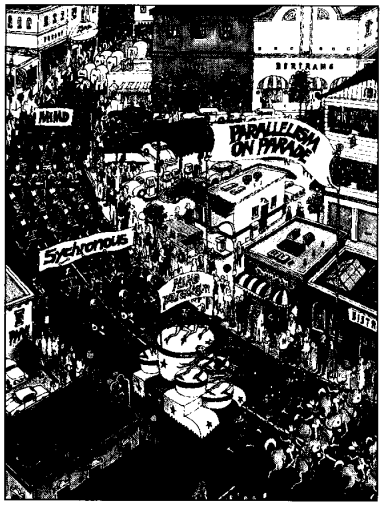
SURVEY & TUTORIAL SERIES

A Survey of Parallel Computer Architectures



Ralph Duncan, Control Data Corporation

This decade has witnessed the introduction of a wide variety of new computer architectures for parallel processing that complement and extend the major approaches to parallel computing developed in the 1960s and 1970s. The recent proliferation of parallel processing technologies has included new parallel hardware architectures (systolic and hypercube), interconnection technologies (multistage switching topologies), and programming paradigms (applicative programming). The sheer diversity of the field poses a substantial obstacle to the nonspecialist who wishes to comprehend what kinds of parallel architectures exist and how their relationship to one another defines an orderly schema.

This discussion attempts to place recent architectural innovations in the broader context of parallel architecture development by surveying the fundamentals of both newer and more established parallel computer architectures and by placing these architectural alternatives in a coherent framework. The survey's primary emphasis concerns architectural constructs rather than specific parallel machines.

Terminology and taxonomy

Problems. Diverse definitions have been proposed for parallel architectures.

The diversity of parallel computer architectures can bewilder the nonspecialist. This tutorial reviews alternative approaches to parallel processing within the framework of a high-level taxonomy.

The difficulty in precisely defining the term is intertwined with the problem of specifying a parallel architecture taxonomy. A central problem for specifying a definition and consequent taxonomy for modern parallel architectures is to satisfy the following set of imperatives:

- Exclude architectures incorporating only low-level parallel mechanisms that have become commonplace features of modern computers.
- Maintain elements of Flynn's useful taxonomy[1] based on instruction and data streams.
- Include pipelined vector processors and other architectures that intuitively seem to merit inclusion as parallel architectures, but which are difficult to gracefully accommodate within Flynn's scheme.

We will examine each of these imperatives as we seek a definition that satisfies all of them and provides the basis for a reasonable taxonomy.

Low-level parallelism. There are two reasons to exclude machines that employ only low-level parallel mechanisms from the set of parallel architectures. First, failure to adopt a more rigorous standard might make the majority of modern computers "parallel architectures," negating the term's usefulness. Second, architectures having only the features listed below do not offer an explicit, coherent framework for developing high-level parallel solutions:

- *Instruction pipelining* — the decomposition of instruction execution into a linear series of autonomous stages, allowing each stage to simultaneously perform a portion of the execution process (such as decode, calculate effective address, fetch operand, execute, and store).
- *Multiple CPU functional units* — providing independent functional units for arithmetic and Boolean

0018-9162/90/0200-0005$01.00 © 1990 IEEE

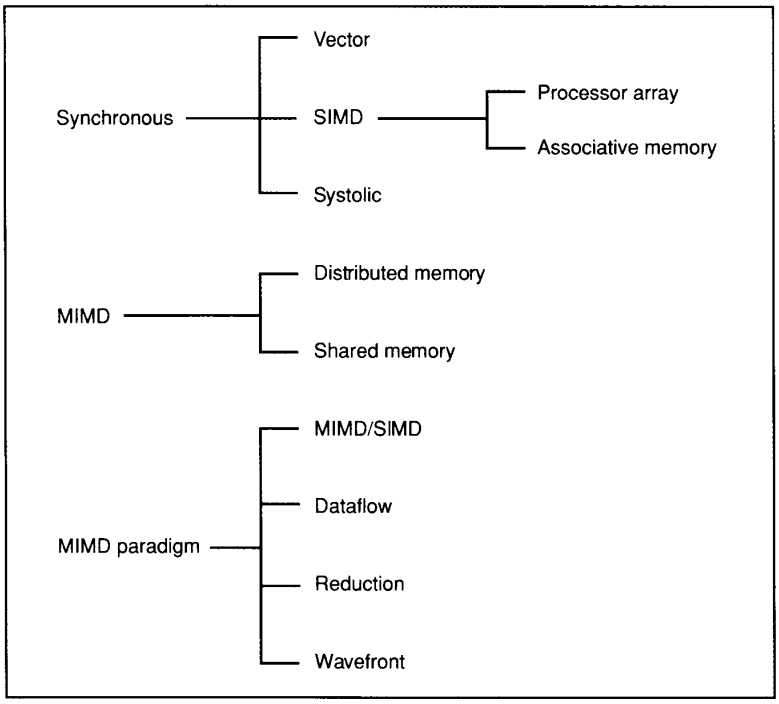


Figure 1. High-level taxonomy of parallel computer architectures.

operations that execute concurrently.
- *Separate CPU and I/O processors* — freeing the CPU from I/O control responsibilities by using dedicated I/O processors; solutions range from relatively simple I/O controllers to complex peripheral processing units.

Although these features contribute significantly to performance engineering, their presence does not make a computer a parallel architecture.

Flynn's taxonomy. Flynn's taxonomy classifies architectures on the presence of single or multiple streams of instructions and data. This yields the four categories below:

SISD (single instruction, single data stream) — defines serial computers.
MISD (multiple instruction, single data stream) — would involve multiple processors applying different instructions to a single datum; this hypothetical possibility is generally deemed impractical.
SIMD (single instruction, multiple data streams) — involves multiple processors simultaneously executing the same instruction on different data (this definition is discussed further prior to examining array processors below).
MIMD (multiple instruction, multiple data streams) — involves multiple processors autonomously executing diverse instructions on diverse data.

Although these distinctions provide a useful shorthand for characterizing architectures, they are insufficient for classifying various modern computers. For example, pipelined vector processors merit inclusion as parallel architectures, since they exhibit substantial concurrent arithmetic execution and can manipulate hundreds of vector elements in parallel. However, they are difficult to accommodate within Flynn's taxonomy, because they lack processors executing the same instruction in SIMD lockstep and lack the asynchronous autonomy of the MIMD category.

Definition and taxonomy. A first step to providing a satisfactory taxonomy is to articulate a definition of parallel architecture. The definition should include appropriate computers that the Flynn schema cannot handle and exclude architectures incorporating only low-level parallelism. Therefore, a *parallel architecture* provides an explicit, high-level framework for the development of parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution.

Figure 1 shows a taxonomy based on the imperatives discussed earlier and the proposed definition. This informal taxonomy uses high-level categories to delineate the principal approaches to parallel computer architectures and to show that these approaches define a coherent spectrum of architectural alternatives. Definitions for each category are provided below.

This taxonomy is not intended to supplant efforts to construct more fully articulated taxonomies. Such taxonomies provide comprehensive subcategories to reflect permutations of architectural characteristics and to cover lower level features. The "Further reading" section at the end references several thoughtful taxonomic studies that address these goals.

Synchronous architectures

Synchronous parallel architectures coordinate concurrent operations in lockstep through global clocks, central control units, or vector unit controllers.

Pipelined vector processors. The first vector processor architectures were developed in the late 1960s and early 1970s[2,3] to directly support massive vector and matrix calculations. Vector processors[4] are characterized by multiple, pipelined functional units, which implement arithmetic and Boolean operations for both vectors and scalars and which can operate concurrently. Such architectures provide parallel vector processing by sequentially streaming vector elements through a functional unit pipeline and by streaming the output results of one unit into the pipeline of another as input (a process known as "chaining").

A representative architecture might have a vector addition unit consisting of six pipeline stages (see Figure 2). If each pipeline stage in the hypothetical architecture shown in the figure has a cycle time of 20 nanoseconds, then 120 ns elapse from the time operands $a1$ and $b1$ enter stage 1 until result $c1$ is available. When the pipeline is filled, however, a result is available every 20 ns. Thus, start-up overhead of pipelined vector units has significant performance implications. In the case of the register-to-register architecture depicted, special high-speed vector registers hold operands and results. Efficient performance for such architectures (for example,

the Cray-1 and Fujitsu VP-200) is obtained when vector operand lengths are multiples of the vector register size. Memory-to-memory architectures (such as the Control Data Cyber 205 and Texas Instruments Advanced Scientific Computer) use special memory buffers instead of vector registers.

Recent vector processing supercomputers (such as the Cray X-MP/4 and ETA-10) unite four to 10 vector processors through a large shared memory. Since such architectures can support task-level parallelism, they could arguably be termed MIMD architectures, although vector processing capabilities are the fundamental aspect of their design.

SIMD architectures. SIMD architectures (see Figure 3) typically employ a central control unit, multiple processors, and an interconnection network (IN) for either processor-to-processor or processor-to-memory communications. The control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The interconnection network allows instruction results calculated at one processor to be communicated to another processor for use as operands in a subsequent instruction. Individual processors may be allowed to disable the current instruction.

Processor array architectures. Processor arrays[5] structured for numerical SIMD execution have often been employed for large-scale scientific calculations, such as image processing and nuclear energy modeling. Processor arrays developed in the late 1960s (such as the Illiac-IV) and more recent successors (such as the Burroughs Scientific Processor) utilize processors that accommodate word-sized operands. Operands are usually floating-point (or complex) values and typically range in size from 32 to 64 bits. Various IN schemes have been used to provide processor-to-processor or processor-to-memory communications, with mesh and crossbar approaches being among the most popular.

One variant of processor array architectures involves using a large number of one-bit processors. In bit-plane architectures, the array of processors is arranged in a symmetrical grid (such as 64×64) and associated with multiple "planes" of memory bits that correspond to the dimensions of the processor grid (see Figure 4). Processor n (P_n), situated in the processor grid at location (x, y), operates on the memory bits at location (x, y) in all the

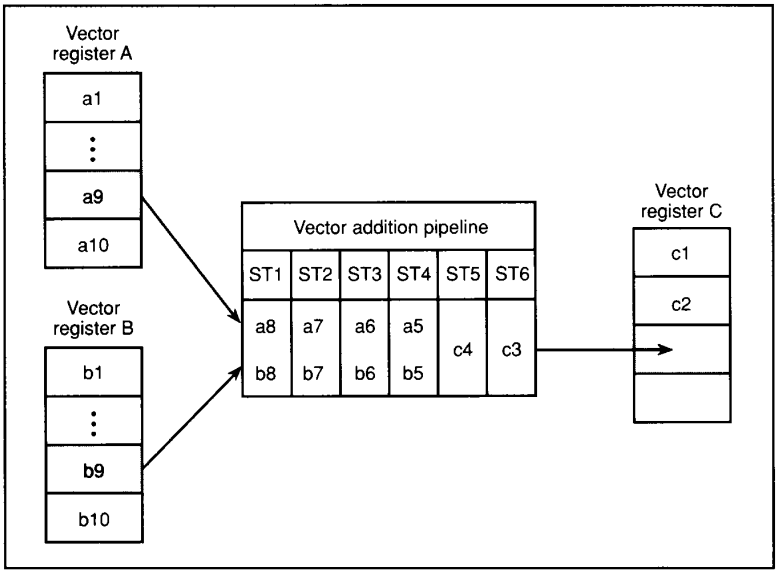


Figure 2. Register-to-register vector architecture operation.

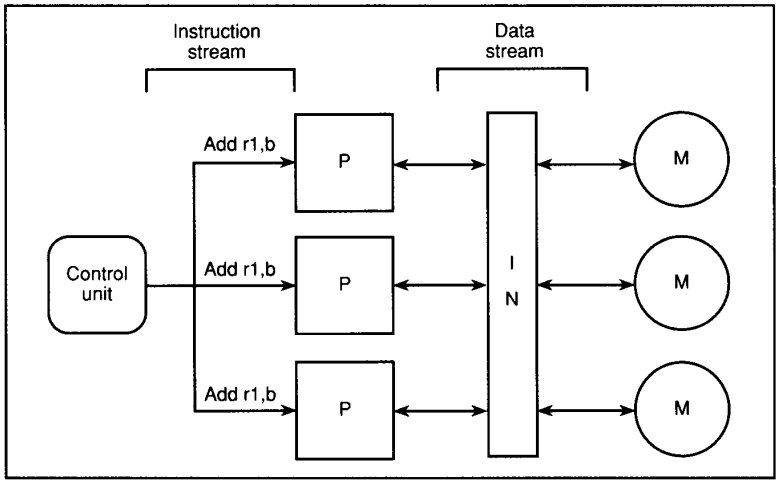


Figure 3. SIMD execution.

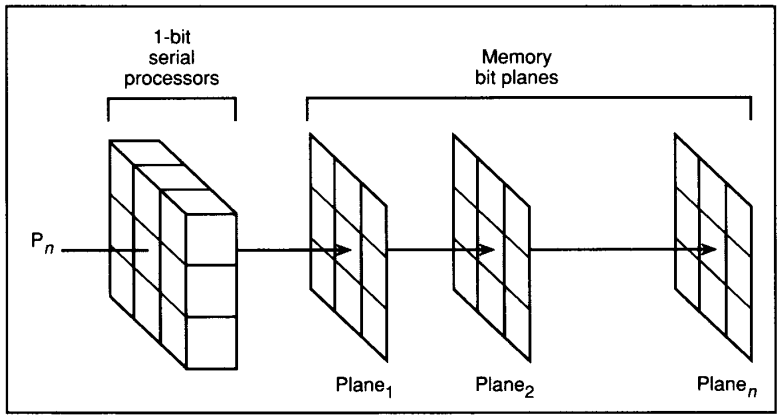


Figure 4. Bit-plane array processing.

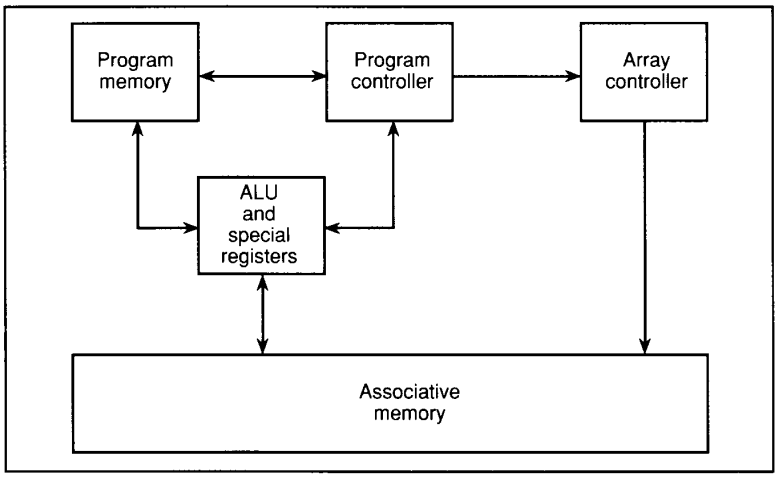


Figure 5. Associative memory processing organization.

associated memory planes. Usually, operations are provided to copy, mask, and perform arithmetic operations on entire memory planes, as well as on columns and rows within a plane.

Loral's Massively Parallel Processor[6] and ICL's Distributed Array Processor exemplify this kind of architecture, which is often used for image processing applications by mapping pixels to the memory's planar structure. Thinking Machines' Connection Machine organizes as many as 65,536 one-bit processors as sets of four-processor meshes united in a hypercube topology.

Associative memory processor architectures. Computers built around an associative memory[7] constitute a distinctive type of SIMD architecture that uses special comparison logic to access stored data in parallel according to its contents. Research in constructing associative memories began in the late 1950s with the obvious goal of being able to search memory in parallel for data that matched some specified datum. "Modern" associative memory processors developed in the early 1970s (for example, Bell Laboratories' Parallel Element Processing Ensemble, or PEPE) and recent architectures (for example, Loral's Associative Processor, or Aspro) have naturally been geared to database-oriented applications, such as tracking and surveillance.

Figure 5 shows the characteristic functional units of an associative memory processor. A program controller (serial computer) reads and executes instructions, invoking a specialized array controller when associative memory instructions are encountered. Special registers enable the program controller and associative memory to share data.

Most current associative memory processors use a bit-serial organization, which involves concurrent operations on a single bit-slice (bit-column) of all the words in the associative memory. Each associative memory word, which usually has a very large number of bits (for example, 32,768), is associated with special registers and comparison logic that functionally constitute a processor. Hence, an associative processor with 4,096 words effectively has 4,096 processing elements.

Figure 6 depicts a row-oriented comparison operation for a generic bit-serial architecture. A portion of the comparison register contains the value to be matched. All of the associative processing elements start at a specified memory column and compare the contents of four consecutive bits in their row against the comparison register contents, setting a bit in the *A* register to indicate whether or not their row contains a match.

In Figure 7 a logical OR operation is performed on a bit-column and the bit-vector in register *A*, with register *B* receiving the results. A zero in the mask register indicates that the associated word is not to be included in the current operation.

Systolic architectures. In the early 1980s H.T. Kung of Carnegie Mellon University proposed systolic architectures to solve the problems of special-purpose systems that must often balance intensive computations with demanding I/O bandwidths.[8] *Systolic architectures* (systolic arrays) are pipelined multiprocessors in which data is pulsed in rhythmic fashion

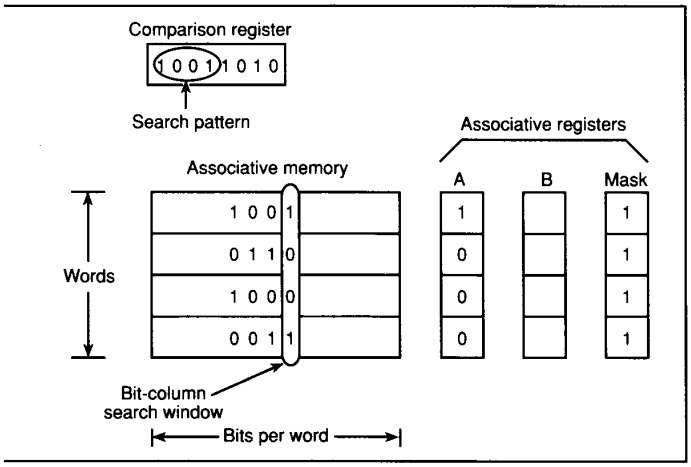


Figure 6. Associative memory comparison operation.

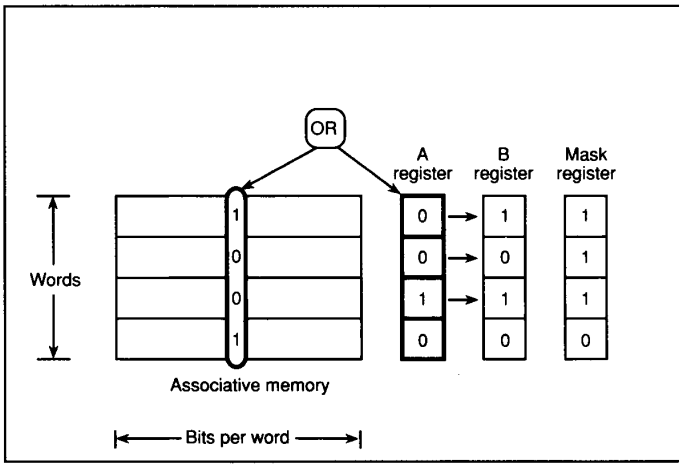


Figure 7. Associative memory logical OR operation.

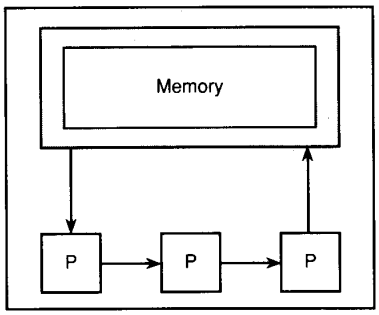


Figure 8. Systolic flow of data from and to memory.

from memory and through a network of processors before returning to memory (see Figure 8). A global clock and explicit timing delays synchronize this pipelined data flow, which consists of operands obtained from memory and partial results to be used by each processor. Modular processors united by regular, local interconnections provide basic building blocks for a variety of special-purpose systems. During each time interval, these processors execute a short, invariant sequence of instructions.

Systolic arrays address the performance requirements of special-purpose systems by achieving significant parallel computation and by avoiding I/O and memory bandwidth bottlenecks. A high degree of parallelism is obtained by pipelining data through multiple processors, typically in two-dimensional fashion. Systolic architectures maximize the computations performed on a datum once it has been obtained from memory or an external device. Hence, once a datum enters the systolic array, it is passed to any processor that needs it, without an intervening store to memory. Only processors at the topological boundaries of the array perform I/O to and from memory.

Figure 9a-e shows how a simple systolic array could calculate the outer product of two matrices,

$$\mathbf{A} = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \text{ and } \mathbf{B} = \begin{vmatrix} e & f \\ g & h \end{vmatrix}$$

The zero inputs shown moving through the array are used for synchronization. Each processor begins with an accumulator set to zero and, during each cycle, adds the product of its two inputs to the accumulator. After five cycles the matrix product is complete.

A growing number of special-purpose systems use systolic organization for algorithm-specific architectures, particularly for signal processing. In addition, programmable (reconfigurable) systolic architectures (such as Carnegie Mellon's Warp and Saxpy's Matrix-1) have been constructed that are not limited to implementing a single algorithm. Although systolic concepts were originally proposed for VLSI-based systems to be implemented at the chip level, systolic architectures have been implemented at a variety of physical levels.

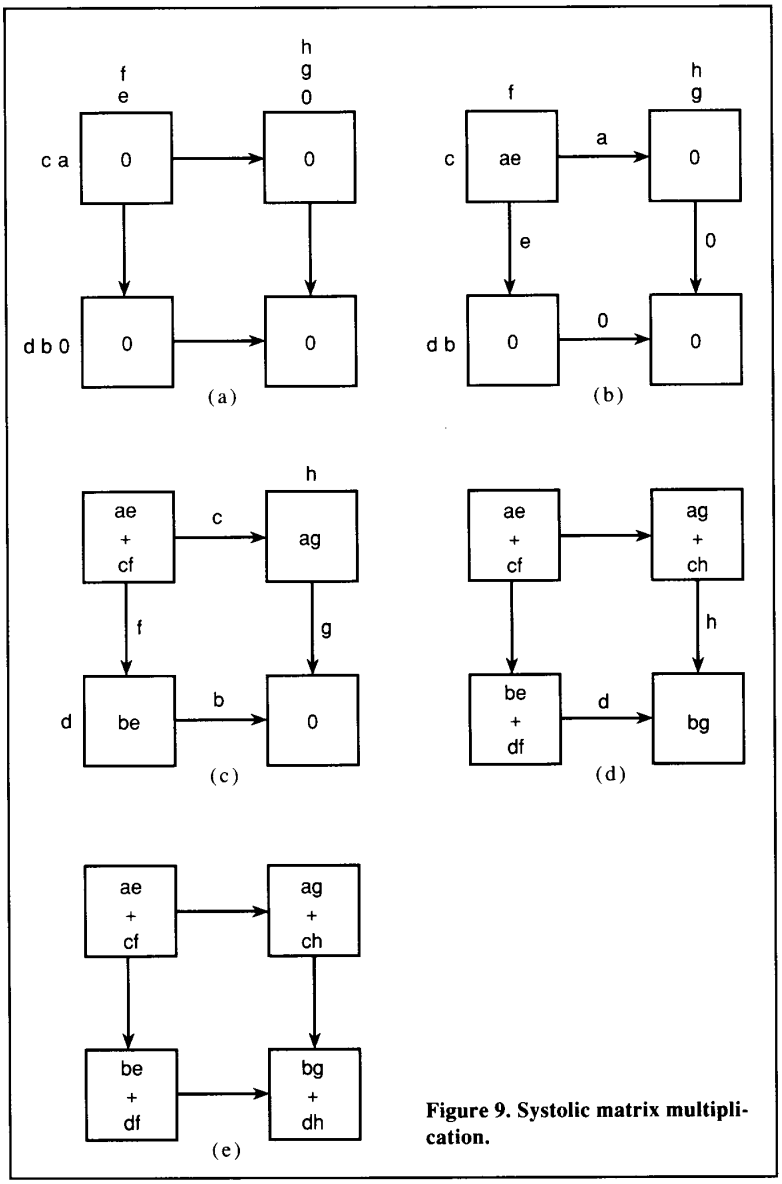


Figure 9. Systolic matrix multiplication.

MIMD architectures

MIMD architectures employ multiple processors that can execute independent instruction streams, using local data. Thus, MIMD computers support parallel solutions that require processors to operate in a largely autonomous manner. Although software processes executing on MIMD architectures are synchronized by passing messages through an interconnection network or by accessing data in shared memory units, MIMD architectures are asynchronous computers, characterized by decentralized hardware control.

The impetus for developing MIMD architectures can be ascribed to several interrelated factors. MIMD computers support higher level parallelism (subprogram and task levels) that can be exploited

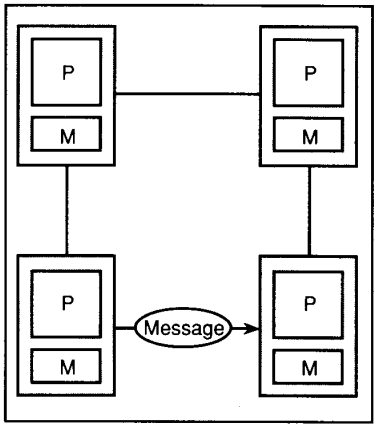


Figure 10. MIMD distributed memory architecture structure.

by "divide and conquer" algorithms organized as largely independent subcalculations (for example, searching and sorting). MIMD architectures may provide an alternative to depending on further implementation refinements in pipelined vector computers to provide the significant performance increases needed to make some scientific applications tractable (such as three-dimensional fluid modeling). Finally, the cost-effectiveness of n-processor systems over n single-processor systems encourages MIMD experimentation.

Distributed memory architectures. Distributed memory architectures (Figure 10) connect processing nodes (consisting of an autonomous processor and its local memory) with a processor-to-processor interconnection network. Nodes share data by explicitly passing messages through the interconnection network, since there is no shared memory. A product of 1980s research, these architectures have principally been constructed in an effort to provide a multiprocessor architecture that will "scale" (accommodate a significant increase in processors) and will satisfy the performance requirements of large scientific applications characterized by local data references.

Various interconnection network topologies have been proposed to support architectural expandability and provide efficient performance for parallel programs with differing interprocessor communication patterns. Figure 11a-e depicts the topologies discussed below.

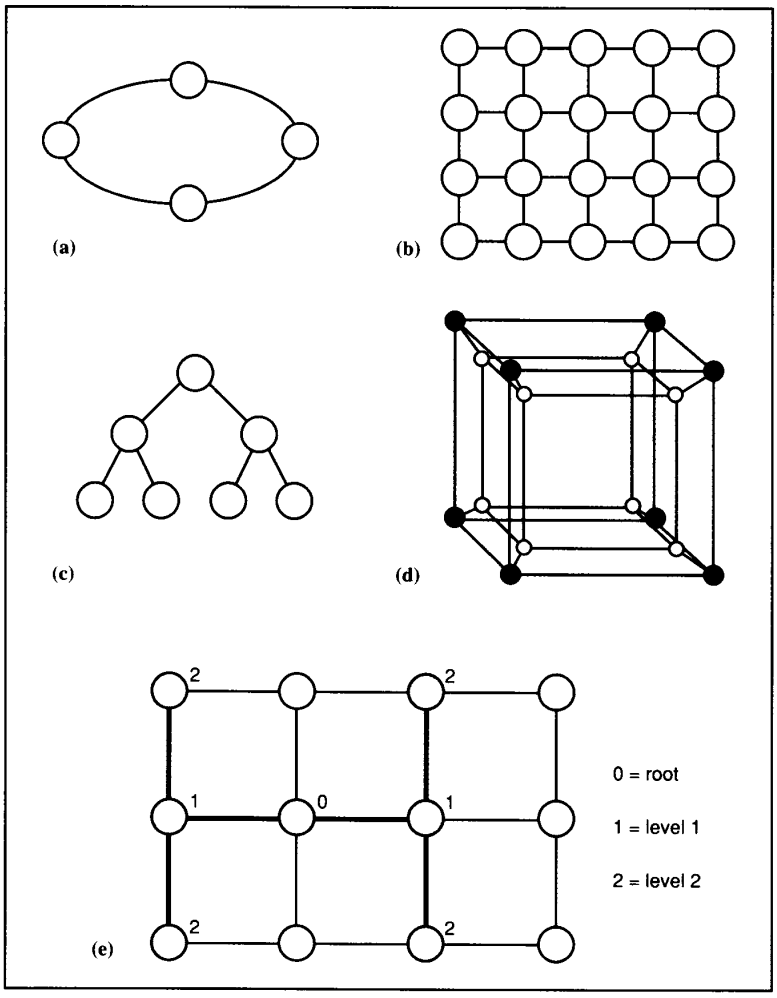


Figure 11. MIMD interconnection network topologies: (a) ring; (b) mesh; (c) tree; (d) hypercube; (e) tree mapped to a reconfigurable mesh.

Ring topology architectures. The communication diameter ($N/2$) of ring topology architectures can be reduced by adding chordal connections. Using chordal connections or multiple rings can increase a ring-based architecture's fault tolerance. Typically, fixed-size message packets are used that include a node destination field. Ring topologies are most appropriate for a small number of processors executing algorithms not dominated by data communications.

Mesh topology architectures. A two-dimensional mesh, or lattice, topology has n^2 nodes, each connected to its four immediate neighbors. Wraparound connections at the edges are sometimes provided to reduce the communication diameter from $2(n-1)$ to 2 * (Integer-part of $n/2$). Communications may be augmented by providing additional diagonal links or by using buses to connect nodes by rows and columns. The topological correspondence between meshes and matrix-oriented algorithms encourages mesh-based architecture research.

Tree topology architectures. Tree topology architectures, such as Columbia University's DADO2 and Non-Von, have been constructed to support divide-and-conquer algorithms for searching and sorting, image processing algorithms, and dataflow and reduction programming paradigms. Although a variety of tree-structured topologies have been suggested, complete binary trees are the most analyzed variant.

Several strategies have been employed to reduce the communication diameter of tree topologies ($2(n-1)$ for a complete binary tree with n levels and 2^n-1 pro-

cessors). Example solutions include adding additional interconnection network pathways to unite all nodes at the same tree level.

Hypercube topology architectures. A Boolean n-cube or "hypercube" topology uses $N = 2^n$ processors arranged in an n-dimensional cube, where each node has $n=\log_2 N$ bidirectional links to adjacent nodes. Individual nodes are uniquely identified by n-bit numeric values ranging from 0 to $N-1$ and assigned in a manner that ensures adjacent nodes' values differ by a single bit. The communication diameter of such a hypercube topology architecture is $n=\log_2 N$.

Hypercube architecture research has been strongly influenced by the desire to develop a "scalable" architecture that supports the performance requirements of 3D scientific applications. Extant hypercube architectures include the Cosmic Cube, Ametek Series 2010, Intel Personal Supercomputer, and Ncube/10.

Reconfigurable topology architectures. Although distributed memory architectures possess an underlying physical topology, reconfigurable topology architectures provide programmable switches that allow users to select a logical topology matching application communication patterns. The functional reconfigurability available in research prototypes ranges from specifying different topologies (such as Lawrence Snyder's Configurable Highly Parallel Computer, or Chip) to partitioning a base topology into multiple interconnection topologies of the same type (such as Howard J. Siegel's Partitionable SIMD/MIMD System, or Pasm). A significant motivation for constructing reconfigurable topology architectures is that a single architecture can act as many special-purpose architectures that efficiently support the communications patterns of particular algorithms or algorithm steps.

Shared-memory architectures. Shared memory architectures accomplish interprocessor coordination by providing a global, shared memory that each processor can address. Commercial shared-memory architectures, such as Flexible Corporation's Flex/32 and Encore Computer's Multimax, were introduced during the 1980s. These architectures involve multiple general-purpose processors sharing memory, rather than a CPU and peripheral I/O processors. Shared memory computers do not have some of the problems encountered by message-passing architectures, such as message sending latency as data is queued and forwarded by intermediate nodes. However, other problems, such as data access synchronization and cache coherency, must be solved.

Coordinating processors with shared variables requires atomic synchronizing mechanisms to prevent one process from accessing a datum before another finishes updating it. These mechanisms provide an atomic operation that subjects a "key" to a comparison test before allowing either the key or associated data to be updated. The "test-and-set" mechanism, for example, is an atomic operation for testing the key's value and, if the test result is true, updating the key value.

Typically, each processor in a shared memory architecture also has a local memory used as a cache. Multiple copies of the same shared memory data, therefore, may exist in various processors' caches at a given time. Maintaining a consistent version of such data is the cache coherency problem, which concerns providing new versions of the cached data to each involved processor whenever a processor updates its copy. Although systems with a small number of processors can use hardware "snooping" mechanisms to determine when shared memory data has been updated, larger systems usually rely on software solutions to minimize performance impact.

Figure 12a-c illustrates some major alternatives for connecting multiple processors to shared memory (outlined below).

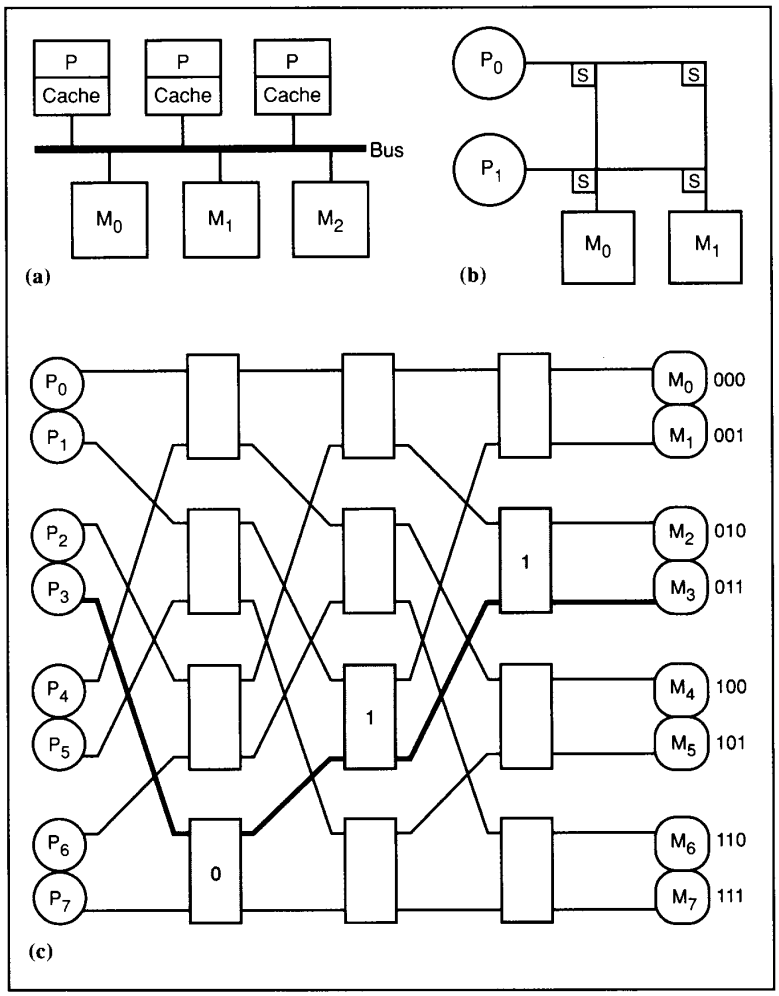


Figure 12. MIMD shared-memory interconnection schemes: (a) bus interconnection; (b) 2×2 crossbar; (c) 8×8 omega MIN routing a P_3 request to M_3.

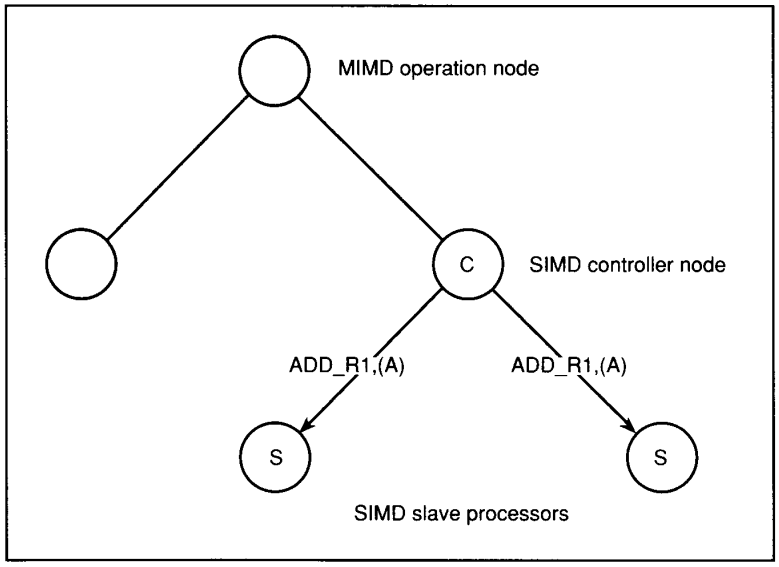


Figure 13. MIMD/SIMD operation.

Bus interconnections. Time-shared buses (Figure 12a) offer a fairly simple way to give multiple processors access to a shared memory. A single, time-shared bus effectively accommodates a moderate number of processors (from four to 20), since only one processor accesses the bus at a given time. Some bus-based architectures, such as the experimental Cm* architecture, employ two kinds of buses — a local bus linking a cluster of processors and a higher level system bus linking dedicated service processors associated with each cluster.

Crossbar interconnections. Crossbar interconnection technology uses a crossbar switch of n^2 crosspoints to connect n processors to n memories (see Figure 12b). Processors may contend for access to a memory location, but crossbars prevent contention for communication links by providing a dedicated pathway between each possible processor/memory pairing.

Power, pinout, and size considerations have limited crossbar architectures to a small number of processors (from four to 16). The Alliant FX/8 is a commercial architecture that uses a crossbar scheme to connect processors and cache memories.

Multistage interconnection networks. Multistage interconnection networks (MINs)[9] strike a compromise between the price/performance alternatives offered by crossbars and buses. An $N \times N$ MIN connects N processors to N memories by deploying multiple "stages" or banks of switches in the interconnection network pathway.

When N is a power of 2, one approach is to employ $\log_2 N$ stages of $N/2$ switches, using 2×2 switches. A processor making a memory access request specifies the desired destination (and pathway) by issuing a bit-value that contains a control bit for each stage. The switch at stage i examines the ith bit to determine whether the input (request) is to be connected to the upper or lower output.

Figure 12c shows an omega network connecting eight processors and memories, where a control bit equal to zero in dicates a connection to the upper output. Expandability is a significant feature of such a MIN, since its communication diameter is proportional to $\log_2 N$. The BBN (Bolt, Beranek, and Newman) Butterfly, for example, can be configured with as many as 256 processors.

MIMD-based architectural paradigms

MIMD/SIMD hybrids, dataflow architectures, reduction machines, and wavefront arrays all pose a similar difficulty for an orderly taxonomy of parallel architectures. Each of these architectural types is predicated on MIMD principles of asynchronous operation and concurrent manipulation of multiple instruction and data streams. However, each of these architectures is also based on a distinctive organizing principle as fundamental to its overall design as MIMD characteristics. These architectures, therefore, are described under the category "MIMD-based architectural paradigms" to highlight their distinctive foundations as well as the MIMD characteristics they have in common.

MIMD/SIMD architectures. A variety of experimental hybrid architectures constructed during the 1980s allow selected portions of a MIMD architecture to be controlled in SIMD fashion (for example, DADO, Non-Von, Pasm, and Texas Reconfigurable Array Computer, or TRAC).[14] The implementation mechanisms explored for reconfiguring architectures and controlling SIMD execution are quite diverse. Using a tree-structured, message-passing computer[10] as the base architecture for a MIMD/SIMD architecture helps illustrate the general concept.

The master/slaves relation of a SIMD architecture's controller and processors can be mapped onto the node/descendents relation of a subtree (see Figure 13). When the root processor node of a subtree operates as a SIMD controller, it transmits instructions to descendent nodes that execute the instructions on local memory data.

The flexibility of MIMD/SIMD architectures obviously makes them attractive candidates for further research. Specific incentives for recent development efforts include supporting parallel image processing and expert system applications.

Dataflow architectures. The fundamental feature of dataflow architectures is an execution paradigm in which instructions are enabled for execution as soon as all of their operands become available. Thus, the sequence of executed instructions is based on data dependencies, allowing dataflow architectures to exploit concurrency at the task, routine, and instruction levels. A major incentive for dataflow architecture research, which dates from J.B. Dennis' pioneering work in the mid-1970s, is to explore new computational models and languages that can be effectively exploited to achieve large-scale parallelism.

Dataflow architectures execute dataflow graphs, such as the program fragment depicted in Figure 14. We can think of

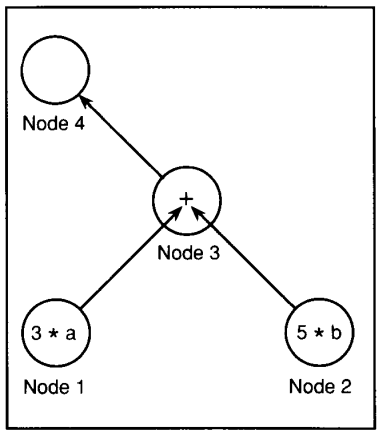


Figure 14. Dataflow graph—program fragment.

graph nodes as representing asynchronous tasks, although they are often single instructions. Graph arcs represent communications paths that carry execution results needed as operands in subsequent instructions.

Some of the diverse mechanisms used to implement dataflow computers (such as the Manchester Data Flow Computer, MIT Tagged Token Data Flow architecture, and Toulouse LAU System)[11] are outlined below. Static implementations load all graph nodes into memory during initialization and allow only one instance of a node to be executed at a time; dynamic architectures allow the creation of node instances at runtime and multiple instances of a node to be executed concurrently.

Some architectures directly store "tokens" containing instruction results into a template for the instruction that will use them as operands. Other architectures use token-matching schemes, in which a matching unit stores tokens and tries to match them with instructions. When a complete set of tokens (all required operands) is assembled for an instruction, an instruction template containing the relevant operands is created and queued for execution.

Figure 15 shows how a simplified token-matching architecture might process the program fragment shown in Figure 14. At step 1, the execution of (3*a*) results in the creation of a token that contains the result (15) and an indication that the instruction at node 3 requires this as an operand. Step 2 shows the matching unit that will match this token and the result token of (5*b*) with the node 3 instruction. The matching unit creates the instruction token (template) shown at step 3. At step 4, the node store unit obtains the relevant instruction opcode from memory. The node store unit then fills in the relevant token fields (step 5), and assigns the instruction to a processor. The execution of the instruction will create a new result token to be used as input to the node 4 instruction.

Reduction architectures. Reduction, or demand-driven, architectures[12] implement an execution paradigm in which an instruction is enabled for execution when its results are required as operands for another instruction already enabled for execution. Most reduction architecture research began in the late 1970s to explore new parallel execution paradigms and to provide architectural support for applicative (functional) programming languages.

Reduction architectures execute programs that consist of nested expressions. Expressions are recursively defined as literals or function applications on arguments that may be literals or expressions. Programs may "reference" named expressions, which always return the same value (the referential transparency property). Reduction programs are function applications constructed from primitive functions.

Reduction program execution consists of recognizing reducible expressions, then replacing them with their calculated values. Thus, an entire reduction program is ultimately reduced to its result. Since the general execution paradigm only enables an instruction for execution when its results are needed by a previously enabled instruction, some additional rule is needed to enable the first instruction(s) and begin computation.

Practical challenges for implementing reduction architectures include synchronizing demands for an instruction's results (since preserving referential transparency requires calculating an expression's results once only) and maintaining copies of expression evaluation results (since an expression result could be referenced more than once but could be consumed

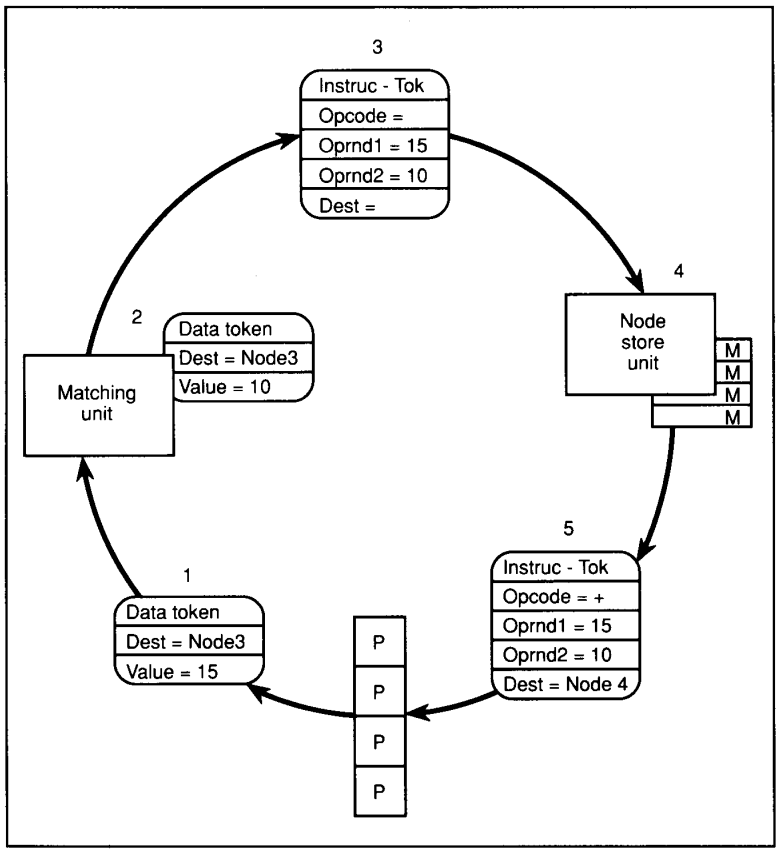


Figure 15. Dataflow token-matching example.

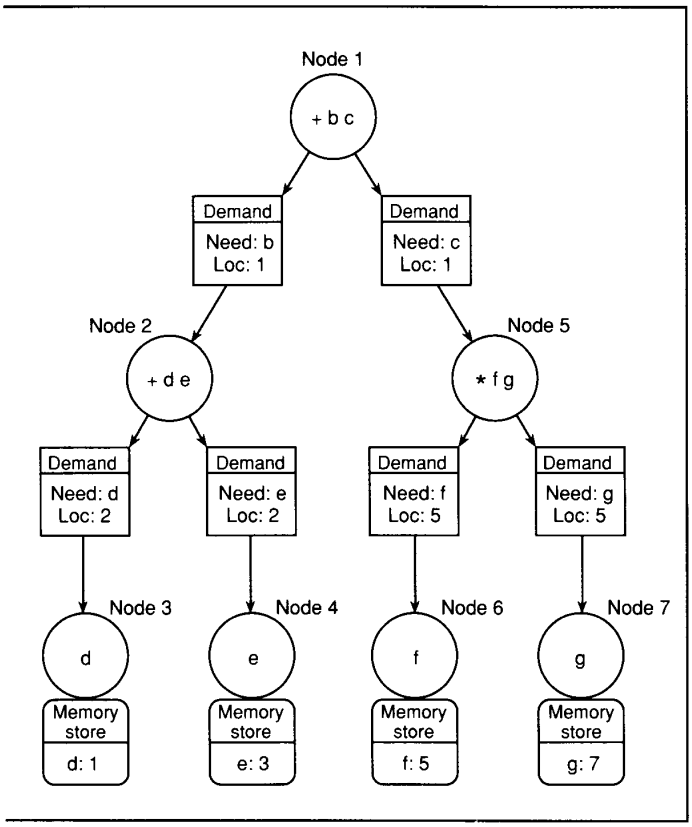


Figure 16. Reduction architecture demand token production.

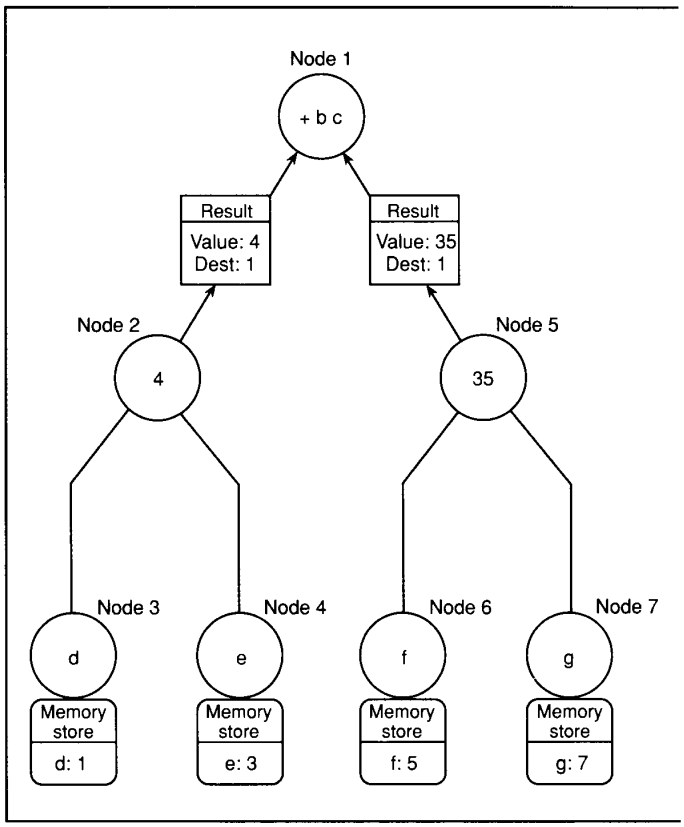


Figure 17. Reduction architecture result token production.

by subsequent reductions upon first being delivered).

Reduction architectures employ either string reduction or graph reduction to implement demand-driven paradigms. String reduction involves manipulating literals and copies of values, which are represented as strings that can be dynamically expanded and contracted. Graph reduction involves manipulating literals and references (pointers) to values. Thus, a program is represented as a graph, and garbage collection reclaims dynamically allocated memory as the reduction proceeds.

Figures 16 and 17 show a simplified version of a graph-reduction architecture that maps the program below onto tree-structured processors and passes tokens that demand or return results. Figure 16 depicts all the demand tokens produced by the program, as demands for the values of references propagate down the tree. In Figure 17, the last two result tokens produced are shown as they are passed to the root node.

```
a = + b c;
b = + d e;
c = * f g;
d = 1; e = 3; f = 5; g = 7.
```

Rather dissimilar architectures (such as the Newcastle Reduction Machine, North Carolina Cellular Tree Machine, and Utah Applicative Multiprocessing System) have been proposed to support both string- and graph-reduction approaches.

Wavefront array architectures. Wavefront array processors[13] combine systolic data pipelining with an asynchronous dataflow execution paradigm. S-Y. Kung developed wavefront array concepts in the early 1980s to address the same kind of problems that stimulated systolic array research — producing efficient, cost-effective architectures for special-purpose systems that balance intensive computations with high I/O bandwidth (see the systolic array section above).

Wavefront and systolic architectures are both characterized by modular processors and regular, local interconnection networks. However, wavefront arrays replace the global clock and explicit time delays used for synchronizing systolic data pipelining with asynchronous handshaking as the mechanism for coordinating interprocessor data movement. Thus, when a processor has performed its computations and is ready to pass data to its successor, it informs the successor, sends data when the successor indicates it is ready, and receives an acknowledgment from the successor. The handshaking mechanism makes computational wavefronts pass smoothly through the array without intersecting, as the array's processors act as a wave propagating medium. In this manner, correct sequencing of computations replaces the correct timing of systolic architectures.

Figure 18a-c depicts wavefront array concepts, using the matrix multiplication example used earlier to illustrate systolic operation (Figure 9). The example architecture consists of processing elements (PEs) that have a one-operand buffer for each input source. Whenever the buffer for

a memory input source is empty and the associated memory contains another operand, that available operand is immediately read. Operands from other PEs are obtained using a handshaking protocol.

Figure 18a shows the situation after memory input buffers are initially filled. In Figure 18b PE(1,1) adds the product *ae* to its accumulator and transmits operands *a* and *e* to neighbors; thus, the first computational wavefront is shown propagating from PE(1,1) to PE(1,2) and PE(2,1). Figure 18c shows the first computational wavefront continuing to propagate, while a second wavefront is propagated by PE(1,1).

Kung argued[13] that wavefront arrays enjoy several advantages over systolic arrays, including greater scalability, simpler programming, and greater fault tolerance. Wavefront arrays constructed at Johns Hopkins University and at the Standard Telecommunications Company and Royal Signals and Radar Establishment (in the United Kingdom) should facilitate further assessment of wavefront arrays' proposed advantages.

The diversity of recently introduced parallel computer architectures confronts the interested observer with what R.W. Hockney has felicitously termed "a confusing menagerie of computer designs."

This discussion has tried to address the difficulty of understanding these diverse parallel architecture designs. An underlying goal was to explain how the principal types of parallel architectures work. The informal taxonomy of parallel architecture types proposed here is meant to show that the parallel architectures reviewed define a coherent spectrum of architectural alternatives. The discussion shows that parallel architectures embody fundamental organizing principles for concurrent execution, rather than disparate collections of hardware and software features.■

Acknowledgments

Particular thanks are due Lawrence Snyder and the referees for constructive comments. Thanks go to the following individuals for providing research materials, descriptions of parallel architectures, and insights: Theodore Bashkow, Laxmi Bhuyan, Jack Dongarra, Paul Edmonds, Scott Fahlman, Dennis Gannon, E. Allen Garrard, H.T. Kung, G.J. Lipovski, David Lugowski, Miroslaw Malek, Robert Masson,

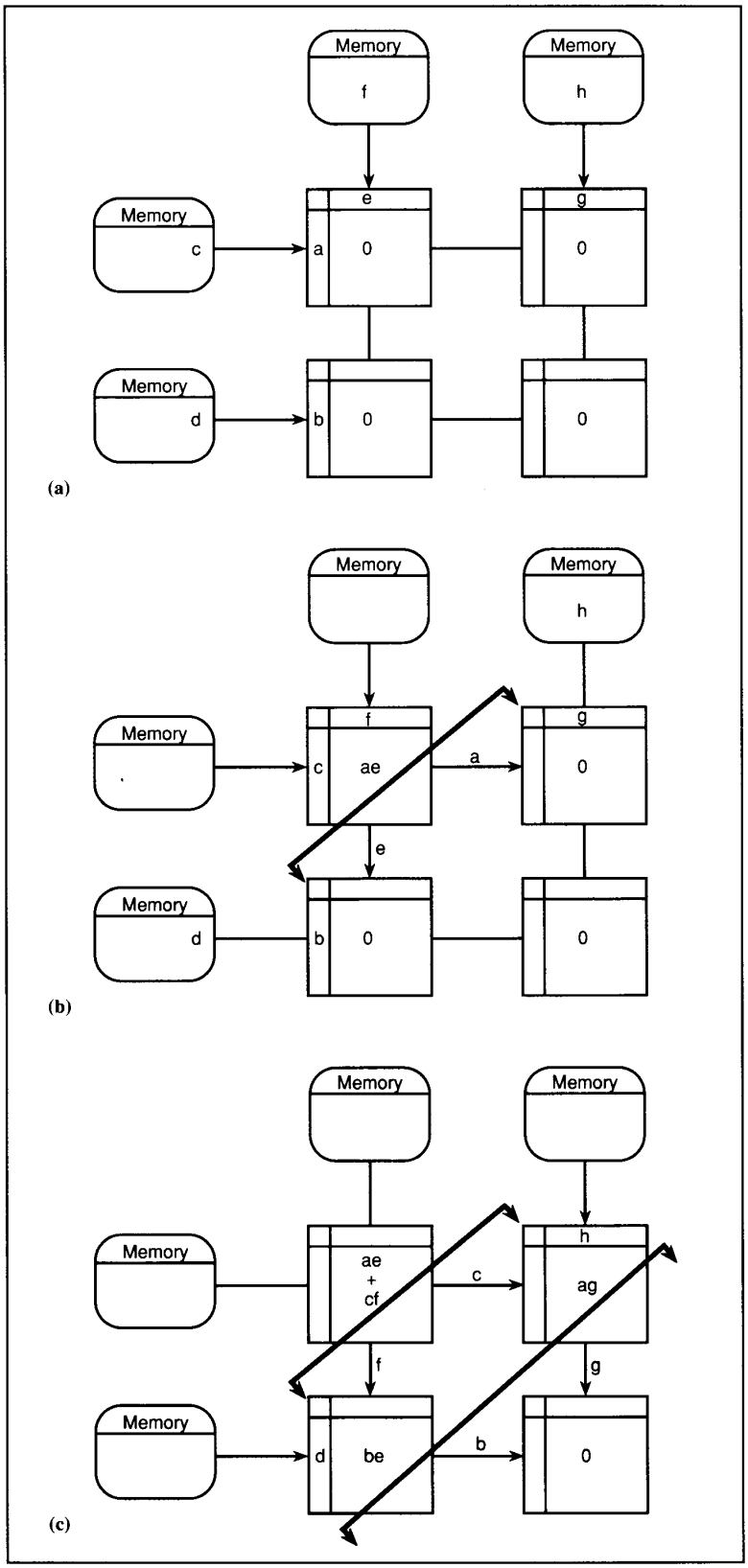


Figure 18. Wavefront array matrix multiplication.

Susan Miller, James Peitrocini, Malcolm Rimmer, Howard J. Siegel, Charles Seitz, Vason Srini, Salvatore Stolfo, David Waltz, and Jon Webb.

This research was supported by the Rome Air Development Center under contract F30602-87-D-0092.

Further reading

J.J. Dongarra, ed., *Experimental Parallel Computing Architectures*, North-Holland, Amsterdam, 1987.

R.W. Hockney, "Classification and Evaluation of Parallel Computer Systems," in *Springer-Verlag Lecture Notes in Computer Science*, No. 295, 1987, pp. 13-25.

D.J. Kuck, "High-speed Machines and Their Compilers," in *Parallel Processing Systems*, D. Evans, ed., Cambridge Univ. Press, 1982.

J. Schwartz, "A Taxonomic Table of Parallel Computers, Based on 55 Designs," Courant Institute, NYU, New York, Nov. 1983.

D.B. Skillicorn, "A Taxonomy for Computer Architectures," *Computer*, Vol. 21, No. 11, Nov. 1988, pp. 46-57.

L. Snyder, "A Taxonomy of Synchronous Parallel Machines," *Proc. 17th Int'l Conf. Parallel Processing*, University Park, Penn., August, 1988.

References

1. M.J. Flynn, "Very High Speed Computing Systems," *Proc. IEEE*, Vol. 54, 1966, pp. 1901-1909.
2. W.J. Watson, "The ASC — A Highly Modular Flexible Super Computer Architecture," *Proc. AFIPS FJCC*, 1972, pp. 221-228.
3. N.R. Lincoln, "A Safari through the Control Data Star-100 with Gun and Camera," *Proc. AFIPS NCC*, June 1978.
4. K. Hwang, ed., *Tutorial Supercomputers: Design and Applications*, Computer Society Press, Los Alamitos, Calif., 1984. (Chapters 1 and 2 contain salient articles on vector architectures.)
5. K. Hwang and F. Briggs, *Computer Architectures and Parallel Processing*, McGraw-Hill, New York, 1984.
6. K.E. Batcher, "Design of a Massively Parallel Processor," *IEEE Trans. Computers*, Vol. C-29, Sept. 1980, pp. 836-844.
7. T. Kohonen, *Content-Addressable Memories*, 2nd edition, Springer-Verlag, New York, 1987.
8. H.T. Kung, "Why Systolic Architectures?," *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37-46.
9. H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, Mass., 1985.
10. S.J. Stolfo and D.P. Miranker, "The DADO Production System Machine," *J. Parallel and Distributed Computing*, Vol. 3, No. 2, June 1986, pp. 269-296.
11. V. Srini, "An Architectural Comparison of Dataflow Systems," *Computer*, Vol. 19, No. 3, Mar. 1986, pp. 68-88.
12. P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *ACM Computing Surveys*, Vol. 14, No. 1, Mar. 1982, pp. 93-143.
13. S-Y. Kung et al., "Wavefront Array Processors — Concept to Implementation," *Computer*, Vol. 20, No. 7, July 1987, pp. 18-33.
14. G.J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons*, Wiley and Sons, New York, 1987. (Includes reprints of original papers describing recent parallel architectures.)

Ralph Duncan is a system software design consultant with Control Data's Government Systems Group. His recent technical work has involved fault-tolerant operating systems, parallel architectures, and automated code generation.

Duncan holds an MS degree in information and computer science from the Georgia Institute of Technology, an MA from the University of California at Berkeley, and a BA from the University of Michigan. He is a member of the IEEE and the Computer Society.

Readers may contact the author at Control Data Corp., Government Systems, 300 Embassy Row, Atlanta, GA 30328.