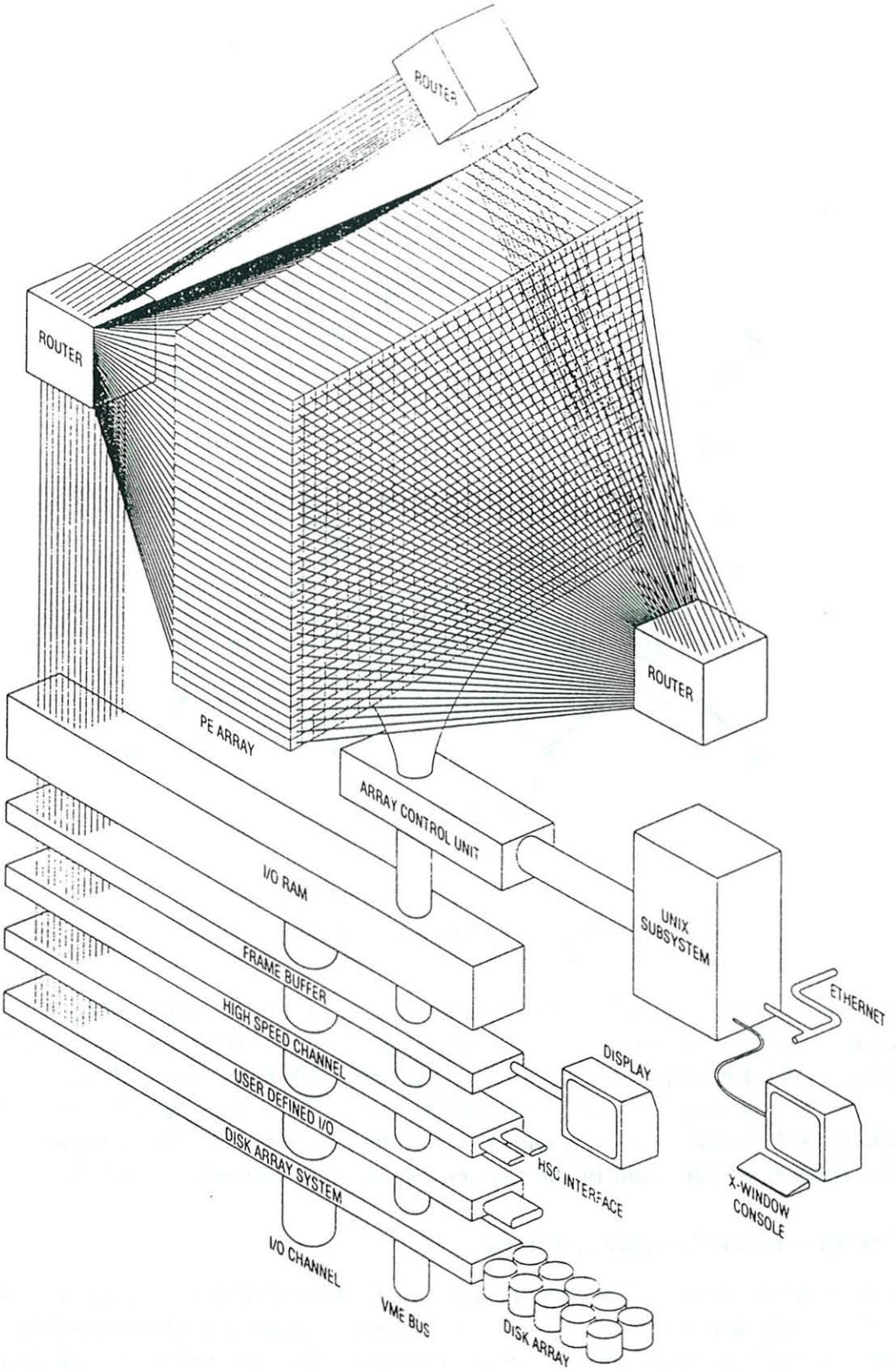

Figure 8: The MP-1 System

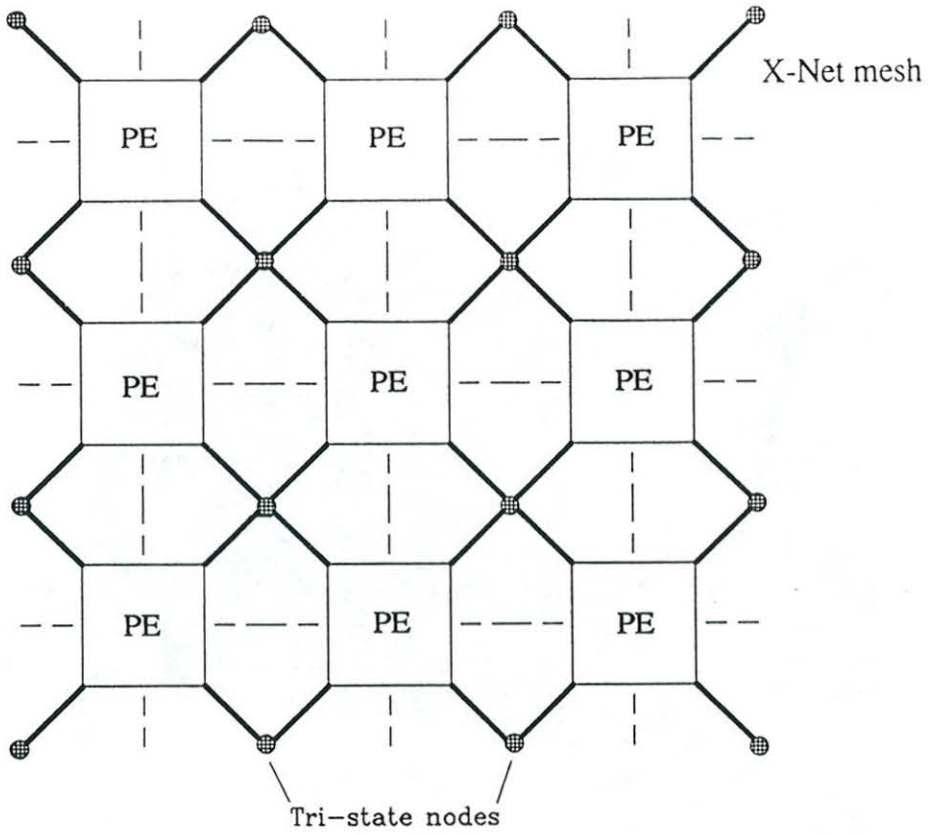

Figure 9: The MP-1 X-Net

processor based on standard TI chips, but may in future be replaced by a faster commercial RISC processor which could also act as the front end. The ACU is linked to the PE array by a 48-bit data bus and a 4-bit instruction bus.

The hardware provides indirect addressing of PE arrays, which means that individual PEs can access variables held at different offsets within their memories. This feature is also found in the Connection Machine but not in the DAP.

Interprocessor Communications

Interprocessor communications are handled by two separate mechanisms. The choice of which is more appropriate for a given application is determined by the regularity of the data transfer. For situations in which an entire array of data is to be moved across the PE lattice then the X-Net communications mesh is used.

Conceptually, the X-Net mesh is a one-bit wide communications network which links each processor with its eight nearest neighbours. In fact each PE has only four interconnects, located at its diagonal corners, thus forming an X-shaped grid at 45 degrees to the PE lattice, rather than the square grid of the DAP. However,

the X-Net mesh is more sophisticated than the DAP mechanism, since the diagonal links do not simply cross over each other, but are connected at nodes which allow communications to be routed to any of the PE's eight nearest neighbours (figure 9). The direction for the outgoing message through this tri-state node is set by the ACU at the same time as the PEs are instructed to transmit, so that there is no latency in the connection and hence the communications of a bit between neighbouring processors takes one clock cycle. At the edges of the PE array the interconnects are wrapped around to form a torus, though the user may select planar boundaries, in which case any differences between the topologies are handled by software. The aggregate X-Net bandwidth for the MP1101 systems is 1.1 Gbyte/s; this increases linearly with systems size.

Random communications between arbitrary processors are possible via a three-stage global router which emulates a crossbar switch. Each PE cluster has a connection to the router stage of the switch and another to the target stage. These ports are shared by all PEs within the cluster. The router and target units of the global switch are connected by an intermediate stage. The address of the target PE is calculated by the originating processor and, if all the links through the router between the start PE cluster and the finishing one are free, then connection is established. Clearly, this may necessitate some PEs waiting, perhaps for some time, for others within the same cluster to finish their data transfer. Once set, the link is bi-directional and on closing the target PE sends an acknowledgement. For the smallest MP-1 machine (with 1024 PEs) this communication hardware acts like a 64×64 crossbar switch. Data transfers through the links are bit-serial and clocked synchronously with the PE clock. Since the router ports are multiplexed for each PE cluster arbitrary communications patterns require a minimum of 16 router cycles to complete. A connection through the router takes 40 clock cycles to establish and a further 10 to close. MasPar claim to be working to reduce these quite considerable overheads but, in any case, applications which require the global router would be more difficult to code on other SIMD computers.

2.3 The Connection Machine

The design philosophy of the Connection Machine [11] challenged the conventional view of what constitutes an efficient computing machine, by shifting the emphasis from an obsession with instruction cycle times to a more realistic consideration of processor-memory bandwidth requirements. The designers observed that technology has evolved to the point where processors and memories are made using a common (VLSI) technology. Thus the physical separation of processors and memory inherent in earlier technologies (and the cause of the so-called 'von Neumann bottleneck') need no longer be maintained, and a processor can be placed *in* the memory to create a *cell* which is then replicated as a unit to create large and highly parallel systems. This form of logic-in-memory is no different in principle from that used in the DAP (or indeed SOLOMON); what the Connection Machine emphasises is the *programmability* of the connections between processing cells.

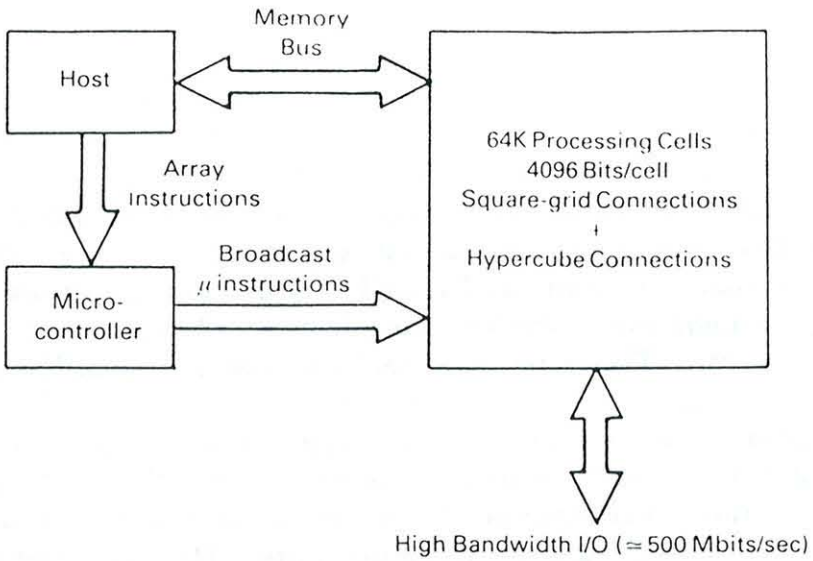

Figure 10: Architecture of the Connection Machine

The first Connection machine, the CM-1, contained 16K processors and was first delivered in 1986. In 1987 a revised version, the CM-2/CM-2a range of machines was announced. The CM-2a can contain 4K or 8K processors, while the CM-2 can contain 16K, 32K or 64K processors.

System architecture

The system level architecture of CM-1 is illustrated in figure 10, in which the similarity with the DAP (and most other SIMD array processors) is clearly visible. The array of processing elements, comprising a simple boolean processor and some local memory, is seen by the host machine simply as an extended region of memory. The host computer directs the connection machine to implement parallel portions of code, and in this respect it differs from the DAP or MP-1 which have an instruction processor built into the array unit. The CM-1 host broadcasts a sequence of instructions to the array micro-controller, which interprets the instructions and broadcasts an appropriate sequence of micro-instructions to the array of PEs, for each received host instruction. In the largest CM-2, this system is replicated four times, and a 4×4 crossbar switch (the Nexus) interconnects the arrays to four Front End Bus Interfaces.

The processor-memory cells, like those of the DAP, are too small and slow to perform meaningful computations individually. In CM-1, which was designed to address the AI/symbolic processing market, the main language supported was CM-Lisp. When running CM-Lisp processor-memory cells are linked together in data-dependent patterns called *active data structures.* Low-level operations on active data structures are then evaluated in parallel by the low-level boolean processors

acting in concert on their local segments of those structures, thus exploiting the parallelism and producing high processing rates.

Network structure

An important feature of a Connection Machine is its support for programmable links between PEs. In the DAP, when one processor communicates with its Northern neighbour all processors must communicate with their Northern neighbour, or not at all. This is because the DAP has a static square-mesh communication network, which only supports eight routing functions. Communication in CM-1 is significantly more powerful than this, since each group of sixteen processing elements shares a link into a packet-switched binary 12-cube network, as well as having individual connections to a DAP-like grid (known as the North-East-West-South, or NEWS grid). Essentially this means that all PEs can compute the address of a PE to which they want to send a message, and then use the 12-cube network to route the message in logarithmic time. A two-dimensional grid routes messages in $O(\sqrt{N})$ time, where N is the number of PEs. A full set of N^N permutations is supported by a dynamic binary k-cube network, where $k = \log N$, and in the case of CM-1 this produces a quoted worst-case bandwidth of $\approx 3.2 \times 10^7$ bits/s and a best-case bandwidth of $\approx 1.0 \times 10^9$ bits/s.

In the CM-2 a redesigned NEWS array uses an n-dimensional grid (n in the range 1 - 31, selectable by software), rather than being restricted to two dimensions. Furthermore, this network is implemented on top of the hypercube network, rather than using its own hardware.

Processing elements

The CM processing elements are implemented using custom VLSI components which contain a group of sixteen boolean processors, a local controller, and a message-routing interface to the cube network. This chip is fabricated in CMOS technology, and contains approximately 50,000 active devices. The processing element is a general-purpose single-bit processor with a private 4K×1-bit memory (64 Kbits to 1Mbits in the CM-2). Whereas in machines like the MPP and STARAN special architectural features, such as shift registers, were introduced to support integer multiplication, in the CM machines the processor cell is kept as simple as possible. It is also highly programmable.

Figure 11 shows the logical structure of a processing element. It consists of a single-bit arithmetic and logic unit, a file of sixteen single-bit registers (called flags) and connections from the local memory to the ALU and from the flags to the message router. The ALU is capable of realising all 256 possible boolean functions of three inputs (two memory operands and one flag), and it does this for both the value to be written back to memory and the value to be written back to one of the flag registers. This requires a total of sixteen bits of control input to the ALUs. In addition, the PE microcontroller must also specify various parameters for each

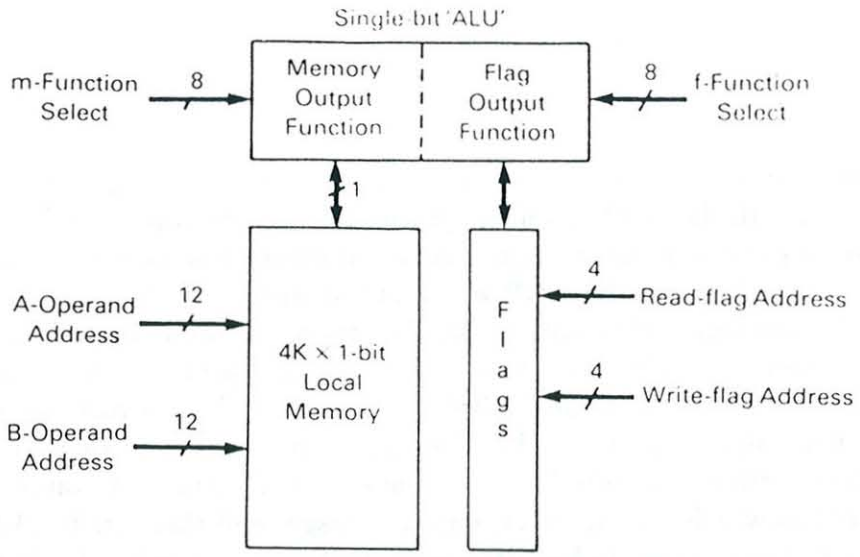

Figure 11: Structure of a CM-1 processing element

operation. These include:

1. A-address and B-address. The two memory operand bits are read from the A and B addresses and the memory output from the ALU is written back to the A address location.

2. Read and write flag addresses. These specify one input flag for the ALU, and one flag register to which the flag output from the ALU is written.

3. Condition flag address. Specifies which of the sixteen flags is to be used to determine whether a conditional operation will take place locally.

4. NEWS direction. Specifies in CM-1 which of the four 2D mesh permutations is selected for operations involving the NEWS grid.

The flag register file contains eight general purpose flags and eight special purpose flags. The special purpose flags provide links between the ALU (and hence memory) and the interconnection networks (that is, the NEWS grid and the router). For example, one read-only flag contains information written from the flag output of the neighbouring ALU in the direction specified by the NEWS direction controls.

The sixteen PEs in each processor chip can also be linked to form a chain of processors, as well as a square mesh, and this permits (rather slow) carry propagation across 16-bit slices of processing elements. So, whilst the design of the processing elements is not highly optimised for speed, the flexibility of the ALU and the flags together compensate somewhat, and the massive replication

of the PEs puts their combined power of about 10^9 integer 32-bit additions per second, well into the supercomputer category.

Although the CM-1 was aimed at the AI/symbolic processing market, TMC soon perceived the need to address the computationally intensive scientific market, and added a floating-point capability in the form of a floating-point unit based on the Weitek 3132 chip. One such unit is provided for each pair of processor chips (*i.e.* per 32 processors). A memory interface unit is incorporated to carry out the necessary transpositions between data stored in serial form in the individual processor memories to the parallel form needed for the Weitek chip, but in fact most floating-point users never use the 1-bit processors, so their data is stored in parallel form anyway.

The router

Each group of sixteen PEs shares a single message router, which itself constitutes one node in a binary k-cube network. In CM-1 $k = 12$, and so there can be a maximum of 4096 routers, with each router being connected directly to twelve other routers. Processors whose node addresses differ in only the ith bit-position have a direct connection in the ith dimension of the cube network. Since any two addresses can only differ in a maximum of twelve bit-positions (*i.e.* one is the inverse of the other) there can be at most twelve unique links forming a path between them. Hence, in a k-cube, no pair of nodes is separated by more than k links.

The routing algorithm is based loosely on the standard routing functions for binary k-cubes. Each message contains an address and a data field. The address comprises a *relative* router address field (12 bits), a PE address within a group of sixteen (4 bits), and an address in the memory of the destination processor where the message is to be deposited on delivery. Router addresses are *relative* because they specify the distance to be moved in order to get from the source to the destination processor. Hence, a 1 in bit position i indicates that the message must be routed through dimension i before it can arrive at its destination. Conversely, a 0 in bit position i means that no routing through dimension i is required. Therefore, when an address is all zeros the message must be at its destination. Also, when a message is routed through dimension i *towards* its destination, bit i must be *cleared*; and when routed away from its destination, bit i must be set.

Each parallel message delivery cycle consists of a sequence of repeated *petit cycles*. In a single petit cycle all messages which do not encounter routing delays (caused typically by contention in the network) will be delivered. These petit cycles are repeated until all messages within a 'burst' of messages have been delivered. Message bursts are associated with Lisp 'beta reduction' operations, for example. Each petit cycle consists of a sequence of twelve *dimension* cycles, and during the ith dimension cycle messages are routed (where required and where possible) through the ith dimension.

Network performance

The performance of the interprocessor communication network in the DAP is easy to analyse since all permutations are homogeneous (all processors communicate using the same routing function). However, in the Connection Machine routing functions are not homogeneous, and hence the distribution of message addresses can have a major effect on the net communication bandwidth.

It follows from the routing algorithm that the number of inter-nodal hops that a message must make is equal to the number of 1's in the destination address. Uniformly distributed message addresses will have a mean of $n/2$ 1's, where n is the number of bits in the address. Only one message can occupy each link during a single petit cycle, and during each dimension cycle only one twelfth of all communication links can be active. This is not a particularly efficient use of wire, the component which most severely limits the extensibility of cube-connected architectures. From the assumptions above we can predict the sustainable bandwidth of the network. A cube network with $N = 2^n$ nodes has $nN = n2^n$ wires in total. Since the number of 1's in all message addresses can only be changed to 0's at a rate of one per wire per petit cycle, in its steady-state the network cannot accept more than $n2^n$ injected address bits which are 1. This means it cannot accept more than twice this number of uniformly distributed messages. Thus there can only be $2N$ injected messages, or two injected messages per node, in each petit cycle.

The network does however contain some message buffering, and so at the beginning of a burst of messages the message-injection rate can be higher than two per node in each petit cycle. Higher levels of message injection can also be sustained when message addresses are localised. This must be considered when allocating elements of an active data structure to processing cells. Some operations naturally require local communications only. For example, steps in each beta-reduction operation specify near-neighbouring processors, and hence the number of 1's in each message address is just 1.

Another important consideration for message delivery in an SIMD system is that each burst of messages only terminates when *all* messages have been delivered. Where routing conflicts occur, additional petit cycles must be provided during the latter stages of the burst. Since all messages destined for the same node must be delivered *sequentially*, the maximum number of messages going to any one node during a burst of messages defines the number of additional petit cycles that will be required. TMC quote typical values of 2.5 Gbyte/s in a 64K processor CM-2 for the NEWS grid (this compares with 5.2 Gbyte/s in the DAP and 4.5 Gbytes/s in the MasPar 1104) and 259 Mbytes/s (without collisions) for the router (compared with a theoretical maximum of 7 Gbytes/).

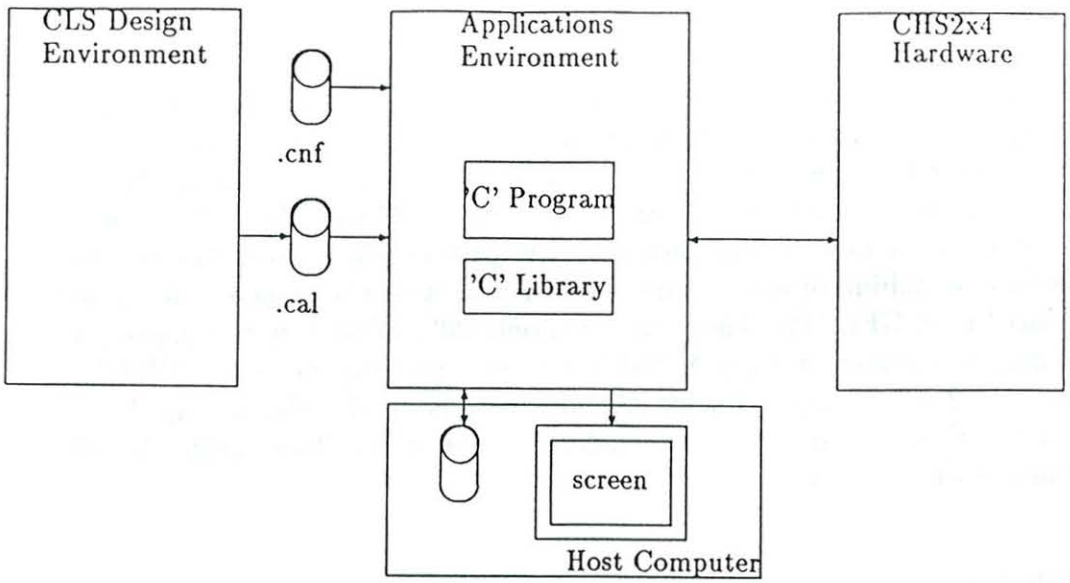

Figure 12: Software Architecture of the Algotronix CAL system

3 Custom Computing

American versions of the history of computing frequently refer to the ENIAC as the world's first electronic digital computer. In fact it was an application specific machine in the sense that its 'program' had to be hardwired. This is a perfectly respectable computational paradigm which has continued to co-exist with the von Neumann model. With the advances now being made in SRAM programmed varieties of field-programmable gate arrays, however, Gray [11] claims that a third, *Configurable Hardware* paradigm can now be supported. Support for this view also comes from Bell [6] who classifies machines built on this principle as *Custom Computers.*

A custom computer is programmed by mapping an algorithm directly into the hardware. In the current version of the Algotronix system (based on their Configurable Array Logic (CAL) chips) both the array itself and its control RAM are mapped into the memory address space of the PC into which the hardware is plugged. Software support involves two major areas, a design environment for creating circuits and an applications environment for running programs (figure 12. Although the current system is quite small (consisting of a 2×4 array of 1024 gate chips), the chip interconnection mechanism allows circuits to be spread across chip boundaries without the programmer being aware of them. Thus very large arrays could in principle be constructed, leading to the possibility of massive parallelism.

4 Conclusions

The term 'massively parallel computers ' can really only be applied to SIMD systems. They have a long history in niche markets but are begining to attack the traditional supercomputer market as levels of IC integration increase and the individual processors become more powerful. In 1989 TMC won the IEEE Gordon Bell Award both for raw performance and best price/performance. For the former a sesimic modelling program run on a 64K processor CM-2 gave a sustained performance of 5.6 GFLOPS. This represents only 20% of peak performance. By comparison, an 8-processor Cray Y-MP has a peak performance of 3 GFLOPS, giving the CM-2 a price/performance advantage of about 12. The next goal is to produce a TeraFLOPs machine, by the middle of the decade. It will probably take till the turn of the century.

References

[1] G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17:746–57, 1968.

[2] K.E. Batcher. STARAN Parallel Processor System Hardware. In *Proc. AFIPS-NCC*, volume 43, pages 405–410, 1974.

[3] K.E. Batcher. The FLIP Network in STARAN. In *Int. Conf. Parallel Proc.*, pages 65–71, 1976.

[4] K.E. Batcher. Multi-dimensional Access Memory in STARAN. *IEEE Transactions on Computers*, C-26:174–177, 1977.

[5] K.E. Batcher. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, C-29:836–840, 1980.

[6] C.G. Bell. The Future of High Performance Computers in Science and Engineering. *Communications of the ACM*, 32, 1989.

[7] T. Blank. The MasPar MP-1 Architecture. In *Proceedings IEEE Compcon*, February 1990.

[8] L.J. Clarke and G.V. Wilson. Tiny: An efficient routing harness for the inmos transputer. Technical Report EPCC-TR90-04, Edinburgh Parallel Computing Centre, 1990.

[9] B.A. Crane, M.J. Gilmartin, J.H. Huttenhoff, P.T. Rus, and R.R. Shively. PEPE Computer Architecture. In *IEEE Compcon*, pages 57–60, 1972.

[10] H. Falk. Reaching for the gigaflop. *IEEE Spectrum*, 13(10):65–70, 1976.

[11] J.P. Gray and T.A. Kean. Configurable Hardware: A New Paradigm for Computation. In *Proceddings Decennial Caltech Conference on VLSI*, Pasadena, Ca, USA, 1989.

[12] J. Gregory and R.C. McReynolds. The SOLOMON computer. *IEEE Transactions on Electronic Computers*, EC-12:774–81, 1963.

[13] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[14] R.N. Ibbett and N.P. Topham. *Architecture of High Performance Computers Vol II.* Macmillan, 1989.

[15] D.J. Kuck and R.A. Stokes. The Burroughs Scientific Processor (BSP). *IEEE Transactions on Computers*, C-31(5):363–376, 1982.

[16] S.F. Reddaway. DAP – a distributed array processor. In *1st Int. Symp. Comp. Architecture*, pages 61–65, 1973.

[17] S.F. Reddaway. The DAP Approach. In C. R. Jesshope and R. W. Hockney, editors, *Infotech State of the Art Report: Supercomputers*, volume Vol 2, pages 311–329, Maidenhead, England, 1979. Infotech Intl Ltd.

[18] C.L. Seitz. Experiments with VLSI Ensemble Machines. *Journal of VLSI & Computer Systems*, 1(4):311–334, 1983.

[19] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, 1985.

[20] D.L. Slotnick, W.C. Borck, and R.C. McReynolds. The SOLOMON computer. In *AFIPS Conf. Proc.*, volume 22, pages 97–107, 1962.

[21] A. Trew and G.V. Wilson. *Past, Present, Parallel: A Survey of Available Parallel Computing Systems.* Springer-Verlag, London.

[22] S.H. Unger. A computer oriented towards spatial problems. In *Proc. Inst. Radio Eng.*, volume 46, pages 1744–50, 1958.

[23] C.R. Vick and J.A. Cornell. PEPE architecture — present and future. In *AFIPS Conf. Proc*, volume 47, pages 981–1002, 1978.

DISCUSSION

Rapporteur: M. J. Elphick

In response to Professor Randell's query about the relative use of multi-Transputer machines for specific applications and for general workloads, Professor Ibbett pointed out that the Edinburgh Meiko system could be split into as many as 128 "domains"; much development work could be done on single-Transputer domains, while large physics applications might use the whole system as a single domain. The Intel i860 version of such systems could be used to imitate the "Cray style" (but with some effort, as the compilers were not really good enough yet); most physics and engineering applications can use vector operations effectively, but Crays are expensive!

Professor Randell then referred to the early versions of the DAP architecture, when there were arguments about the most effective use of developing technology -- should one make the individual processors more powerful, or increase the array size? The latter option had been favoured then -- had this view changed now? The speaker observed that while the processor elements were still 1-bit, additional 8-bit co-processors had been added. Several people commented (not always favourably) on the attempts to extend sequential languages to cope with parallel computation; Professor Levy felt that some aspects (e.g. the 'slicewise' storage patterns used by some extended FORTRAN's) were an admission of failure, being in effect just elaborate strategies for feeding data to multiple floating-point units.

Turning to the question of abstraction, Professor Randell asked what kinds of abstraction are affordable; Professor Levy had found that for very high performance, very few abstractions could be afforded. This effect of the low-level details of an implementation was similar to the importance of the sequential component of a parallel program (expressed by Amdahl's law). The speaker's response was that while we understood sequential machines (with their need for large fast memories being satisfied by suitably managed memory hierarchies) parallelism introduced another dimension of complexity, where we don't understand the problem's spatial locality well enough. In reply to comments by Professor Shepherd, he said that there seemed to be two classes of users: those who don't want to know about these problems, and those (principally scientists and engineers) who are willing to invest the time and effort in organising their computations to take advantage of the speedups possible. It might be that we need two classes of languages for these two approaches to parallelism. Professor Whitfield felt that some people in the latter class might be more interested in their specific application (and the possibility of Nobel prizes?) than in in exploring more general methods for a wider class of problems.

Professor Levy commented that a lot of commercial interest was solely in achieving improved price:performance ratios, and in coping with "code museums", while Professor Lee said that there was much resistance from engineers in particular to such new architectures; however, in many cases inspection of "dusty decks" had revealed that much effort had been expended originally on what were now inappropriate trade-offs (e.g. in economising on memory usage), and that considerable speedups were possible by eliminating this trade-off.

Finally, Mr. Kerr said that he had intended to make the following points after Professor Levy's next talk but the foregoing discussion had already touched on some of them; one is frivolous, one possibly naive and a third intended to be neither but which may turn out to be both!

(1) "Any realistic person has to attach some weight to Gerhard Goos's comments yesterday on the ever-continuing dominance of FORTRAN in the presence of superior programming systems. I offer a glimmer of hope. Over the past 30 years, I have observed with wry amusement the gradual evolution of FORTRAN into an approximation to ALGOL. If this trend continues, we can expect FORTRAN to turn into some sort of SIMULA during the next 20 years, giving it some object-oriented properties which might enable the past to co-exist with the present and future! As with humour, frivolity often has a thread of truth."

(2) "Regarding yesterday's comments on the ability to apply object-oriented principles in the diametrically opposite scenarios of shared and distributed memory architectures, I would hypothesise that OOP in the parallel context is no more tied to the shared/distributed issue than the virtual memory concept is to the physical characteristics of memory boards. Both present an illusion which is accomplished with ingenuity on ill-matched hardware."

(3) "As a comparative novice in the parallel field, I am concerned at a practical level about the diversity of competing parallel architectures and the deep divisions between their eminent and expert designers. I agree with Iann Barron's comments in his after-dinner speech that the central issues revolve around thinking parallel in the first place but, more than that, we need the ability to express these thoughts in a notation which preserves the parallel perspectives.

In my view, object-orientation is currently the best vehicle we have for this. In saying this, I am drawing on a broader view of object-orientation than that held by most of its proponents. Almost 25 years ago were laid down three fundamental concepts of object-orientation: encapsulation, inheritance and autonomous activity. Of these, the last has been largely ignored in modern incarnations of the original inspiration (SIMULA). However, this notion is the one which addresses the description of inherently independent computational activity which makes parallelism possible. (This is related to the threads notion already mentioned in this seminar.) Application of this concept will result in parallel systems descriptions whose structure are determined, a priori, by the parallelism inherent in them. Having said that, we can expect to observe certain relationships and regularities within the sets of objects identified which may well suggest that their execution will be best served by one or other of the diverse architectures available.

As Iann Barron observed last night, our thinking has been dominated and conditioned by the strait-jacketing influence of serial processing. OOP is our current best hope of avoiding falling into the same trap in the context of some other stylised form of processing. "