Advance Computer Architecture

UNIT-I

THE STATE OF COMPUTING

System Attributes to Performance

The ideal performance of a computer system demands a perfect match between machine capability and program behaviour. Machine capability can be enhanced with better hardware technology, innovative architectural features, and efficient resources management. However, program behaviour is difficult to predict due to its heavy dependence on application and run-time conditions.

Consider the execution of a program on a given computer. The simplest measure of program performance is the turnaround time, which includes disk and memory accesses, input and output activities, compilation time, OS overhead, and CPU time. In order to shorten the turnaround time, one must reduce all these time factors.

Clock Rate and CPI

The CPU of today's digital computer is driven by a clock with a constant cycle time τ. The inverse of the cycle time is the clock rate. The size of a program is determined by its instruction count in terms of the number of machine instructions to be executed in the program. Different machine instructions may require different numbers of clock cycles to execute. Therefore, the Cycle Per Instruction (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

Performance Factors

Let I_c be the number of instructions in a given program or the instruction count. The CPU time needed to execute the program is estimated by finding the product of three contributing factors:

$$T = I_c \times CPI \times \tau$$

The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand fetch, execution and store results. In this cycle, only the instruction decodes, and execution phases are carried out in the CPU.

System Attributes

The above five performance factors (I_c, p, m, k, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, cache and memory hierarchy.

The instruction-set architecture affects the program length and processor cycles needed. The compiler technology affects the values of I_c, p and the memory reference count. The CPU implementation and control determine the total processor time needed. Finally, the memory technology and hierarchy design affect the memory access latency. The above CPU time can be used as a basis in estimating the execution rate of a processor.

Floating Point Operations per Second

Most compute-intensive applications in science and engineering make heavy use of floating-point operations. Compared to instructions per second, for such applications a more relevant measure of performance is floating point operations per second, which is abbreviated as flops.

Throughput Rate

Another important concept is related to how many programs a system can execute per unit time, called the system throughput W_s. in a multiprogrammed system, the system throughput is often lower than the CPU throughput W_p defined by:

$$W_p = \frac{f}{I_c \times CPI}$$

Programming environment

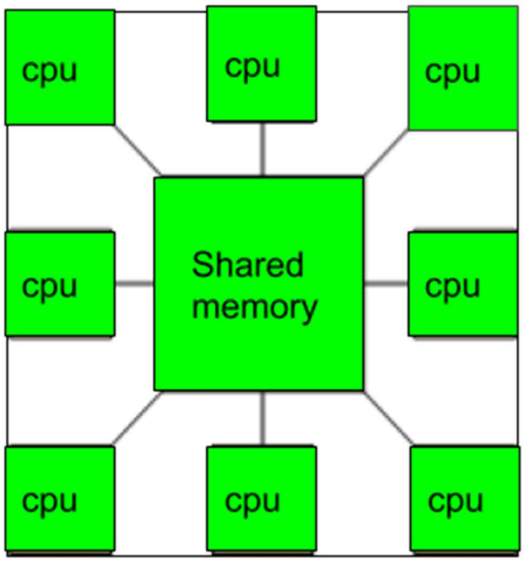
The programmability of a computer depends on the programming environment provided to the users. In fact, the marketability of any new computer system depends on the creation of a user-friendly environment in which programming becomes a productive undertaking rather than a challenge.

Multiprocessor and Multicomputer

Multiprocessor

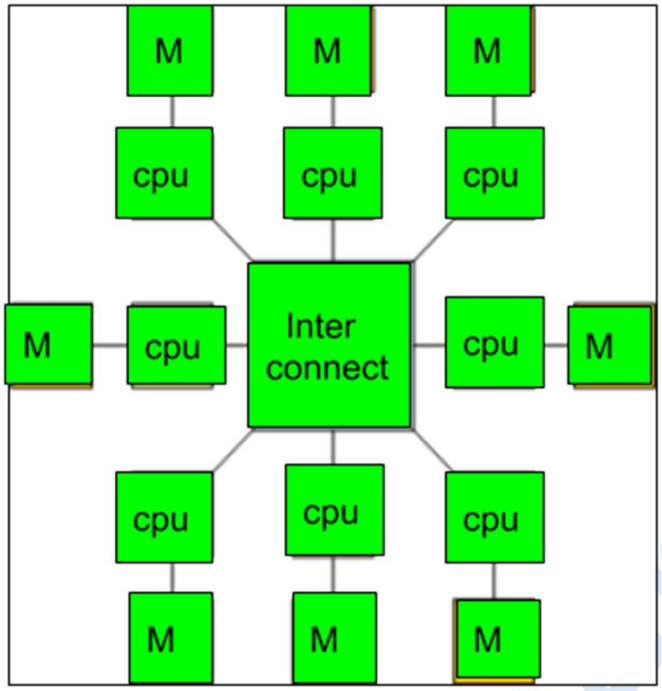
A Multiprocessor is a computer system with two or more central processing units (CPUs) share full access to a common RAM. The main objective of using a multiprocessor is to boost the system's execution speed, with other objectives being fault tolerance and application matching.

There are two types of multiprocessors, one is called shared memory multiprocessor, and another is distributed memory multiprocessor. In shared memory multiprocessors, all the CPUs shares the common memory but in a distributed memory multiprocessor, every CPU has its own private memory.



Multicomputer

A multicomputer system is a computer system with multiple processors that are connected together to solve a problem. Each processor has its own memory and it is accessible by that particular processor and those processors can communicate with each other via an interconnection network.



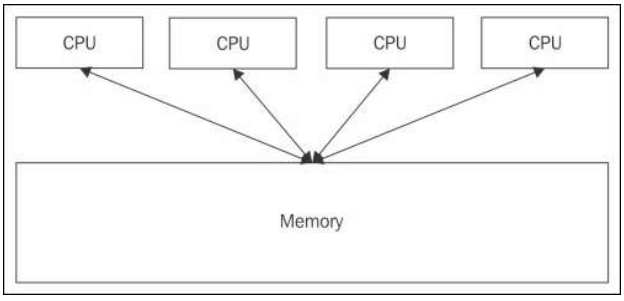
As the multicomputer is capable of messages passing between the processors, it is possible to divide the task between the processors to complete the task. Hence, a multicomputer can be used for distributed computing. It is cost effective and easier to build a multicomputer than a multiprocessor.

Difference between Multiprocessor and Multicomputer

Multiprocessor	Multicomputer
1. Multiple processors in a single computer.	1. Interlinked multiple autonomous computers.
2. A system with two or more CPUs that allows simultaneous processing of programs	2. A set of processors connected by the communication network that works jointly to solve a computation problem.
3. Easier to process.	3. Less easy to program.
4. Supports parallel computing.	4. Supports distributed computing.

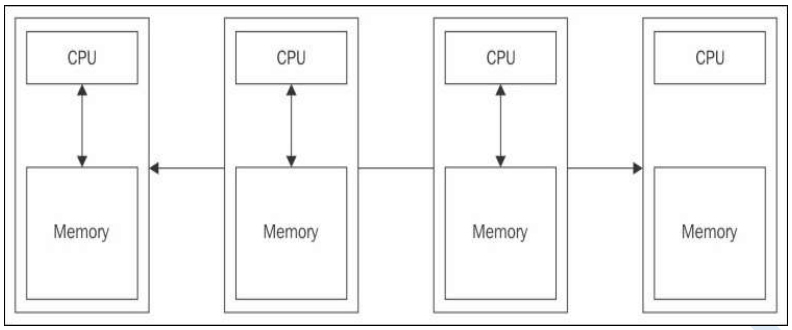
Shared Memory:

In the following figure, we see a typical shared-memory architecture where four processors (the four CPU boxes in the following diagram) can all access the same memory address space (that is, the Memory box).



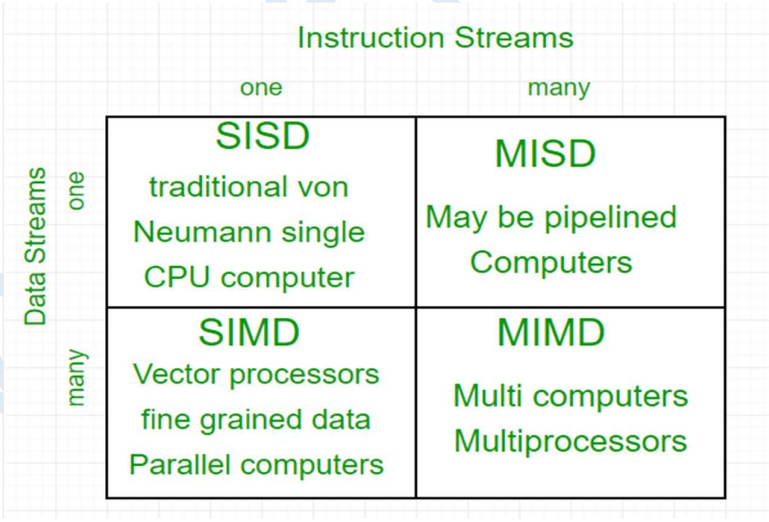
Distributed Memory:

In the following figure, we have the same four CPUs as before, that are organized now in a shared-memory architecture. Each CPU has access to its own private memory and cannot see any other CPU memory space. The four computers (indicated by the boxes surrounding their CPU and memory) communicate through the network (the black line connecting them).



Taxonomy of MIMD Computers

Parallel computing is a computing where the jobs are broken into discrete parts that can be executed concurrently. Each part is further broken down to a series of instructions. Instructions from each part execute simultaneously on different CPUs. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.



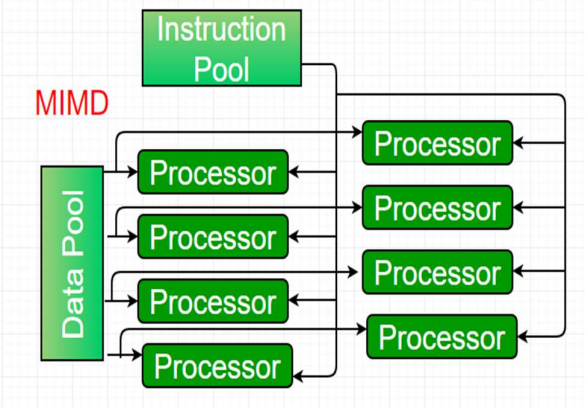
Instruction Stream: It as a sequence of instructions read from memory.

Data Stream: The operation performed on data in processors.

Single Instruction Single Data Stream

Multiple-instruction, multiple-data (MIMD) systems

In computing, MIMD (multiple instruction, multiple data) is a technique employed to achieve parallelism. Machines using MIMD have a number of processors that function asynchronously and independently. At any time, different processors may be executing different instructions on different pieces of data.



MIMD machines are broadly categorized into **shared-memory MIMD** and **distributed-memory MIMD** based on the way PEs are coupled to the main memory.

In the **shared memory MIMD** model (tightly coupled multiprocessor systems), all the PEs are connected to a single global memory and they all have access to it. The communication between PEs in this model takes place through the shared memory, modification of the data stored in the global memory by one PE is visible to all other PEs.

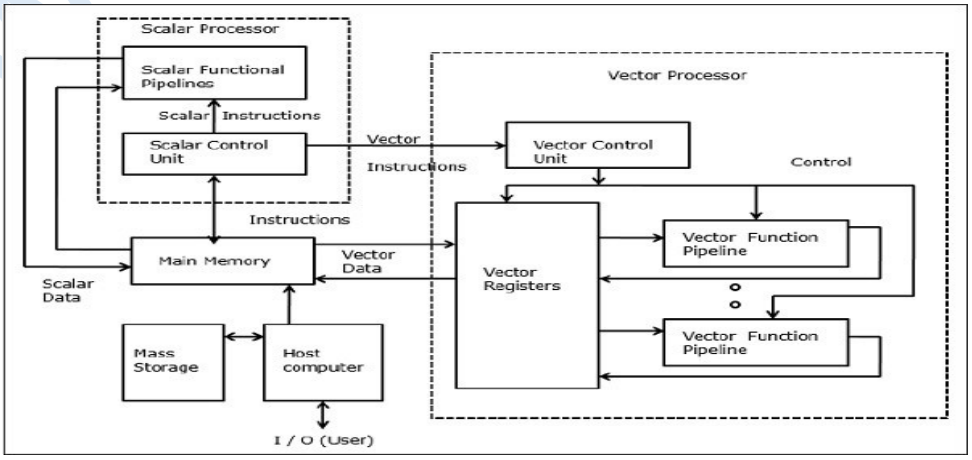
In **Distributed memory MIMD** machines (loosely coupled multiprocessor systems) all PEs have a local memory. The communication between PEs in this model takes place through the interconnection network (the inter process communication channel, or IPC). The network connecting PEs can be configured to tree, mesh or in accordance with the requirement.

Multivector and SIMD Computers

Vector Supercomputers

In a vector computer, a vector processor is attached to the scalar processor as an optional feature. The host computer first loads program and data to the main memory. Then the scalar control unit decodes all the instructions. If the decoded instructions are scalar operations or program operations, the scalar processor executes those operations using scalar functional pipelines.

On the other hand, if the decoded instructions are vector operations then the instructions will be sent to vector control unit.

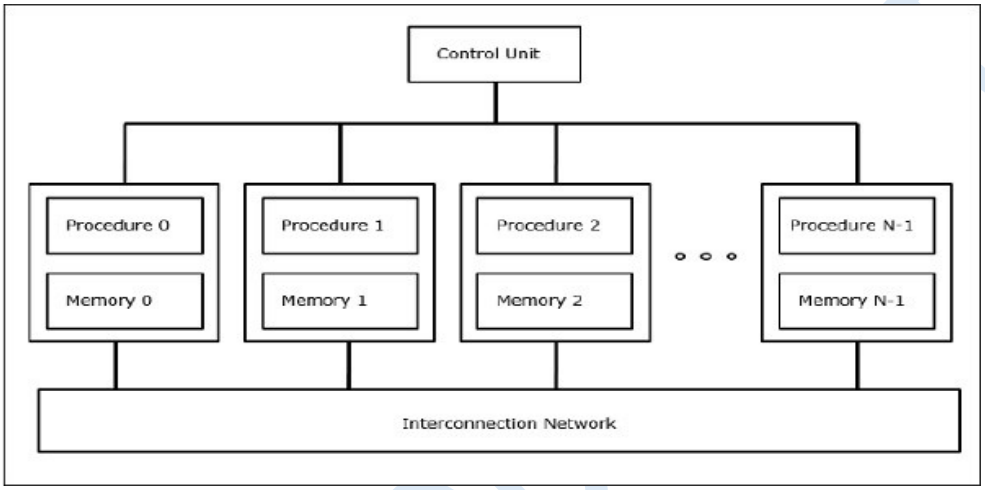


Vector Processor Models

The above diagram shows a register-to-register architecture. Vector registers are used to hold the vector operands, intermediate and final vector results. The vector functional pipelines retrieve operands from and put results into the vector registers. All vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

SIMD Supercomputers

In SIMD computers, 'N' number of processors are connected to a control unit and all the processors have their individual memory units. All the processors are connected by an interconnection network.



PRAM and VLSI Models

The models can be applied to obtain theoretical performance bounds on parallel computers or to estimate VLSI (Very Large Scale Integration) complexity on chip area and execution time before the chip is fabricated. The abstract models are also useful in scalability and programmability analysis when real machines are compared with an idealized parallel machine without worrying about communication overhead among processing nodes.

The ideal model gives a suitable framework for developing parallel algorithms without considering the physical constraints or implementation details.

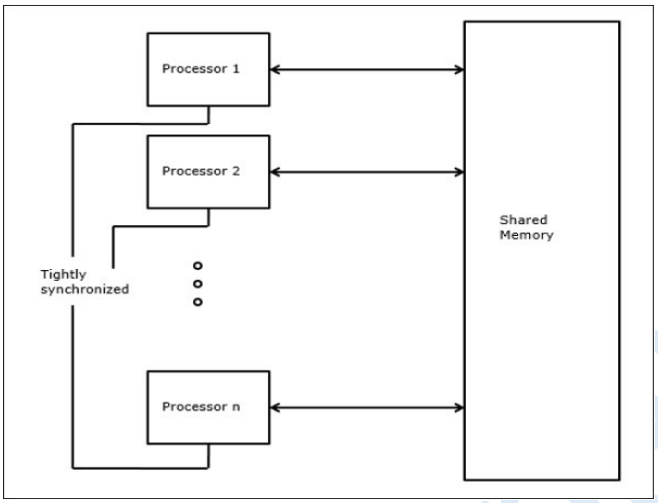
Parallel Random-Access Machines

Time and space complexities: The complexity of an algorithm for solving a problem of size s on a computer is determined by the execution time and the storage space required. The time complexity is a function of the problem size. The time complexity function in order natation is the asymptotic time complexity of the algorithm.

The space complexity can be similarly defined as a function of the problem size s. the asymptotic space complexity refers to the data storage of large problems. The time complexity of a serial algorithm is simply called serial complexity.

PRAM models

Shepardson and Sturgis (1963) modelled the conventional Uniprocessor computers as random-access-machines (RAM). Fortune and Wyllie (1978) developed a parallel random-access-machine (PRAM) model for modelling an idealized parallel computer with zero memory access overhead and synchronization.



An N-processor PRAM has a shared memory unit. This shared memory can be centralized or distributed among the processors. These processors operate on a synchronized read-memory, write-memory and compute cycle. So, these models specify how concurrent read and write operations are handled.

Following are the possible memory update operations −

- **Exclusive read (ER)** − In this method, in each cycle only one processor is allowed to read from any memory location.
- **Exclusive write (EW)** − In this method, at least one processor is allowed to write into a memory location at a time.
- **Concurrent read (CR)** − It allows multiple processors to read the same information from the same memory location in the same cycle.
- **Concurrent write (CW)** − It allows simultaneous write operations to the same memory location. To avoid write conflict some policies are set up.

VLSI Complexity Model

Parallel computers use VLSI chips to fabricate processor arrays, memory arrays and large-scale switching networks.

Nowadays, VLSI technologies are 2-dimensional. The size of a VLSI chip is proportional to the amount of storage (memory) space available in that chip.

We can calculate the space complexity of an algorithm by the chip area (A) of the VLSI chip implementation of that algorithm. If T is the time (latency) needed to execute the algorithm, then A.T gives an upper bound on

the total number of bits processed through the chip (or I/O). For certain computing, there exists a lower bound, f(s), such that

$A \times T2 >= O(f(s))$

Where A=chip area and T=time

Memory Bound on Chip Area: There are many computations which are memory-bound, due to the need to process large data sets. To implement this type of computation in silicon, one is limited by how densely information can be placed on the chip.

PARALLALISM

In Parallel or Sequential Execution of programs there are some concepts of dependence that we need to understand. These dependencies are known as Data, Control and Resource Dependence.

Data and Resource Dependencies

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. The independence comes in various forms defined below separately.

Data Dependencies

Relation between statements is shown by data dependences. There are 5 types of data dependencies given below:

(a) Antidependency: A statement S2 is anti-dependent on statement S1 if S2 follows S1 in order and if the output of S2 overlap the input to S1 .

(b) Input dependence: Read & write are input statement input dependence occur not because of same variables involved put because of same file is referenced by both input statements.

(c) Unknown dependence: The dependence relation between two statement cannot be found in following situation

The subscript of variable is itself subscribed.

The subscript does not have the loop index variable.

Subscript is nonlinear in the loop index variable.

(d) Output dependence: Two statements are output dependence if they produce the same output variable.

(e) Flow dependence: The statement ST2 is flow dependent if a statement ST1, if an expression path exists from ST1 to ST2 and at least are output of ST, feeds in an input to ST2.

Resource Dependencies

This is different from data dependency, which demands the independence of the work to be done. Resource dependence is concerned with the conflicts in using shared resources, such as integer units, floating-point units, registers and memory areas, among parallel events. When the conflicts resource is an ALU, we call it ALU dependence.

Hardware and software dependencies

Hardware Dependencies

Hardware parallelism is defined by hardware multiplicity & machine hardware. It is a function of cost & performance trade off. It present the resource utilization patterns of simultaneously executable operations. It also indicates the performance of the processor resources.

One method of identifying parallelism in hardware is by means by number of instructions issued per machine cycle.

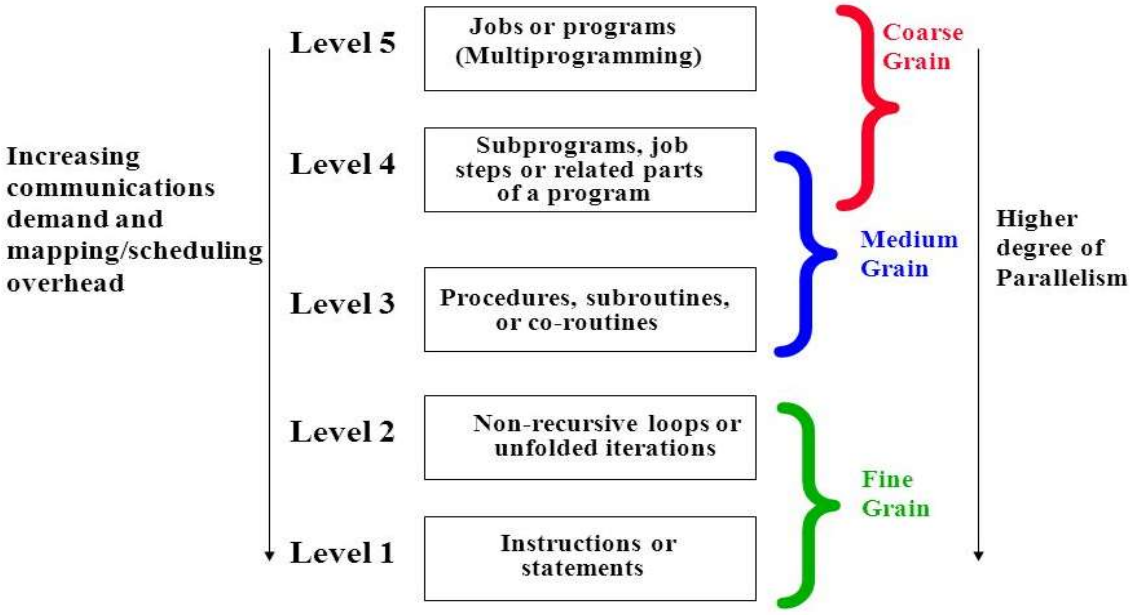
Software Dependencies

Software dependency is defined by control and data dependency of programs. Degree of parallelism is revealed in the program profile or in program flow graph. Software parallelism is a function of algorithm, programming style and compiler optimization. Program flow graphs shows the pattern of simultaneously executable operation. Parallelism in a program varies during the execution period.

Program Partitioning and Scheduling

Grain sizes

Grain size or granularity is a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as fine, medium, or coarse. depending on the processing levels involved.



Instruction Level: At the lowest level, a typical grain contains less than 20 instructions, called fine grain. Depending on individual programs, fine grain parallelism at this level may range from two to thousands.

Loop Level: This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent in successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines.

Procedure Level: This level corresponds to medium-grain parallelism at the task, procedural, subroutine and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels.

Subprogram Level: This corresponds to the level of job steps and related subprograms. The grain size may typically contain tens or hundreds of thousands of instructions. Job steps can overlap across different jobs.

Job Level: This corresponds to the parallel execution of essentially independent jobs on a parallel computer. The grain size can be as high as millions of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical.

Latency

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the memory latency is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the synchronization latency.

Grain Packing and Scheduling

Two fundamental questions to ask in parallel programming are: (i) How can we partition a program into parallel branches, program modules, micro tasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?

This grain-size problem demands determination of both the number and the size of grains (or micro tasks) in a parallel program. Of course, the solution is both problem dependent and machine-dependent. The goal is to produce a short schedule for fast execution of subdivided program modules.

There exists a trade-off between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads the program partitioning involves the algorithm designer, programmer, compiler, operating system support, etc.

Program Flow Mechanism

Conventional computers are based on a control flow mechanism by which the order of program execution is explicitly stated in the user programs. Dataflow computers are based on a data-driven mechanism which allows the execution of an instruction to be driven by data (operand) availability. Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level. Reduction computers are based on a demand-driven mechanism which initiates an operation based on the demand for its results by other computations.

Control flow versus Data flow

Conventional Von Neumann computers use a program counter to sequence the execution of instructions in a program. The program counter is sequenced by instruction flow in a program. This sequential execution style has been called control-driven, as program flow is explicitly controlled by programmers. A uniprocessor computer is inherently sequential, due to use of the control driven mechanism.

In a dataflow computer, the execution of an instruction is driven by data availability instead of being guided by a program counter. In theory, any instruction should be ready for execution whenever operands become available. The instructions in a data-driven program are not ordered in any way. Instead of being stored separately in a main memory, data are directly held inside instructions.

Demand Driven Mechanism

In a reduction machine, the computation is triggered by the demand for an operation's result. Consider the evaluation of a nested arithmetic expression a = ((b+1)*c-(d/e)). The data-driven computation seen above chooses a bottom-up approach, starting from the innermost operations b+1 and d/e, the proceeding to the operation, and finally to the outermost operation -.

A demand-driven computation corresponds to lazy evaluation, because operations are executed only when their results are required by another instruction. The demand driven approach matches naturally with the functional programming concept. The removal of side effects in functional programming makes programs easier to parallelize. There are two types of reduction machine models, both having a recursive control mechanism as characterized below.

Reduction Machine Models

In a string reduction model, each demander gets a separate copy of the expression for its own evaluation. A long string expression is reduced to a single value in a recursive fashion. Each reduction step has an operator followed by an embedded reference demand the corresponding input operands. The operator is suspended while its input arguments are being evaluated. An expression is said to be fully reduced when all the arguments have been replaced by literal values.

In a graph reduction model, the expression is represented as a directed graph. The graph is reduced by evaluation of branches or subgraphs. Different parts of a graph or subgraphs can be reduced or evaluated in parallel upon demand. Each demander is given a pointer to the result of the reduction. The demander manipulates all references to that graph.

Comparisons of Flow Mechanisms

Machine Model	**Control Flow**	**Data Flow**	**Demand Driven (Reduction)**
Basic Definition	Conventional computation; token of control indicates when a statement should be executed.	Eager evaluation; statements are executed when all of their operands are available	Lazy evaluation; statements are executed only when their result is required for another computation
Advantages	Full control The most successful model for commercial products	Very high potential for parallelism	Only required instructions are executed
	Complex data and control structures are easily implemented	High throughput	High degree of parallelism
		Free from side effects	Easy manipulation of

			data structure
Disadvantages	In theory, less efficient than the other two	Time lost waiting for unneeded arguments	Does not support sharing of objects with changing local state
	Difficult in preventing run time errors	High control overhead	Time needed to propagate demand tokens
		Difficult in manipulating data structures	

System interconnect architectures

An **interconnection network** in a parallel machine transfers information from any source node to any desired destination node. This task should be completed with as small latency as possible. It should allow a large number of such transfers to take place concurrently. Moreover, it should be inexpensive as compared to the cost of the rest of the machine.

Network properties and routing

The topology of an interconnection network can be either static or dynamic. Static networks are formed of point-to-point direct connections which will not change during program execution. Dynamic networks are implemented with switched channels, which are dynamically configured to match the communication demand in user programs. Packets switching and routing is playing an important role in modern multi-processor architecture.

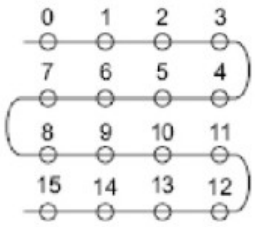
Static networks are used for fixed connections among subsystems of a centralized system or multiple computing nodes of a distributed system. Dynamic networks include buses, crossbar switches, multistage networks and routers which are often used in shared memory multiprocessors.

Static connection networks

Static networks use direct links which are fixed once built. This type of network is more suitable for building computers where the communication patterns are predictable or implementable with static connections.

Linear Arrays

This is a one-dimensional network in which N nodes are connected by N-1 links in a line. Internal noeds have degree 2 and the terminal nodes have degree 1. The diameter is N – 1, which is rather long for large N. The bisection width b = 1.

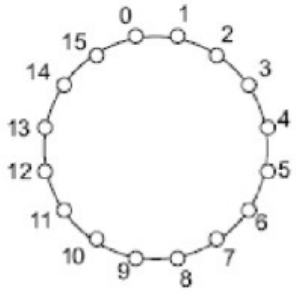


(a) Linear array

Linear Arrays are the simplest connection topology.

Ring and Chordal Ring

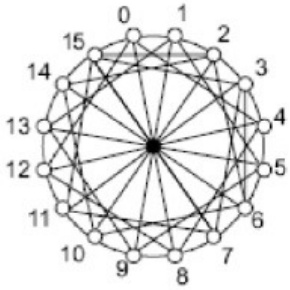
A ring is obtained by connecting the two terminal nodes of a linear array with one extra link. A ring can be unidirectional or bidirectional.



(b) Ring

Barrel Shifter

As shown below, a network of N = 16 nodes, the barrel shifter is obtained from the ring by adding extra links from each node to those nodes having a distance equal to an integer power of 2.



(e) Barrel shifter

UNIT-II

PERFORMANCE METRICS AND MEASURES

Parallelism Profile in Programs

The degree of parallelism reflects the extent to which software parallelism matches hardware parallelism. We characterize below parallelism profiles, introduce the concept of average parallelism and define an ideal speedup with infinite machine resources.

Degree of Parallelism

The execution of a program on a parallel computer may use different numbers of processors at different time periods during the execution cycle. For each time period, the number of processors used to execute a program is defined as the degree of parallelism.

When the DOP exceeds the maximum number of available processors in a system, some parallel branches must be executed in chunks sequentially. However, parallelism still exists within each chunk, limited by the machine size.

Harmonic Mean Performance

The harmonic mean is a type of numerical average. It is calculated by dividing the number of observations by the reciprocal of each number in the series. Thus, the harmonic mean is the reciprocal of the arithmetic mean of the reciprocals.

The harmonic mean helps to find multiplicative or divisor relationships between fractions without worrying about common denominators. Harmonic means are often used in averaging things like rates (e.g., the average travel speed given duration of several trips).

Efficiency, Utilization and Quality

Ruby Lee has defined several parameters for evaluating parallel computations. These are fundamental concepts in parallel processing. Trade-offs among these performance factors are often encountered in real-life applications.

System Efficiency

Let O(n) be the total number of unit operations performed by an n-processor system and T(n) be the execution time in unit time steps. In general, $T(n)<O(n)$ if more than one operation is performed by n processors per unit time, where $n>=2$. Assum T (1) = O (1) in a uniprocessor system.

$S(n) = T(1)/T(n)$

The system efficiency for an n-processor system is defined by

$E(n) = S(n)/n = T(1)/nT(n)$

Efficiency is an indication of the actual degree of speedup performance achieved as compared with the maximum value.

The lowest efficiency corresponds to the case of the entire program code being executed sequentially on a single processor, the other processors remaining idle. The maximum efficiency is achieved when all n processors are fully utilized throughout the execution period.

Redundancy and Utilization

The redundancy in a parallel computation is defined as the ratio of O(n) to O(1):

R(n) = O(n)/O(1)

The ratio signifies the extent of matching between software parallelism and hardware parallelism. Obviosly 1<=R(n) <=n. The system utilization in a parallel computation is defined as:

U(n) = R(n)E(n) = O(n)/nT(n)

The system utilization indicates the percentage of resources that was kept busy during the execution of a parallel program.

Quality of Parallelism

The quality of a parallel computation is directly proportional to the speedup and efficiency and inversely related to the redundancy. Thus we have

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3(1)}{nT^2(n)O(n)}$$

Since E(n) is always a fraction and R(n) is a number between 1 and n, the quality Q(n) is always upper-bounded by the speedup S(n).

Standard Performance Measures

Speedup Performance Law

Amdahl's law for a fixed workload

Amdahl's law is a formula used to find the maximum improvement possible by improving a particular part of a system. In parallel computing, Amdahl's law is mainly used to predict the theoretical maximum speedup for program processing using multiple processors.

In Amdahl's law, computational workload W is fixed while the number of processors that can work on W can be increased. Denote the execution rate of i processors as Ri , then in a relative comparison they can be simplified as and 1 R = 1 Rn = n . The workload is also simplified. We assume that the workload consists of sequential work αW and n parallel work (1) −α W where α is between 0 and 1. More specifically, this workload can be written in a vector form as, W W = − (,0,...,0, 1) α α , or, W W 1 = α , (1) W W n = −α , and 0 Wi = for all i n ≠ 1, .

The execution time of the given work by n processors is then computed as,

$$T_n = \frac{W_1}{R_1} + \frac{W_n}{R_n}$$

Speedup of n processor system is defined using a ratio of execution time, i.e.,

$$S_n = \frac{T_1}{T_n}$$

Substituting the execution time in relation W gives,

$$S_n = \frac{W/_1}{\frac{aW}{1} + \frac{(1-a)W}{n}} = \frac{n}{1+(n-1)a} \quad \ldots.(1)$$

Eq.(1) is called the Amdahl's law. If the number of processors is increased infinity, the speedup becomes,

$$S_\infty = \frac{1}{a}$$

Notice that the speedup can NOT be increased to infinity even if the number of processors is increased to infinity.

Gustafson's Law

This law says that increase of problem size for large machines can retain scalability with respect to the number of processors. Assume that the workload is scaled up on an n-node machine as,

$W' = aW + (1-a)nW$

Speedup for the scaled up workload is then,

$$S'_n = \frac{Single\ Processor\ Execution\ Time}{n - Processer\ Execution\ Time}$$

$$S'_n = \frac{(aW + (1-a)nW)/1}{\frac{aW}{1} + \frac{(1-a)nW}{n}} \quad \ldots 3$$

Simplifying Eq.(3) produces the Gustafson's law:

$$S'_n = a + (1-a)n$$

Notice that if the workload is scaled up to maintain a fixed execution time as the number of processors increases, the speedup increases linearly. What Gustafson's law says is that the true parallel power of a large multiprocessor system is only achievable when a large parallel problem is applied.

Scalability Analysis and Approaches

The performance of a computer system depends on a large number of factors, all affecting the determine how scalability helps in scalability of the computer evaluating performance of parallel architecture and the application computers program involved.

The simplest definition of Scalability is that the performance of a computer system increases linearly with respect to the number of processors used for a given application.

Scalability analysis of a given computer system must be conducted for a given application program. The analysis can be performed under different constraints on the growth of the problem size and on the machine size.

Scalability Metrics and Goals

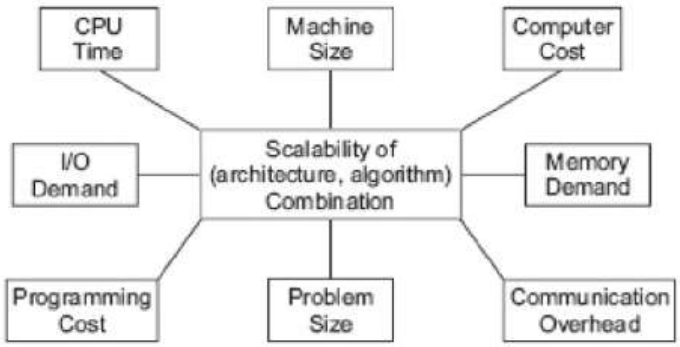
Scalability studies determine the degree of matching between computer architecture and an application algorithm. For different pairs, the analysis may end up with different conclusions. Thus, a good computer architecture should be efficient in implementing a large class of application algorithms.

Scalability Metrics

Below are the basic metrics affecting the scalability of a computer system for a given application:

Machine size (n)

The number of processors employed in a parallel computer system. A large machine size implies more resources and more computing power.



Clock rate (f)

The clock rate determine the basic machine cycle. We hope to build a machine with components driven by a clock which can scale up with better technology.

Problem size (s)

The amount of computational workload or the number of data points used to solve a given problem. The problem size is directly proportional to the sequential execution time T(s,1) for a uniprocessor system because each data point may demand one or more operations.

CPU time (T)

The actual CPU time (in seconds) elapsed in executing a given program on a parallel machine with n processors collectively.

I/O demand (d)

The input /output demand in moving the program, data and results associated with a given application run. The I/O operations may overlap with the CPU operations in a multiprogrammed environment.

Memory Capacity

The amount of main memory used in a program execution. Note that the memory demand is affected by the problem size, the program size, the algorithms and the data structures used.

Computer cost (c)

The total cost of hardware and software resources required to carry out the execution of a program.

Evolution of Scalable Computers

The idea of massive parallelism is rather old, the technology is advancing steadily and the software is relatively unexplored, as was observed by Cybenko and Kuck. One evolutional trend is to build scalable supercomputers with distributed shared memory and standardized UNIX/LINUX for parallel processing.

Size Scalability

The study of system scalability started with the desire to increase the machine size. A size scalable computer is designed to have a scaling range from a small to a large number of resource components. The expectations is to achieve linearly increased performance with components include computers, processors or processing elements, memories etc.

Generation (Time) Scalability

Since the basic processor nodes become obsolete every three years, the time scalability is equally important as the size scalability. Not only should the hardware technology be scalable, such as the CMOS circuits and packaging technologies in building processors and memory chips but also the software which demands software compatibility and portability with new hardware systems.

Problem Scalability

The problem size corresponds to the data set size. This is the key to achieving scalable performance as the program granularity changes. A problem scalable computer should be able to perform well as the problem size increases. The problem size can be scaled to be sufficiently large in order to operate efficiently on a computer with a given granularity.

UNIT-III

MULTIPROCESSOR SYSTEM INTERCONNECTS

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O and peripherals devices. Hierarchical buses, crossbar switches, and multistage networks are often used for this purpose.

Hierarchical bus system

A bus system consists of a hierarchy of buses connecting various system and subsystem components in a computer. Each bus is formed with a number of signals, control and power lines. Different buses are used to perform different interconnection functions.

Local Bus: Buses implemented within processor chips or on printed-circuit boards are called local buses. On a processor board one may find a local bus which provides a common communication path among major components mounted on the board. A memory board uses a memory bus to connect the memory with the interface logic. An I/O or network interface chip or board uses a data bus. Each of these local buses consists of signal and utility lines.

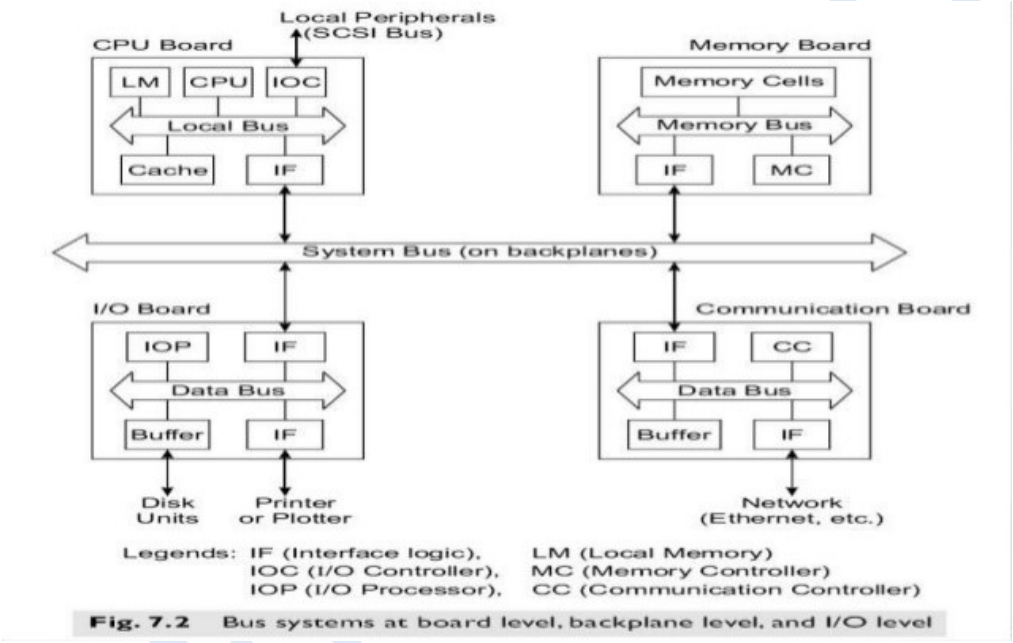


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

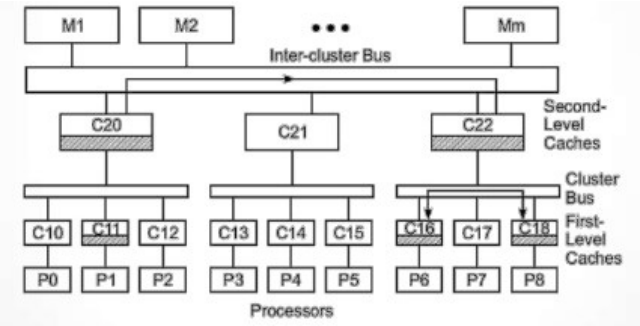
Backplane Bus: A backplane is a printed circuit on which many connectors are used to plug in functional boards. A system bus, consisting of shared signal paths and utility lines, is built on the backplane. This system bus provides a common communication path among all plug-in boards.

I/O Bus: Input/output devices are connected to a computer system through an I/O bus such as the SCSI (small computer systems interface) bus. This bus made of coaxial cables with taps connecting disks, printer and other devices to a processor through an I/O controller. Special interface logic is used to connect various board types to the backplane bus.

Hierarchical Buses and Caches: Wilson proposed a hierarchical cache/bus architecture as shown below.

This is a multilevel tree structure in which the leaf nodes are processors and their private caches. These are divided into several clusters, each of which is connected through a cluster bus.

An intercluster bus is used to provide communications among the clusters. Second level caches are used between each cluster bus and the intercluster bus. Each second-level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.



Crossbar switch and multiport memory

Switched networks provide dynamic interconnections between the inputs and outputs. Major classes of switched networks are specified below, based on the number of stages and blocking or nonblocking.

Network Stages: Depending on the interstage connections used, a single-stage network is called a recirculating network because data items may have to recirculate through the single stage many times before reaching their destination. A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections. The crossbar switch and multiport memory organization are both single-stage networks.

Blocking vs Nonblocking networks: A multistage network is called blocking if the simultaneous connections of some multiple input-output pairs may result in conflicts in the use of switches or communication links.

A multistage network is called nonblocking if it can perform all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair.

Crossbar Networks: In a crossbar network, every input port is connected to a free output port through a crosspoint switch without blocking. A crossbar network is a single-stage network built with unary switches at the crosspoints.

Once the data is read from the memory, its value is returned to the requesting processor along the same crosspoint switch. In general, such a crossbar network requires the use of n x m crosspoints switches. A square crossbar can implement any of the n! permutations without blocking.

Crosspoint Switch Design: Out of n crosspoint switches in each column of an n x m crossbar mesh, only one can be connected at a time. To resolve the contention for each memory module, each crosspoint switch must be designed with extra hardware.

Furthermore, each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals. This means that each crosspoint has a complexity matching that of a bus of the same width.

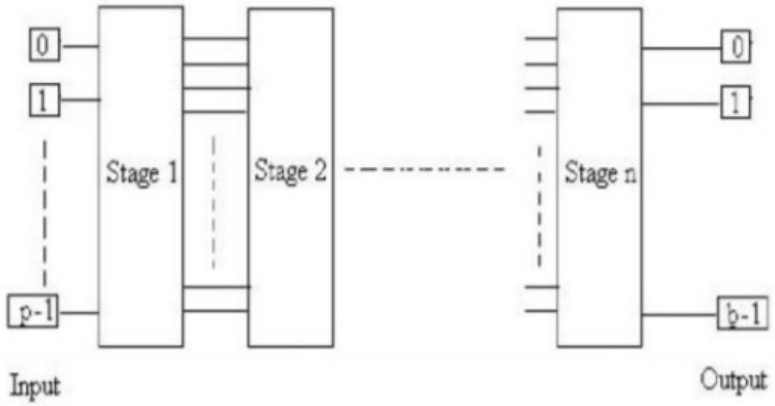
Crossbar Limitations: A single processor can send many requests to multiple memory module. For an n x n crossbar network, at most n memory words can be delivered to at most n processors in each cycle.

The crossbar network offers the highest bandwidth of n data transfers per cycle, as compared with only one data transfer per bus cycle. Since all necessary switching and conflict resolution logic are built into the crosspoint switch, the processor interface and memory port logic are much simplified and cheaper.

Multiport Memory: Because building a crossbar network into a large system is cost prohibitive, some mainframe multiprocessors used a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

Multistage and combining networks

Multistage networks consist of more than one stages of small interconnection elements called switching elements and links interconnecting them. A multistage network normally connects N inputs to N outputs and is referred as an N × N multistage network. The parameter N is called the size of the network. Figure below illustrates a structure of multistage network. This figure shows the connection between p inputs and b outputs, and connection between these is via number of stages.



Multistage network is actually a compromise between crossbar and shared bus networks of various types of multiprocessor networks. Multistage networks are used in multiprocessing systems to provide cost-effective, high bandwidth communication between processors and/or memory modules. Multistage networks attempt to reduce cost and decrease the path length.

Cache coherence and synchronization mechanisms

Cache coherence protocols for coping with the multicache inconsistency problem are considered below. Snoopy protocols are designed for bus-connected systems. Directory-based protocols apply to network-connected systems.

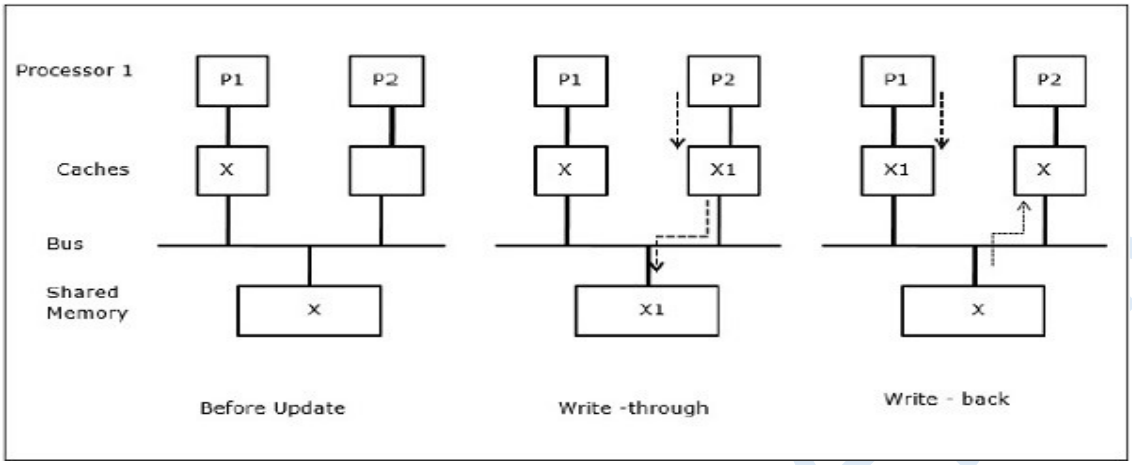
The cache coherence problem

Caches in a multiprocessing environment introduces the cache coherence problem. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data.

Inconsistency in Data Sharing: The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: sharing of writable data, process migration and

I/O activity. Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

If processor p1 writes new data X into the cache, the same copy will be written immediately into the shared memory under a write-through policy. In this case, inconsistency occurs between the two copies.



Process Migration and I/O: In the above figure, the occurrence of inconsistency after a process containing a shared variable X migrates from processor 1 to processor 2 using the write-back cache on the right. In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.

In both cases, inconsistency appears between the two cache copies, labelled X and X'. special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely migrate from one processor to another.

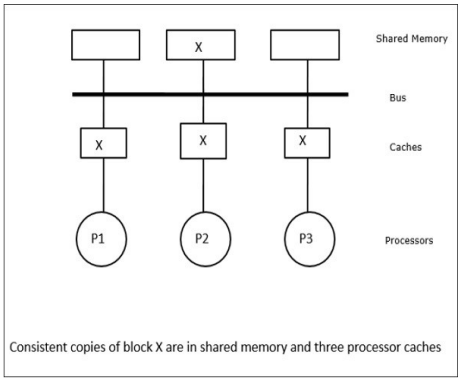
Two Protocol Approaches: Many of the early commercially available multiprocessors used bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point links in the direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system.

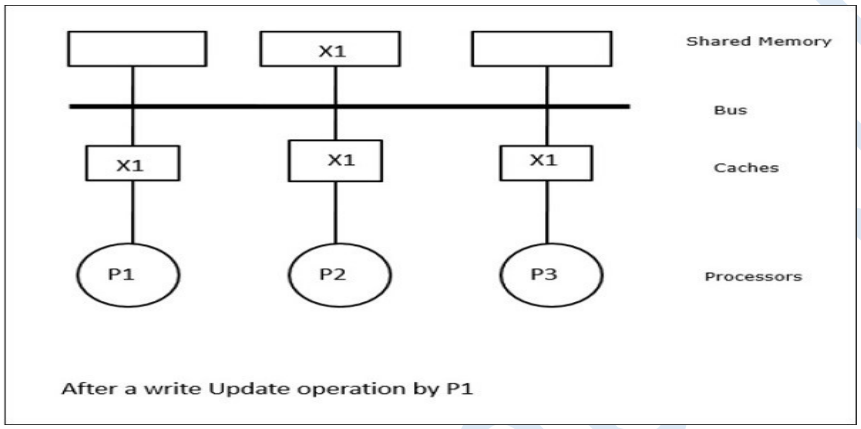
Snoopy Bus Protocol

Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism. Consider three processors (P_1, P_2, and P_n) maintaining consistent copies of block X in their local caches and in the shared-memory module marked X.

Using a write-invalidate protocol, the processor P1 modifies its cache from X to X' and all other copies are invalidated via the bus. Invalidated blocks are sometimes called dirty, meaning they should not be used. The write-update protocol demands the new block content X' be broadcast to all cache copies via the bus.



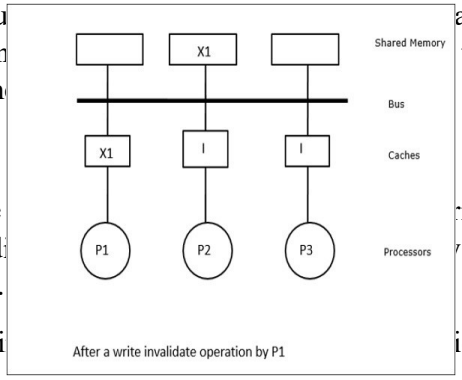
Consistent copies of block X are in shared memory and three processor caches



After a write Update operation by P1

Directory-Based Protocols

When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to su adcasting is expensive to perform in such a network, consistency comman that keep a copy of the block. This leads to directory-based protocols for n



After a write invalidate operation by P1

Directory Structure

In a multistage or packet switched network, cache coherence ies to store information on where copies of cache blocks reside. Various di in how the directory maintains information and what information it stores.

Different types of directory protocols fall under three pri ies, limited directories and chained directories.

Full-Map Directories: The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache. If the dirty bit is set, then one and only one processor's bit is set, and that processor can write into the block.

Limited Directories: Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

Chained Directories: Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks. This type of cache coherence scheme is called a chained scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.

Hardware Synchronization Mechanisms

Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating processes residing in the same or different processors. Synchronization enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data. Synchronization can be implemented in software, firmware and hardware through controlled sharing of data and control information in memory.

VECTOR PROCESSING PRINCIPLES

The need to increase computational power is a never-ending requirement. In scientific and research areas, the computational involved are quite extensive and hence high-power computers are the must.

Vector Processing

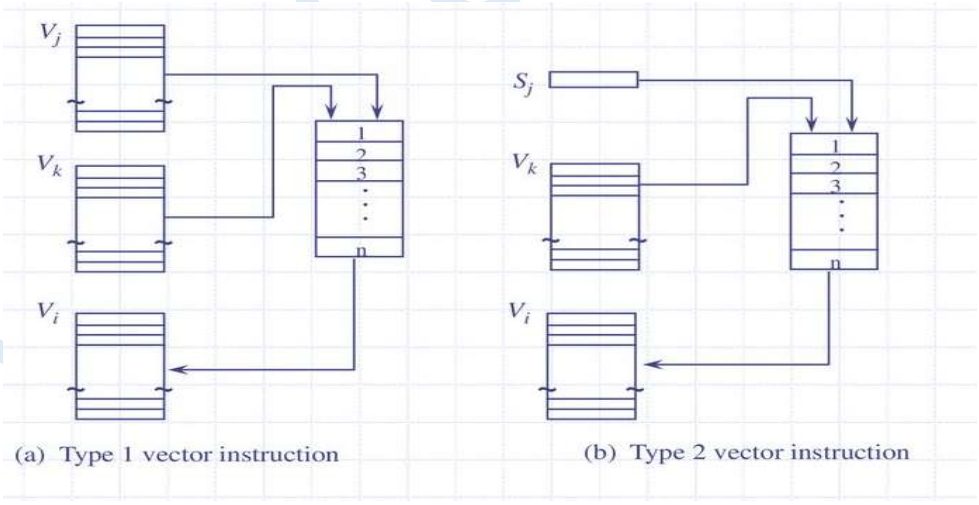
A vector is an ordered set of scalar data items, all of the same type, stored in memory. Usually, the vector elements are ordered to have a fixed addressing increment between successive elements, called the stride.

A vector processor is an ensemble of hardware resources, including vector registers, functional pipelines, processing elements and register counters, for performing vector operations. Vector processing occurs when arithmetic or logical operations are applied to vectors. It is distinguished from scalar processing which operates on one datum or one pair of data. The conversion from scalar code to vector code is called vectorization.

Vector Instruction Types

Six types of vector instructions are:

1. **Vector-Vector instructions:** As shown in Fig(a) one or two vector operands are fetched from the respective vector registers, enter through a functional pipeline unit and produce results in another vector register.



(a) Type 1 vector instruction (b) Type 2 vector instruction

2. **Vector-scaler instructions:** Above Fig(b) shows a vector-scalar instruction corresponding to the following mapping:

$$\boldsymbol{f3 : s \times vk \rightarrow vi}$$

An example is a scalar product $\boldsymbol{s \times v1 = v2}$, in which the elements of v1 are each multiplied by a scalar s to produce vector v2 of equal length.

3. **Vector-memory instructions:** This corresponds to vector load or vector store, element by element, between the vector register and the memory as defined below:

$f_4 : M \rightarrow V$ **vector load**

$f_5 : V \rightarrow M$ **vector store**

4. **Vector reduction instructions:** These correspond to the following mapping:

$$f6 : vi \rightarrow s$$
$$f7 : Vi \times vj \rightarrow s$$

Example of f6 include finding the maximum, minimum, sum and mean value of all elements in a vector.

5. **Gather and Scatter Instructions:** These instructions use two vector registers to gather or to scatter vector elements randomly throughout the memory, corresponding to the following mapping:

$f_8 : M \rightarrow V_1 \times V_0$ Gather

$f_9 : V_1 \times V_0 \rightarrow M$ Scatter

Gather is an operation that fetches from memory the none-zero elements of a sparse vector using indices that themselves are indexed. Scatter does the opposite, storing into memory a vector in a sparse vector whose nonzero entries are indexed.

6. **Masking instructions:** This type of instruction uses a mask vector to compress or to expand a vector to a shorter or longer index vector, respectively.

Vector Access Memory Schemes

The flow of vector operands between the main memory and vector registers is usually pipelined with multiple access paths.

Vector Operand Specification

Vector operands may have arbitrary length. Vector elements are not necessarily stored in contiguous memory locations. For example, the entries in a matrix may be stored in row major or in column major order. Each row, column or diagonal of the matrix can be used as a vector.

Vector operands should be stored in memory to allow pipelined or parallel access. The memory system for a vector processor must be specifically designed to enable fast vector access. The access rate should match the pipeline rate.

MULTIVECTOR MULTIPROCESSORS

The architectural design of supercomputers continues to be upgraded based on advances in technology and past experience. Design rules are provided for high performance and we review these rules in case studies of well-known early supercomputers, high-end mainframes and mini-supercomputers.

Performance-Directed Design Rules

Supercomputers are targeted towards large-scale scientific and engineering problems. They should provide the highest performance constrained only by current technology. In addition, they must be programable and accessible in a multiuser environment.

Supercomputer architecture design rules are presented below. These rules are driven by the desire to offer the highest available performance in a variety of respects, including processor, memory, and I/O performance, capacities and bandwidths in all subsystems.

Architecture Design Goals

Smith, Hsu and Hsiung identified the following four rules in the development of future general-purpose supercomputers:

- Maintaining a good vector/scalar performance balance.
- Supporting scalability with an increasing number of processors.
- Increasing memory system capacity and performance.
- Providing high-performance I/O and easy-access network.

Balanced vector/scalar ratio

In a supercomputer, separate hardware resources with different speeds are dedicated to concurrent vector and scalar operations. Scalar processing is indispensable for general-purpose architectures. Vector processing is needed for regularity structured parallelism in scientific and engineering computations. These two types of computations must be balanced.

I/O and Networking Performance

With the aggregate speed of supercomputers increasing at least three to five times each generation, problem size has been increasing accordingly, as have I/O bandwidth requirements.

The I/O is defined as the transfer of data between the processor/memory and peripherals or a network. In the earlier generation of supercomputers, I/O bandwidths were not always well correlated with computational performance. I/O processor architecture were implemented by Cray Research with two different approaches.

Memory Demand

A large scale-scale memory system must provide a low latency for scalar processing, a high bandwidth for vector and parallel processing and a large size for grand challenge problems and throughput.

Over the last two decades with advances in VLSI technology, the processing power available on a chip has tended to double every two years or so. Memory sizes available on a chip have also grown rapidly.

Supporting Scalability

Multiprocessor supercomputers must be designed to support the triad of scalar, vector and parallel processing. The dominant scalability problem involves support of shared memory with an increasing number of processors and memory ports. Increasing memory-access latency and interprocessor communication overhead impose additional constraints on scalability.

Cray Y-MP, C-90 and MPP

We study below the architecture of the Cray Research Y-MP, C-90 and MPP. Besides architectural features, we examine the operating systems and target performance of these machines.

Table 8.3 *Architectural Characteristics of Three Supercomputers of the 1990s*

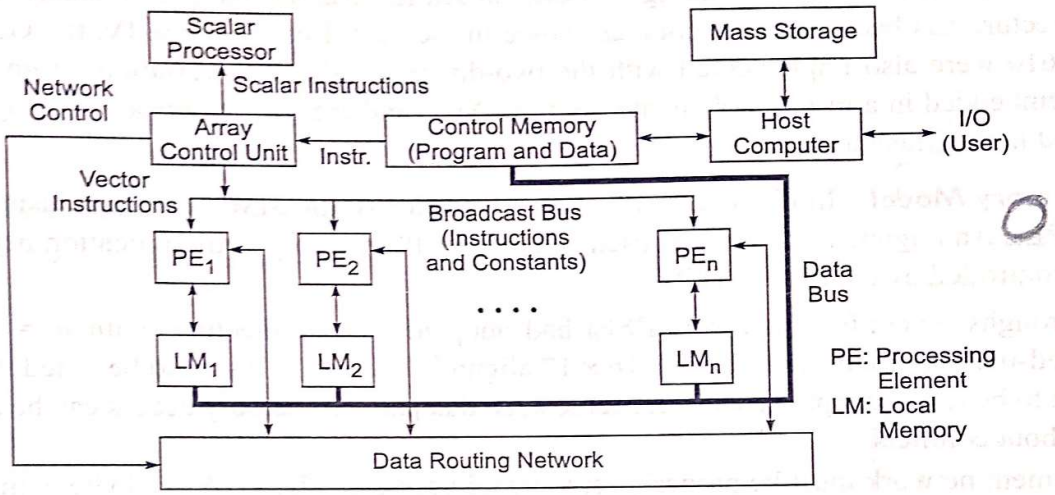
Machine Characteristics	*Cray Y-MP C90/16256*	*NEC SX-X Series*	*Fujitsu VP-2000 Series*
Number of processors	16 CPUs	4 arithmetic processors	1 for VP2600/10, 2 for VP2400/40
Machine cycle time	4.2 ns	2.9 ns	3.2 ns
Max. memory	256M words (2 Gbytes).	2 Gbytes, 1024-way interleaving.	1 or 3 Gbytes of SRAM.
Optional SSD memory	512M, 1024M, or 2048M words (16 Gbytes).	16 Gbytes with 2.75 Gbytes/s transfer rate.	32 Gbytes of extended memory.
Processor architecture: vector pipelines, functional and scalar units	Two vector pipes and two functional units per CPU, delivering 64 vector results per clock period.	Four sets of vector pipelines per processor, each set with two adder/shift and two multiply/logical pipelines. A separate scalar pipeline.	Two load/store pipes and 5 functional pipes per vector unit, 1 or 2 vector units, 2 scalar units could be attached to each vector unit.
Operating system	UNICOS derived from UNIX/V and BSD.	Super-UX based on UNIX System V and 4.3 BSD.	UXP/M and MSP/EX enhanced for vector processing.
Front-ends	IBM, CDC, DEC, Univac, Apollo, Honeywell.	Built-in control processor and 4 I/O processors.	IBM-compatible hosts.
Vectorizing languages / compilers	Fortran 77, C, CF77 5.0, Cray C release 3.0	Fortran 77/SX, Vectorizer/XS, Analyzer/SX.	Fortran 77 EX/VP, C/VP compiler with interactive vectorizer.
Peak performance and I/O bandwidth	16 Gflops, 13.6 Gbytes/s.	22 Gflops, 1 Gbyte/s per I/O processor.	5 Gflops, 2 Gbyte/s with 256 channels.

Implementation Models

Two SIMD computer models are described below based on the memory distribution and addressing scheme used. Most SIMD computers are a single control unit and distributed memories, except for a few that use associative memories.

Distributed-Memory Model

Spatial parallelism is exploited among the processing element in a SIMD computer. A distributed-memory SIMD computer consists of an array of processing elements which are controlled by the same array control unit. Program and data are loaded into the control memory through the host computer.

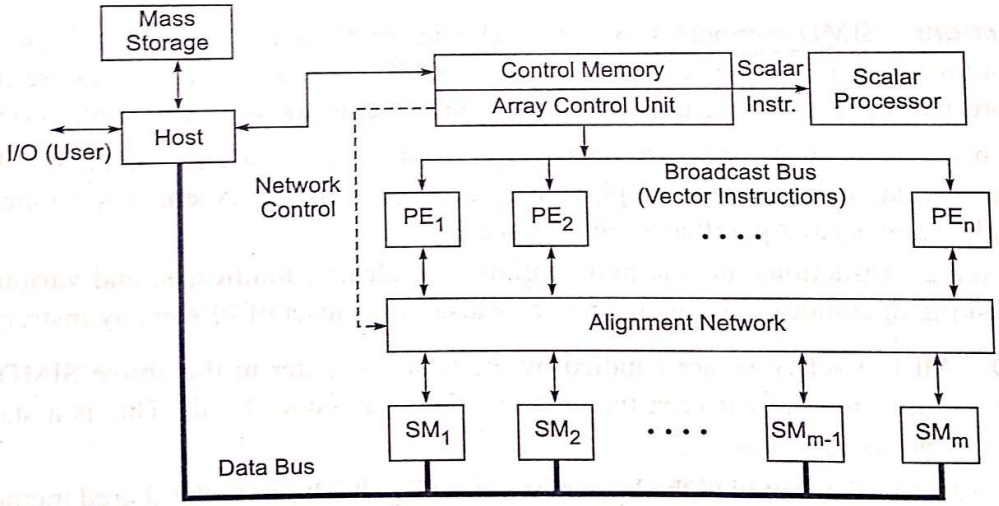


(a) Using distributed local memories (e.g. the Illiac IV)

An instruction is sent to the control unit for decoding. If it is a scalar or program control operation, it will be directly executed by a scalar processor attached to the control unit. If the decoded instruction is a vector operation, it will be broadcast to all the processing elements for parallel execution.

Almost all SIMD machines built have been based on the distributed-memory model. Various SIMD machines differ mainly in the data-routing network chosen for inter processing elements communications. The four neighbours mesh architecture has been the most popular choice in the past.

Shared-Memory Model



In the above fig, we show a variation of the SIMD computer using shared memory among the processing elements. An alignment network is used as the inter processing element memory communication network. Again, this network is controlled by the control unit.

The alignment network must be properly set to avoid access conflicts. Most SIMD computers were built with distributed memories. Some SIMD computers used bit-slice processing elements, such as the DAP610 and CM / 200.

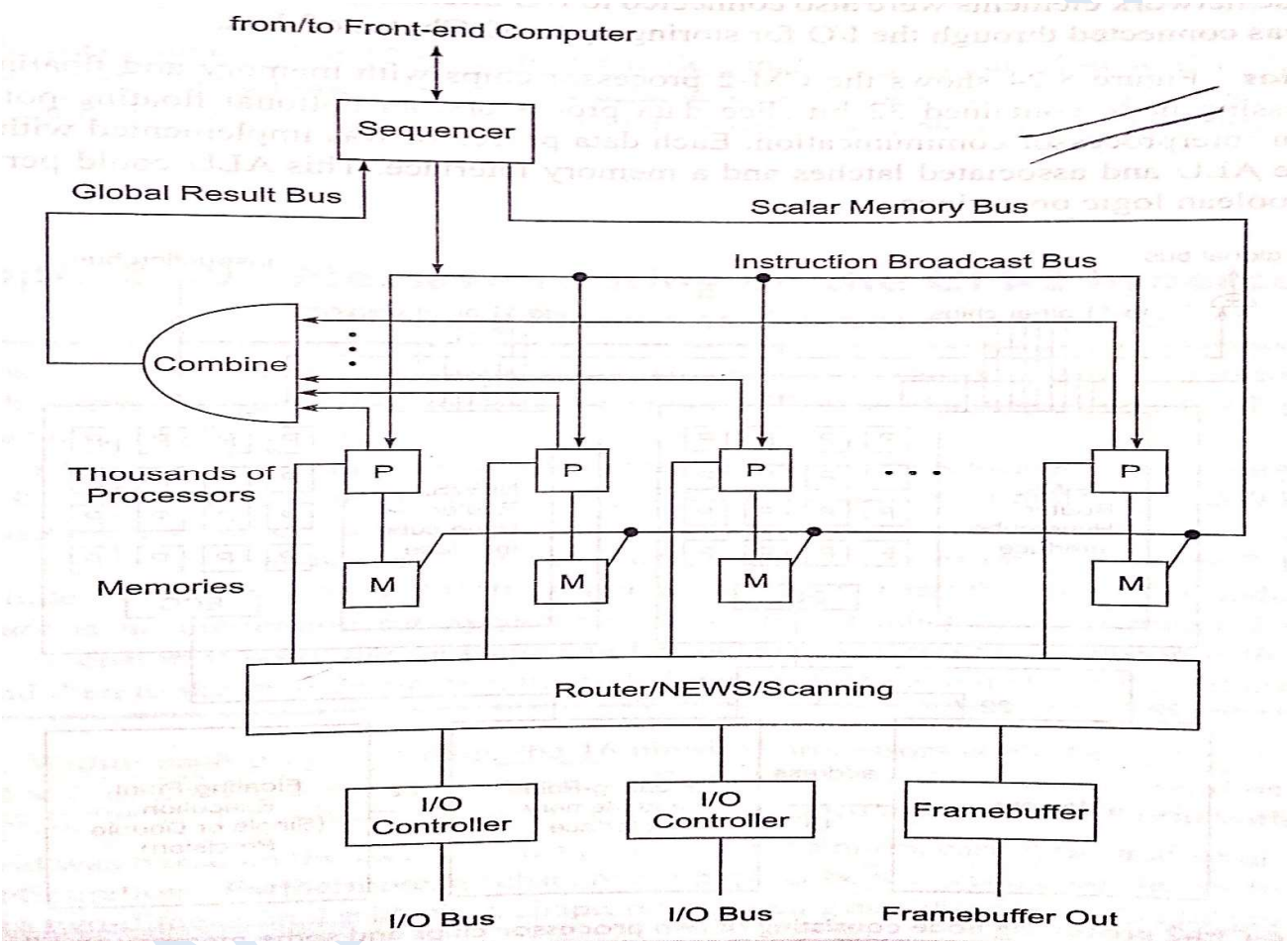
The CM-2 Architecture

The Connection Machine CM-2 produced by Thinking Machines Corporation was a fine-grain MPP (Massively Parallel Processing) computer using thousands of bit-slice processing elements in parallel to achieve a peak processing speed of above 10 Gflops. We describe the parallel architecture built into the CM-2.

Program Execution Paradigm

All programs started execution on a front-end, which issued microinstructions to the back-end processing array when data-parallel operations were desired. The sequencer broke down these microinstructions and broadcast them to all data processors in the array.

Data sets and results could be exchanged between the front-end and the processing array in one of three ways: broadcasting, global combining and scalar memory bus.



The processing Array

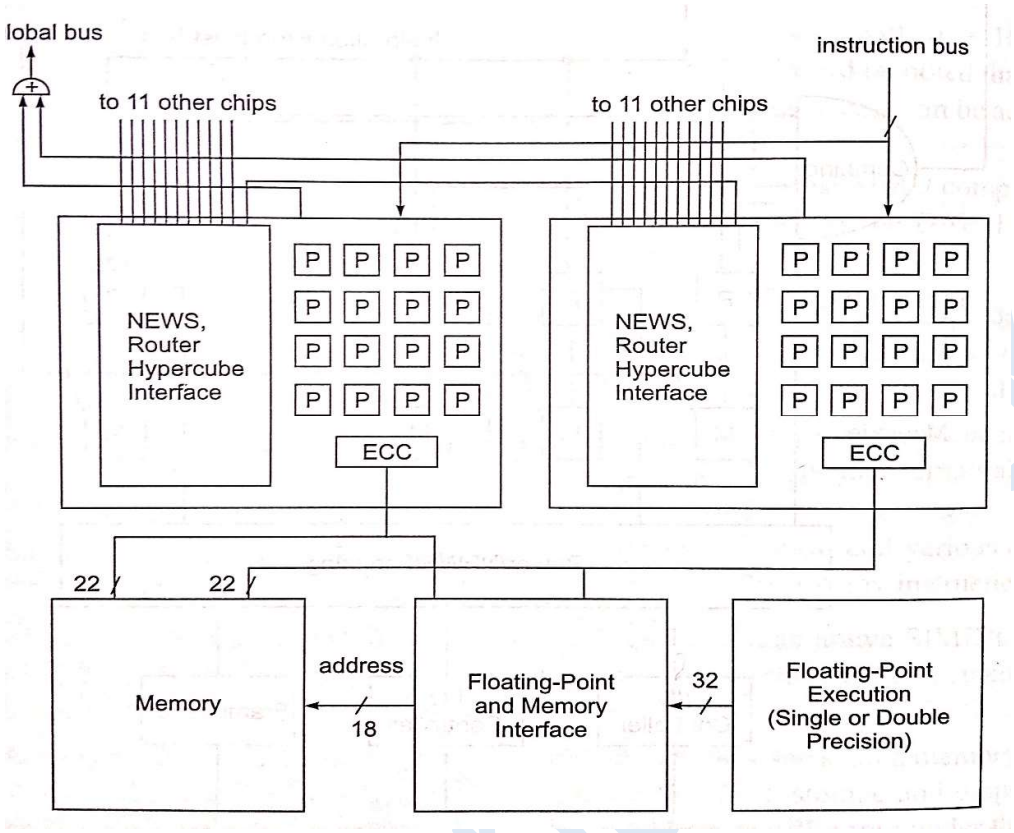
The CM-2 was a back-end machine for data-parallel computation. The processing array contained from 4K to 64K bit-slice data processors, all of which were controlled by a sequencer.

The sequencer decoded microinstructions from the backend and broadcast nanoinstructions to the processors in the array. All processors could access their memories simultaneously. All processors executed the broadcast instructions in a lockstep manner.

Processing Nodes

Figure below shows the CM-2 processor chips with memory and floating-point chips. Each data processing node contained 32 bit-slice data processors, an optional floating-point accelerator, and interfaces for interprocessor communication. Each data processor was implemented with a 3-input and 2-output bit-slice ALU

and associated latches and a memory interface. This ALU could perform bit-serial full-adder and Boolean logic operations.



Hypercube Routers

Special hardware was built on each processor chip for data routing among the processors. The router nodes on all processor chips were together to form a Boolean n-cube. A full configuration of CM-2 had 4096 router nodes on processor chips interconnected as a 12-dimensional hypercube.

Each router node was connected to 12 other router nodes, including its paired node. All 16 processors belonging to the same node equally capable of sending a message from one vertex to any other processor at another vertex of the 12-cube.

UNIT-IV

PARALLEL PROGRAMMING MODELS

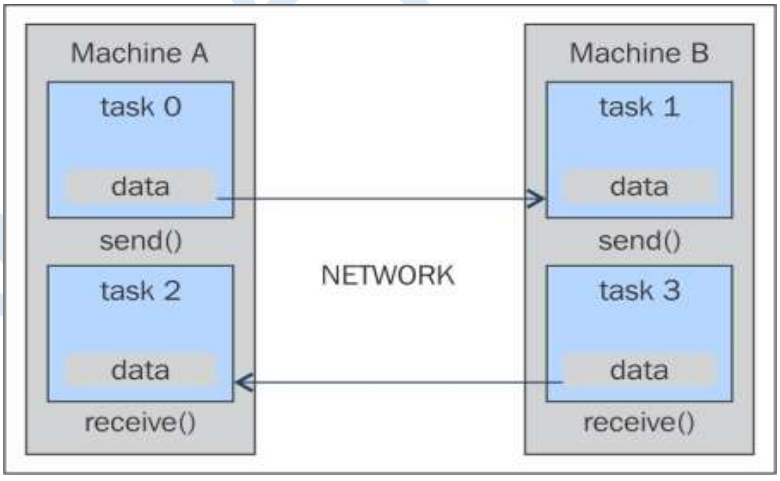
A programming model is a collection of program abstractions providing a programmer a simplified and transparent view of the computer hardware/software system. Parallel programming models are specifically designed for multiprocessors, multicomputers or vector/SIMD computers.

Shared variable model

In this model the tasks share a single shared memory area, where the access (reading and writing data) to shared resources is asynchronous. There are mechanisms that allow the programmer to control the access to the shared memory, for example, locks or semaphores. This model offers the advantage that the programmer does not have to clarify the communication between tasks. An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality; keeping data local to the processor that works on it conserves memory accesses, cache refreshes, and bus traffic that occur when multiple processors use the same data.

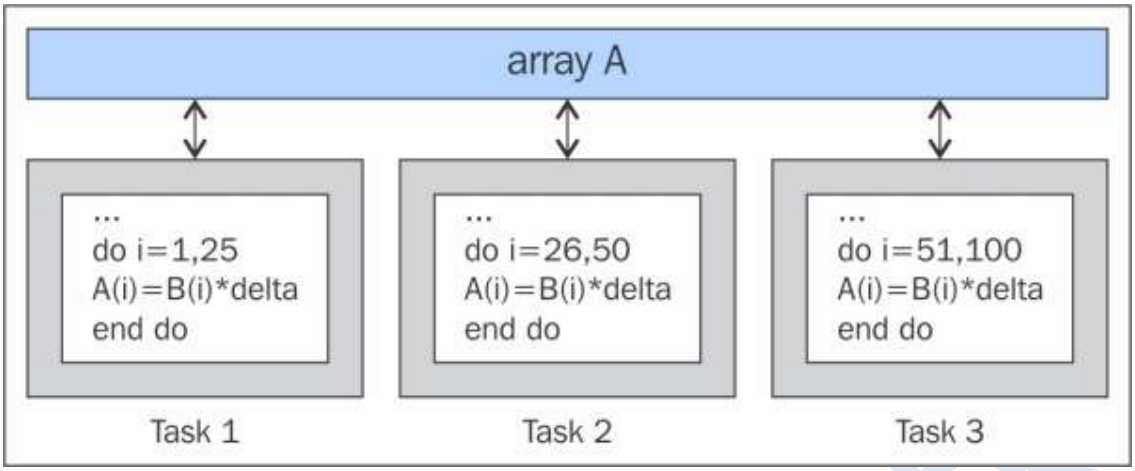
Message passing model

The message passing model is usually applied in the case where each processor has its own memory (distributed memory systems). More tasks can reside on the same physical machine or on an arbitrary number of machines. The programmer is responsible for determining the parallelism and data exchange that occurs through the messages. The implementation of this parallel programming model requires the use of (ad hoc) software libraries to be used within the code. Numerous implementations of message passing model were created: some of the examples are available since the 1980s, but only from the mid-90s, was created to standardized model, coming to a de facto standard called MPI (the message passing interface). The MPI model is designed clearly with distributed memory, but being models of parallel programming, multiplatform can also be used with a shared memory machine.



Data parallel model

In this model, we have more tasks that operate on the same data structure, but each task operates on a different portion of data. In the shared memory architecture, all tasks have access to data through shared memory and distributed memory architectures, where the data structure is divided and resides in the local memory of each task. To implement this model, a programmer must develop a program that specifies the distribution and alignment of data. The current generation GPUs operates high throughout with the data aligned.



Object oriented model

In this model, objects are dynamically created and manipulated. Processing is performed by sending and receiving messages among objects. Concurrent programming models are built up from low-level objects such as processes, queues and semaphores into high-level objects like monitors and program modules.

Functional and Logic Models

Functional Programming Model

A functional programming language emphasizes the functionality of a program and should not produce side effects after execution. There is no concept of storage, assignment, and branching in functional programs. In other words, the history of any computation performed prior to the evaluation of a functional expression should be irrelevant to the meaning of the expression.

Logic Programming Model

Based on predicate logic, logic programming is suitable for knowledge processing dealing with large databases. This model adopts an implicit search strategy and supports parallelism in the logic inference process. A question is answered if the matching facts are found in the database. Two facts match if their predicates and associated arguments are the same. The process of matching and unification can be parallelized under certain conditions.

PARALLEL LANGUAGE AND COMPILERS

The environment for parallel computers is much more demanding than that for sequential computers. A programming environment is a collection of software tools and system software support. Users should not have to spend a lot of time programming hardware details: they should focus instead on program parallelism using high-level abstractions.

Language Features for Parallelism

Some of the features are described below:

Optimization Features

These features are used for program restructuring and compilation directives in converting sequentially coded programs into parallel forms. The purpose is to match the software parallelism with the hardware parallelism in the target machine.

- Automated parallelizer: Examples are- Express C automated parallelizer and the Alliant FX Fortran compiler.
- Semiautomated parallelizer: Needs compiler directives or programmer's interaction, such as DINO.
- Interactive restructure support: Static analyser, run-time statistics, dataflow graph and code translator for restructuring Fortran code.

Availability Features

These are features that enhance the user-friendliness, make the language portable to a large class of parallel computers and expand the applicability of software libraries.

- Scalability: The language is scalable to the number of processors available and independent of hardware topology.
- Compatibility: The language is compatible with an established sequential language.
- Portability: The language is portable to shared-memory multiprocessors, message-passing multicomputers or both.

Synchronization/Communication Features

Listed below are desirable language features for synchronization or for communication process:

- Single-assignment language
- Shared variables for IPC
- Logically shared memory such as the tuple space in Linda
- Send/Receive for message passing
- Remote procedure call

Control of Parallelism

Listed below are features involving control constructs for specifying parallelism in various forms:

- Coarse, medium or fine gold
- Global parallelism in the entire program
- Loop parallelism in iterations
- Task-split parallelism
- Divide and conquer paradigm

Data Parallelism Features

Data parallelism is used to specify how data are accessed and distributed in either SIMD and MIMD computers.

- Run-time automatic decomposition – Data are automatically distributed with no user interventions, as in Express
- Mapping specification – Provides a facility for users to specify communication patterns or how data and processes are mapped onto the hardware.
- Virtual processor support- The compiler maps the virtual processors dynamically or statically onto the physical processors.
- Direct access to shared data – Shared data can be directly accessed without monitor control, as in Linda.

Parallel Language Constructs

Special language constructs and data array expressions are presented below for exploiting parallelism in programs.

Fortran 90 Array Notations

A multidimensional data array is represented by an array name indexed by a sequence of subscript triplets, one for each dimension. Triplets for different dimensions are separated by commas. Examples are:

$$e_1 : e_2 : e_3$$

$$e_1 : e_2$$

$$e_1 : * : e_3$$

$$e_1 : *$$

$$e_1$$

$$*$$

Where each e is an arithmetic expression that must produce a scalar integer value. The first expression e_1 is a lower bound, the second e_2 an upper bound and third e_3 an increment.

When the third expression in a triplet is missing, a unit stride is assumed. The * notation in the second expression indicates all elements in that dimension starting from e_1 or the entire dimension if e_1 is also omitted.

Parallel Flow Control

The conventional Fortran Do Loop declares that all scalar instructions within the pair are executed sequentially and so are the successive iterations. To declare parallel activities, we use the (Doall, Endall) pair. All iterations in the Doall loop are totally independent of each other. This implies that they can be executed in parallel if there are sufficient processors to handle different iterations.

When the successive iterations of a loop depend on each other, we use the (Doacross, Endacross) pair to declare parallelism with loop-carried dependences. Synchronizations must be performed between the iterations that depend on each other.

Optimizing Compilers for Parallelism

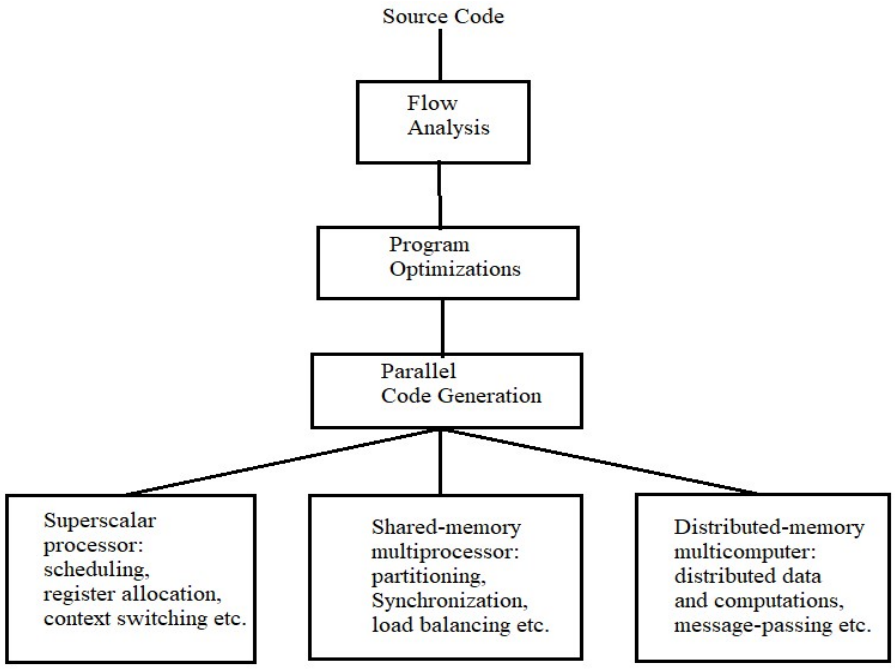
Because high-level languages are used almost exclusively to write programs today, compilers have become a necessity in modern computers. The role of a compiler is to remove the burden of program optimization and code generation from the programmer. A parallelizing compiler consists of the following three major phases:

Flow Analysis

This phase reveals the program flow patterns in order to determine data and control dependences in the source code. Scalar dependence analysis is extended below to structure data arrays or matrices. Depending on the machine structure, the granularities of parallelism to be exploited are quite different. Thus the flow analysis is conducted at different execution levels on different parallel computers.

Program Optimizations

This refers to the transformation of user programs in order to explore the hardware capabilities as much as possible. Transformation can be conducted at the loop level, locality level or prefetching level with the ultimate goal of reaching global optimization.



In reality most transformations are constrained by the machine architecture. This is the main reason why many such compilers are machine dependent. At least, we want to design a compiler which can run on most machines with only minor modifications. One can also conduct certain transformations preceding the global optimization.

Parallel Code Generation

Code generation usually involves transformation from one representation to another, called an intermediate form. A code model must be chosen as an intermediate form. Parallel code is even more demanding because parallel constructs must be included. Code generation is closely tied to the instruction scheduling policies used.

Parallel code generation is very different for different computer classes. For example, a superscalar processor may be software-scheduled or hardware-scheduled. How to optimize the register allocation on a RISC or superscalar processor, how to reduce the synchronization overhead when codes are partitioned for multiprocessor execution.

PARALLEL PROGRAMMING ENVIRONMENT

The last two decades have seen a revolution in massively parallel computer architecture, with system performance reaching hundreds of teraflops and even petaflops. Huge advances in processors, memory, display systems, system interconnects and networking have contributed to this revolution, while the range of applications of such systems has also grown enormously.

Software Tools and Environments

Parallel programming languages such as Linda and Strand 88 provided the minimal parallel programming environments. Integrated environments can be divided into basic, limited and well-developed classes, depending on the maturity of the tool sets.

A basic environment provides a simple program tracing facility for debugging and performance monitoring or a graphic mechanism for specifying the task dependence graph in SCHEDULE, the process call graph in FAUST and the process component graph in PIE.

Environment Features

In designing a parallel programming language, one often faces a dilemma involving compatibility, expressiveness, ease of use, efficiency and portability. Parallel languages are developed either by introducing new languages such as Linda and Occam or by extending existing sequential language such as Fortran 90, C* and Concurrent Pascal.

Most parallel computer designers choose the language extension approach to solving the compatibility problem. High-level parallel constructs were added to Fortran, C, Pascal and Lisp to make them suitable for use on parallel computers. Special optimizing compiler are needed to automatically detect parallelism and transform sequential constructs into paralleled ones.

Y-MP, Paragon and CM-5 Environments

The software and programming environments of the Cray Y-MP, Intel Paragon XP/S and Connection Machine CM-5 are examined below.

Cray Y-MP Software

The Cray Y-MP ran with the Cray operating systems UNICOS or COS. Two Fortran compiler, CFT77 and CFT, provided automatic vectorizing, as did C and Pascal compilers. Large library routines, program management utilities, debugging aids and a Cray assembler were included in the system software.

UNICOS for Y-MP was written in C and was a time-sharing OS extension of UNIX. UNICOS supported optimizing, vectorizing, concorrentizing Fortran compilers and optimizing and vectorizing Pascal and C compilers.

COS was a multiprogramming, multiprocessing and multitasking OS. It offered a batch environment to the user and supported interactive jobs and data transfers through the front-end system. COS programs could run with a maximum of four processors up to 16M words of memory on the Y-MP system.

Intel Paragon XP/S Software

The Intel Paragon XP/S system was an extension of the Intel iPSC/860 and Delta systems. It was claimed to be a scalable, wormhole, mesh-connected multicomputer using distributed memory. The processing nodes used were 50-MHz i860 XP processors.

CM-5 Software

The CM-5 designers aimed at independent scalability of processing, communication and I/O. this aim must necessarily be supported by extensive software, language and application libraries.

The operating system was CMOST, an enhanced version of UNIX/OS which supported time-sharing and batch processing. The low-level languages and libraries matched the hardware instruction-set architecture.

CMOST provided not only standard UNIX services but also supported fast IPC capabilities and data-parallel and other programming models. It also extended the UNIX I/O environment to support parallel reads and writes and managed large files on data vaults.

Visualization and Performance Testing

The performance of a parallel computer can be enhanced at several stages: At the machine design stage, the architecture and OS should be optimized to yield high resource utilization and maximum system throughput. At the algorithm design/data structuring stage, the programmer should match the software with the target hardware.

Visualization Support

In I/O and program development areas, visualization is also needed. Performance tuning requires extensive software experiments with the help of GUI and utilization support. Tuning an operating system and an application program requires effort from both ends. System tuning involves the tuning of virtual memory and process priorities, such as adjusting the resident set size and scheduling policy.

Even the best computer architect cannot guarantee the performance of a system until the machine is tested by actually running real software programs. During the architecture design stage, simulation may be used to predict performance.

Performance Tuning

If special hardware/software mechanism are available, one can also collect run-time information, such as processor and memory utilization patterns, to guide the code transformation. One can also conduct a critical-path analysis of programs in order to reveal the bottleneck. Bottleneck removal or shortening the critical path through grain packing or other techniques can improve the system performance.