Assignment - I

1] ① Unifunⁿ pipeline

:- When a fixed & dedicated funⁿ is performed through a pipline.

② Data dependency Hazards

:- It occurs when an instruction depends on the result of previous instruction & that result of instruction has not yet been computed.

There are 4 types of data dependencies

a] Read After Write (RAW), b] Write After Read (WAR)
c] Write After write (WAW) d] Read After Read (RAR)

③ Pipeline Throughput

: Pipeline is a technique where multiple instructions are over-lapped during exception. Pipeline is divided into stages & these stages are connected with one another to form a pipe like structure. Insⁿ enter from one end & exit from other end. Pipling increases overall insⁿ throughput

④ Static Pipeline

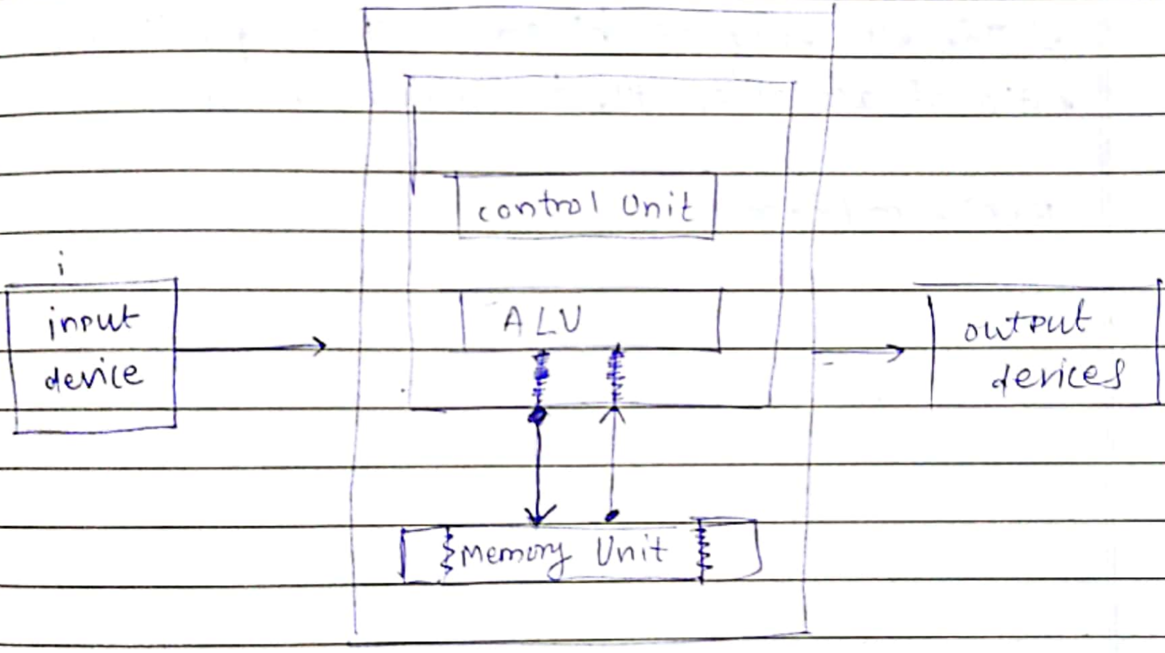
:- A static pipeline can be viewed as a two-stage processor with the two stages being fetch & everything after fetch. As discussed in next sub-section, the statically pipelined insⁿ are already partially decoded as compared to traditional insⁿ.

It divides an arithmetic problem into various sub problems for execution in various pipeline segments.

⑤ scalar pipeline

:- This type of pipeline processes ~~under~~ scalar operands of repeated scalar insⁿ. It multiply the funⁿ units, the same sub-task from different operation run in parallel.
It considered to be the simplest of all processors, works on one or two computer data items at a given time

2] Architecture of CPU



① It is the electronic circuity within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logical, control & i/p operations specified by insⁿ

② parts of CPU
a] ALU :- executes all calculations within CPU
b] CU :- coordinates how data moves around, decodes insⁿ.
c] PC :(Program counter):- stores address of next insⁿ in RAM.
d] MAR:(Memory Address Register):- stores address of current insⁿ being executed.
e] MDR:-(Memory Data Register):- stores data that is to be sent to or fetched from memory.

* pipeline structure in CPU

:- Pipelining is the process of accumulating ins^n from the processor through pipeline. It allows storing & executing instructions in an orderly process. It is also known as pipeline processing. Pipelining is a technique where multiple ins^n. are overlapped during execution.

* efficiency of linear pipeline.

:- The efficiency of 'n' stages in a pipeline is defined as ratio of actual speedup to maximum speed.

$E(n) = m/n+m-1$

4] characteristics of vector processing

① virtual memory plays major supporting role in super-computer.

Assit in memory management :- The OS can commit & decommit arbitany blocks of real memory without having to ensure the physical contiguity of user's workspace. This reduces overhead for actions such as the accumulation of unused space, which could be quite costly in large memory systems.

② provide identically appearing execution of all jobs :- This means that a program's dimension statements & input parameters can remain unchanged whether or not a 4-hour run is being contemplated or a simple debugging run of a particular phase is intended.

③ Eliminate working space constraints from algorithm developement

:- Maths & programmers can begin developing an algorithm as if they had available workspace in which to put data & temporary results. Once the algorithm is developed, the programmer must introduce the means to handle paging of information in order to optimize the performance of system & to eliminate the thrashing that can occur in virtual memory machines moving data to & from real central memory

Assignment - 2

1] ① Associative memory

:- a] It can be defined as a memory system with the property that stored data items can be retrieved by their content or part of their content.

b] A It has been also called catalog memory, content addressed memory, data-addressed memory, parallel search memory.

c] The major advantages of associative memory over RAM is its capability of performing parallel search & parallel comparison operiations.

d] There are no. of registers & counters in associative memory

e] The comparand register C is used to hold the key operand being searhed for or being compared with.

f] The indicator register I & temporary register T are used to hold the current & previous match patterns respectively.

★ How its (Associative memory) different from RAM
(AM)
:

① In RAM, the user supplies a memory address & RAM returns data word stored at the address. In AM the user supplies data word & associative memory searches its entire ~~more~~ memory.

② RAM is used to store computer programs & data that CPU needs in real time.
AM is used to stored data words & AM searches its memory for provided word

2) Latency & latency hiding w/ techniques

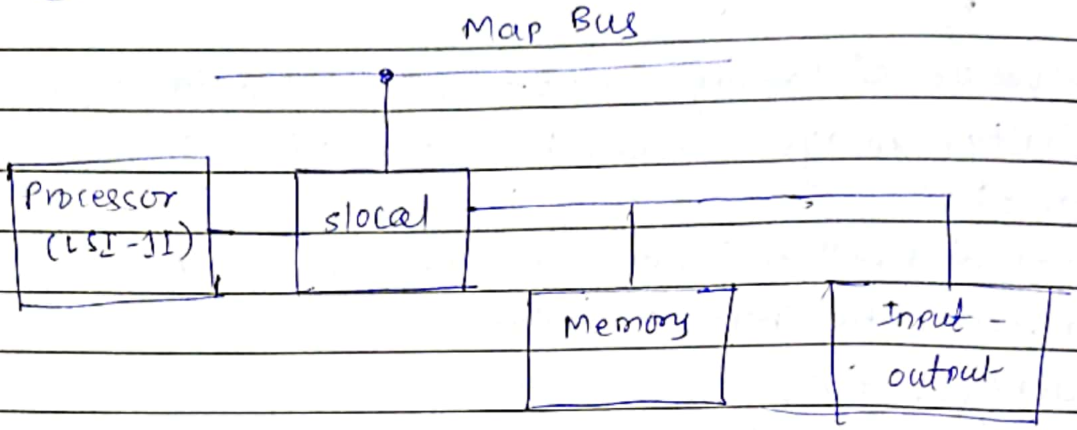
→ The value of latency includes:- network delays, cache-miss penalty, delays caused by contentions in split transactions.
Interval betn switches refers to cycles betn switches triggered by remote reference.

*Latency hiding
1. Prefetching technique
2. using coherent caches
3. Using relaxed memory consistency models
4. Using multiple contexts supports

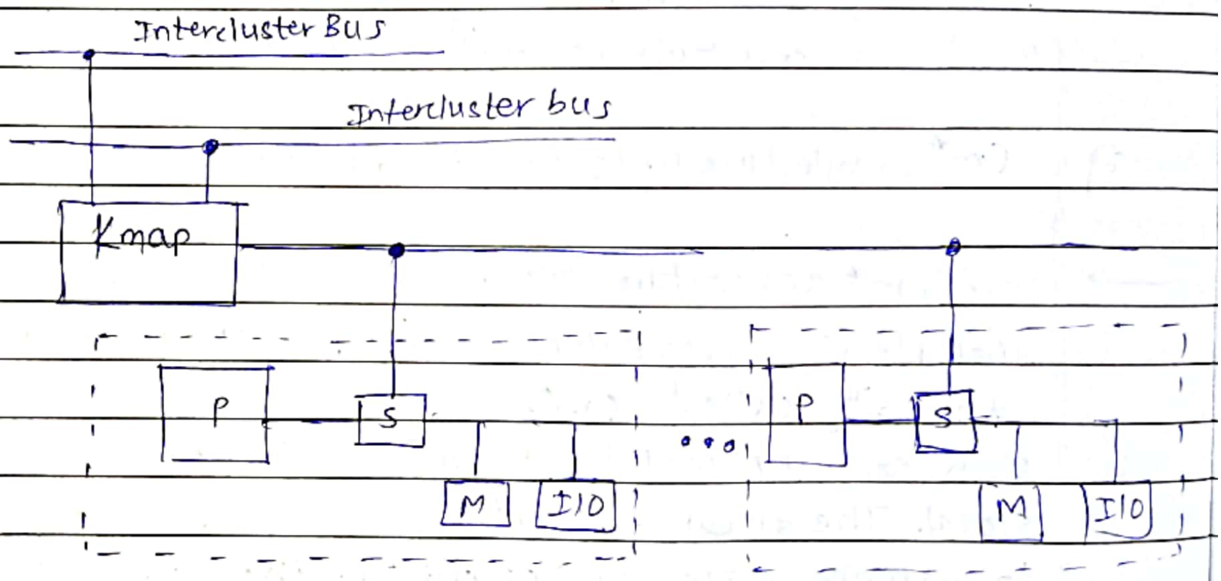
3.) Cm* architecture in loosely coupled system

→ The cm* architecture for hierarchical LCS, we consider the example of a computer system project at Carnegie Mellon university called cm*.
Each computer module of cm* includes a local switch called Slocal. The slocal intercepts & routes the processors requests to memory & I/O devices outside the computer module via a map bus. It also accepts references from other computer modules to its local memory & I/O devices.
The address translation uses four high order bits of the processors address along with the current address space, as indicated by x-bit of LSI-11 processor status word

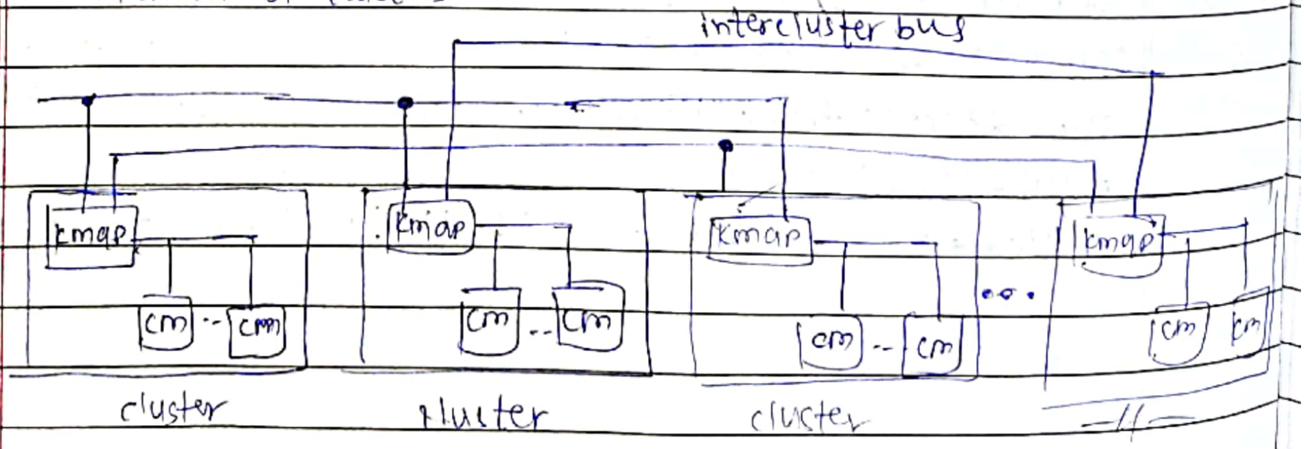
(a) computer module



(b) cluster of computer Modules



(c) Network of clusters.



Assignment No.3

1) **What are different components of Kmap in Cm* architecture? State function of each component in detail**

2) **With block diagram explain dynamic and static dataflow architecture**

3) **What are different parallel programming models?**

   1) Shared Memory Model-

      In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks / semaphores may be used to control access to the shared memory. An advantage of this model from the programmer's point of view is that program development can often be simplified. An important disadvantage of this model (in terms of performance) is that for this model, data management is difficult.

   2) Threads Model-

      In this model a single process can have multiple, concurrent execution paths. The main program is scheduled to run by the native operating system. It loads and acquires all the necessary softwares and user resources to activate the process. A thread's work may best be described as a subroutine within the main program. Any thread can execute any one subroutine and at the same time it can execute other subroutine. Threads communicate with each other through global memory. This requires Synchronization constructs to insure that more than one thread is not updating the same global address at any time. Threads can be created and destroyed, but the main program remains live to provide the necessary shared resources until the application has completed. Threads are commonly associated with shared memory architectures and operating systems.

   3) Message Passing Model-

      In the message-passing model, there exists a set of tasks that use their own local memories during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines. Tasks exchange data by sending and receiving messages. In this model, data transfer usually requires cooperation among the operations that are performed by each process. For example, a send operation must have a matching receive operation.

   4) Data Parallel Model –

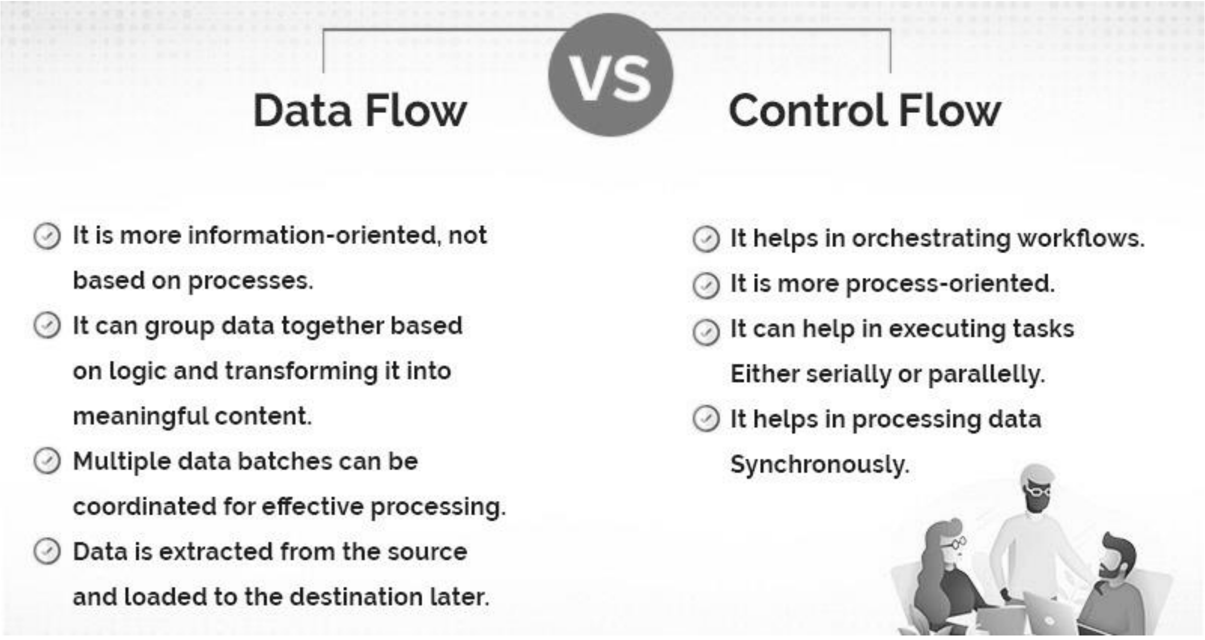
      In the data parallel model, most of the parallel work focuses on performing operations on a data set. The data set is typically organised into a common structure, such as an array or a cube. A set of tasks work collectively on the same data structure with each task working on a different portion of the same data structure. Tasks perform the same operation on their partition of work, for example, “add 3 to every array element” can be one task. In shared memory architectures, all tasks may have access to the data structure through the global memory. In the distributed memory architectures, the data structure is split up and data resides as “chunks” in the local memory of each task.

   5) Hybrid model-

      The hybrid models are generally tailormade models suiting to specific applications. Actually these fall in the category of mixed models. Such type of application-oriented models keep cropping up. Other parallel programming models also exist, and will continue to evolve corresponding to new applications. In this types of models, any two or more parallel programming models are combined.

Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines. Another common example of a hybrid model is combining data parallel model with message passing model. As mentioned earlier in the data parallel model, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data transparently between tasks and the programmer.

4) Compare between control flow and data flow computing models



VS

Data Flow

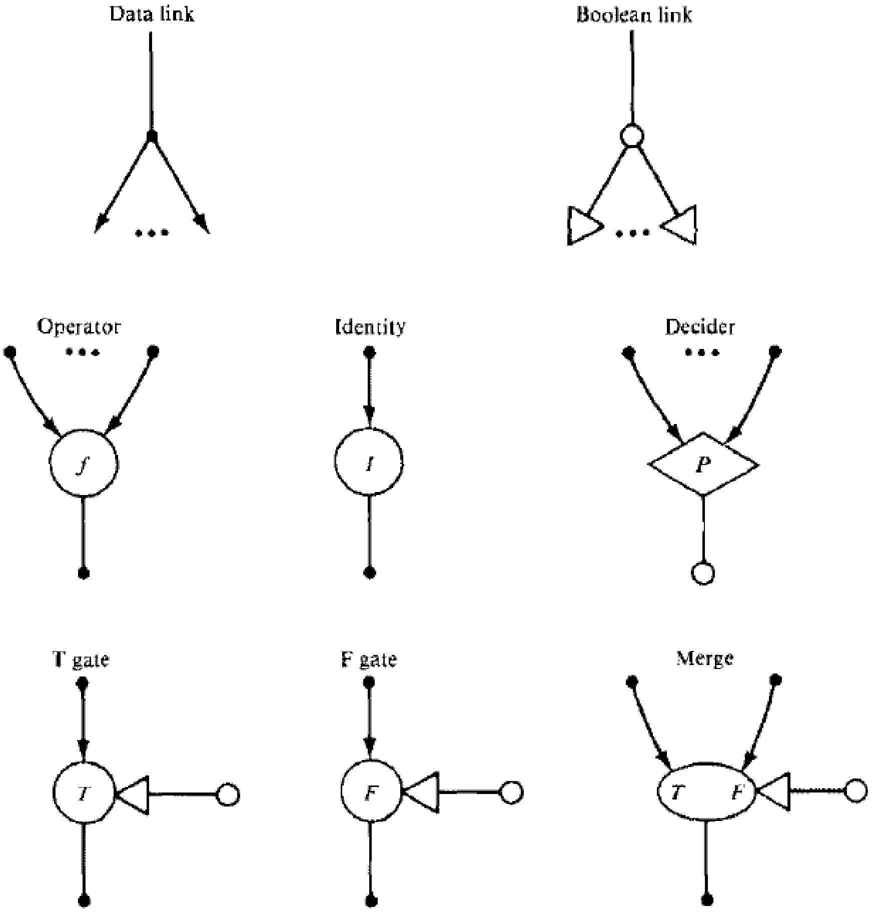
- It is more information-oriented, not based on processes.
- It can group data together based on logic and transforming it into meaningful content.
- Multiple data batches can be coordinated for effective processing.
- Data is extracted from the source and loaded to the destination later.

Control Flow

- It helps in orchestrating workflows.
- It is more process-oriented.
- It can help in executing tasks Either serially or parallelly.
- It helps in processing data Synchronously.

Assignment No.4

1. **Draw and explain different operators and links used for construction of dataflow graphs.**



a. Above figure shows various Operators and Links used to construct data-flow graph.
b. Numerical data links transmit integer, real or complex numbers and boolean links carry only boolean values for control purpose.
c. An identity operator is a special operator that has one input and transmit its input value unchanged.
d. Deciders, gates and merge operators are used to represent conditional and iterative computation in data flow graphs. A Deciders requires a value from each input are and produces the truth value resulting from applying the predicate P to the value received
e. Control tokens bearing boolean values control the flow of data tokens by means of the T gate, the F gate and the merge operator.
f. A T gate passes a data token from its data input arc to its output arc when it receives the true value on its control input arc
g. A merge operator has T and F data input arcs and a truth value control arc. When a truth value is received, the merge actor places the token from the true input arc on its output arc.

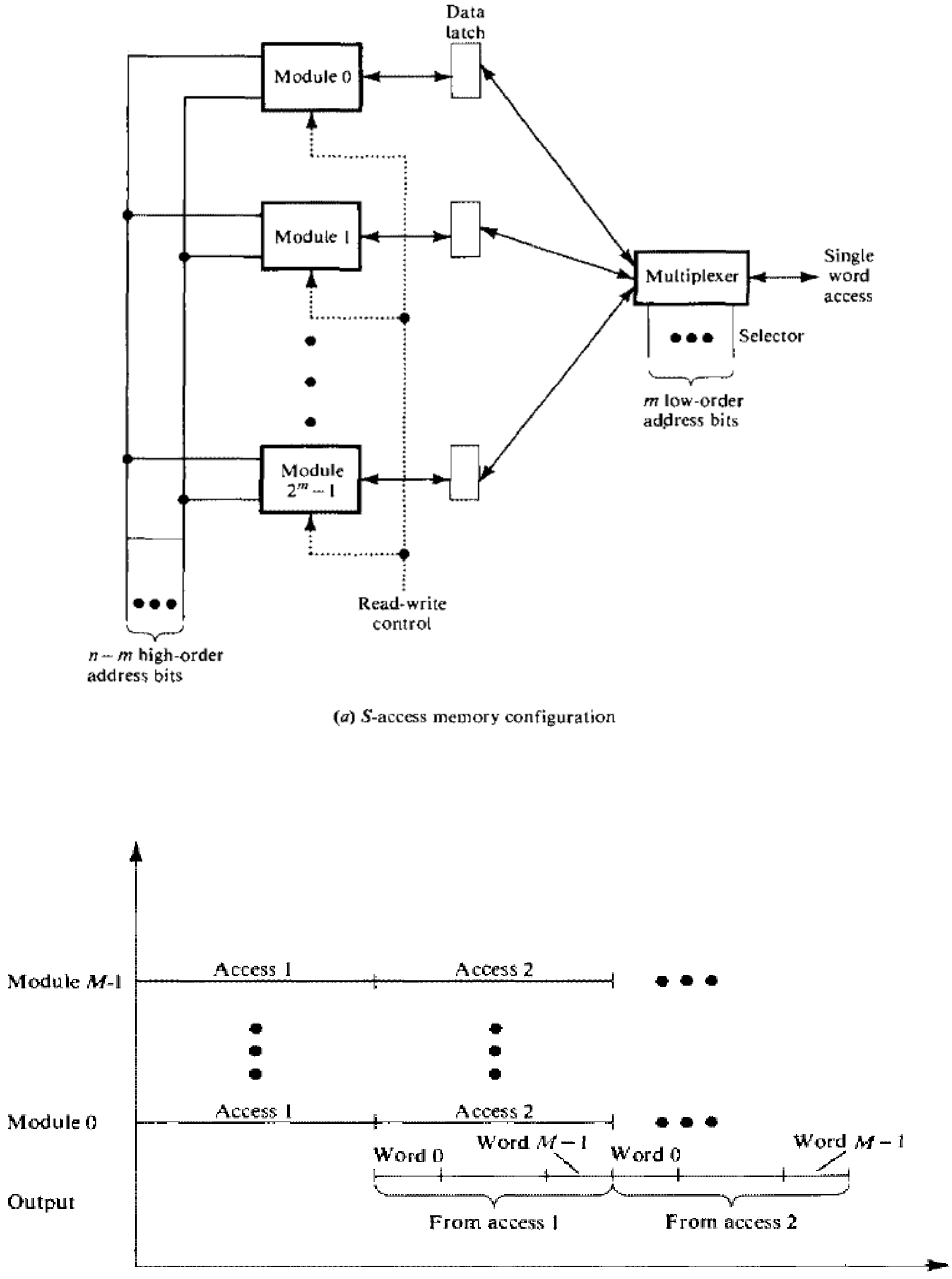
2. What are interleaved memory organizations used in Pipeline Vector Processing.

i) S-access

ii) C-access

iii) C/S access

i) S-access: -



(*a*) *S*-access memory configuration

(*b*) Timing diagram for *S*-access configuration

Figure: - The S-access interleaved memory configuration.

a. One of the simplest memory configuration for pipelining vector processor use lower-order interleaving and applies the higher (n – m) bits of the address to all M = 2^m memory modules simultaneously in one access. The single access returns M consecutive words of information from the M memory models. Using the lower-order m address bots, the information from a particular modules can be accessed.

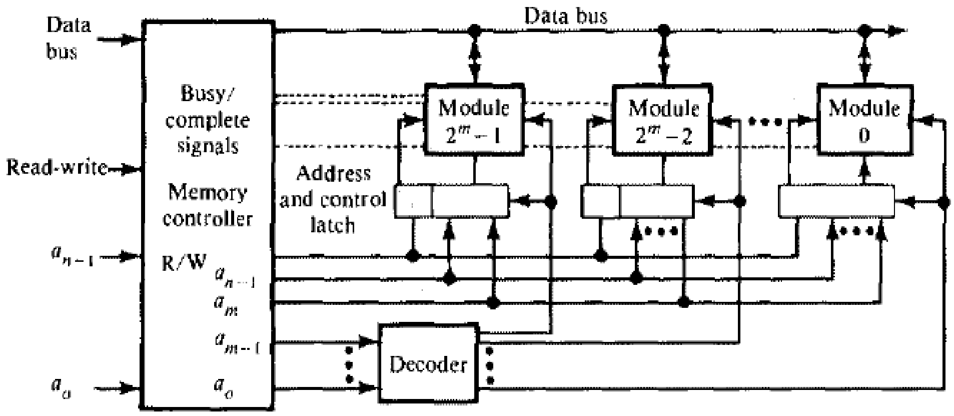
b. The configuration, which is shown in figure (a) is called as S-access because all modules are accessed simultaneously.

c. A data latch is associated with each modules. For a fetch operation, the information from each modules is gated into its latch, whereupon the multiplexer can be used to direct the desired data to the single-word bus.

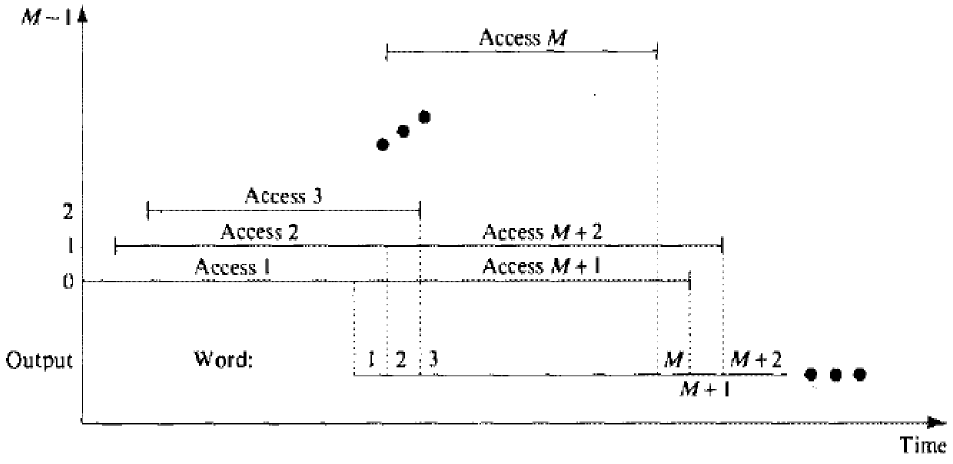
d. Figure (b) shows the timing diagram for sample multiple-word read accesses using the S-access configuration.

e. It can be used to access a block of information or a pipeline processor with a cache

ii) C-access: -



(a) C-access memory configuration



(b) Timing diagram for accesses to consecutive addresses

a. When a memory operation is initiated in a module, it causes the bank to be activated for t_a seconds and the module to be active for t_c seconds. If t_a is much less than t_c the initiated module uses the bank for a duration much less than one memory cycle per access. Therefore, more than one module can share a bank, increasing the bank utilization and reducing the bank cost. This configuration is called as C-access because modules are accessed concurrently, as shown in figure (a).
b. The lower order m bits are used to select the module and the remaining n – m bits address the desired element within the module.
c. The memory controller is used to buffer a request which both references a busy module and initiate services when the module completes its current cycle.
d. Figure (b) shows an example timing diagram where K consecutive words are fetched.

iii)

3. Draw multithreaded architecture

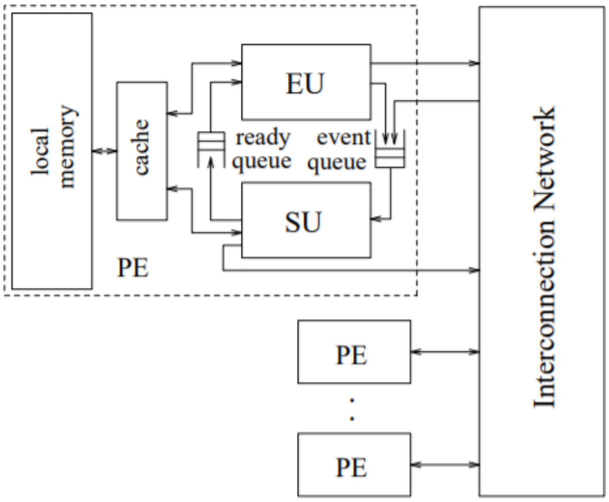


Figure 1: The Multi-Threaded Architecture

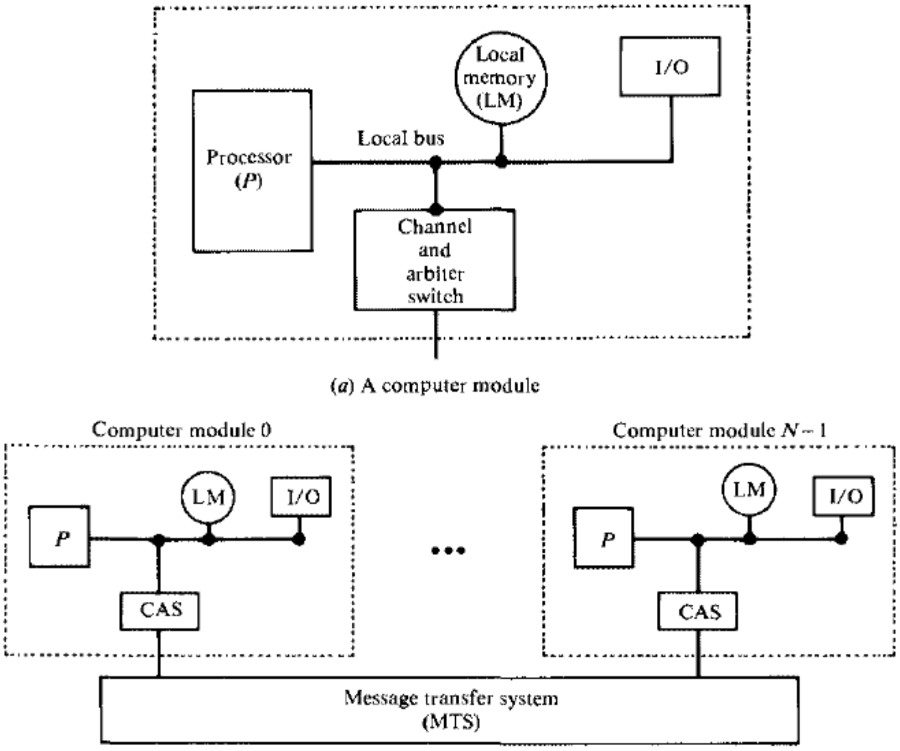
a. Each MTA node contains an Execution Unit (EU) and a Synchronization Unit (SU) linked together by buffers (see Figure 1). In simple terms, the EU executes the instructions that comprise the program, while the SU tells the EU which instructions to execute. This organization is derived from the Argument-Fetch Data flow Processor [6], in which the analogous units are called the Pipelined Instruction Processing Unit (PIPU) and the Data flow Instruction Scheduling Unit (DISU), respectively.
b. The SU and EU share the node's local memory, which is cached for better performance. Accessing data in a remote node requires explicit `request' and `send' messages; thus hardware wise, the MTA multiprocessor is a distributed memory machine. However, this does not preclude the use of software and/or limited hardware mechanisms for projecting a global address space to the compiler; which would make the MTA a distributed shared memory machine.
c. The EU consists of a register file and a RISC execution pipeline. The register file contains registers for storing temporary values, and special registers such as a frame pointer register (%fp) and a status register (%sr). The execution pipeline processes instructions in an active thread, using standard RISC register-to-register and load/store operations. Moreover, it can inject messages into the interconnection network and issue synchronization signals to the local SU.
d. The buffer between the SU output and EU input is called the ready queue, and holds the threads that are waiting for execution. A thread becomes active (initiated for execution) when the EU fetches its ready thread identifier (or ready thread id for short) from the ready queue. The execution pipeline of the EU processes an active thread until the thread relinquishes control of the pipe with a special instruction which fetches the next available ready thread id from the ready queue and context-switches to the new thread. It is the

responsibility of the SU to generate ready thread ids of threads to be executed and deposit them into the ready queue.

e. The SU is also responsible for processing messages in the event queue. Events in the event queue include local synchronization signals emitted by the EU of the local processor, as well as remote synchronization signals and communication messages delivered from the interconnection network. Lastly, the SU is solely responsible for handling remote requests for data.

4. **How the degree of memory conflicts is not encountered in loosely coupled multiprocessor systems? Draw and explain Non-hierarchical loosely coupled multiprocessor system.**

Loosely coupled multiprocessor system do not generally encounter the degree of memory conflict experienced by tightly coupled system. In such systems, each processor has a set of input-output devices and a large local memory where it accesses most of instructions and data. We refer to the processor, its local memory and I/O interfaces as a computer module. Processes which execute on a different computer modules communicate by exchanging message through a message-transfer (MTS). The degree a coupling in such system is very loose. Hence it if often referred to as a distributed system. The determinant factor of the degree of coupling if the communication topology of the associated message transfer system.



(*a*) A computer module

(*b*) Loose coupling of computer modules

Figure: - Non- hierarchical loosely coupled multiprocessor system.

a. Figurer (a) shows a computer module of a non-hierarchical loosely coupled multiprocessor system
b. It consist of a processor, a local memory, local input-output devices and an interface to another computer module. The interface may contain a channel and arbiter switch (CAS).
c. Figure (b) shows the connection between the computer modules and a message-transfer system.
d. If request from two or more different computer modules collide in accessing a physical segment of the MTS, the arbiter is responsible for choosing one of the simultaneous request according to a given service discipline. It is also responsible for delaying other request until the serving of the selected request is completed.
e. The channel within the CAS may have high speed communication memory which is used for buffering block transfer of message. The communication memory is accessible by all processors.
f. The message transfer system for a non-hierarchical Loosely Coupled multiprocessor System could be a simple time shared bus, as in the PDP-11, or a shared memory system.
g. The letter case can be implemented with a set of memory modules and a processor-memory interconnection network or a multiport main memory.

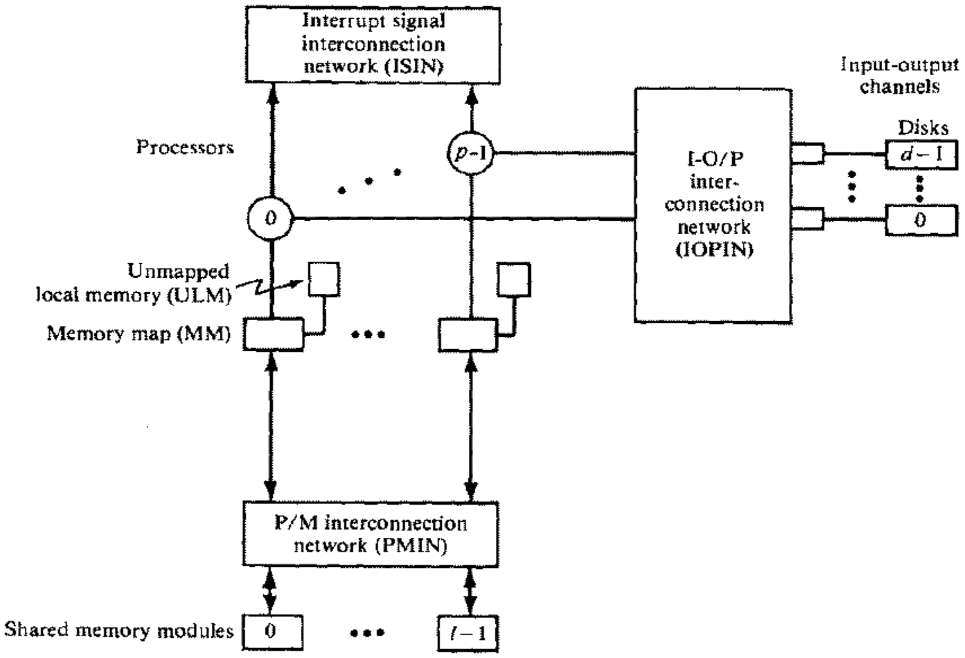
Assignment No.5

1. **How data -driven computation is different from conventional von Neumann machine? Compare concept of control flow and data flow computing**

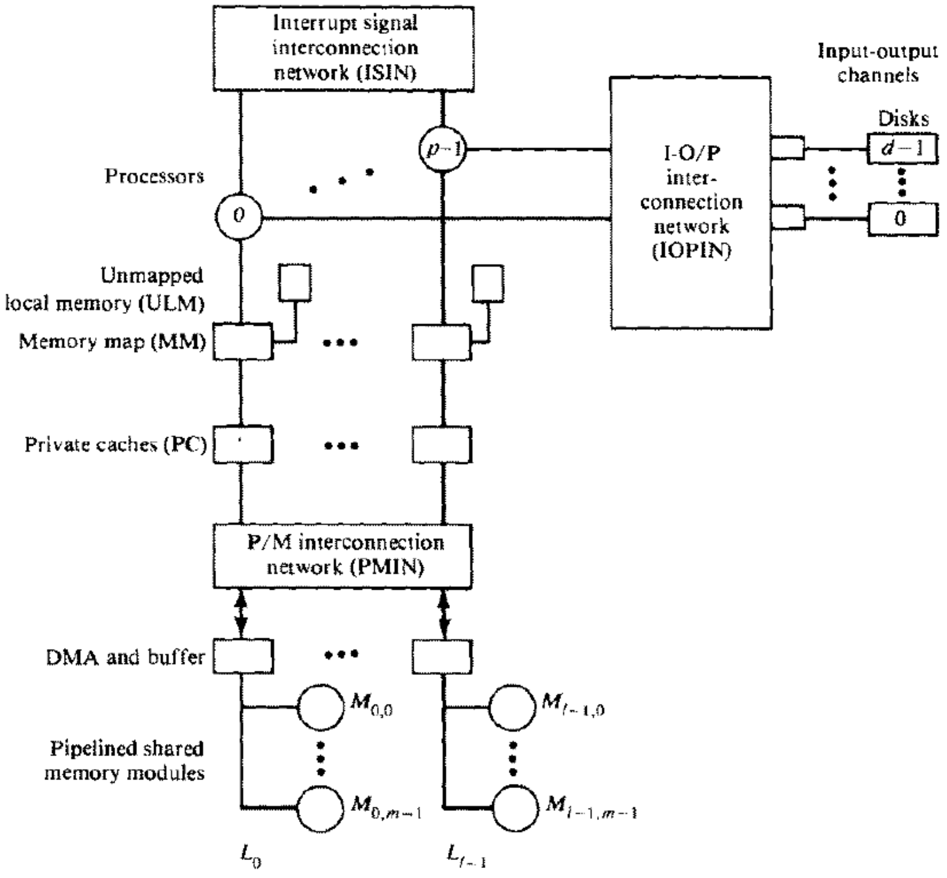
Sr. No.	Control Flow Computing	Data Flow Computing
1.	Computer Flow Computers uses Shared Memory to store program instructions and data.	There is no shared memory in data flow computers. Hence data is stored inside instructions.
2.	Control flow computers have a control driven organization. This means that the program has complete control over instruction sequencing.	Data flow computers have a data-driven organization that is categorized by a passive examine stage.
3.	The data is not transferred between the tasks.	The data is transferred between the tasks.
4.	In control flow task required to be completed to move to the next task.	In data flow one task not wait for the other to finish.
5.	We can add constraint in the link.	We can't add constraint in the links.
6.	Conventional von Neumann machine is based on the control flow computers.	Data-driven machines are based on the data flow computes.
7.	Execution of one instruction may produce side effect on other instructions since memory is shared	Execution of one instruction can't make side effect on other instruction since memory is not shared

2. **What are the functions of Kmap processor in Cm* architecture? Explain with steps how intra cluster memory access is achieved in Cm* architecture**

3. Why tightly coupled systems are preferred for high speed and real time processing? Draw and explain tightly coupled multiprocessor configuration.


Interrupt signal interconnection network (ISIN)
Input-output channels
Disks
Processors
p-1
d-1
I-O/P inter-connection network (IOPIN)
0
0
Unmapped local memory (ULM)
Memory map (MM)
P/M interconnection network (PMIN)
Shared memory modules
0
l-1

(*a*) Without private cache


Interrupt signal interconnection network (ISIN)
Input-output channels
Disks
p-1
d-1
Processors
I-O/P inter-connection network (IOPIN)
0
0
Unmapped local memory (ULM)
Memory map (MM)
Private caches (PC)
P/M interconnection network (PMIN)
DMA and buffer
$M_{0,0}$
$M_{l-1,0}$
Pipelined shared memory modules
$M_{0,m-1}$
$M_{l-1,m-1}$
L_0
L_{l-1}

(*b*) With private cache

a. Above figure shows the tightly coupled multicomputers system configuration.

b. In this system the processor shares clock generator, bus control logic, entire memory and I/O system.
c. This system communicate through memory.
d. One of the limitation of this system is the performance degradation due to memory contentions which occur when two or more processors attempts the same memory simultaneously.
e. When high speed of real-time processing is desired, this system may be used.
f. This tightly coupled multicomputer system is divided into two different models
   i) Tightly coupled multicomputers without private cache.
   ii) Tightly coupled multicomputers with private cache.

i) Tightly coupled multicomputers without private cache.

a. The above figure (a) show the tightly coupled multicomputer without private cache.
b. This system is consist of following things
   1) *p* processor
   2) *l* memory modules
   3) *d* input-output channel.
c. These units are connected through a set of three interconnection network namely, the processor memory interconnection networks (PMIN), the I/O processor interconnection network (IOPIN), the interrupt-signal interconnection network (ISIN).
d. The PMIN is a switch which is used to connect every processor to every memory module
e. The IOPIN is used to allow a processor to communicate with the I/O channel which is connected to I/O devices.
f. The ISIN is used for two purpose: To direct an interrupt to any other inter-processor network and to initiate hardware alarm in case of processor failure.
g. There are no separate private cache memory to any processor.

ii) Tightly coupled multicomputers with private cache.

a. In previous module each memory reference goes through the PMIN, it encounters delay in the processor or memory switch and hence the instruction cycle time is increases.
b. It reduces system throughput. This delay can be reduced by associating a cache with each processor.
c. The advantage of the cache is that the traffic through the crossbar switch can be reduced.
d. More than one inconsistent copy of data may exist in the system as this multicomputer system encounters cache coherence problem

4. What is processor -memory interconnection network in tightly coupled multiprocessor?

a. Processor – memory interconnection network in tightly coupled multicomputer is a switch which can connect every processor to every memory modules.
b. Typically, this switch is a p by l crossbar which has pl set of cross points.

c. A set of cross points for a particular processor-memory pair includes (n + k) cross points, where n is the width of the address within a module and k is the data path. Hence crossbar switch for a p by l multiprocessor system has a complexity O (pl(n + k)).

d. A memory module can satisfy only one processor’s request in a given memory cycle. Hence, if two or more processors attempts to access the same memory module, a conflict occurs which is resolved ore arbitrated by the PMIN. If necessary the PMIN may be designed to permit broadcasting of data from one processor to two or more memory modules. To avoid excessive conflict, the number of memory modules l is usually as large as p.

e. Another method is used to reduce the degree of conflict is to associate a reserved storage area with each processor. This is the unmapped local memory (ULM). It is used to store kernel code and operating system tables often used by the processes running on that processor. For example, if each processor is Multiprogrammed, each time a task switch is desired the state of the processor to be blocked may be saved in the ULM. The ULM helps in reducing the traffic in the PMIN and hence the degree of conflicts.

5. Compare between tightly coupled Multiprocessor and Loosely coupled systems.

Sr. No.	Loosely Coupled Multiprocessor System.	Tightly coupled Multiprocessor system.
1	There is distributed memory in loosely coupled multicomputers system.	There is shared memory in tightly coupled multicomputer system.
2	Loosely coupled multicomputers system has low data rate.	Tightly coupled multicomputers system has high data rate.
3	Loosely coupled multicomputers systems has low degree of interaction between tasks.	Loosely coupled multicomputers systems has high degree of interaction between tasks.
4	In loosely coupled multicomputers systems there is direct communication between processor and I/O devices.	In tightly coupled multicomputers system there is IOPIN to help communication between the processor and I/O devices.
5	The cost of loosely coupled multicomputers systems is less.	The cost of loosely coupled multicomputer system is more costly.
6	In loosely coupled multicomputers system, Memory conflict don’t take place.	In tightly coupled multicomputers system have memory conflict.
7	Application of loosely coupled multicomputers system are in distributed computing system.	Application of tightly coupled multicomputers system are in parallel processing systems.
8.	In loosely coupled multicomputers system, modules are connected through message transfer system network.	In tightly coupled multicomputers system there is PMIN, IOPIN and ISIN network.

Unit 6 – Parallel Programming

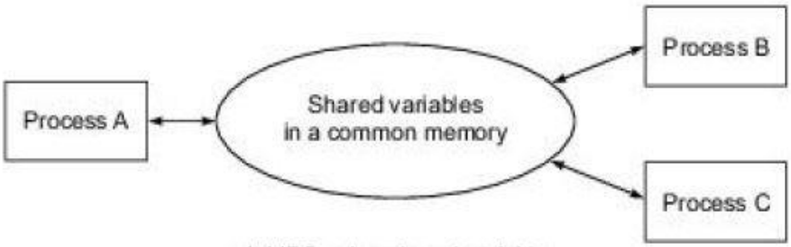
6.1) PARALLEL PROGRIKMMING MODELS:-

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, multicomputer, or vector/SIMD computers. Five models are characterized below tor these computers that exploit parallelism with different execution paradigms.

6.1.1) Shared-Variable Model: -

1) In all programming systems, we consider processors active resources and memory and I/O devices passive resources. The basic computational units in a parallel program are processes corresponding to operations performed by related code segments. The granularity of a process may vary in different programming models and applications.
2) A Program is a collection of processes. Parallelism depends on how interprocess communication (IPC) is implemented. Fundamental issues in parallel programming are centered around the specification, creation, suspension, reactivation, migration, termination, and synchronization of concurrent processes residing in the same or different processors.
3) By limiting the scope and access rights, the process address space may he shared or restricted. To ensure orderly IPC, a mutual exclusion property requires the exclusive access of a shared object by one process at a time. We address these issues and explore their solutions below.

A. Shared-Variable Communication:-



(a) IPC using shared variable

1) Multiprocessor programming is based on the use of shared variables in a common memory for IPC. As Specified in above figure shared-variable IPC demands the use of shared memory and mutual exclusion among multiple processes accessing the same set of variables.
2) Fine-grain MIMD parallelism is exploited in tightly coupled multiprocessors. Interprocessor synchronization can he implemented either unconditionally or conditionally, depending on the mechanisms used.
3) The main issues in using this model include protected access of critical sections, memory consistency, atomicity of memory operations, fast synchronization, shared data structures, and fast data movement technique.

B. **Critical Section: -** A critical section (CS) is a code segment accessing shared variables, which must be executed by only one process at a time and which, once started, must be completed without interruption. In other words, a CS operation is indivisible and satisfies the following requirements:

   i) **Mutual Exclusion** - At most one process executing the CS at a time.
   ii) **No deadlock in waiting-** No circular wait by two or more processes trying to enter the CS; at least one will succeed.
   iii) **Nonpreemption-** No interrupt until completion, once entered the CS.
   iv) **Eventual Entry -** A process attempting to enter its CS will eventually succeed.

C. **Protected Access: -**

   1) The main problem associated with the use of a CS is avoiding race conditions where concurrent processes executing in different orders produce different results. The granularity of a CS affects the performance. If the boundary of a CS is too large, it may limit parallelism due to excessive waiting by competing processes.
   2) When the CS is too small, it may add unnecessary code complexity or software overhead. The trick is to shorten a heavy-duty CS or to use conditional CS5 to maintain a balanced performance.
   3) Shared-variable programming requires special atomic operations for IPC, new language constructs for expressing parallelism, compilation support for exploiting parallelism, and OS support for scheduling parallel events and avoiding resource conflicts
   4) Shared-memory multiprocessors use shared variables for Interprocessor communications. Multiprocessing takes various forms, depending on the number of users and the granularity of divided computations. Four operational modes used in programming multiprocessor systems are specified below:

   i) **Multiprogramming: -**

      a. Traditionally, Multiprogramming is defined as multiple independent programs running on a single processor or on a multiprocessor by time-sharing use of the system resources. A multiprocessor can be used in solving a single large problem or in running multiple programs across the processors.
      b. A Multiprogrammed multiprocessor allows multiple programs to run concurrently through time-sharing of all the processors in the system. Multiple programs are interleaved in their CPU and U0 activities. When a program enters HO mode, the processor switches to another program. Therefore, multiprogramming is not restricted to a multiprocessor. Even on a single processor, multiprogramming is usually implemented.

ii) **Multiprocessing**:-

a. When multiprogramming is implemented at the process level on a multiprocessor, it is called multiprocessing. Two types of multiprocessing are specified below. If interprocessor communications are handled at the instruction level, the multiprocessor operates in MIMD mode. If interprocessor communications are handled at the program, subroutine, or procedural level, the machine operates in MPMD (multiple programs" over multiple data streams) mode.

b. In other words, we define MIMD multiprocessing with line-grain instruction-level parallelism. MPMD multiprocessing exploits coarse-grain procedure-level parallelism. In both multiprocessing modes, shared variables are used to achieve interprocessor communication. This is quite different from the operations implemented on a message-passing system.

iii) **Multitasking: -**

a. A single program can be partitioned into multiple interrelated tasks concurrently executed on a multiprocessor. This has been implemented as multitasking on Cray multiprocessors. Thus multitasking provides the parallel execution of two or more parts of a single program. A job efficiently multitasked requires less execution time. Multitasking is achieved with added codes in the original program in order to provide proper linkage and synchronization of divided tasks.

b. Tradeoffs do exist between multitasking and not multitasking. Only when overhead is short should multitasking be practiced. Sometimes, not all parts of a program can he divided into parallel tasks. Therefore, multitasking tradeoffs must he analyzed before implementation.

iv) **Multithreading: -**

a. The traditional LFNIXIUS has a single-threaded kernel in which only one process can receive OS kernel service at a time. In a multiprocessor, we want to extend the single kernel to be multithreaded. The purpose is to allow multiple threads of lightweight processes to share the same address space and to he executed by the same or different processors simultaneously.

b. The concept of multithreading is an extension of the concepts of multitasking and multiprocessing. The purpose is to exploit fine-grain parallelism in modern multiprocessors built with multiple-context processors or superscalar processors with multiple-instruction issues. Each thread will use a separate program counter. Resource conflicts are the major problem to be resolved in a multithreaded architecture.

c. The levels of sophistication in securing data coherence and in preserving event order increase from monoprogramming to multitasking, to multiprogramming, to multiprocessing, and to multithreading in that order. Memory management and special protection mechanisms must be developed to ensure correctness and data integrity in parallel thread operations.

D. Partitioning and Replication: -

1) The goal of parallel processing is to exploit parallelism as much as possible with the lowest overhead. Program partitioning is a technique for decomposing a large program and data set into many small pieces for parallel execution by multiple processors.
2) Program partitioning involves both programmers and the compiler. Parallelism detection by users is often explicitly expressed with parallel language constructs. Program restructuring techniques can be used to transform sequential programs into a parallel form more suitable for multiprocessors. Ideally, this transformation should be carried out automatically by a compiler.
3) Program replication refers to duplication of the same program code for parallel execution on multiple processors over different data sets. Partitioning is often practiced on a shared-memory multiprocessor system, while replication is more suitable for distributed-memory message-passing multicomputer.
4) So far, only special program constructs, such as independent loops and independent scalar operations, have been successfully parallelized. Clustering of independent scalar operations into vector or VLIW instructions is another approach toward this end.

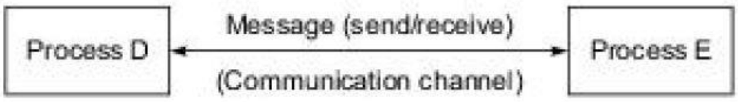
E. Scheduling and Synchronization: -

1) Scheduling of divided program modules on parallel processors is much more complicated than scheduling of sequential programs on a uniprocessor. Static scheduling is conducted at post-compile time. Its advantage is low overhead but the shortcoming is a possible mismatch with the runtime profile of each task and therefore potentially poor resource utilization.
2) Dynamic scheduling catches the run-time conditions. However, dynamic scheduling requires fast context switching, preemption, and much more OS support. The advantages of dynamic scheduling include better resource utilization at the expense of higher scheduling overhead. Static and dynamic methods can be jointly used in a sophisticated multiprocessor system demanding higher efficiency.
3) In a conventional UNIX system, interprocessor communication (IPC) is conducted at the process level. Processes can be created by any processor. All processes asynchronously accessing the shared data must be protected so that only one is allowed to access the shared writable data at a time. This mutual exclusion property is enforced with the use of locks, semaphores, and monitors.
4) At the control level, virtual program counters can be assigned to different processes or threads. Counting semaphores or barrier counters can be used to indicate the completion of parallel branch activities. One can also use atomic memory operations such as Test&Set and Fetch&Add achieve synchronization. Software-implemented synchronization may require longer overhead. Hardware barriers or combining networks can be used to reduce the synchronization time.

F. Cache Coherence and Protection: -

1) Besides maintaining data coherence in a memory hierarchy, multiprocessors must assume data consistency between private caches and the shared memory. The multicache coherence problem demands an invalidation or update after each write operation. These coherence control operations require special bus or network protocols for implementation as noted in previous chapters. A memory system is said to be coherent if the value returned on a read instruction is always the value written by the latest write instruction on the same memory location. The access order to the caches and to the main memory makes a big difference in computational results.
2) The shared memory of a multiprocessor can be used in various consistency models. Sequential consistency demands that all memory accesses be strongly ordered on a global basis. A processor cannot issue an access until the most recently shared writable memory access has been globally performed. A weak consistency model enforces ordering and coherence at explicit synchronization points only. Programming with the processor consistency or release consistency may be more restricted, but memory performance is expected no improve.

6.1.2) <u>Message Passing Model:-</u>



(b) IPC using message passing

1) Multicomputer programming is specified in above figure. Two processes D and E residing at different processor nodes may communicate with each other by passing messages through a direct or indirect network. The messages may be instructions, data, synchronization, or interrupt signals, etc. The commination delay caused by message passing is much longer than that caused by accessing shared variables in a common memory.
2) Multicomputer are considered loosely coupled multiprocessors. Two message-passing programming models are introduced below.

i) **Synchronous Message Passing: -**

1) Since there is no shared memory, there is no need for mutual exclusion. Synchronous message passing must synchronize the sender process and the receiver process in time and space, just like a telephone call using circuit-switched lines. In general, no buffers are used in the communication channels. That is why synchronous communication can be blocked by channels being busy or in error since only one message is allowed to be transmitted via a channel at a time.
2) In a synchronous paradigm, the passing of a message must synchronize the sending process and the receiving process in time and space. Besides having a time connection, the sender and receiver must also be linked by physical communication channels in space. A path of channels must be ready to enable the message passing between them.

3) In other words, the sender and receiver must be coupled in both time and space synchronously. If one process is ready to communicate and the other is not, the one that is ready must be bloc-ked (or wait). In this sense, synchronous communicate has been also called a blocking communication scheme.

ii) **Asynchronous Message Passing: -**

1) Asynchronous communication does not require that message sending and receiving be synchronized in time and space. Buffers are often used in channels, which results in nonblocking in message passing provided sufficiently large buffers are used or the network traffic is not saturated.
2) However, arbitrary communication delays may be experienced because the sender may not know if and when the message has been received until acknowledgment is received from the receiver. This scheme is like a postal service using mailboxes {channel buffers] with no synchronization between senders and receivers.
3) Nonblocking can be achieved by asynchronous message passing in which two processes do not have to be synchronized either in time or in space. The sender is allowed to send a message without blocking, regardless of whether the receiver is ready or not.
4) Asynchronous communication requires the use of buffets to hold the messages along the path of the connecting channels. Since channel buffers are finite, the sender will eventually be blocked. In c synchronous multicomputer, buffers are not needed because only one message is allowed to pass through a channel at a time.
5) The critical issue in programming this model is how to distribute or duplicate the program codes and data sets over the processing nodes. Tradeoffs between computation time and communication overhead must be considered.
6) As explained in Chapter 9, fine-grain concurrent programming with global naming was aimed at merging the shared-variable and message-passing mechanisms for heterogeneous processing.

iii) **Distributing the Computation:** Program replication and data distribution are used in multicomputer. The processors in a multicomputer [or a NORMA machine) are loosely coupled in the sense that they do not share memory. Message passing in a multicomputer is handled at the subprogram level rather than at the instructional or fine-grain process level as in a tightly coupled multiprocessor. That is why explicit parallelism is more attractive for multicomputer.

6.1.3) <u>Data-Parallel Model</u>

1) With die lockstep operations in SIMD computers, the data-parallel code is easier to write and to debug because parallelism is explicitly handled by hardware synchronization and Flow control. Data-parallel languages are modified directly from standard serial programming languages. For example, Fortran 90 is specially designed for data parallelism. Thinking Machines C* was specially designed for programming the erstwhile Connection Machines.
2) Data-parallel programs require the use of pre-distributed data sets. Thus the choice of parallel data structures makes a big difference in data-parallel programming. Interconnected data structures are also needed to facilitate data exchange operations. In summary, data-parallel programming emphasizes local computations and data routing operations (such as permutation,

replication, reduction, and parallel prefix). It is applied to fine-grain problems using regular grids, stencils, and multidimensional signal image data sets.

3) Data parallelism can he implemented either on SIMD computers or on SPMD multicomputer, depending on the grain size and operation mode adopted. In this section, we consider mainly parallel programming on SIMD computers that emphasize fine-grain data parallelism under synchronous control. Data parallelism often leads to a high degree of parallelism involving thousands of data operations concurrently. This is rather different from control parallelism which offers a much lower degree of parallelism at the instruction level.
4) Synchronization of data-parallel operations is done at compile time rather than at run time. Hardware synchronization is enforced by the control unit to carry out the lockstep execution of SIMD programs.

i) Dara Parallelism: -

Ever since the introduction of the illiac IV computer, programming SIMD array processors has been a challenge for computational scientists. The main difficulty in using the illiac IV had been to match the problem size with the fixed machine size. In other words, large arrays or matrices had to be partitioned into 64-element segments before they could be effectively processed by the 64 processing elements (PEs) in the illiac IV machine.

ii) Army Language Extension: -

1) Array extensions in data-parallel languages are represented by high-level data types. The array syntax enables the removal of some nested loops in the code and should reflect the architecture of the array processor.
2) Examples of array processing languages are CFD for the illiac IV, DAP FORTRAN for the AMT/Distributed Array Processor, C* for the TMC/Connection Machine, and MPF for the MasPar family of massively parallel computers.

iii) Compiler Support: -

1) To support data-parallel programming, the array language expressions and their optimizing compilers must be embedded in familiar standards such as Fortran 77, Fortran 90, and C. The idea is to unify the program execution model, facilitate precise control of massively parallel hardware, and enable incremental migration to data-parallel execution.
2) Compiler-optimized control of SIMD machine hardware allows the programmer to drive the PE array transparently. The compiler must separate the program into scalar and parallel components and integrate with the US environment. The compiler technology must allow array extensions to optimize data placement, minimize data movement, and virtualize the dimensions of the PE array. The compiler generates data-parallel machine code to perform operations on arrays.

6.1.4 Object-Oriented Model: -

1) If one considers special language features and their implications, additional models for parallel programming can he introduced. An object-oriented programming model is characterized below.
2) 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.

Concurrent OOP: -

1) The popularity of object oriented programming [OOP] is attributed to three application demands: First, there is increased use of interacting processes by individual users, such as the use of multiple windows. Second: Workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving. Third, multiprocessor technology in several variants has advanced to the point of providing supercomputing power at a fraction of the traditional cost.
2) As a matter of fact, program abstraction leads to program modularity and software reusability as is commonly experienced with OOP. Other areas that have encouraged the growth of OOP include the development of CAD (computer-aided design) tools and other sophisticated applications with graphics capabilities. Objects are program entities which encapsulate data and operations into single computational units. It turns out that concurrency is a natural consequence of the concept of objects. In fact, the concurrent use of coroutines in conventional programming is very similar to the concurrent manipulation of objects in OOP.
3) The development of concurrent object oriented programming (COOP) provides an alternative model for concurrent computing on multiprocessors or on multicomputer. Various object models differ in the internal behavior of objects and in how they interact with each other.

An Actor Model: -

1) COOP must support patterns of reuse and classification, for example, through the use of inheritance which allows all instances of a particular class to share the same property. An actor model developed at MIT is presented as one framework for COOP.
2) Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. In an actor model, message passing is attached with semantics. Basic actor primitives include:
   i) Create: Creating an actor from a behavior description and a set of parameters.
   ii) Send-to: Sending a message to another actor.
   iii) Become: An actor replacing its own behavior by a new behavior.
3) State changes are specified by behavior replacement. The replacement mechanism allows one to aggregate changes and to avoid unnecessary control-flow dependences. Concurrent computations are visualized in learns of concurrent actor creations, simultaneous communication events, and behavior replacements. Each message may cause an object (actor) to modify its state, create new objects, and send new messages.
4) Concurrency control structures represent particular patterns of message passing. The actor primitives provide a low-level description of concurrent systems. High-level constructs are also needed for missing the granularity of descriptions and for encapsulating faults. The actor model is particularly suitable for multicomputer implementations.

Parallelism in COOP:-

Three common patterns of parallelism have been found in the practice of COOP.

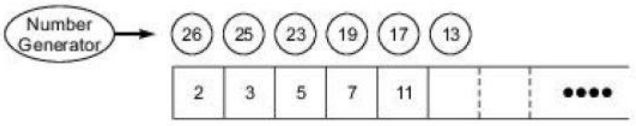
1) First, pipeline concurrency involves the overlapped enumeration of successive solutions and concurrent testing of the solutions as they emerge from an evaluation pipeline.
2) Second, divide-and-conquer concurrency involves the concurrent elaboration of different subprograms and the combining of their solutions to produce a solution to the overall problem. In this case, there is no interaction between the procedures solving the sub problems. These two patterns are illustrated by the following examples taken from the paper by Agha (1990).
3) A third pattern is called cooperative problem solving. A simple example is the dynamic path evaluation (computational objects) of many physical bodies {objects} under the mutual influence of gravitational fields. In this case, all objects must interact with each other; intermediate results are stored in objects and shared by passing messages between them. Interested readers may refer to the book on actors by Agha (1986).

Example: Concurrency in object-oriented programming: -

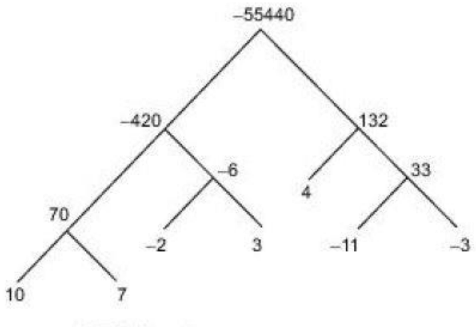
A prime-number generation pipeline is shown Fig. (A). Integer numbers are generated and successively tested for divisibility by previously generated primes in a linear pipeline of primes. The circled numbers represent those being generated.

A number enters the pipeline from the left end and is eliminated if it is divisible by the prime number tested at a pipeline stage. All the numbers being forwarded to the right of a pipeline stage are those indivisible by all the prime numbers nested on the left of that stage.

Figure (B) shows the multiplication of a list of numbers [[0, 7, -2, 3, 4, ~11, -3] using a divide and conquer approach. The numbers are re-presented as leaves of a tree. The problem can be recursively subdivided into sub problems of multiplying two sublists, each of which is concurrently evaluated and the results multiplied at the upper node.



(a) Pipeline concurrency



(b) Divide-and-conquer concurrency

6.1.5 Functional and Logic Models: -

Two language-oriented programing models for parallel processing are described below. The first model is based on using functional programming languages such as pure Lisp, SISAL, and Sn-and 83. The second model is based on logic programming languages such as Concurrent Prolog and Parlog. We reveal opportunities for parallelism in these two models and discuss their potential in AI applications.

Functional Programming Model: -

1) 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.
2) The lack of side effects opens up much more opportunity for parallelism. Precedence restrictions occur only as a result of function application. The evaluation of a function produces the same value regardless of the order in which its arguments are evaluated. This implies that all arguments in a dynamically created structure of a functional program can be evaluated in parallel. All single-assignment and clam-flow languages are functional in nature. This implies that functional programming models can be easily applied to data driven multiprocessors. The functional model emphasizes fine-grain MIMD parallelism and is referentially transparent.
3) The majority of parallel computers designed to support the functional model were oriented toward Lisp, such as Multilisp developed at MIT. Other dataflow computers have been used to execute functional programs, including SISAL used in the Manchester dataflow machine.

Logic Programming Model:-

1) 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. Clauses in logic programming can be transformed into dataflow graphs. Parallel unification has been attempted on some dataflow computers built in Japan.
2) Concurrent Prolog, developed by Shapiro (1986), and Parlog, introduced by Clark (1931), are two parallel logic programming languages. Both languages can implement relational language features such as AND-parallel execution of conjunctive goals, IPC by shared variables, and DR-parallel reduction.
3) In Parlog, the resolution tree has one chain at AND levels, and OR levels are partially or fully generated. In concurrent Prolog, the search strategy follows multiple paths or depth first. Stream parallelism is also possible in these logic programming systems.
4) Both functional and logic programming models have been used in artificial intelligence applications where parallel processing is very much in demand. Japan's Fifth-Generation Computing System (FGCS) project attempted to develop parallel logic systems for problem solving, machine inference, and intelligent human-machine interfacing.

5) In many ways, the FGCS project was a marriage of parallel processing hardware and AI software. The Parallel Inference Machine (PIM-I) in this project was designed to perform ID million logic inferences per second (MLIPS). However, more recent Al applications tend to be based on other techniques, such as Bayesian inference.

6.2) PARALLEL LANGUAGES AND COMPILERS

6.2.1) Language Features for Parallelism: -

Chang and Smith (1990) classified the language features for parallel programming into six categories according to functionality. These features are idealized for general-purpose applications. In practice, the real languages developed or accepted by the user community might have some or no features in some of the categories. Some of the features are identified with existing language/compiler development. The listed features set guidelines for developing a user-friendly programming environment.

1) **Optimization Feature:** These features are used for program restructuring and compilation directives in convening sequentially coded programs into parallel forms. These 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 Fortrancompiler.
    - Semiautomated parallelizer—Needs compiler directives or programmer's interaction, such as DINO.
    - Interactive restructure support—Static analyzer, run-time statistics, dataflow graph, and code translator for restructuring FORTRAN code, such as the MIMDizer from Pacific Sierra.

2) **Availability Feature:** 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 multicomputer, or both.

3) **Synchronization/Communication Feature:** Listed below are desirable language features for synchronization or for communication purposes:
    - Single-assignment languages.
    - Shared variables (locks) for IPC.
    - Logically shared memory such as the tuple space in Linda.
    - Send/receive for message passing.
    - Rendezvous in Ada.
    - Remote procedure call.

- Dataflow languages such as Id.
- Barriers. Mailbox, semaphores, monitors.

4) **Control of Parallelism**: Listed below are features involving control constructs for specifying parallelism in various forms:
   - Coarse, medium, or fine grain.
   - Explicit versus implicit parallelism.
   - Global parallelism in the entire program.
   - Loop parallelism in iterations.
   - Task-split parallelism.
   - Shared task queue.
   - Divide-and-conquer paradigm.
   - Shared abstract data types.
   - Task dependency specification.

5) **Dara Parallelism Feature:**
   Data parallelism is used to specify how data are accessed and distributed in either SIMD or MIMI) computers.
   - Run-time automatic decomposition- Data are automatically distributed vvitl1 no user intervention, as in Express.
   - Mapping specification- Provides facility for users to specify communication patterns or how data and processes are mapped onto the hardware, as in DINO.
   - Virtual processor support - The compiler maps the virtual processors dynamically or statically onto the physical processors, as in PISCES 2 and DINO.
   - Direct access to shared data- Shared data can be directly accessed without monitor control, as in Linda.
   - SPMD (single program multiple data) support—SPMD programing, as in DINO and Hypertasking.
6) **Process Management Feature:** These features are needed to support the efficient creation of parallel processes, implementation of multithreading or multitasking, program partitioning and replication, and dynamic load balancing at run lime.
   - Dynamic process creation at run time.
   - Lightweight processes (threads) - Compared to UNIX (heavyweight) processes.
   - Replicated workers- Same program on every node with different data (SPMD mode)
   - Partitioned networks- Each processor node might have more than one process and all processor nodes might run different processes.
   - Automatic load balancing—The workload is dynamically migrated among busy and idle nodes to achieve the same amount of work at various processor nodes

6.2.2) Parallel Language Constructs: -

Special language constructs and data array expressions are presented below for exploiting parallelism in programs. We first specify Fortran 90 array notations. Then we describe commonly used parallel constructs for program flow control.

1) **Fortran 90 Army 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_i is an arithmetic expression that must produce a scalar integer value. The first expression e, is a lower bound, the second E2 an upper bound, and the third e3 an increment (stride). For example, B (l : 4 : 3, 6 : 3 : 2,3) represents four elements B(l, 6, 3), B(4 ,6, 3), B(l, 8, 3), and B(4, 8, 3) of a three-dimensional array.

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. When both e_2 and e_3 are omitted, the e_1 alone represents a single element in that dimension. For example, A (5) represents the fifth element in the array A (3: 7: 2). This notation allows us to select array sections or particular array elements.

Array assignments are permitted under the following constraints: The array expression on the right must have the same shape and the same number of elements as the array on the left. For example, the assignment A(2 : 4, 5 : 8) =A(3 : 5, 1 : 4) is valid, but the assignment .A (l : 4, 1 : 3) =A(l:2, 1: 6) is not valid, even tempt each side has 12 elements. When a scalar is assigned to an array, the value of the scalar is assigned to every element of the array. For instance, the statement B (3 : 4, 5) = 0 sets B(3, 5) and B(4, 5) to G.

2) **Parallel Flaw Control: -**

The conventional Fortran Do loop declares that all scalar instructions within the (Du, Enddo) pair are executed sequentially, and so are the successive iterations. To declare parallel activities, we use die (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. However, the computations within each iteration are still executed serially in program order.

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. For example, dependence along the J-dimension exists in the following program. We use Doacross to declare parallelism along the I-dimension, but synchronization between iterations is required. The (Forall Endall) and (Pardo, Parend) commands can be interpreted either as a Doall loop or as a Doacross loop.

```
Doacross I = 2, N
Do J = 2, N
S1: A (I, J) = (A (I, (J – 1)) + A (I, ( J + 1))/2
Enddo
Endacross
```

Another program construct is the (Cobegin, Coend) pair. All computations specified within the block could be executed in parallel. But parallel processes may be created with a slight time difference in real implementations. This is quite different from the semantics of the Doall loop or Doacross loop structures. Synchronization among concurrent processes created within the pair are implied. Formally, the command

```
Cube-gin
P1
P2
.
.
.
Pn
Coend
```

causes processes P_1, P_2,... , P_n, to start simultaneously and to proceed concurrently until they have all ended. The command (Parbegin, Parend) has equivalent meaning.

Finally, we introduce the Fork and Join commands in the following example. During the execution of a process P, we can use a Fork Q command to spawn a new process Q:

```
Process P          Process Q
.                  .
.                  .
.                  .
Fork Q             End
.
.
.

Join Q
```

The Join Q command recombines the two processes into one process. Execution of Q is initialized when the Fork Q statement in P is executed. Programs P and Q are executed concurrently until either P executes the Join Q statement or Q terminates. Whichever one finishes first must wait for the other to complete execution, before they can he re-joined.

In a UNIX or LINUX environment, the Fork-Join statements provide a direct mechanism for dynamic process creation including multiple activations of the same process. The Cobegin-Coend statements provide a structured single-entry, single-exit control command which is not as dynamic as the Furl:-Jain. The (Parbegin, Parend) command is equivalent to the (Cobegin, Coend) command.

6.2.3) Optimizing Compilers for Parallelism: -

Because high-level languages are used almost exclusively to write programs today, compilers have become a necessity in modem 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 analysts, optimizations, and code generation, as shown in below figure.

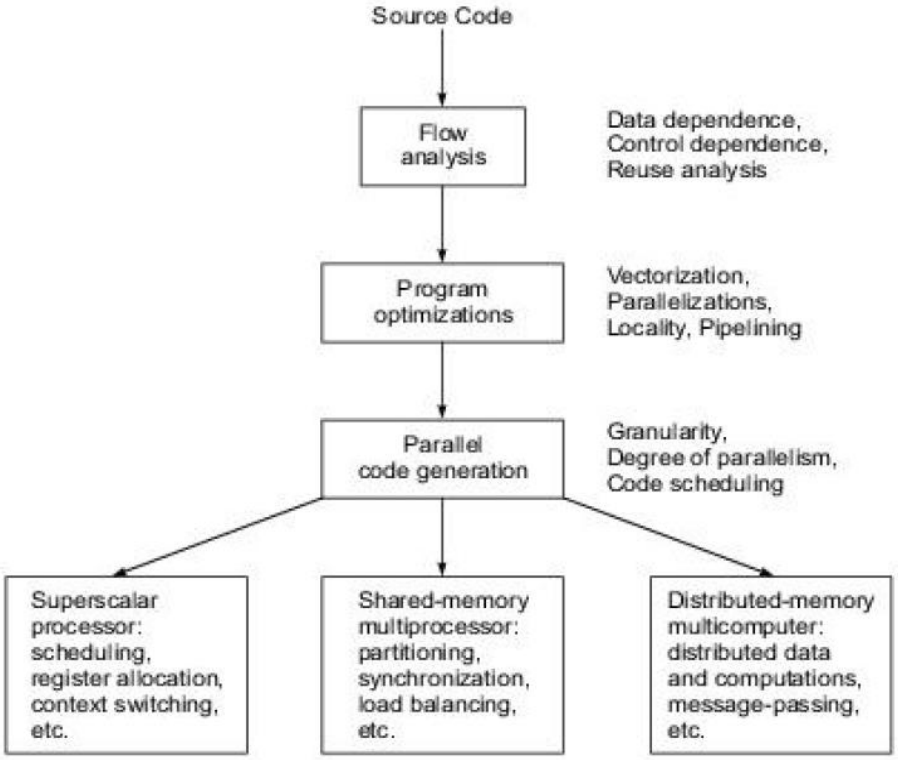

Source Code
Flow analysis
Data dependence, Control dependence, Reuse analysis
Program optimizations
Vectorization, Parallelizations, Locality, Pipelining
Parallel code generation
Granularity, Degree of parallelism, Code scheduling
Superscalar processor: scheduling, register allocation, context switching, etc.
Shared-memory multiprocessor: partitioning, synchronization, load balancing, etc.
Distributed-memory multicomputer: distributed data and computations, message-passing, etc.

Fig. 10.4 Compilation phases in parallel code generation

Flow Analysis: -

a. This phase reveals t11e program flow patterns in order to determine data and \ control dependences in the source code. We have discussed data dependence relations among scalar-type instructions in previous chapters. Scalar dependence analysis is extended below to structured data arrays or matrices. Depending on the machine structure, the granularities of parallelism to be exploited are quite different. Thus the l-‘tow analysis is conducted at different execution levels on different parallel computers.

b. Generally speaking, instruction-level parallelism is exploited in superscalar or VLSI processors; loop level in SIMD, vector, or systolic computers; and task level in multiprocessors, multicomputers, or a network of workstations. Of course, exceptions do exist. For example. fine-grain parallelism can in theory be pushed down to multicomputers with a globally shared address space. The flow analysis must also reveal code/data reuse and memory-access patterns.

Program Optimization: -

a. 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. The optimization often transforms a code into an Equivalent but "better" form in the same representation language. These transformations should be machine-independent.
b. In reality, most transformations are constrained by the machine architecture. This is the main reason why many such compilers are machine-dependent. At the least, we want to design a compiler which can run on most machines with only minor modifications. One can also conduct curtain transformations preceding the global optimization. This may require a source-to-source optimization (sometimes carried out by a precompiler), which transforms the program from one high-level language to another before using a dedicated compiler for the second language on a target machine.
c. The ultimate goal of program optimization is to maximize the speed of code execution. This involves the minimization of code length and of memory accesses and the exploitation of parallelism in programs. The optimization techniques include vectorization using pipelined hardware and parallelization using multiple processors simultaneously. The compiler should be designed to reduce the running time with minimum resource binding. Other optimizations demand the expansion of routines or procedure integration with inlining. Both local and global optimizations are needed in most programs. Sometimes the optimization should be conducted at the algorithmic level and must involve the programmer.
d. Machine-dependent transformations are meant to achieve more efficient allocation of machine resources, such as processors, memory, registers, and functional units. Replacement of complex operations by cheaper ones is often practiced. Other optimizations include elimination of unnecessary branches or common expressions. Instruction scheduling can he used to eliminate pipeline or memory delays in executing consecutive instructions.

Parallel Code Generation: -

a. 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. Basic blocks linked by control-flow commands are often optimized to encourage a high degree of parallelism. Special data structures are needed to represent instruction blocks.
b. 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, and how to implement message-passing commands when codes/data are distributed (or replicated) on a multicomputer are added difficulties in parallel code generation. Compiler directives can be used to help generate parallel code when automated code generation cannot he implemented easily.

c. Two well-known exploratory optimizing compilers were developed over mid-1980: one was Parafrase at the University of Illinois, and the other was the PFC (Parallel Fortran Converter] at Rice University. These systems are briefly introduced below.

Parafrase and Parafrase 2: -

a. This system, developed by David Kuck and coworkers at Illinois, is a source-to-source program restructurer (or compiler preprocessor) which transforms sequential Fortran 77 programs into forms suitable for vectorization or parallelization. Parafrase contains more than 100 program transformations which are encoded as passes. A pass list is used to identify the particular sequence of transformations needed for restructuring a given sequential program. The output of Parafrase is the converted concurrent program.
b. Different programs use different pass list and thus go through different sequences of transformations. The pass lists can be optimized for specific machine architectures and specific program constructs. Parafrase 2 was developed for handling programs written in C and Pascal, in addition to convening FORTRAN codes. Information on Parafrase can he found in [Kuck84] and on Parafrase 2 in [PolychronopoulosB9].
c. Parafrase is retargetable to produce code for different classes of parallel/vector computers. The program transformed by Parafrase still needs a conventional optimizing compiler to produce the object code for the target machine. The Parafrase technology was later transferred to implement the RAP vectorizer by Kuck and Associates, Inc.

The PFC and ParaScope: -

a. Ken Kennedy and his associates at Rice University developed PFC as an automatic source-to-source vectorizer. It translated Fortran T7 code into FORTRAN 9|] code. A categorized dependence testing scheme was developed in PFC for revealing opportunities for loop vectorization. The PFC package was also extended to PFC+ for parallel code generation on shared-memory multiprocessors. PFC and PFC l also supported the ParaScope programming environment.
b. PFC [Allen and Kennedy, I934] performed syntax analysis, including the following four steps:
   (I) Interprocedural flow analysis using call graphs.
   (2) Standard transformations such as Do-loop normalization, subscript categorization, deletion of dead codes, etc.
   (3) Dependence analysis which applied the separability, GCD, and Banerjee tests jointly.
   (4) Vector code generation. PFC+ further implemented a parallel code generation algorithm (Callahan et al, 1938).

Commercial Compiler: - Optimizing compilers have also been developed in a number of commercial parallel/vector computers, including the Alliant FX/F Fortran compiler, the Convex parallelizing/vectorizing compiler, the Cray CFT compiler, the IBM vectorizing Fortran compiler, the VAST vectorizer by Pacific Sierria, Inc., and Intel iPSC-VX compiler. IBM also developed a PTRAN (Parallel Fortran) system based on control dependence with Interprocedural analysis.

6.3) Code Optimization and Scheduling: -

In this section, we describe the roles of compilers in code optimization and code generation for parallel computers. In no case can one expect production of a true optimal code which matches the hardware behavior perfectly. Compilation is a software technique which transforms the source program to generate better object code, which can reduce the running time and memory requirement. On a parallel computer, program optimization often demands an effort from both the programmer and the compiler.

6.3.1) Scalar Optimization with Basic Blocks: -

a. Instruction scheduling is often supported by both compiler techniques and dynamic scheduling hardware. In order to exploit instruction-level parallelism (ILP), we need to optimize the code generation and scheduling process under both machine and program constraints. Machine constraints are caused by mutually exclusive use of functional units, registers, data paths, and memory. Program constraints are caused by data and control dependences. Some processors, like those with VLIW architecture, explicitly specify ILP in their instructions. Others may use hardware interlock, out-of-order execution, or speculative execution. Even machines with dynamic scheduling hardware can benefit from compiler scheduling techniques.

b. There are two alternative approaches to supporting instruction scheduling. One is to provide an additional set of nontrapping instructions so that the compiler can perform aggressive static instruction scheduling. This approach requires an extension of the instruction set of existing processors. The second approach is to support out-of-order execution in the micro-architecture so that the hardware can perform aggressive dynamic instruction scheduling. This approach usually does not require the instruction set to be modified but requires complex hardware support.

c. In general, instruction scheduling methods ensure that control dependences, data dependences, and resource limitations are properly handled during concurrent execution. The goal is to produce a schedule that minimizes the execution time or the memory demand, in addition to enforcing correctness of execution.

d. Static scheduling at compile time requires intelligent compilation support, whereas dynamic scheduling at rim time requires sophisticated hardware support. In practice, dynamic scheduling can be assisted by static scheduling in improving performance.

Precedence Constraint: -

a. Speculative execution requires the use of program profiling to estimate effectiveness. Speculative exceptions must not terminate execution. In other words, precise exception handling is desired to alleviate the control dependence problem. The data dependence problem involves instruction ordering and register allocation issues.

b. If a flow dependence is detected, the write must proceed ahead of the read operation involved. Similarly, output dependence produces different results if two writes to the same location are executed in a different order. Antidependence enforces a read to be ahead of the write operation involved. We need to analyze the memory variables. Scalar data dependence is much easier to detect. Dependence among arrays of data elements is much more involved. Other difficulties lie in Interprocedural analysis, pointer analysis, and register allocations interacting with code scheduling.

Basic Block Scheduling: - A basic black (or just a block) s a sequence of statements satisfying two properties:

(1) No statement but the first can be reached from outside the block; i.e. there are no branches into the middle of the block.

(2) All statements are executed consecutively if the first one is. Therefore, no branches out or halts are allowed until the end of the block. All blocks are required to be maximal in the sense that they cannot be extended up or down without violating these properties.

For local optimization only, an extended basic block is defined as a sequence of statements in which the first statement is the only entry point. Thus an extended block may have branches out in the middle of the code but no branches into it. The basic steps for constructing basic blocks are summarized below:

(l) Find the leaders, which are the first statements in a block. Leaders are identified as being one or more of the following:

(a) The first statement of the code.

(b) The target of a conditional or unconditional branch.

(c) A statement following a conditional branch.

(2) For a leader, a basic block consists of the leader and all statements following up to but excluding the next leader. Note that the beginning of inaccessible code (dead code) is not considered a leader. In fact, dead code should be eliminated.

<u>6.3.1) Local and Global Optimizations: -</u>

We first describe local code optimization within basic blocks. Then we study global optimizations among basic blocks. Boflt iniraprooodural and interprocedutal optimizations are discussed. Finally, we identify some machine-dependent optimizations. Readers will realize the limitations and potentials of these code optimization methods.

1) Local Optimization: -

These are code optimizations performed only within basic blocks. The information needed for optimization is gathered entirely from a single basic block, not from an extended basic block. No control-flow information between blocks is considered. Listed below are some local optimizations often performed:

(1) Local Common Subexpression Elimination: - If a Subexpression is to be evaluated more than once within a single block, it can be replaced by a single evaluation. For Example |o.s, in block B7, t9 and tl5 each compute 4 "' (j - i}, and 112 and tlii each compute 4 “' j. Replacing 115 by t9, and 118 by 112, we obtain the following revised code for B1’, which is shorter to execute.

```
tfi := j l
t'-1 := 4 " til
temp := A[t'5l']
tl2 := 4 * j
ti3 := .4.[ti2]
ans] -.=t13
A[tl'.'-'3] := temp
```

(2)Local Constant Folding or Propogorion Sometimes some constants used in instructions can be computed at compile time. This often takes place in the initialization blocks. The compile-time generated constants are then folded to eliminate unnecessary calculations at run time. in other cases, a local copy may be propagated to eliminate unnecessary calculations.

{3} Algebraic Optimization to Si'npl'ifiv Earpressiorrs For example, one can replace the t'derrtr'.l}r statement A :=
B + 0 or A := B * 1 by A := B and later even replace references to this A by references to B. Or one can use the commutative law to combine expressions C := A + B and D := B + A. The associative and di'srri£mrr've
i‘aw.'r can also be applied on equal-priority operators, such as replacing (rt — b) + c by ti (b — c) if (b — c) has already been evaluated earlier.
{4} instruction Reordering Code reordering is often practiced to maximize tlte pipeline utilization or to enable overlapped memory accesses. Some orders yield better code than others. Reordered instructions lead to better scheduling, preventing pipeline or memory delays. In the following example, instruction I3 may be delayed in memory accesses:
ll: Load R1, A
12: Load R2, B
13: Add R2, R1, R2 — delayed
14: Load R3, C
With reordering, the instruction 13 may experience no delay:
Load R1, A
Load R2, B
Load R3, CAdd
R2, R1, R2 — not delayed
{5} Elimination of Dead Code or Unorv Operators Code segments or even basic blocks which are not accessible or will never be referenced can be eliminated to save compile time, run time, and spaoe requirements.
GEE:
Unary operators, sueh as arithmetic negation and logical cornplement, can often be eliminated by applying algebraic laws, such as x I {—_v) = x — __\-', —{.t — y] = _t-' — x, (—x] " (—_t-'} = x ' _1-', Not(Not A) = A, etc. Boolean expression evaluation can often be optimized after some form of minimization.
Gabe] Optimizations These are code optimizations performed across basic block boundaries. Controlflow information among basic blocks is needed. John I-lennessy (1992) has classified intraprocedural global optimizations into three types:
{I} Global Versions of local Optimizations These include global common subeitpression elimination, global constant propagation, dead code elimination, etc. The following example fiirtber optimizes the code in
Example 10.6 if some global optimizations are performed.