Module 6

Multithreaded and Data Flow Architectures

MULTI-THREADED AND DATA FLOW ARCHITECTURE

Ques 1) What is multi-threaded processor?

Ans: Multi-Threaded Processors
Consider one processor in a multiprocessor system needs two memory loads of two variables from two remote processors. The issuing processor will use these variables simultaneously in one operation.

In case of large-scale MPP (massively parallel processors) systems, the following two problems arise:

1) **Remote-load Latency Problem:** When one processor needs some remote loading of data from other nodes, then the processor has to wait for these two remote load operations. The longer the time taken in remote loading, the greater will be the latency and idle period of the issuing processor.
2) **Synchronization Latency Problem:** If two concurrent processes are performing remote loading, then it is not known by what time two processes will load, as the issuing processor needs two remote memory loads by two processes together for some operation. That means two concurrent processes return the results asynchronously and this causes the synchronization latency for the processor.

These problems increase in the design of large-scale multiprocessors such as MPP as discussed above. Therefore, a solution for optimizing this latency should be acquired at. The concept of Multithreading offers the solution to these problems.

When the processor activities are multiplexed among many threads of execution, then problems are not occurring. In single threaded systems, only one thread of execution per process is present. But if we multiplex the activities of process among several threads, then the multithreading concept removes the latency problems.

In the above **example**, if multithreading is implemented, then one thread can be for issuing a remote load request from one variable and another thread can be for remote load for second variable and third thread can be for another operation for the processor and so on.

Ques 2) What is latency-hiding? What are the solutions to the latency problem?

Ans: Latency-Hiding
Basically there are many scalable parallel systems which use distributed memory architecture. As the memory is distributed in these systems, so any PE in multiprocessor system may access the remote memory. As the memory is remote (i.e., not local to PE), so some time is required to fetch that memory data. This time delay is called memory latency. This delay is due to the delays of interconnection hardware, communication delay and memory response time.

The latency hiding is a technique that hides the time consumed in micro-tasks by communication and/or remote memory.

Solution to the Latency Problem
To improve the performance of large scale computer systems, it is necessary to enhance their scalability and programmability. As far as the programmability is concerned, the effective performance of the program depends upon many factors and memory latency is one of them. There are many ways to tackle this problem of latency. Out of this, the three important mechanisms are as follows:

1) Latency avoidance mechanism,
2) Latency reduction mechanism, and
3) Latency hiding mechanism.

Ques 3) Discuss the latency avoidance and latency reduction mechanism.

Ans: Latency Avoidance Mechanism
Latency Avoidance mechanism emphasises on improving and enhancing the temporal and spatial locality of the program. This will reduce the cache miss operations and hence latency is avoided.

There are many techniques to avoid latency. One of the simplest techniques is to properly organise the user application, so as to improve the localities. This

Scanned by CamScanner

technique requires that the parallel computing system should provide the supporting architecture such as cache coherency techniques, fast messing technique. In yet another technique, user can explore the locality in the program by use of specific languages. The programmer can represent certain data structure in a way to improve the locality properties. The other techniques involve the compiler optimisation to improve the locality:

Latency Reduction Mechanism

The latency avoiding mechanism can be effective upto certain extent. At some critical point, it becomes difficult to enhance the data locality. At this stage, one must give attention on reducing the latency. The latency can be reduced by using appropriate communication hardware and software. The scalable parallel system should use efficient communication hardware such as network interfaces and interconnection network.

Ques 4) Outline the latency hiding techniques/mechanism.

Ans: Latency Hiding Techniques

This technique emphasises on hiding the latency by the use of overlapping operations during latency. This means that the operation of remote memory access is overlapped with another useful operation by the processor. Hence, the processor need not wait till the remote data is accessed.

There are four main techniques to hide latency. They are as follows:

1) **Prefetching Techniques:** Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. Prefetching can be classified based on whether it is binding or non-binding, and whether it is controlled by hardware or software.

   With binding prefetching, the value of a later reference (e.g., a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching may result in a significant loss in performance due to such limitations. In contrast, non-binding prefetching also brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value. Hardware-controlled prefetching includes schemes such as long cache lines and instruction lookahead.

   The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications, while instruction lookahead is limited by branches and the finite lookahead buffer size.

   With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large. The disadvantages of software control include the extra instruction overhead required to generate the prefetches, as well as the need for sophisticated software intervention. In our study, we concentrate on non-binding software controlled perfecting.

2) **Use of Coherent Caches:** In large-scale computers, the caches are distributed and they are made coherent by implementing cache coherence protocols. Please note that the small-scale multiprocessor systems use snoopy protocol whereas large scale systems use directory based protocol. Coherent caches help to hide latency by increasing the cache hit ratio for read operations. When the cache miss occurs, the number of cycles are wasted. The benefit of using distributed coherent caches is to reduce the cache miss.

3) **Use of Relaxed Memory Consistency Models:** There is different memory models developed for shared-memory multiprocessor systems. These memory models specify the ordering of global memory reads and memory writes generated by the same processor or different processors. The ordering of these reads/writes may cause the consistency problem. A memory consistency model indicates how the memory reads or writes generated by one processor should be observed by other processor in the multiprocessor system.

   There are sequential consistency and relaxed consistency memory models. In sequential consistency model, the ordering of shared memory reads, writes and exchanges by all processors in system must be in manner to maintain the program order in the individual processor. In relaxed memory model, the sequential constraint of ordering is relaxed. The different relaxed memory models are weak consistency model, processor consistency model and release consistency model. The weak consistency model has only restricted the sequential ordering to synchronising variable which are recognised by hardware. But between the synchronisation points, this model relaxes the ordering of read/write operations, i.e., read/write operations may not follow program order.
   In processor consistency relaxed model, there is need of a sequential consistency for writes issued within each processor but writes from different

Scanned by CamScanner

processors may be out of program order. This model relaxes the write orders from different processors. It enables the pipelined memory accesses which hides the latency. The release consistency model, also allows buffering and pipelined memory accesses between two synchronisation accesses which hides the latency.

Thus, the relax memory consistency models relax the ordering of memory events such as reads, writes to some extent. This relaxation allows the reordering or memory reads/writes and facilitates the buffering and pipelined memory accesses. This reduces the long latencies of remote memory accesses.

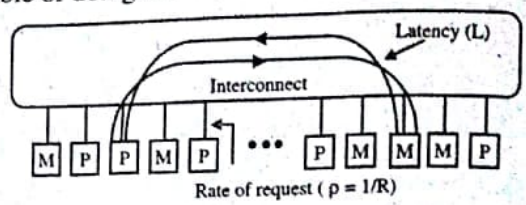
4) **Use of Multithreading to do Context Switching to Hide the Latency:** The multithreaded processor, also called as the multiple context processor is designed in a way to support multithreading. The processor has multiple sets context storage. Therefore, when the processor executes a single thread, which requires a data from remote memory, then this processor does not have to wait. It simply switches to another thread and executes it. Thus, the memory access waiting period due to one thread is overlapped by the execution of another thread. If this new thread also generates cache miss, i.e., if the required data is not in the cache then processor again switches to new thread and starts executing it. By that time, memory data required by initial thread may be available. Thus, the processor always remains busy in doing important work during memory latency.

Ques 5) Discuss the principles of multithreading.

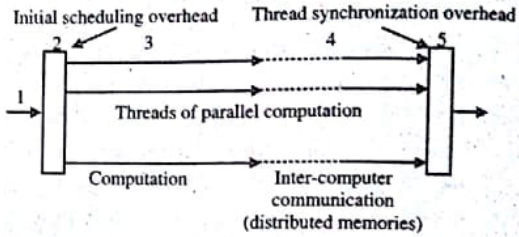
Ans: Principles of Multithreading

Bell (1992) has described the structure of the multithreaded parallel computations model shown in **figure 6.1 (b).** The computation starts with a sequential thread followed by supervisory scheduling where the processors being threads of computation, by intercomputer messages that update variables among the nodes when the computer has a distributed memory, and finally by synchronisation prior to beginning the next unit of parallel work. The communication overhead period inherent in distribution memory structures is usually distributed throughout the computation and is possibly completely overlapped. Message-passing overhead (send and received calls) in multi-computers can be reduced by specialised hardware operating in parallel with computation. Communication bandwidth limits granularity, since a certain amount of data has to be transferred with other nodes in order to complete a computational grain.

Message-passing calls and synchronisation are non-productive. Fast mechanisms to reduce or to hide these delays are therefore needed. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.



(a) The Architecture Environment



(b) Multithreaded computation model

Figure 6.1: Multithreaded Architecture and Its Computation Model for a Massively Parallel Processing System

Ques 6) What are the multithreading issues?

Ans: Multithreading Issues

We have already seen a multithreading computation model in previous section. Now, let us examine a multithreaded system, architecture environment. It is shown in **figure 6.2.**

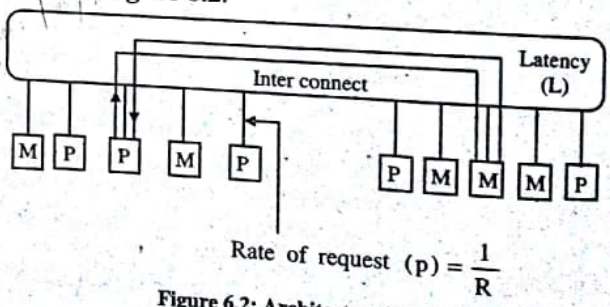


Figure 6.2: Architecture Environment

The distributed memories form a global address space. Four machine parameters are defined below to analyse the performance of this network:

1) **Latency (L):** This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty and delays caused by contentions in split transactions.
2) **Number of Threads (N):** This is the number of threads that can be interleaved in each processor. A thread is represented by a context, consisting of a program counter (PC), a register set and the required context status words.
3) **Context Switching Overhead (C):** This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanisms and the amount of processor states devoted to maintaining active threads.

Scanned by CamScanner

4) **Interval Between Switches (R):** This refers to the cycles between switches triggered by remote reference. The inverse, $P = \frac{1}{R}$ is called the rate of requests for remote accesses. This reflects a combination of program behaviour and memory systems design.

Ques 7) What is the problem of asynchrony?

Ans: Problem of Asynchrony

Massively parallel processors operate asynchronously in a network environment. The asynchrony triggers two fundamental latency problems:

1) Remote Loads
2) Synchronizing Loads

Example: Latency Problems for Remote Loads or Synchronising Loads

The remote load situation is illustrated in **figure 6.3 (a).** Variables A and B are located on nodes N2 and N3 respectively. They need to be brought to node N1 to compute the difference A – B in variable C. The basic computation demands the execution of two remote loads (rload) and then the subtraction.

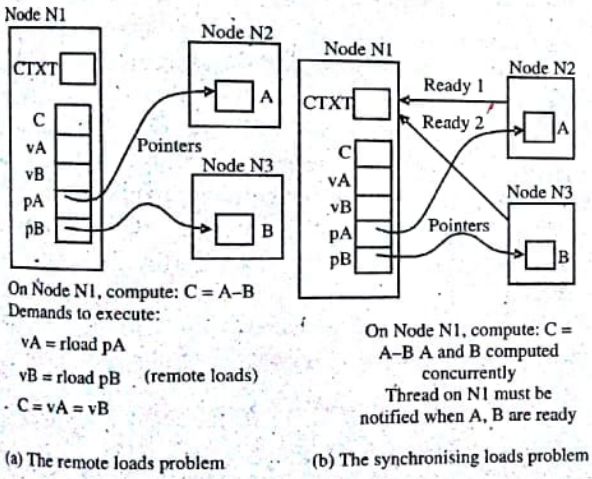


Figure 6.3: Two Common Problems Caused by Asynchrony and Communication Latency in Massively Parallel Processors

Let pA and pB be the pointers to A and B, respectively. The two rloads can be issued from the same thread or from two different threads. The context of the computation on N1 is represented by the variable CTXT. It can be a stack pointer, a frame pointer, a current-object pointer, a process identifier, etc. In general, variable names line vA, vB, and C are interpreted relative to CTXT.

In **figure 6.4 (b)**, the idling due to synchronising loads is illustrated. In this case, A and B are computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to read. The ready signals (Ready1 and Ready2) may reach node N1 asynchronously. This is a typical situation in the producer-consumer problem. Busy-waiting may result.

The key issue involved in remote loads in how to avoid idling in node N1 during the load operations. The latency caused by remote loads is an architectural property. The latency caused by synchronising loads also depends on scheduling and the time it takes to compute A and B, which may be much longer than the transit latency. The synchronisation latency is often unpredictable, while the remote-load latencies are often predictable.

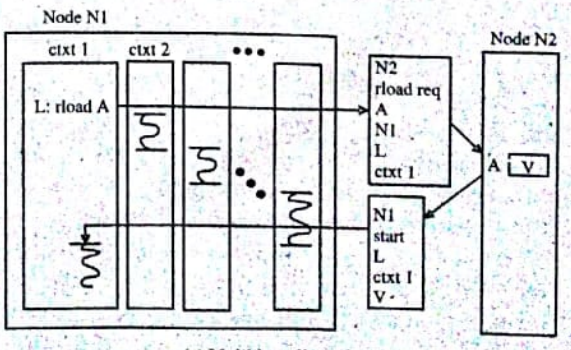
Ques 8) What are the multithreaded solutions?

Ans: Multithreaded Solutions

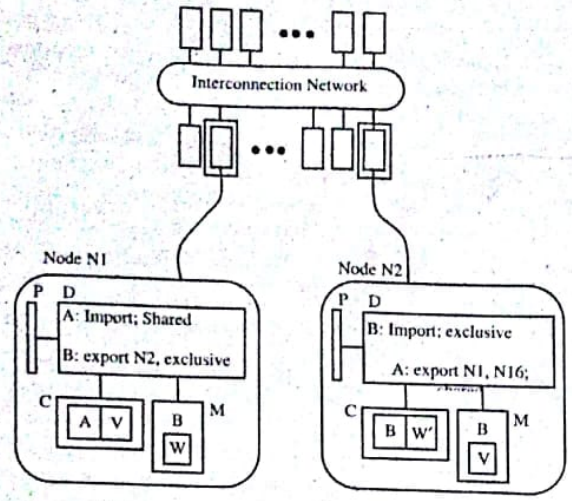
This solution to asynchrony problems is to multiplex among many threads – when one thread issues a remote-load request, the processor begins work on another thread, and so on (**figure 6.5 (a)**. Clearly, the cost of thread switching should be much smaller than that of the latency of the remote load, or else the processor might as well wait for the remote load's response.

As the internode latency increases, more threads are needed to hide it effectively. Another concern is to make sure that messages carry continuations. Suppose, after issuing a remote load from thread T1 (**figure 6.59 (a)**, we switch to thread T2, which also issues a remote load. The responses may not return in the same order. This may be caused by requests traveling different distance, through varying degrees of congestion, to destination nodes whose loads differ greatly, etc. One way to cope with the problem is to associate each remote load and response with an identifier for the appropriate thread, so that it can be re-enabled on the arrival of a response. These thread identifiers are referred to as continuations on messages. A large continuation name space should be provided to name an adequate number of threads waiting for remote responses.

The size of the hardware-supported continuation is a name space varies greatly in different system design – from 1 in the Dash, 4 in the Alewife, 64 in the HEP, and 1024 in the Tera, to the local memory address space in the Monsoon, Hybrid Dataflow/von Neumann, MDP, and *T. Of course, if the hardware-supported name space is small, one can always virtualise it by multiplexing in software, but this has an associated overhead.



Scanned by CamScanner



P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed caching

Figure 6.5: Two Solutions for Overcoming the Asynchrony Problems

Ques 9) What are multiple context processors?

Ans: Multiple Context Processors

While prefetching is a mostly software solution to hiding latency, using multiple context processors is a mostly hardware solution. The basic idea is that when a long-latency operation such as a cache miss is encountered, the processor switches to another context and continues computation while the long-latency operation is in progress. Multiple-context processors take advantage of intercontext parallelism to hide latency.

Several processes are assigned to each processor and are switched in hardware when a long-latency operation is encountered. With processor caches, the interval between long-latency operations become large enough for just a handful of contexts to effectively hide most of the latency. This situation is in contrast to the early cacheless multiple-context processors such as the HEP, where context switches occurred on every cycle and thus a large number of contexts were required to keep processor utilisation high. More recent work by **Laudon** shows that these two schemes can be effectively combined. The result is a processor that is able to switch contexts on every cycle, but does not force a context switch on every cycle. Latency tolerance is good, yet single-context performance is not compromised.

The performance gain to be expected from multiple-context processors depends on several factors:

1) First, there is the number of contexts. With more contexts available, the processor is less likely to be out of ready-to-run contexts. The number of contexts, however, is constrained by hardware costs and available parallelism in the application. Using more contexts also leads to a less desirable point on the speedup graph of an application.
2) Second, there is the context switch overhead. If the overhead is a sizeable fraction of the typical run length (time between misses), a significant fraction of time may be wasted switching contexts. Shorter context switch times, however, require a more complex and possibly slower processor.
3) Third, the performance depends on the application behaviour. Application with clustered misses and irregular miss latencies make it difficult to completely overlap computation of one context with memory accesses of other contexts. Multiple context processors thus achieve a lower processor utilisation on these programs than on applications with more regular miss behaviour.
4) Fourth, multiple contexts themselves affect the performance of the memory system. The different contexts share a single processor cache and can interfere with each other, both constructively and destructively. Also, as is the case with weaker consistency models and prefetching, the memory system is more heavily loaded by multiple contexts, and thus latencies may increase.
5) Another practical issue is that there are no commercially available multiple-context processors. The only microprocessor with multiple contexts is SPARCLE, a modified SPARC processor that uses a register window for each context and is used in the MIT Alewife multiprocessor. Unless multiple contexts are implemented in a state-of-the-art microprocessor, their performance will remain of primarily academic interest. And since the benefit to uniprocessors is limited, the addition of multiple contexts to state-of-the-art microprocessors is not very likely.

Ques 10) What are the context-switching policies?

Ans: Context-Switching Policies

There are four switching policies:

1) **Switch on Cache Miss:** This policy corresponds to the case where a context is pre-empted when it causes a cache miss. In this case, R is taken to be the average interval between misses, and L the time required to satisfy the miss.
2) **Switch on Every Load:** This policy allows switching on every load, independent of whether it will cause a miss or not. In this case, R represents the average interval between loads.
3) **Switch on Every Instruction:** This policy allows switching on every instruction, independent of whether it is a load or not. It interleaves the instructions from different threads on a cycle-by-cycle basis.

Scanned by CamScanner

4) **Switch on Block of Instruction:** Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

Ques 11) Explain the concept of fine-grain multicomputer.

Ans: Fine-Grain Multicomputer
The architecture of the multicomputer is described and contrasted with that of the shared-memory multiprocessor, and the concept of grain size (which depends on the size of the individual memories) is explained. Medium-grain and fine-grain multicomputers, with nodes containing megabytes and tens of kilobytes of memory, respectively.

Fine-grain multi-computers, by operating at a fine grain, increase the amount of concurrency. Programming is simplified because programs may be partitioned into natural units of methods and objects and these objects are addressed uniformly whether they are local or remote. The construction of these machines poses challenging problems in reducing overhead, increasing communication bandwidth, and developing resource management techniques.

Message-passing multi-computers are used to execute medium-grain programs with approximately 10-ms task size as in the iPSC/1. In order to build MPP systems, one may have to explore a higher degree of parallelism by making the task grain size even smaller.

Fine-grain parallelism was utilized in SIMD or data-parallel computers like the CM-2 or on the message – driven J-Machine and Mosaic C.

Ques 12) What is fine-grain parallelism?

Ans: Fine-Grain Parallelism
In fine-grained parallelism, a program is broken down to a large number of small tasks. These tasks are assigned individually to many processors. The amount of work associated with a parallel task is low and the work is evenly distributed among the processors. Hence, fine-grained parallelism facilitates load balancing. As each task processes less data, the number of processors required to perform the complete processing is high. This in turn, increases the communication and synchronization overhead.

Fine-grained parallelism is best exploited in architectures which support fast communication. Shared memory architecture which has a low communication overhead is most suitable for fine-grained parallelism.

It is difficult for programmers to detect parallelism in a program, therefore, it is usually the compilers' responsibility to detect fine-grained parallelism.

An example of a fine-grained system (from outside the parallel computing domain) is the system of neurons in our brain.

Connection Machine (CM-2) and J-Machine are examples of fine-grain parallel computers that have grain size in the range of 4-5 μs.

Ques 13) Write short note on the following fine-grain multi-computers:
1) MIT J-Machine
2) Caltech Mosaic C

Ans: MIT J-Machine
The MIT J-Machine multicomputer has been constructed to study the role of a set of primitive mechanisms in providing efficient support for parallel computing. Each J-Machine node consists of an integrated multicomputer component, the Message-Driven Processor (MDP), and 1MByte of DRAM.

The MDP incorporates a 36-bit integer processor (32 bits of data augmented with 4 bits of tag), a memory management unit, a router for a 3-D mesh network, a network interface, a 4K-word 36-bit SRAM, and an ECC DRAM controller in a single 1.1M transistor VLSI chip.

Rather than being specialized for a single model of computation, the MDP incorporates primitive mechanisms for communication, synchronization, and naming that permit it to efficiently support threads with 50 to 150 instructions which exchange small data objects frequently with low latency and synchronize quickly. A 512 node J-Machine is in daily use at MIT and will be expanded to 1024 nodes in March 1993

The MDP supports communication using a set of send instructions for message formatting, a fast network for delivery, automatic message buffering, and task creation upon message arrival. A series of send instructions is used to inject messages at a rate of up to 2 words per cycle.

Messages are routed through the 3D-mesh network using deterministic, e-cube, wormhole routing. The channel bandwidth is 0.5 words / cycle and the minimum latency is 1 cycle/hop. Upon arrival, messages are buffered in a hardware queue. When a message arrives at the head of the queue, a task is dispatched to handle it in four processor cycles. During these cycles the Instruction Pointer is loaded from the message header, an address register is set to point to the

Scanned by CamScanner

new message so that the thread's arguments may be accessed, and the thread's first instruction is fetched and decoded.

Synchronization is provided by the ability to signal events effectively using the low-latency communication primitives and by the use of data-tagging in both the register file and memory.

The MDP supports a global namespace with segmented memory management and with name translation instructions. Local memory is referenced using indexed accesses via segment descriptors that specify the base and length of each memory object.

Caltech Mosaic C

The Caltech Mosaic C is an experimental, fine-grain multicomputer that employs single-chip nodes and advanced packaging technology to demonstrate the performance/cost advantages of the fine-grain-multicomputer architecture. Each Mosaic node includes 64KB of memory, an 11MIPS processor, a packet interface, and a router. The nodes are tied together with a 60MBytes/s, two-dimensional, routing-mesh network (**figure 6.6**).

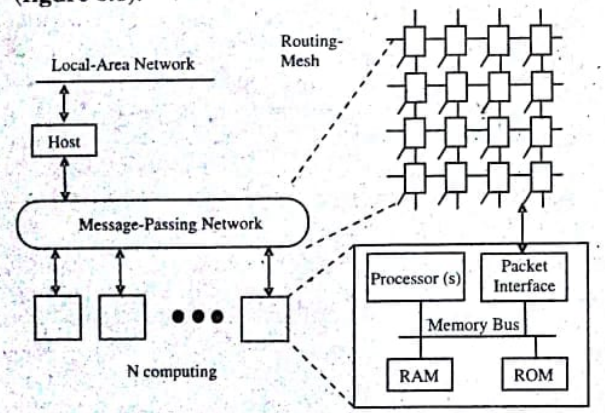


Figure 6.6: Caltech Mosaic Architecture

The compilation-based programming system allows fine-grain, reactive-process, message-passing programs to be expressed in an extension of C++, and the runtime system performs automatic, distributed management of system resources.

Mosaic C Node

The Mosaic C multicomputer node was a single 9.25mm × 10.00mm chip fabricated in a 1.2µm-feature-size, two-level-metal CMOS process. At 5-V operation, the synchronous parts of the chip operated with large margins at a 30-MHz clock rate, and the chip dissipated ≈ 0.5W.

The processor also included two program counters and two sets of general-purpose registers to allow zero-time context switching between user programs and message handling. Thus, when the packet interface received a complete packet, received the header of a packets, completed the sending of a packet, exhausted the allocated space for receiving packets, or any of several other events that could be selected, it could interrupt the processor by switching it instantly to the message-handling context.

Mosaic C 8 × 8 Mesh Boards

The choice of a two-dimensional mesh for the Mosaic was based on a 1989 engineering analysis; originally, a three-dimensional mesh network was planned. But the mutual fit of the two-dimensional mesh network and the circuit board medium provided high packaging density and allowed the high speed signals between the routers to be conveyed on shorter wires.

Sixty-four Mosaic chips were packaged by Tape-Automated Bonding (TAB) in an 8 × 8 array on a circuit board. These boards allowed the construction of arbitrarily large, two-dimensional arrays of nodes using stacking connectors. This style of packaging was meant to demonstrate some of the density, scaling, and testing advantages of mesh-connected systems. Host-interface boards were also used to connect the Mosaic arrays and workstations.

Ques 14) Explain dataflow architecture.

Ans: Dataflow Architecture

Dataflow machines are programmable computers of which the hardware is optimized for fine-grain data-driven parallel computation.

Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or (at least conceptually) the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable: i. e. behavior is indeterministic.

Although no commercially successful general-purpose computer hardware has used a dataflow architecture, it has been successfully implemented in specialized hardware such as in digital signal processing, network routing, graphics processing, telemetry, and more recently in data warehousing. It is also very relevant in many software architectures today including database engine designs and parallel computing frameworks.

Synchronous dataflow architectures tune to match the workload presented by real-time data path applications such as wire speed packet forwarding. Dataflow architectures that are deterministic in nature enable

Scanned by CamScanner

programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.

Using various latency-hiding mechanisms and multiple contexts per processor makes the conventional von Neumann architecture quite expensive to implement, and only certain types of parallelism can be exploited efficiently. Dataflow architectures represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine languages.

Ques 15) What do you understand by dataflow graphs?

Ans: Dataflow Graphs

A data flow graph is a graph where processes form nodes and channels (or buffers) form edges, through which "flow" data. It is a directed graph and may or may not be fully connected. A data flow diagram is a pictorial representation of the data flow graph. **Figure 6.7** shows an example.

No information is contained in the data flow graph about when the processes run, i.e., which must run concurrently and which must run successively. This information must be drawn separately out of the requirement specification.

The analysis of data flow renders clear the communication inherent in the requirement. Now we are able to identify precisely what we require for a process interface specification. It is simply a formal or informal specification of the protocol of both incoming and outgoing data.

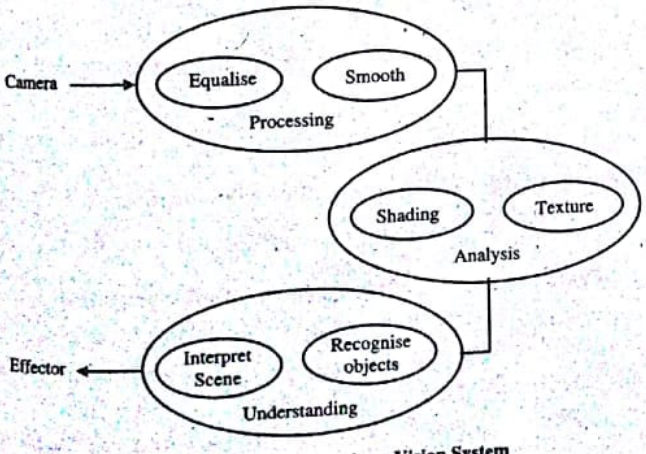


Figure 6.7: Data Flow Diagram for a Vision System

Ques 16) Explain the hybrid architecture of multithreaded computer.

Ans: Hybrid Architecture

General purpose hybrid dataflow/von-Neumann architectures are gaining attraction as effective parallel platforms. Although different implementations differ in the way they merge the conceptually different computational models, they all follow similar principles: they harness the parallelism and data synchronization inherent to the dataflow model, yet maintain the programmability of the von-Neumann model.

One can classify hybrid dataflow/von-Neumann models according to two different taxonomies:

1) One based on the execution model used for inter and intrablock execution, and
2) The other based on the integration level of both control and dataflow execution models.

Hybrid and unified architectures are architectures which combines the positive features from the von Neumann and dataflow architectures. Some common **examples** of these architectures are:

1) **MIT P-RISC:** The P-RISC is a "RISC-ified" dataflow architecture. It allows tighter encodings of the dataflow graphs and produces longer threads for best performance. This can be achieved by dividing the complex dataflow instructions into separate simple component instructions that can be composed by the compiler. In this architecture the traditional instruction sequencing is used. All intra-processor communication via memory are performed by it and implemented by jointly explicitly using memory locations. It replaces some of the dataflow synchronization with conventional program counter-based synchronization.

2) **The IBM Empire:** The Empire is a von Neumann/dataflow hybrid architecture under development at IBM.

3) **MIT/Motorola *T:** *T is a joint project between MIT and Motorola. It is a hybrid dataflow/von Neumann machine using commercial microprocessors that are modified to provide a tightly coupled message interface and thread dispatch mechanism. Two processors, a memory controller, and an optional I/O controller are integrated onto a multichip module.

Scanned by CamScanner