L. M. Patnaik
R. Govindarajan
M. Špegel**
and J. Šilc**
Indian Institute of Science BANGALORE
and Jožef Stefan Institute**

UDK 681.3.001.519.6

The need for parallel processing is felt in many disciplines. In this paper we discuss the essential issues involved in these systems. Various architectural proposals reported in the literature are classified based on the execution order and parallelism exploited. Design and salient features of some architectures which have gained importance are highlighted. Architectual features of non-von Neumann systems such as data-driven, demand-driven, and neural computers, which open the horizont for research in the new models of computations, are presented in this paper. Principles and requirements of programming languages and operating systems for parallel architectures are also reviewed.

Contents

1. Introduction

There has been an ever-increasing need for more and more computing power. In a real-time enviroment, the demands are much more. While the computers built around a single processor cannot stretch their processing speed beyond a few millions of floating point operations per second (FLOPS), an operating speed of a few giga FLOPS is required in many applications. Parallel processing techniques offer a promising scope for these applications.

Several applications such as computer graphics, computer aided design (CAD) of electronic and mechanical systems, wheather modeling and robotics have a high greed for high computing power. To quote an example, ray tracing techniques [32] used to display 3-dimensional objects in computer graphics have to process $5x10^8$ rays, each ray intersecting 10 to 20 surfaces, and each intersection computation requiring 20 to 30 floating point operations on an average. In order to provide a fast, interactive and flicker-free display, each frame has to be computed 30 times per second. Thus, the total computation power required 60 giga FLOPS. Comparing this with the execution speed of the present day supercomputers -- whose expected/measured performance is about 100 millions FLOPS [53] -- we observe that the demand is more by an order of magnitude of two.

The need for parallel processing is conspicuous from the above illustration. The recent advances witnessed in VLSI technology and the consequent decline in the hardware cost further encourage the construction of massively parallel processing systems.

The paper is organized in the following manner. In Section 2, we point out the major issues associated with multiprocessing systems, with a brief discussion on each of these items.

We classify parallel processing architectures broadly into two classes. The first of them, the von Neumann type, includes all multiprocessing systems that adopt the principles of the von Neumann computation model. Pipeline and vector processing, SIMD machines, MIMD machines and VLSI systems which fall under this category are surveyed in Section 3. Supercomputers employ one or more of these techniques for achiving high performance. More emphasis is given to current and recent developments in these areas. In non-von Neumann architecture, we include data-driven, demand-driven and neural computers. Their methodology of computation is functionally different from the ones discussed earlier. An overview of these architectural configurations is presented in Section 4.

The programming language and operating system support required for working with these complex parallel processing systems are reviewed in detail in the subsequent section. Parallel programming languages are classified into two categories: the conventional von Neumann type (that adop the imperative style of programming) and the non-von Neumann languages. Case studies of some of the popular languages and their salient features are also reported.

We remark that an exhaustive coverage of all contributions and research work reported in this fascinanting area of parallel processing systems is beyond the scope of this paper and we do not attempt that. Rather, we will concentrate on the basic principles behind the design of the various architectures. We will pay more attention to some specific architectures and models of computation which have gained importance.

2. Issues in Parallel Processing Systems

In the this section we discuss the various associated with multiprocessing systems. First, it is interesting to look at the models of computation, the way one has evolved from the other, leading to myrial architectural configurations and machines.

2.1. Models of Computation

Depending on the instruction execution order, computer systems can be classified as control-driven, data-driven or demand-driven machines.

Control Flow Conventional von Neumann uniprocessing and multiprocessing systems have an explicit execution order specified by the programmer. This hinders the extent of parallelism that can be exploited. However, these control models perform well for structured data items such as arrays [34]. Also the three decades of programming experience we had with these models still makes it a proponent candidate for future generation computers.

Data-Driven Model In this model, the execution order is specified by the data dependency alone [26]. Instructions are executed as soon as all their input arguments become available to them. Data flow model is suitable for expression evaluation [34]. Since, only data dependency determines the execution order, and the granualarity of parallelism exploited is as low as a single instruction, this model of computation exploits maximum parallelism. However, because of its eagerness in evaluating functions, it executes an instruction, if its operands are available, irrespective of whether or not it is required for the computation of the final result.

Demand-Driven Model In demand-driven (also called reduction) execution, pioneered by Berkling [15] and Backus [9], the demand for result triggers the execution which is turn triggers the evaluation of its arguments and so on. This demand propagation continues until constants are encountered; then a value is returned to the demanding node and execution proceeds in the opposite direction. Since a demand-driven computer performs only those computations required to obtain the results, it will perform less computations, in general, than a data-driven computer. As the computation model is 'lazy'[1] in evaluating expressions, instructions accept partially filled data structures[2]. This makes demand-driven model to support infinite data structures. Such a facility is not available in the other models of computation. However, this model suffers overheads in terms of execution time due to the additional propagation of the demand (for arguments). The control mechanism and the management of program memory add further to the inefficiency [91].

2.2. Concurrency

The granularity and nature of parallelism exploited, significantly influence the performance of the parallel processing systems.

Temporal, Spatial and Asynchronous Parallelism By performing overlapped computation one can exploit temporal parallelism. Pipeline computers [53,77] execute instructions in an overlapped manner as in the assembly line of manufacturing to achieve parallelism. Examples of pipeline processing are the execution of the different phases of an instruction namely, instruction fetch, decode, opcode fetch and execution; execution of the various steps involved in floating point arithmetic operations at various stages of the pipeline. These machines are ideally suited for processing vector instructions [53] namely, vector add, vector multiply and dot product.

Spatial parallelism is the parallelism inherent in performing an operation over the elements of structured data, such as arrays. Spatial parallelism is easy to detect and exploit [34]. The synchronization and scheduling overheads involved in exploiting spatial parallelism are much less compared to those experienced in the other two models.

Multiple instruction multiple data (MIMD) [53] machines achieve asynchronous parallelism through a set of interactive processors with shared resources. Several independent processes running asynchronously on different processors work to accomplish a common goal by exchanging messages or by sharing common variables.

While spatial parallelism is easy to detectet and exploit, the range of applications on which it is inherent is limited. MIMD machines which exploit asynchronous parallelism, on the other hand, offer flexibility in programming. Hence it can be used for a wide range of problems.

Granularity of Parallelism For high performance, we want to exploit as much

---

[1] The nature of the model in executing an instruction only on is termed 'lazy'.

[2] The computation of a recursively defined data structure, for example SEQ(N) = CONS(N,SEQ(N+1)) never terminates, and hence is infinite. Such a data structure can partially be filled by evaluating only those terms which are required for further computation.

parallelism as possible. This means that the granularity of the tasks should be low. We observe that as the level of granularity goes down, the parallelism that can be exploited increases. However, lower the level of parallelism more will be the synchronization overheads. A tradeoff between the parallelism exploited and the synchronization overheads is required to achieve high performance.

Another major issue involved is parallelism detection. Parallelism can either be explicitly specified by the user, or can be detected from a program written in a sequential language using an intelligent compiler. For expressing parallelism explicitly, language such as CSP [50], Occam [55] and Concurrent Pascal [45] can be used. The user need to have a knowledge of the architecture of the machine on which the program is executed, the application program and its runtime characteristics. The second approach which uses program restructuring techniques [71] to transform a sequential program into a parallel form, does not required a knowledged-user. However, these techniques are inefficient and cannot detect parallelism to the fullest extent. Active research to develop efficient parallel programs using these paradigms is underway.

2.3. Scheduling

In a multiprocessor system, scheduling algorithms assign each task to one or more processors with the goal of achieving high performance. Scheduling can again be static or dynamic (refer [6] for a comparison). In static scheduling, the tasks are allocated to processors either during the algorithm design by the user or at compile time by an intelligent compiler. In both these approaches, the scheduling costs are paid only once even if the program is run many times with different data. Morover, there is no runtime overhead. The disadvantage of static scheduling is possible inefficiency in guessing the runtime profile of each task.

Dynamic scheduling at runtime offers better utilization of processors, but at the cost of additional scheduling time. The dynamic scheduling algorithm can be distributed or centralized.

2.4. Synchronization

Synchronization methods are required to coordinate parallel execution of tasks in a multiprocessor system. Architectures with shared storage achive synchronization by semaphores [27] and monitors [49]. In a massage passing multiprocessing system, synchronization of processes is achived using remote procedure calls [46].

In order to update the shared variables in a multiprocessor in a consistent manner (avoiding read-read, read-write races [53]), the updating of the variables is done in a region called critical region. Only one process is allowed to enter the critical region at a time. Preventing access by other processes when the critical region is being accessed by a process is known as mutual exclusion. Synchronization primitives are used to achieve mutual exclusion.

Of the many synchronization primitives proposed (refer [35] for a survey), a few worth mentioning are

(i) the 'test & set' primitive of IBM 360/370 machines [17];
(ii) 'lock' instruction used in C.mmp [101];
(iii) NYU Ultracomputer's 'fetch & add' [42].

The subsequent sections highlight the architectural features of parallel processing systems. The whole spectrum of architectures proposed in the literature can broadly be classified as conventional von Neumann parallel processing systems and the non-von Neumann systems.

3. Von Neumann Parallel Processing Systems

Von Neumann parallel processing systems can be divided into three major architectural configurations: pipeline, SIMD, and MIMD architectures. These three architectural models exploit the three kinds of parallelism, namely temporal, spatial and asynchronous parallelism respectively. Each of the above mentioned architectures is discussed in detail and the design and salient features of certain parallel processing machines are highlighted in this section. Reconfigurable architectures and VLSI systems, though not entirely distinct from the above discussed models, attract the attention of researchers due to their many interesting features. In particular VLSI systems offer new scope in computing for designing dedicated architectures for a variety of applications. Constituent elements of the VLSI systems are systolic [63] and wavefront [64] architectures.

3.1. Pipeline and Vector Processing

Pipeline Architecture Pipelining [77] offers an economical way to realize temporal parallelism. The concept of pipeline processing in a computer is similar to manufacturing in assembly lines in an industrial plant. To achieve pipelining one must subdivide the input task into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are pumped into the pipe and get executed in an overlapped fashion at the subtask level. Pipeline processing leads to a tremendous improvement in system throughput. A k-stage linear pipeline could be at most k times faster. However, due to memory conflicts, data dependency, branch and interrupts this ideal speedup cannot be achieved.

One way of classifying a pipeline is based on the function performed by the pipeline. There are two classes of pipelines based on this classification, namely the instruction pipeline and the arithmetic pipeline. In the instruction pipeline, the various phases of instruction execution such as instruction fetch, instruction decode, operand fetch, and instruction execution, are identified and are executed in the successive stages of the linear pipeline. Thus, from pipeline, after an initial delay, instructions are executed once every clock cycle.

Arithmetic pipelines subdivide the arithmetic operations such as floating-point add or floating-point multiply into subtasks and execute them on specialized arithmetic and logic units. In an instruction pipeline, the instruction execution unit can itself be an arithmetic pipeline to further improve the performance. IBM 360/91 machine employs both these pipelines. Much research has already been done on scheduling of pipelines; buffering and delaying techniques are used to improve the execution speed. In Cray 1 [82], there are 12 functional pipeline units to perform both scalar and floating point arithmetic operations. Cyber 205 supports four vector pipelines in addition to a scalar arithmetic

unit.

Vector Processing In this section, we explaint the basic concepts of vector processing [53]. Vector processors operate on vector data and execute vector instructions. Vector pipelines, unlike scalar pipelines, are assured of continuous stream of data. The overheads involved in initializing the vector pipeline is compensated by the speedup improvement gained, as the number of tasks executed is large. Loop termination conditions are performed by specialized hardware in various stage of the vector pipelines. These features make vector processing more efficient than the scalar pipelines.

Vectorizing compilers [6] transform programs written in conventional imperative languages into vector instructions suitable for execution on vector processors. The fact that present day supercomputers have vectorizing compilers and vector processors as their major components demonstrates that pipeline processing is an easy and efficient way of realizing high speed computation. However, not all programs can be vectorized. Pipeline processors perform poorly in such cases.

3.2. SIMD Machines

A synchronous array of parallel processors executing in a lock-step fashion, consisting of multiple processing elements under the supervision of one control unit is called an array processor. The system and user programs are executed under the control of control unit. The array processor can handle single instruction and multiple data (SIMD) stream. SIMD machines exploit spatial parallelism. The processing cells are interconnected by a data-routing network. The interconnection pattern to be established for specific computation is under program control. Vector instructions are broadcast to the processing cells for distributed execution over different processors. The cells are passive devices without instruction decoding capabilities.

Array processors became well publicized with development of Illiac IV [12], Clip 4 [29], and Massively Parallel Processors (MPP) [14]. Future research in SIMD machines is towards designig and implementing multiple-SIMD (MSIMD) machines [53], consisting of more than one control unit. Each control unit in this class of machines shared a pool of dynamically allocatable processors.

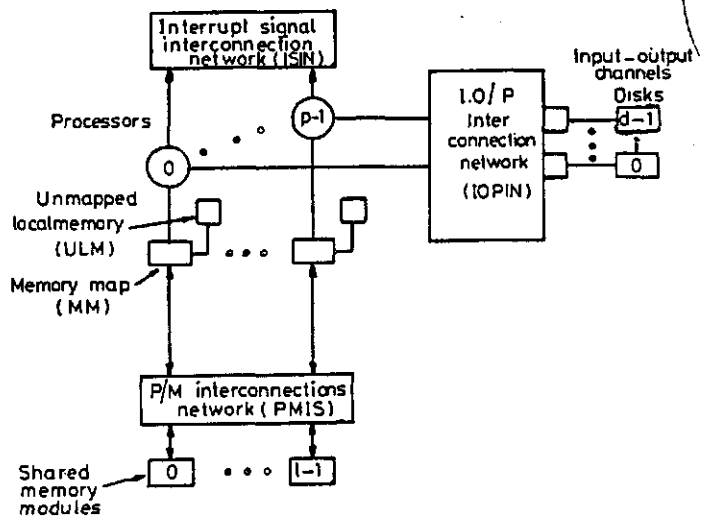
Array processors are characterized by their ability to support parallelism at a low-level. They are ideally suited for specific applications in the areas of image and signal processing [81]. For example, cellular array [81] is a two dimensional array of processors, each of which communicate directly with its neighbors. The structure of the array is very appropriate in terms of layout on a chip. Work in this direction has led us to the design of SIMD architectures for CAD applications [78].

However, as mentioned in section 2.2, the range of applications over which SIMD machines can be put into efficient use is restricted and hence they are not candidate architectures for general purpose parallel processing machines.

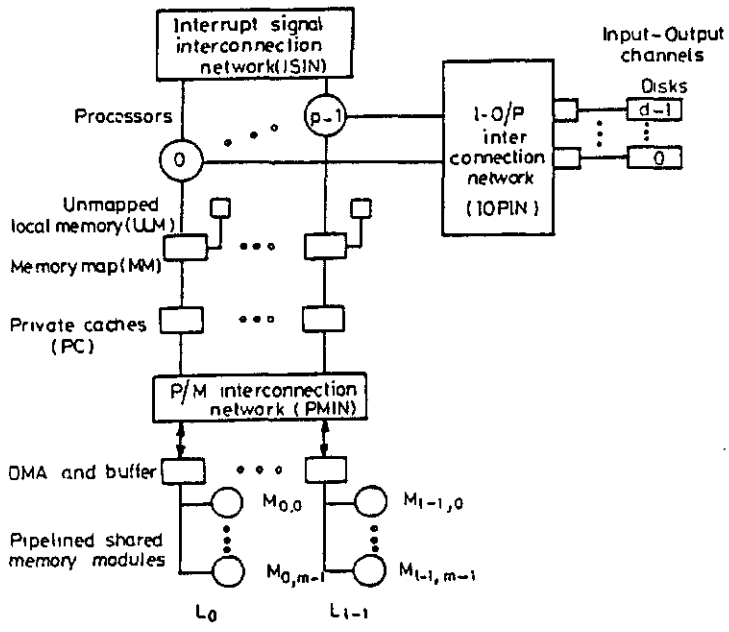
3.3. MIMD Architecture

MIMD machines can be grossly characterized by two attributes: first, a multiprocessor system is a single computer that includes multiple processors and second a multicomputer that has several autonomous computers which are geographically distributed and connected through a communication network. There exists an important distinction between two systems. In the first case, the multiple processors work concurrently in order to achieve a single goal. Interaction between two processors is essentially in terms of intermediate results and synchronization messages. Whereas multicomputers communicate among themselves basically to share expensive resources. Each autonomous computer works on an independent task. In this section we will concentrate more on multiprocessors. Discussion on distributed computing systems using computer network is beyond the scope of this paper.

Multiprocessing systems can be classified into two groups, based on how the processing elements interact among themselves [53]. When several processors communicate among themselves through a shared global memory, we classify them as tightly coupled systems (refer Fig. 1). Hence the rate at which the processors can communicate is of the order of the bandwidth of the memory. On the other hand, we have loosly coupled systems where processors do not share a common memory, but communicate using message passing primitives.



(a) WITHOUT PRIVATE CACHE



(b) WITH PRIVATE CACHE

Fig. 1 Model of a Tightly Coupled System

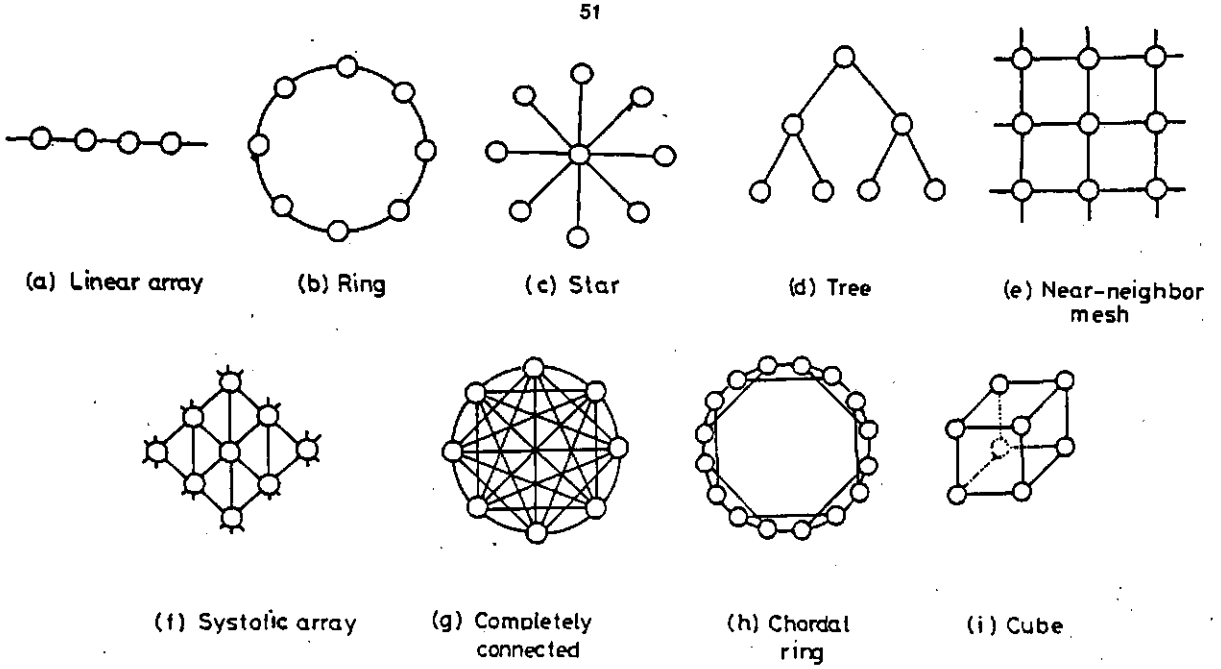


Fig. 2 Static Interconnection Topologies

The interconnection network plays an important role in multiprocessor systems, and significantly influence the performance of the system. A quick overview of the interconnection network is presented in this section, which will be followed by a siscussion on the two configurations of multiprocessing systems.

Interconnection Networks The increasing popularity of many proposed multiprocessing systems with 10^4 to 10^5 processing elements makes the concept, design and implementation of the interconnection network a crucial factor in the design of such systems. A typical interconnection network consists of a set of switching elements. The network can be classified based on the following four design factors: operation mode, control strategy, switching method and network topology [30]. A network can send and receive messages in a synchronous or asynchronous mode. Classification on the control strategy is based on whether the switching elements are controlled in a centralized or distributed manner.

Circuit switching and packet switching are two switching methods adopted in a network. In circuit switching, a physical path is actually established between the source and destination nodes. In contrast, in packet switching, data is put in a packet which is routed through the network without establishing a physical path. Fig. 2 and Fig. 3 depict some of the static and dynamic topologies of computer network.

The bus interconnection scheme connects the various processing cells through a common shared bus. The bandwidth of the bus is very low, and contention is a serious consequence when a large number of cells are connected to the bus. Various schemes [11] such as daisy chain, parallel priority and time-sliced schemes, have been proposed to resolve bus contention. Fig. 4 shows the organization of a shared bus multiprocessing system with daisy chain scheme for priority resolving. Despite its shortcomings, shared bus still attracts system designers because of its low cost and complexity and easy upgradability of the system.

Cross bar switch [53] on the other hand is very expensive, but provides high memory bandwidth. The cost of the network is of the order of n^2, where n is the number of the processing cells in the system.

Multistage interconnection network (MIN) strikes a balance between cost and performance. It is dynamic network using a number of switching elements, unlike a static network where dedicated links are used. A MIN of n x n size establishes a maximum of n links at any instance, at a cost of n log n. This attractive feature makes MIN a proponet in many multiprocessing systems such as the New York university Ultracomputer (NYU) [42].

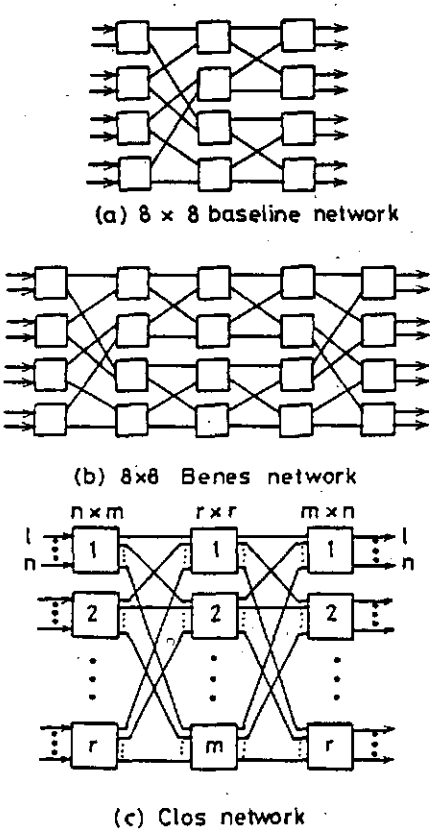


Fig. 3 Dynamic Interconnection Topologies

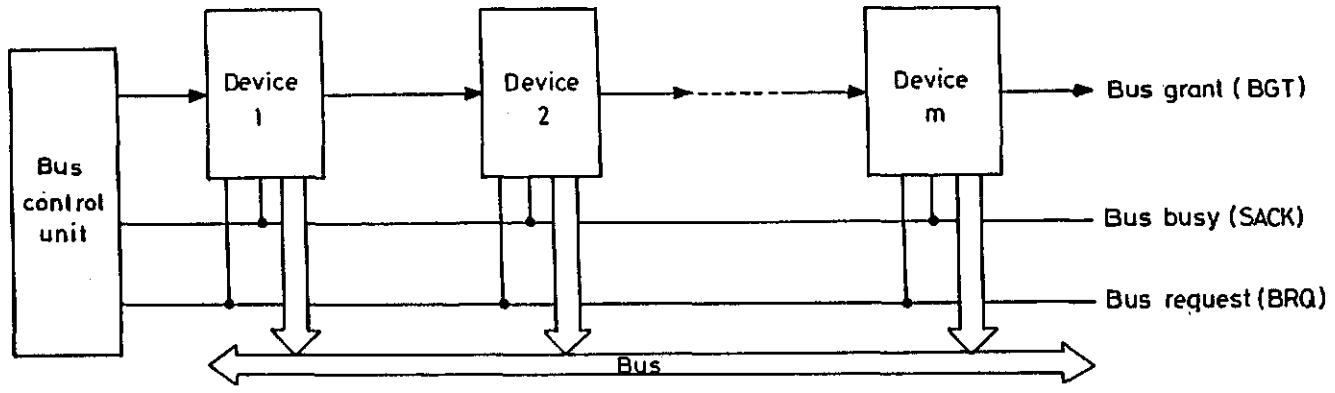


Fig. 4 Daisy Chain Implementation of Shared Bus

Tightly Coupled Multiprocessors Tightly coupled systems are ideally suited if high speed or real-time processing is desired. In a tightly coupled system a set of processing elements is connected to a set of memory modules through an interconnection network.

If two or more processors attempt to access the same memory module, a conflict occurs. Hence the memory modules are either low-order interleaved or high-order interleaved to reduce the number of conflicts. In a variant of tightly coupled systems, each processor is allowed to have a private memory, called the cache for that processor. This will greatly reduce the traffic through the network as local data can be stored in and accessed from the cache.

Processors communicate the intermediate results through the shared memory modules. The set of processors may be homogeneous or heterogeneous. It is homogeneous if the processors are functionally identical. Also, a processor may differ from others in its capability to access I/O systems, performance and reliability.

Various multiprocessing systems have been proposed using a shared memory architecture. Examples of these are, the C.mmp system [101], the Heterogenous Element Processor (HEP) [25], the New York university Ultracomputer (NYU) [42], Honeywell 60/66, Univac 1100/80 and IBM 3084 AP. A dedicated multiprocessing architecture with time shared bus has been designed and experimented by use for computer graphics applications [36,37,38].

Tightly coupled systems can tolerant a higher degree of interaction without much deterioration in performance. However, one of the limiting factor of tightly coupled systems is the performance degradation due to memory conflicts when the number of processors in the system is increased.

Loosely Coupled Multiprocessors Loosely coupled multiprocessor systems do not generally encounter the degree of memory conflicts experienced by tightly coupled systems. In such systems, each processor has a set of input/output devices and large local memory where it accesses most of the instructions and data. Processes which execute on different computer modules communicate by exchanging messages through a message transfer system. The degree of coupling in such a system is very loose (and hence the name). The determining factor in the degree of coupling is the communication topology of the associated message transfer system. Loosely coupled systems are efficient when the interactions among the tasks are minimal.

Fig. 5 illustrates the model of a loosely coupled system. The channel and arbiter switch in each of the computer modules may have a high speed communication memory which is used for buffering block transfers of messages. The communication memory is accessible by all processors through the communication interface. The message transfer system for non-hierarchical system could be a simple time shared bus. The performance of such configuration is limited by the message arrival rate on the bus, the mesage length, and the bus capacity (in bits per second).

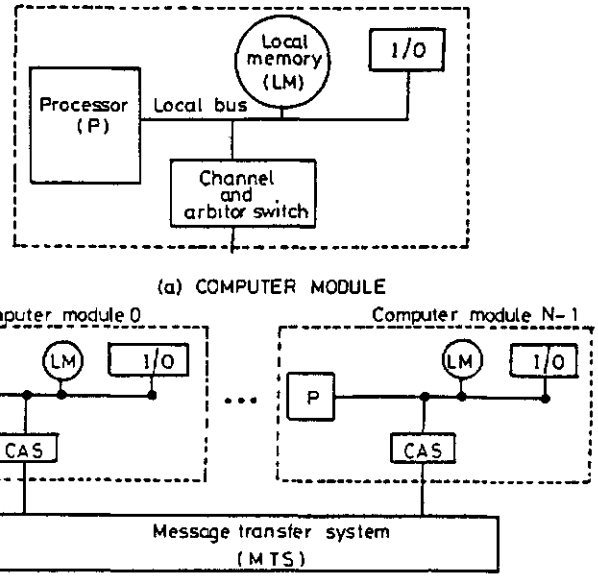


Fig. 5 Model of a Loosely Coupled System

Loosely coupled systems where processors are connected using dedicated links, are also popular. In such a network of processors, locality (nearness between a pair of processors) can be exploited by scheduling tasks which interact among themselves more to a group of processors which are closer to each other. Failure of a node or link will not catastrophically affect the system, as alternate paths can be established and the system can perform with graceful degradation.

The Cm* architecture [54] developed at the Carnegie Mellon University is one example of a loosely coupled system. It consists of a set of clusters connected by an intercluster bus (Fig. 6). Each cluster has a set of processing elements connected over a map bus. The system forms a good example for hierarchical structure.

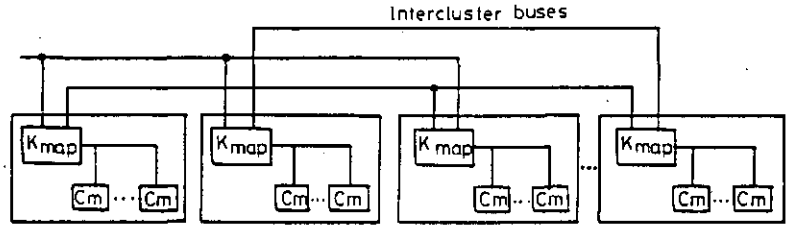


Fig. 6 Cm* Architecture

A loosely coupled system designed for artificial intelligence and image processing application is ZMOB [80]. In this architecture, a set of 256 Zilog microprocessors are connected by a pipeline system.

Another loosely coupled architecture which has attracted the attention of many researchers due to its high performance and relatively low cost, is the hypercube architecture [47,83]. Because of the increased attention it has received, we devote the following paragraphs to this architecture.

The concept of a hypercube computer can be traced back to the work done in the early 1960's by Squire and Palais [93] at yhe University of Michigan. They carried out a detailed paper design of a 4096-node (12-dimensional) hypercube.

The hypercube topology is an n-dimensional generalization of the simple cube. The hypercube of dimension n has 2^n cells. Each cell is connected to n neighboring cells which are at a Hamming distance of one. The connection between adjacent nodes is by point-to-point link. Fig. 7 shows the topology of hypercube architecture for dimensions three and four. The cube manager provides a high level interface for system users. It serves as a local host for the cube, supporting the programming environment, compilation, program loading, input/output and error handling.

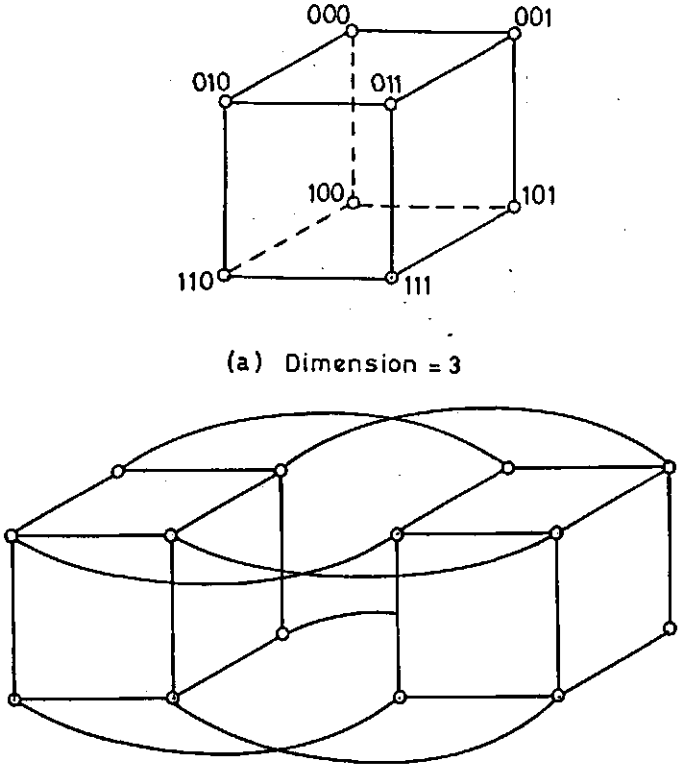


Fig 7. Hypercubes

Hypercube architecture has many attractive properties. The hypercube topology yields a regular array in which the node are close to one another: no more than n steps apart. At the same time, the number of connections from each node to its neighbors is quite low (also equal to n). It thus strikes a balance between a two-dimensional array in which the internode connection costs id low, but the nodes are far apart $O(n^{1/2})$ steps on an average, and a completely connected array in which the internode connection costs is high, but the nodes are only one step apart.

The hypercube architecture is homogeneous in the sense that all the nodes are identical. Further, the hypercube architecture is hierarchical and eminently partitionable. For example, a hypercube of dimension n+1 can be pertitioned into two hypercubes of dimension n. This means that it is quite easy to allocate subcubes to subtasks, especially for problems which adopt a divide-and-conquer strategy. Lastly, the hypercube architecture can itself embed other regular topologies such as tree, mesh, pyramid or hexagonal structure.

3.4. Reconfigurable Architecture

Another interesting aspect of multiprocessing systems in which active research is under progress is reconfigurability [85]. Often, full potentials of multiprocessing systems are not realised in many applications. The reason for this is the mismatch between the application and the architecture. In order to alleviate this problem, we build dedicated systems where the architecture matches the application. Another approach which is actively pursued by researchers is building multiprocessing systems with programmable switches; using these switches it is possible to reconfigure the system depending on the application. Such a reconfigurable system, by nature, is flexible and hence can be used for a variety of applications. Configurable Highly Parallel computer (CHiP) [85] is a reconfigurable system designed to suit the topologies of various applications. In this section we will highlight the salient features of another reconfigurable architecture, the Connection Machine [48]. Its organization is similar to an SIMD machine, but it functions as a reconfigurable architecture executing multiple instructions on multiple data streams, and hence it is hard to classify this as SIMD or MIMD.

Connection Machine The desire to build a machine that will be able to perform the functions of a human mind, the thinking machine, is the motivating force behind the design of Connection Machine. Specifically, retrieving commonsense knowledge from a

semantic network was the application in the designer's mind.

The Connection Machine computes through the interaction of many, say a million, simple identical processing/memory cells. The two requirements for the connection machine are:
(i) each processing element must be as small as possible so that we can afford to have as an many of them as we need;
(ii) the processing elements should be connected by software.

The Connection Machine architecture follows directly from these two requirements. It provides a large number of tiny processor/memory cells connected by a programmable network. Each cell is sufficiently small so that it is incapable of performing meaningful computations on its own. Instead, multiple calls are connected together into data-dependent patterns, called 'active data structures' that both represent and process the data. The activities of these active data structures are directed from outside the Connection Machine by a conventional host computer. This host computer stores data structures on the Connection Machine in much the same way that a conventional machine stores them in a memory. Unlike a conventional memory, though, the Connection Machine has no processor/memory bottleneck. The memory cells themselves do the processing. More precisely, the computation takes place through the coordinated interaction of the cells in the active data structure. Because thousands or even millions of processing cells work on the problem simultaneously, the computation proceeds much more rapidly than would be possible on a conventional machine.

A Connection Machine is connected to a conventional computer much like a conventional memory. Its internal state can be read and written a word at a time from the conventional memory. It differs from a conventional memory in three aspects. First, associated with each cell of storage is a processing cell that can perform local computations based on the information stored in that cell. Second, there exits a general intercommunication network that can connect all the cells in an arbitrary pattern. Third, there is a high-bandwidth input/output channel that can transfer data between the Connection Machine and peripheral devices at a much higher rate than would be possible through the host.

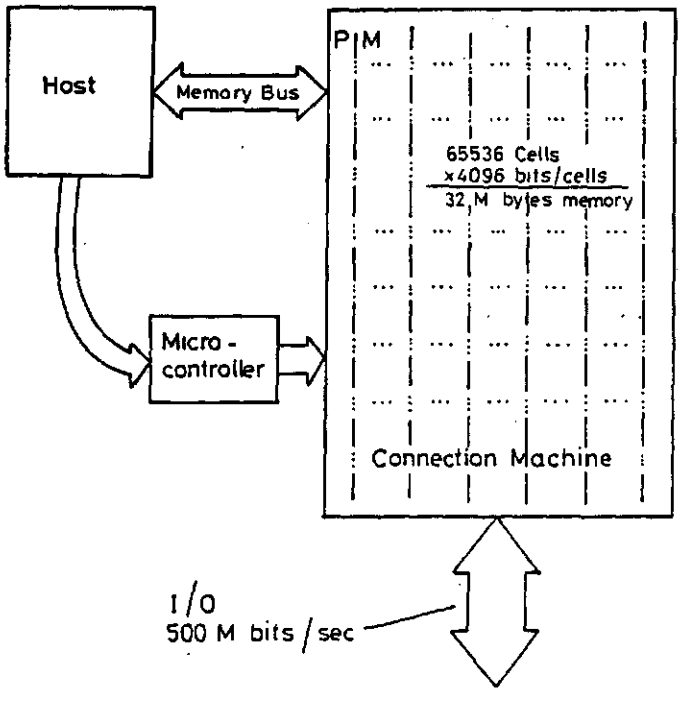


Fig. 8 Connection Machine

A connection is formed between two processing memory cells by storing a pointer in the memory. These connections can be set up by the host, loaded through the input/output channel, or determined dynamically by the Connection Machine itself. In this prototype system, there are 65,536 (2^{16}) processor/memory cells each with 4096 bits of memory. The block diagram of the Connection Machine with host, processor/memory cells, communication network, and input/otput is shown in Fig. 8.

The control of the individual processor/memory cells is orcheastrated by the host of the computer. For example, the host may ask each cell that is in a certain state to add two of its (cell's) memory locations locally and pass the resulting sum to a connected cell through the communication network. Thus a single command from the host can result in tens of thousands of additions and a permutation of data that depends on the pattern of connections. Each processor/memory cell is so small that it is essentially incapable of computing or even storing the results of any significant computation on its own. Instead, the computation takes place in the orchestrated interaction of thousands of cells through the communication network.

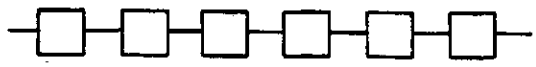
The ability to configure the topology of the machine to match the topology of the problem turns out to be one of the most important features of the Connection Machine. The Connection Machine can also be used as a content addressable or associative memory, but it is also able to perform non-local computations through its communication network.

3.5. VLSI Systems

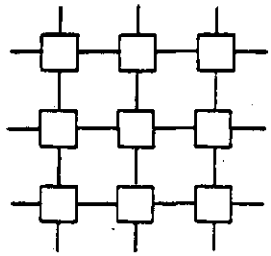
While the previous subsection discusses the features and requirements of reconfigurable architectures, this section is devoted to the architecture of dedicated systems. Nowdays, mature VLSI/WSI (Very Large Scale Integration / Wafer Scale Integration) technology permits the manufacture of circuits whose layouts have minimum feature size of 1 to 3 microns [69]. The effective yields of VLSI/WSI fabrication processes make possible the implementation of circuits with upto half a million transistors at reasonable cost -- even for relatively small production quantities. This opens the horizont for building systems with thousands of processors in a cost-effective and compact manner.

The key attributes of VLSI computing structures [33,63] are
(i) simplicity and regularity,
(ii) concurrency and communication and
(iii) computation-intensiveness.
A VLSI structure should be such that its basic building block is simple and regular, and is used repetitively with simple interfaces. This helps us to cope with high complexity. The algorithm designed for these structures should support a high degree of concurrency, and employ only, simple, regular communication and control to allow efficient implementation. VLSI processors are suitable for implementing compute-bound algorithms. In VLSI processors, we discuss the two classes of architectures namely, systolic and wavefront processors.

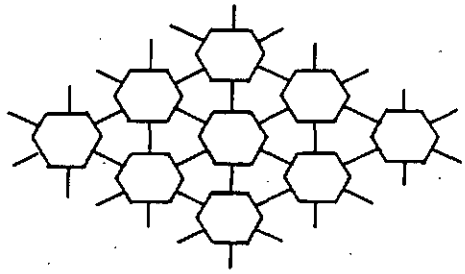
Systolic Arrays Systolic arrays are admired for their elegance and potential for



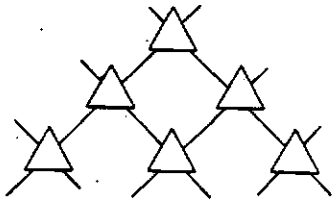
a LINEAR SYSTOLIC ARRAY



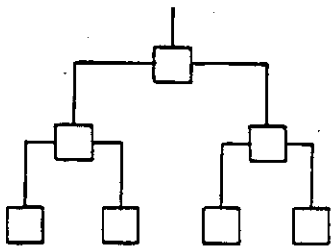
b MESH CONNECTED SYSTOLIC ARRAY



c HEXAGONAL SYSTOLIC ARRAY



d TRIANGULAR SYSTOLIC ARRAY



e TREE STRUCTURED SYSTOLIC ARRAY

Fig. 9 Systolic Arrays

high performance. Systolic arrays belong to the generation of VLSI/WSI architectures for which regularity and modularity are important to achieve area-efficient layouts. The concept of systolic architecture is a general methodology -- rather than being an ad hoc approach -- for mapping high-level computations into hardware structures. In systolic system data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. Systolic arrays derive their computational efficiency from multiprocessing and pipelining. The data items pumped into the systolic processors are reused many times as the move through the pipelines in the array. This results in balancing the processing and input/output bandwidths, a requirement for any parallel processing system for alleviating the von Neumann bottleneck.

The topologies for interconnecting the processing elements of a systolic array are many. Some of the most commonly used topologies, namely linear, mesh, triangular, hexagonal and tree structures are shown in Fig. 9.

Essentially, the whole operation of the systolic system is synchronized with a global clock and it may be visualized as a sequence of computation and data transfer cycles. Incidentally, the clock is the only global signal allowed in systolic architecture apart from the power and ground lines [93]. During the data transfer cycle, all the processing elements pump data into the existing data channels, to be accepted by the neighboring processing elements connected to the data channels. After this cycle is over, all the processing cells enter the computation cycle where each of the cells computes concurrently till the end of the computation cycle. This sequence goes on rhythmicall and perpetually in strict synchronism with the clock beats. Systolic architectures thus capture the concepts of parallel processing, pipelining and regular interconnection structure in a unified framework [63].

Having all the desirable properties of an efficient special purpose system, the systolic architecture is an interesting area of research for a variety of applications, namely digital and image processing [62], linear algebra [28,69], database systems [63], computer graphics [67,96,97], computer-aided design, and solid modeling [60]. The systolic algorithms are characterized by repeated computations of a few types of relatively simpleoperations that are common to many input data items. Often, the algorithms can be described with nested loops or by recurrence equations that describe computations performed on indexed data.

The systolic architectures were originally conceived as devices capable of performing a specialized task. Essentially, these architectures consist of a large number of identical processors, each having only a single arithmetic or logic operation build into its hardware. This greatly limits the applicability of a system to many areas. The latest trend in research in this direction is towards the development of systolic cells which are versatile enough to implement the compute-intensive functions. The design of programmable systolic cells [31] has been suggested as an effective way towards achieving high performance systolic systems. Each systolic cell now possesses a rich instruction set along with some amount of local storage, which were completely absent in the original versions of the systolic architecture. By suitably programming these systolic cells, a variety of operations can be performed. The programmable nature of the systolic cells offers a high degree of flexibility in operation and high performance.

Wavefront Array Processors The data movements in a systolic array are controlled by global timing-reference 'beats'. The burden of synchronizing the operations of the entire systolic computing network becomes heavy for

very large arrays. A simple solution is to take advantage of the data flow computing principle [91] (which will be discussed in detail in Section 4.1), which leads the designer to wavefront array [64] processing.

The wavefront array combines systolic pipelining principle with the data flow computing concept. In fact, the wavefront array can be viewed as a static data flow array that supports the direct hardware implementation of regular data flow graphs. Exploitation of the data flow principle makes theextraction of parallelism and programming for wavefront arrays relatively simpler.

Synchronizing with the global clock and consequently the large surge of current (due to the simultaneous energizing or changing of states of the components) are two major problems in systolic arrays. These can be alliviated in wavefront arrays, because of their asynchronous nature. When the processing times of the individual cells are not uniform, a synchronous array may have to accommodate the slowest cell by using a slower clock. In contrast, wavefront arrays, because of their data-driven nature, do not have to hold back faster cells in order to accommodate the slower one. Wavefront arrays also yield higher speed when the computing times are data-dependent. Lastly, programming of wavefront arrays is easier than that of systolic arrays because wavefront arrays require only the assignment of computations to processing elements, whereas systolic arrays require both assignment and scheduling of computations.

3.6. Critique on von Neumann Systems

The most serious problem with the multiprocessors which use the von Neumann model, as discussed in [10], is the presence of globally updatable shared memory. Special mechanisms are required to ensure correctness of results while updating a memory cell. The explicit execution order to be specified by the programmer is another bottleneck of von Neumann systems. This has led to research in non-von Neumann architectures.

4. Non-von Neumann Architectures

The principal stimuli for developing the non-von Neumann machines have come from the pioneering work on data flow machines by Jack Dennis [26], and on reduction languages and machines by John Backus [9] and Klaus Berkling [15]. In data-driven model, the availability of operands triggers the execution of the operation to be performed on them, whereas in demand-driven model, the requirement for a result triggers the operation that will generate the result. Of late, the realization of the suitability of biological nervous systems for many applications and the use of artificial nets have triggered the design of a new system, the neural computer. In this section, we present the details of these three non-von Neumann approaches.

4.1. Data-Driven Model

In a data flow computer, an instruction is executed as soon as all its operands are available. Since the data availability solely dictates the execution order of instructions, there is no need for having a program counter. Also, the data are passed as values between instructions; this eliminates shared memory and makes the synchronization mechanism simpler. Thus data flow model alleviates the shortcoming of the von Neumann model. Also, parallelism is exploited at instruction level. Hence, a very high speed of computing is possible.

The machine language for a data flow computer is the data flow graph [24]. The data flow graph consists of nodes, to represent operators, and arcs, to carry data between the nodes. Tokens carry data values along the arcs to the nodes. When all the required operands are available, the node is 'fired'. As a result, the input tokens are removed and output tokens are produced.

Data flow architectures are classified as static and dynamic architecture. In static model, an additional constraint -- no token should be present on any of the output arcs of node -- is required to enable the execution of an instruction. The implication of this is that the static data flow model cannot support execution of reentrant routines. This severely restricts the extent of parallelism and asynchrony that can be exploited in the static model. In a dynamic model, additional tags, called environment tags (color), are associated with the token to distinguish the various instantiations of a rentrant routine. Thus, the dynamic model supports fine grain parallelism with full asynchrony. In this paper we describe the architecture of a data flow computer, using the Manchester data flow computer [98] as an illustrative example.

In Fig. 10, the switch unit serves as an interface between the host computer and the data flow machine. It routes the intermediate results to the token queue and the final results to the host computer. The token queue unit is a FIFO buffer which smoothes the flow of tokens in the ring. Tokens in the token queue are checked for their operand type. All tokens directed to single input nodes are directly routed to the node store. Other tokens are sent to the matching unit.

Tokens that arrive in the matching unit search for their matching partner. If the search fails, the token is stored in the matching store; it awaits its partner. On the other hand, if the matching partner is found, it (the matching partner) is removed from the matching store; the incoming token is merged with its partner to form a group token. The group token which contains the information about the operand values, address (of the node to which the token is destined to) and environment tag is sent to the node store.

The node store unit stores the program graph; it stores the opcode for the operator, destination and operand type of result tokens. Tokens entering the node store get the above information to form an executable token. The executable packets are sent to the distributor unit which distributes the tokens to one of the free processing elements. The processing element performs the operation specified by the token to produce result tokens. The result tokens are collected by the arbitrator and are sent to the switch unit. Fig. 10 depicts the block schematic of the Manchester data flow computer.

Research in the area of data flow computation is a rapidly expanding area in United States, Japan and Europe. There are a number of data flow projects that are underway in many universities. Some of them worth mentioning here are M.I.T. data flow computer [26] developed by Dennis, Irvine data flow machine [7,39] by Arvind, Manchester data flow system [98], its extended version (EXtended MANchester architecture) proposed by the

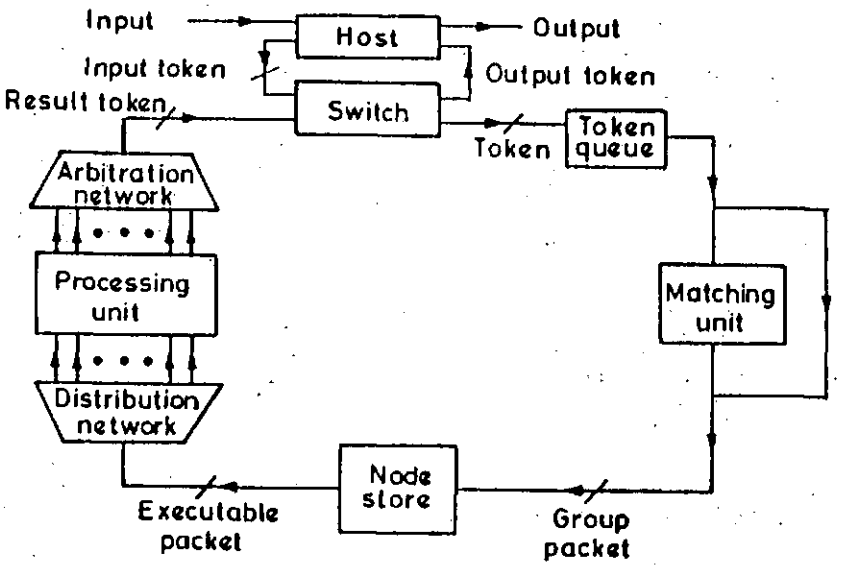


Fig. 10 Block Schematic of Manchester Data Flow Computer

authors [74], Texas Instruments distributed data processor [22], Utah data driven machine [23], Toulouse LAU system [20,76], Newcastle data-control flow computer [89], the efficient static dataflow architecture for specialized computation proposed by the authors [87], and the high speed data flow system developed by Nippon Telegraph and Telephone Systems [3].

Much work needs to be carried out in the design of languages for data flow machines, and implementation of compilers for converting the programs into data flow graphs. (Refer [88] for the initial work on these issues.) Efficient methods to overcome the inherent overheads associated with exploiting fine-grain parallelism have to be developed. Although many data flow machines have been proposed in the literature, no effort is made to prototype them. Demonstrating the feasibility of the data flow model of computation is thus a positive step towards the commercialization of such systems.

4.2. Demand-Driven Systems

In contrast to control flow and data flow programs which are built from fixed-size instructions, demand-driven (reduction) [91] programs are built from nested expressions. The need for result triggers the execution of a particular instruction.

An important point to note is that, in a reduction machine, a program is mathematically equivalent to its value. Demanding the result of definition a, defined as x = (y+1) * (y-z), means that the embedded reference to x is to be rewritten in a simple form. This requires that only one definition of x may occur in a program, and all references to it give the same value, a property known as referential transparency [91].

There are two form of reduction, called string reduction and graph reduction. The basis for string reduction is that each instruction that accesses a particular definition will take and manipulate a seperate copy of the function definition. Whereas, in graph reduction each instruction that accesses a particular definition will manipulate references to that definition. That is, graph reduction is based on sharing of arguments using pointers. In string reduction each access for a definition

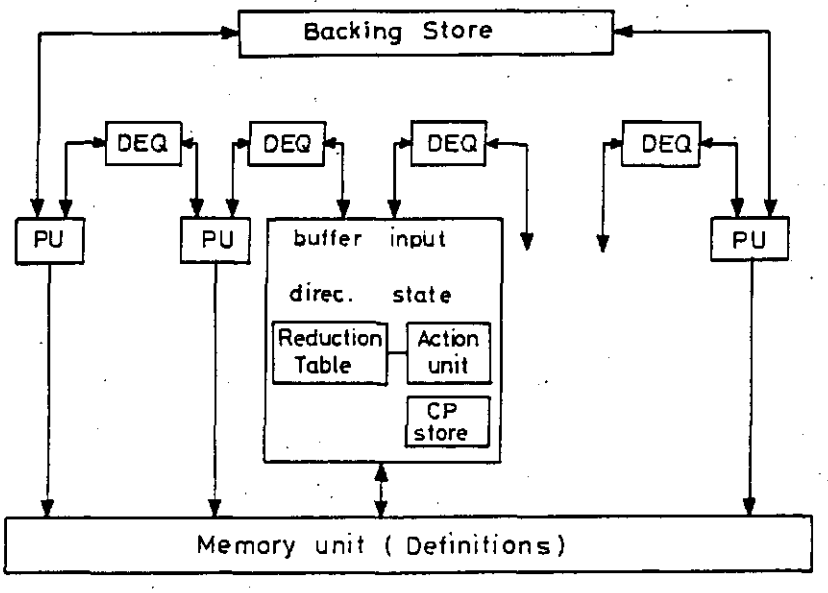


Fig. 11 The Newcastle Reduction Machine

will result in the evaluation of the definition. Reduced definitions or data values are accessed when the demanded definition has already been evaluated. While graph reduction machine takes advantage of the shared definition (in term of the number of definitions evaluated), it is more complex than string reduction.

There are two basic problems in supporting reduction approach on a machine organization [90]: first, managing dynamically the memory of the program structure being transformed and, second, keeping control information about information about the state of the transformation. The basic organization of a reduction machine, the Newcastle reduction machine [90], is presented below (refer Fig. 11)

The reduction machine organisation discussed here supports reduction by expression evaluation. To find work, each processing element traverses the subexpression in its memory looking for a reducible expression. When a processing element locates a reference to be replaced by the corresponding definition, it sends a request to the communication unit via its communication element. The communication unit in such a computer frequently organized as a tree-structured network on the assumption that the majority of communications will exhibit properties of locality of reference. Concurrency in such reduction computers is related to the number of reducible subexpressions at any instant and also to the number of processing elements that traverse these expressions.

Apart from the pioneering work of Klaus Berkling [15], the stage of development of reduction computers somewhat lags behind that of data flow computers [91]. However, researchers have demonstrated the principle and feasibility of reduction machine organization by designing many prototype system such as the GMD reduction machine [58], Newcastle reduction machine [90], Mago's cellular tree machine [66], Applicative Multiprocessing Systems (popular known as AMPS) [57], and Combridge University SKIM reduction machine [92].

4.3. Neural Computers

In the fields of image processing and speech recognition, the ability to adapt and continue learning is essential. Traditional techniques used in these applications are not adaptive. It has been realized that the biological nervous system is more suitable for applications involving pattern recognition and learning [2]. Artificial neural nets have been studied during the last few years in the hope of achieving human like performance in the fields of image processing and speech recognition. The neural net models [65] attempt to achieve good performance via dense interconnection of simple computational elements. The interest in this type of non-von Neumann computing techniques in recent years is due to the development of new net topologies and algorithms, new analog VLSI implementation techniques and the growing fascinantion for the understanding of the functioning of the human brain as well as the realization that human-like performance is required for applications involving enormous amount of processing [51,65]. Several mathematical models have been proposed to exhibit some of the essential qualities of human mind: the ability to recognize patterns and relationships, to store and use knowledge, to reason and plan, to learn from experience and to undrstand what is observed.

Neural net models are specified by the net topology, node characteristics and training or learning rules. The computational elements or nodes used in neural net models have nonlinear characteristics, typically analog, and are specified by the type of nonlinearity. Most net algorithms also adapt in time to improve performance based on current results. Any artificial neural model must necessarily be a speculation: definitive experimental evidence about the structure and function of the neurological circuitry in the brain is extremely difficult to obtain since it is hard to measure the neural activity without interfering with the flow of information in the neural circuit. Further, the neurons are intricately interconnected and the flow of information is complicated by the presence of multiple feedback loops.

Nevertheless enough is known about some parts of the brain to fuel the desire for constructing mathematical models of the neural circuit. In general, the models propose to generate a sensory-activated goal-directed behavior and control a multilevel hierarchy of computing modules. At each level of hierarchy, input commands are decomposed into strings of output subcommands, that form input commands to the next lower level. Feedback from external environment or from internal sources drives the decomposition process, and steers selection of subcommands to achieve the goal successfully (refer Fig. 12).

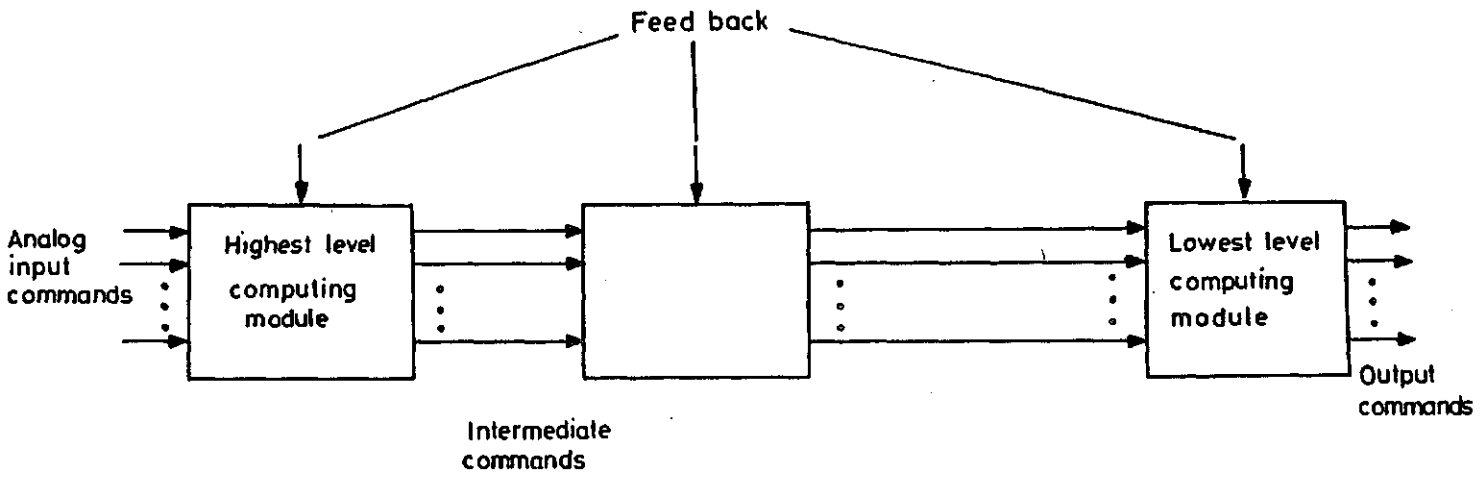


Fig. 12 Neural Net Modules

The benefits of neural net include high computation rates, provided by massive parallelism and a greater degree of fault-tolerant since there are many processing nodes each with primarily local connections. Designing artificial neural nets to solve problems and studying real biological nets may also change the way we think about problems and lead us to new insights and algorithmic improvements.

5. Software Issues Related to Multiprocessing Systems

Having discussed the various multiprocessing systems, it is worth probing further into two aspects of parallel processing: the language to program and the operating system support to handle these complex systems.

5.1. Parallel Programming Languages

One of the motivations behind the development of concurrent languages has been the structuring of software -- in particular, operating system -- by means of high-level language constructs. The need for liberating the production of real-time applications from assembly language has been another driving force. In the following discussion, we classify the concurrent high-level languages into two groups: 'traditional' languages of the von Neumann type (based on imperative style of programming) and the unconventional languages, such as data flow, functional and logic-based languages. A quick review of some of these languages is presented below.

Conventional Parallel Languages Based on the imperative style of programming, these languages are just the extensions of their sequential counterparts. A concurrent language should allow programmers to define a set of sequential activities to be executed in parallel, to initiate their evolution and to specify their interaction (refer [4] for an excellent survey on the concepts and notations for parallel programming). One important point regards the 'granularity of parallelism', i.e. the kinds of granules that can be processed in parallel. Some languages specify concurrency at statement level, and certain others at task level. Constructs for specifying inter-activity interaction are probably the most critical linguistic aspects of concurrency. Language constructs ensuring mutual exclusion are called synchronization primitives. Some of the best-known and landmark solutions that have been adopted to solve these problems are the semaphores [27], mailboxes, monitors [49] and remote procedure calls [46]. Research in this direction is towards designing new machanisms for interprocessor communication, such as ordered ports [13]. In the following discussion we restrict ourselves to a few parallel programming languages and their salient features.

Communicating Sequential Processes (CSP) [50] is a language designed especially for distributed architectures. In CSP, activities communicate via input/output commands. Communication requires both the participating processes to issue their commands. Also CSP achieves process synchronization using the input/output commands. Another interesting feature of this language is its ability to express non-determinism using guarded commands. An implementation of a subset of CSP [73] has been has been successfully attempted by the authors' research group. Their work on the design of an architecture to execute CSP is reported in [79].

Distributed Processes (DP) [46], developed by Hansen, is proposed for real-time applications controlled by microcomputer networks with distributed storage. In DP, a task consists of a fixed number of subtasks that are started simultaneously and exist forever. A process can call common procedures defined within other processes. These procedures are executed when the other processes are waiting for some conditions to become true. This is the only means of communication among the processes. Processes are synchronized by means of non-deterministic guarded commands.

Occam language [55], which is based on CSP has also been designed to support concurrent applications by using concurrent processes running in a distributed architecture. These processes communicate through channels. The Transputer [56], also developed by Inmos Corporation, supports the direct execution of this language.

Languages which adopt monitor-based solution for synchronization are oriented towards architectures with shared memory. Examples of these languages are Concurrent Pascal [45], Ada [5], and Modula [99]. These languages, in general, support strong type checking and seperate compilation, and express concurrent actions using explicit constructs.

Concurrent Pascal [45] extends the sequential programming language Pascal using the concurrent programming tools, processes and monitors. The main contribution of this language is extending the concept of the monitor using an explicit hierarchy of access rights to shared data structures that can be started in the program text and checked by a compiler.

The programming language Modula [99] is primarily intended for programming dedicated computer systems. This language borrows many ideas from Pascal, but in addition to conventional block structure, it introduces a so-called module structure. A module is a set of procedures, data types and variables where the programmer has precise control over the names that are imported from and exported to the environment [99]. Modula includes general multiprocessing facilities such as processes, interface modules and signals.

In Ada [5], we only have active components, the tasks. Information may be exchanged among tasks via entries. An entry is very similar to a procedure. The call to an entry is like a procedure call; parameters should be passed if the called entry requires them. A randezvous is said to occur when the caller is in the call state and the called task is in the accept state. After executing the entry subprogram both the tasks resume their parallel execution. Ada provides specific and elaborate protocols for task termination. Ada is designed to support reliable and efficient real-time programming.

Non-von Neumann Languages Conventional languages imitate the von Neumann computer. The dependency of these languages on the basic organization of the von Neumann machine is essentially a limitation to express and exploit parallelism [10]. These imperative languages perform a task by changing the state of a system rather than modifying the data directly. In parallel processing applications, it makes more sense to use a language with a

nonsequential semantic base. Various paradigms have been adopted and new programming languages based on these approaches have evolved. We will restrict ourselves in this paper to two such paradigms, namely the applicative and the non-procedural style of programming, and the resulting parallel versions of the languages that adopt these approaches.

Applicative languages (also referred to as functional languages) avoid side-effects, such as those caused by an assignment statement. The lack of side-effects accounts, at least partially, for the well-known Church-Rosser property, which essentially states that no matter what order of computation is chosen in executing a program, the program is guaranteed to give the same result (assuming termination). This marvellous determinacy property is invaluable in parallel systems. Another key point is that in functional languages the parallelism is implicit and supported by their underlying semantics.

A system of languages known as Functinal Programming languages (FP) [10] and Lisp [68] are two major outcomes of the applicative style of programming. Languages for data flow architectures, which avoid side-effects and encourage single assignment, are also included in the set of applicative languages. Dennis' Value oriented Algorithmic Language (VAL) [1], Arvind's Irvine Data flow language (Id) [8], and Keller's Flow Graph Language (FGL) [57] are candidate examples in this category.

Considerable work has been done by us in the area of aplicative programming languages. A high level language for data flow computers, called Data Flow Language (DFL) [72], has been proposed by us, and a compiler to convert the programs written in this language into data flow graphs has been implemented. The concepts borrowed from CSP and DP when embedded into data flow systems results in two new languages for distributed processing, namely Communicating Data Flow Channels (CDFC) and Distributed Data Flow (DDF) respectively [75]. Communication and non-determinism features have been added to FP by us [40,41] to strengthen its power as a programming language. We have also proposed that FP can be used as a language for program specification [41].

Although parallelism in a program is expressed by the functional languages in a natural way, their automatic detection and mapping to processors do not result in optimal performance. It is desirable to provide the user with the ability to explicitly express parallelism and mapping, retaining the functional style of programming. Languages which allow the programmers to annotate the parallelism and mapping scheme for the target architecture lead to optimal performance on a particular machine. Two languages developed with this motivation are ParAlfl (the Para-Functional language) [52] and Multilisp [44]. Efforts have been taken to exploit the advantages offered by the functional languages to the maximum extent by developing new machines based on non-von Neumann architecture (refer [95] for a recent survey).

Applications such as the design of knowledge base systems and natural language processing revealed the inadequacies of the conventional programming languages to offer elegant solutions. The use of predicate logic, which is a high-level human oriented language for describing problem and problem solving methods for computers, promised great scope for these applications. Logic programming languages combine simplicity with a lot of powerful features. They separate the logic and control [59], the two major components of an algorithm, and thus enable the programmer to write more correct, more easily improved and more readily adapted programs. The powerful features of these languages also include the declarative style, the unification mechanism for parameter passing and execution strategy offered by non-deterministic computation rule. The powerful execution mechanism provided by these languages is due to the non-procedural paradigm. An outcome of the research carried out in this area with these motivations is the design of the language Prolog [19].

Logic programming languages offer three kinds of parallelism, namely the 'AND', 'OR' and 'argument' parallelism [21,43]. The inability of the von Neumann architecture to efficiently execute logic programming language (in essence supporting non-procedural paradigm) has led to the design of many parallel logic programming machines [16,43,70,94,100]. Further research on these languages has led to the design of three parallel logic programming languages, the PARLOG (PARallel LOGic programming) [18], P-Prolog [103] and Concurrent Prolog [84].

With the increasing size and complexity of parallel processing systems, it becomes essental to design efficient operating systems, without which handling of such systems would be impossible. The general principles and requirements of multiprocessor operating systems are discussed in the following section.

5.2. Multiprocessor Operating Systems

The basic goals for an operating system are to provide programmer-interface to the machine, manage resources, provide mechanisms to implement policies, and facilitate matching applications to the machine. There is conceptually little difference between the operating system requirements of a multiprocessor and those of a large computer system with multiprogramming. The operating system for a multiprocessor should be able to support multiple asynchronous tasks which execute concurrently, and hence is more complex.

The functional capabilities of a multiprocessor operating system include resource allocation and management schemes, memory and data set protection, prevention of system deadlock and abnormal process termination or exception handling and processor load-balancing. Also, the operating system should be capable of providing system reconfiguration schemes to support graceful degradation of performance in the event of a failure. In the following discussion, we introduce brifly the three basic configurations, namely master-slave, separate supervision and floating-supervision systems [53].

In a 'master-slave' configuration, one processor, called the master, maintains the status of all processors in the system and apportions the work to all the slave processors. Service calls from the slave processors are sent to the master for executive service. Only one processor (the master) uses the supervisor and its associated procedures. The merit of this configuration is its simplicity. However, parallel processing system which has the master-slave configuration is susceptible to catastrophic failures, and a low utilization of the slave processors may result if the master cannot despatch processes fast enough. Cyber 170 and DEC System 10 use this mode of operation. The master-slave

configuration is most effective for special applications where the work load is well-defined.

In a 'seperate supervisor system', each processor contains a copy of a basic kernel. Each processor services its own needs. However, since there is some interaction among the processors, it is necessary for some of the supervisory code to be reentrant, unlike in the master-slave mode. Separate supervisor mode is more reliable then master-slave mode. But the replication of the kernel in all the processors causes an under-utization of memory.

The 'floating supervisor' scheme treats all the processors as well as other resources symmetrically or as an anonymous pool of resources. In this mode, the supervisor routine floats from one processor to another, although several of the processors may be executing service routines simultaneously. Conflicts in service requests are resolved by priorities. Table access should be carefully controlled to maintain the system integrity. The floating supervisor mode of operation has the advantages of providing graceful degradation and better availability of reduced capacity systems. Furthermore, it is flexible and it provides true redundancy and makes the most efficient use of available resources. Examples of the operating systems that execute in this mode are the MVS and VM in the IBM 3081 and the Hydra [102] on the C.mmp.

6. Conclusions

In this survey, we have identified the various issues involved in parallel processing systems. Approaches followed to solved the associated problems have also been discussed and their relative merit are put forth. The principles and the requirements of language and operating system support for complex multiprocessing systems are elaborately described. For the wide spectrum of architectures proposed in the litarature, their design principles and salient features are brought out in a comparative menner.

While the envisaged potentials offer a promising scope for parallel processing systems for many applications, hardly a few systems are commercialized. The reasons fot this is the lack of good software support for these systems. Design of intelligent compilers which can identify parallel subtasks in a program (written in a sequential language), schedule the subtasks to the processing elements and manage communication among the scheduled tasks, is a step toward this end. Although there are many existing proposals in this line, none of them seems to achive all the three goals in an integrated manner, relieving the burden from the user completely.

Another question that remains unanswered is whether or not to continue with von Neumann approach for building complex parallel processing machines. While familiarity and the past experience with control flow model make it a proponent candidate, its inherent inefficiencies, such as the explicit specification of control and global updatable memory, limit its capabilities. Although data-driven and demand-driven computers exploit maximum parallelism in a program, their complex structure and inadequate software support force the designer to have a second throught on these approaches.

With the advent of VLSI technology and RISC design, dedicated architectures are becoming more and more popular. However, the inapplicability of these systems to a variety of applications causes a serious concern. At the other end of the spectrum, we have general purpose parallel processing systems which give degrade performance due to the mismatch of the architecture and algorithm, and the reconfigurable machines. Considerable and on design efficient algorithms (for general purpose computing systems) which will bridge the gap between the application program and architecture.

Finally, the research on neural computer and molecular machines is at its infancy. Modeling the neural circuits and understanding the functioning of human brain have to be considerable refined before one could make use them for building high speed computing systems.

The vastness of this fascinanting area in which active research is underway, and the innumerable problems that remain to be solved are themselves standing evidences for the promising future of parallel processing. With the ever-growing greed for very high speed of computing, and with the inability of the switching devices to cope up with the need, parallel processing techniques seem to be the only alternative.

7. References

1. W.B. Ackerman and J.B. Dennis, "VAL - A Value Oriented Algorithmic Language, Preliminary Reference Manual", Tech. Rep. TR-218, Lab. for Computer Science, M.I.T. Cambridge, MA, June 1979.

2. J.S. Albus, Brains, Behavior and Robotics, Byte Books, McGraw-Hill, New York, NY, 1981.

3. M. Ammamiya, R. Hasegawa and H. Mikami, "List Processing and Data Flow Machines", Lecture Note Series, No.436, Research Institute for Mathematical Sciences, Kyoto University, September 1980.

4. G.R. Andrews and F.B. Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol.15, No.1, pp.3-43, March 1983.

5. ANSI/MIL-STD 1815A, Reference Manual for Ada Programming Language, 1983.

6. C.N. Arnold, "Performance Evaluation of Three Automatic Vectorizer Packages", Proc. Int'l Conf. Parallel Processing, pp.235-243, 1982.

7. Arvind and K.P. Gostelow, "A Computer Capable of Exchanging Processors for Time", Proc. IFIP Congress, pp.849-854, 1977.

8. Arvind, K.P. Gostelow and W. Plouffe, "An Asynchronous Programming Language and Computing Machine", Tech. Rep. TR-114a, Dept. Information and Computer Science, University of California, Irvine, CA, December 1978.

9. J. Backus, "Reduction Languages and Variable Free Programming", Rep. RJ.1010, IBM T.J. Watson Research Center, Yorktown Heights, NY, April 1972.

10. J. Backus, "Can Programming be Liberated from von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ASM, Vol.21, No.8, pp.613-641, August 1978.

11. W.L. Bain Jr. and S.R. Ahuja, "Performance Analysis of High-Speed Digital Busses for Multiprocessing Systems", Proc. 8th Ann. Symp. Computer Architecture, pp.107-131, May 1981.

12. G.H. Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick and R.A. Stokes, "The ILLIAC IV Computer", IEEE Trans. on Computers, Vol.C-17, No.8, pp.746-757, August 1968.

13. J. Basu, L.M. Patnaik and A.K. Goswami, "Ordered Ports - A Language Construct for High Level Distributed Programming", The Computer Journal, Vol.30, 1987.

14. K.E. Batcher, "Design of a Massively Parallel Processor", IEEE Trans. on Computers, Vol.C-29, No.9, pp.836-840, September 1980.

15. K. Berkling, "Reduction Languages for Reduction Machines", Proc. 2nd Int'l Symp. Computer Architecture, Januar 1975.

16. L. Bic, "A Data-Driven Model for Parallel Implementation of Logic Programs", Dept. Information and Computer Science, University of California, Irvine, CA, January 1984.

17. R.P. Case and A. Padegs, "Architecture of the IBM System 370", Communication of the ACM, Vol.21, No.1, pp.73-96, January 1978.

18. K. Clark and S. Gregory, "PARLOG: PARallel Programming in LOGic", ACM Trans. on Programming Languages and Systems, Vol.8, No.1, pp.1-49, January 1986.

19. W.F. Clocksim and C.S. Mellish, Programming in Prolog, Springer-Verlag, New York, NY, 1984.

20. D. Comte, A. Durrieu, O. Gelly, A. Plas and J.C. Syre, "TEAU 9/7 SYSTEME LAU - Summary in English", CERT Tech. Rep. #1/3059, Centre d'Etudes et de Recherches de Toulouse, October 1976.

21. J.S. Conery and D.F. Kibler, "AND Parallelism and Nondeterminism in Logic Programs", New Generation Computing, Vol.3, No.1, pp.43-70, 1985.

22. M. Cornish, "The TI Data Flow Architectures: The Power of Concurrency for Avionics", Proc. 3rd Conf. Digital Avionics Systems, pp.19-25, November 1979.

23. A.L. Davis, "The Architecture and System Method of DDM1: A Recursive Structured Data Driven Machine", Proc. 5th Ann. Symp. Computer Architecture, pp.210-215, April 1978.

24. A.L. Davis and R.M. Keller, "Data Flow Graphs", Computer, Vol.15, No.2, pp.26-41, February 1982.

25. Deneclor, Inc. Heterogeneous Element Processor: Principles of Operation, April 1981.

26. J.B. Dennis and D.P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor", Proc. 2nd Int'l Symp. Computer Architecture, pp.126-132, January 1975.

27. E.W. Dijkstra, A Discipline of Programming, Prentice-Hall Inc., Englewood Cliffs, NJ, 1976.

28. B.L. Drake, F.L. Luk, J.M. Speiser and J.J. Symguski, "SLAPP: A Systolic Linear Algebra Parallel Processor", Computer, Vol.20, No.7, pp.45-49, July 1987.

29. M.J. Duff, "CLIP-4", in Special Computer Architecture for Pattern Recognition, K.S. Fu and T. Ichigawa (Eds.) CRC Press, 1982.

30. T.Y. Feng, "A Survey of Interconnection Networks", Computer, Vol.14, No.12, pp.12-27, December 1981.

31. A.L. Fisher, H.T. Kung, L.M. Monier, H. Walker and Y. Dohi, "Architecture of the PSC: A Programmable Systolic Chip", Proc. 10th Ann. Symp. Computer Architecture, 1983.

32. J.D. Foley and A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley, Reading, MA, 1982.

33. J.A.B. Fortes and B.W. Wah, "Systolic Arrays -- From Concept to Implementation", Guest Editor's Introduction, Computer, Vol.20, No.7, pp.12-17, July 1987.

34. D.D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", Computer, Vol.18, No.6, pp.9-27, June 1985.

35. C. Ghezzi, "Concurrency in Programming Languages: A Survey", Parallel Computing, Vol.2, pp.229-241, 1985.

36. D. Ghosal and L.M. Patnaik, "Performance Analysis of Hidden Surface Removal Algorithms on MIMD Computers", Proc. IEEE Int'l Conf. on Computers, Systems and Signal Processing, pp.1666-1675, December 1984.

37. D. Ghosal and L.M. Patnaik, "Parallel Polygon Scan Conversion Algorithms: Performance Evaluation on a Shared Bus Architecture", Computers and Graphics, Vol.10, No.1, pp.7-25, 1986.

38. D. Ghosal and L.M. Patnaik, "SHAMP: An Experimental Multiprocessor System for Performance Evaluation of Parallel Algorithms", Multiprocessing and Microprogramming, Vol.19, No.3, pp.179-192, 1987.

39. K.P. Gostelow and R.E. Thomas, "Performance of a Data Flow Computer", Tech. Rep. 127a, Dept. Information and Computer Science, University of Califirnia, Irvine, CA, October 1979.

40. A.K. Goswami and L.M. Patnaik, "An Algebraic Approach Towards Formal Development of Functional Programs", accepted for publication, New Generation Computing.

41. A.K. Goswami, "Non-Determinism and Communication in Functional Programming Systems: A Study in Formal Program Development", Ph.D. Dissertation, Dept. Computer Science and Automation, Indian Institute of Science, Bangalore, India, October 1985.

42. A. Gottlieb, R. Grishman, C.P. Krushkal, K.P. McAaliffe, L. Randolph and M. Snir, "The NYU Ultracomputer -- Designing an MIMD Shared Memory Parallel Computer", IEEE Trans. on Computers, Vol.C-32, No.2, pp.175-189, February 1983.

43. Z. Halim, "A Data-Driven Machine for OR Parallel Evaluation of Logic Programs",

New Generation Computing, Vol.4, No.1, pp.5-33, 1986.

44. R.H. Halstead Jr., "Multilisp: A Language for Concurrent Symbolic Computation", ACM Trans. on Programming Languages and Systems, October 1985.

45. P.B. Hansen, "The Programming Language Concurrent Pascal", IEEE Trans. on Software Engineering, Vol.SE-1, No.2, pp.199-209, June 1975.

46. P.B. Hansen, "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol.21, No.11, pp.934-941, November 1978.

47. J.P. Hayes, R. Jain, W.R. Martin, T.N. Mudge, L.R. Scott, K.G. Shin and Q.F. Stout, "Hypercube Computer Research at the University of Michigan", Proc. Second Conf. Hypercube Multiprocessors, September/October 1986.

48. W.D. Hillis, The Connection Machine, M.I.T. Press, Cambridge, MA, 1986.

49. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", Communication of the ACM, Vol.17, No.10, pp.549-557, October 1974.

50. C.A.R. Hoare, "Communicating Sequential Processes", Communication of the ACM, Vol.21, No.8, pp.666-677, August 1978.

51. J.J. Hopfieed and D.W. Tank, "Computing with Neural Circuits", Science, pp.625-633, August 1986.

52. P. Hudak, "Para-Functional Programming", Computer, Vol.19, No.8, pp.60-70, August 1986.

53. K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company, New York, NY, 1984.

54. A.K. Jones and E.F. Gehringer, "Cm* Multiprocessor Project: A Research Review", Tech. Rep. CMU-CS-80-131, Carnegie-Mellon University, July 1980.

55. Inmos Ltd., Occam Reference Manual, 1986.

56. Inmos Ltd., Transputer Reference Manual, Ref. No. 72TRN 04802, 1986.

57. R.M. Keller, S. Patil and G. Lindstrom, "A Loosely Coupled Applicative Multiprocessing System", Proc. Nat. Computer Conf., AFIPS, pp.861-870, 1979.

58. W.E. Kluge and H. Schlutter, "An Architecture for the Direct Execution of Reduction Languages", Proc. Int'l Workshop High-Level Language Computer Architecture, pp.174-180, May 1980.

59. R.A. Kowalski, "Algorithm = Logic + Control", Communication of the ACM, Vol.22, No.7, pp.424-435, July 1979.

60. D. Krishnan and L.M. Patnaik, "Systolic Architecture for Boolean Operations on Polygons and Polyhedra", Eurographics, The Europian Association for Computer Graphics, Vol.6, No.3, July 1987.

61. C.P. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors", Proc. Int'l Conf. Parallel Processing, pp.236-240, August 1984.

62. H.T. Kung and S.W. Song, "A Systolic 2D Convolution Chip", Dept. Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-81-110, March 1981.

63. H.T. Kung, "Why Systolic Architectures?", Computer, Vol.15, No.1, pp.37-46, January 1982.

64. S.Y. Kung, S.C. Lo, S.N. Jean and J.N. Hwang, "Wavefront Array Processors -- Concept to Implementation", Computer, Vol.20, No.7, pp.18-33, July 1987.

65. R.P. Lippmann, "An Introduction to Computing with Neural Nets", IEEE ASSP, pp.4-20, April 1987.

66. G.A. Mago, "A Network of Multiprocessors to Execute Reduction Languages", Int'l J. Computer and Information Science, Part 1: Vol.8, No.5, pp.349-385, Part 2: Vol.8, No.6, pp.435-571, 1979.

67. P.C. Mathias and L.M. Patnaik, "A Systolic Evaluation of Linear, Quadratic and Cubic Expressions", accepted for publication, Journal of Parallel and Distributed Computing.

68. J. McCarthy, et.al., LISP 1.5 Programmer's Reference Manual, M.I.T. Press, Cambridge, MA, 1965.

69. C.A. Mead and L.A. Conway, Introduction to VLSI Systems, Addison Wesley, Reading, MA, 1980.

70. R. Onai, M. Aso, H. Shimizu, K. Masuda and A. Matsumoto, "Architecture of a Reduction Based Parallel Inference Machine: PIM-P", New Generation Computing, Vol.3, No.2, pp.197-228, 1985.

71. D.A. Padua, D.J. Kuck and D.L. Lawrrie, "High Speed Multiprocessor and Compilation Techniques", IEEE Trans. on Computers, Vol.C-29, No.9, pp.763-776, September 1980.

72. L.M. Patnaik, P. Bhattacharya and R. Ganesh, "DFL: A Data Flow Language", Computer Languages, Vol.9, No.2, pp.97-106, 1984.

73. L.M. Patnaik and B.R. Badrinath, "Implementation of CSP-S for Description of Distributed Algorithms", Computer Languages, Vol.9, No.3/4, pp.193-202, 1984.

74. L.M. Patnaik, R. Govindarajan and N.S. Ramadoss, "Design and Performance Evaluation of EXMAN: An EXtended MANchester Data Flow Computer", IEEE Trans. on Computers, Vol.C-35, No.3, pp.229-244, March 1986.

75. L.M. Patnaik and J. Basu, "Two Tools for Interprocess Communication in Distributed Data Flow Systems", The Computer Journal, Vol.29, No.6, pp.506-521, December 1986.

76. A. Plas, et.al., "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment", Proc. 1976 Int'l Conf. Parallel Processing, pp.293-302, August 1976.

77. C.V. Ramamoorthy and H.F. Li, "Pipeline Architectures", Computing Surveys, Vol.9, No.1, pp.61-102, March 1977.

78. C.P. Ravikumar and L.M. Patnaik, "Parallel Placement Based on Simulated Annealing",

IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors, October 1987.

79. C.P. Ravikumar and L.M. Patnaik, "An Architecture for CSP and its Simulation", Int'l Conf. Parallel Processing, 1987.

80. C. Rieger, "ZMOB: Doing it in Parallel", Proc. Workshop Computer Architecture for PAIDM, pp.119-125, 1981.

81. A. Rosenfield, "Parallel Image Processing Using Cellular Arrays", Computer, Vol.16, No.1, pp.14-20, January 1983.

82. R.M. Russel, "The Cray-1 Computer System", Communications of the ACM, Vol.21, No.1, pp.63-72, January 1978.

83. C.L. Seitz, "The Cosmic Cube", Communication of the ACM, Vol.28, No.1, pp.22-33, January 1985.

84. E.Y. Shapiro, "Concurrent Prolog: A Progress Report", Computer, Vol.19, No.8, pp.44-58, August 1986.

85. L. Snyder, "Introduction to Configurable Highly Parallel Computer", Computer, Vol.15, No.1, pp.47-64, January 1982.

86. J.S. Squire and S.M. Palais, "Programming and Design Considerations for a Highly Parallel Computer", Proc. AFIP Conf., Vol.23, pp.395-400, 1963.

87. J. Silc and B. Robič, "Efficient Static Dataflow Architecture for Specialized Computatins", Proc. 12th IMACS World Congress on Scientific Computing, Paris, France, July 1988.

88. K.R. Traub, "A Compiler for M.I.T. Tagged-Token Data Flow Architecture", M.S. Thesis, M.I.T., Cambridge, MA, 1986.

89. P.C. Treleaven, "Principle Components of a Data Flow Computer", Proc. 1978 Euromicro Symp., pp.366-374, October 1978.

90. P.C. Treleaven and G.F. Mole, "A Multiprocessor Reduction Machine for User-Defined Reduction Languages", Proc. 7th Int'l Symp. Computer Architecture, pp.121-130, 1980.

91. P.C. Treleaven, D.R. Brownbridge and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", Computing Surveys, Vol.14, No.1, pp.93-143, March 1982.

92. D.A. Turner, "A New Implementation Technique for Applicative Languages", Software Practice and Experience, Vol.9, No.5, pp.31-49, September 1979.

93. J.D. Ullman, Computational Aspect of VLSI, Computer Science Press, 1984.

94. S. Umeyama and K. Tamura, "Parallel Execution of Logic Programs", Proc. 10th Ann. Symp. Computer Architecture, pp.349-355, 1983.

95. S.R. Vehdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages", IEEE Trans. on Computers, Vol.C-33, No.12, pp.1050-1071, December 1984.

96. A.G. Venkataramana and L.M. Patnaik, "Design and Performance Evaluation of a Systolic Architecture for Hidden Surface Removal", accepted for publication, Computer and Graphics.

97. A.G. Venkataramana and L.M. Patnaik, "Systolic Architecture for B-Spline Surfaces", accepted for publication, International Journal of Parallel Programming.

98. I. Watson and J. Gurd, "A Prototype Data Flow Computer with Token Labelling", Proc. Nat. Computer Conf. New York, AFIPS Press, pp.623-628, 1979.

99. N. Wirth, "Modula: A Programming Language for Modular Multiprogramming", Software Practice and Experience, Vol.7, No.1, pp.3-35, January 1977.

100. M.J. Wise, "A Parallel PROLOG: The Construction of Data-Driven Model", ACM Conf. LISP and Functional Programming, 1982.

101. W.A. Wulf and C.G. Bell, "C.mmp -- A Multi-miniprocessor", Proc. AFIPS Fall Joint Computer Conf., Vol.41, pp.765-777, 1972.

102. W.A. Wulf, et.al., "Overview of the Hydra Operating System", Proc. 5th Symp. Operating System Principles, pp.122-131, November 1975.

103. R. Yang and H. Aiso, "P-Prolog: A Parallel Logic Language Based on Exclusive Relation", New Generation Computing, Vol.5, No.1, pp.79-95, 1987.