

CHAPTER

1

Parallel Processing Concepts

Syllabus Topics

Motivating Parallelism, Scope of Parallel Computing, Parallel Programming Platforms : Implicit Parallelism, Trends in Microprocessor and Architectures, Limitations of Memory, System Performance, Dichotomy of Parallel Computing Platforms, Physical Organization of Parallel Platforms, Communication Costs in Parallel Machines, Scalable design principles, Architectures : N-wide superscalar architectures, Multi-core architecture.

1.1 Introduction to Parallel Computing

- Parallel computing is typically used to describe about the solving of a single problem using two or more processors. In the current developments the use of multiple cores also termed as parallel computing.
- A simple example will better state about it, suppose we have to perform arithmetic operations and we have quad core machine.
- The program is divided in four functions for obvious reasons and each function will be assigned into individual cores. Once the inputs required to perform the operations specified in the functions become available all functions execute independently in individual core and computations will be carried out in less amount of time.
- It becomes a need to compare parallel computing with serial computing because we are all habitual to do serial computing using the computers. One question always arises in our mind about parallel computing is whether it is harder than serial computing or not? Parallel computations can be performed only when interdependencies amongst operations will not take any significant time.
- The programmer is responsible to sort out the details about these and also need to deal with the communications required among the operations implemented in different functions.

- As we all know the largest and most powerful computers are called supercomputers. In fact supercomputers are highly parallel machines equipped with number of processors to work on the same problem.
- It is always possible that the parallelism can appear at different levels and that makes it difficult to define precisely.
- One of the levels of parallelism is the instruction level parallelism in which, within a CPU several instructions work simultaneously.
- In instruction level parallelism a compiler derives the instructions and processor decides which instructions can be processed simultaneously. Another form of parallelism called as task level parallelism is based upon the fact that multiple CPUs are assigned to process multiple instruction streams simultaneously.

Syllabus Topic : Motivating Parallelism

1.2 Motivation for Parallelism

Q. 1.2.1 Write short note on Motivation for Parallelism.
(Refer section 1.2) (4 Marks)

- There are many factors included behind the use of parallel computing in the current trend, some of the significant factors will be mentioned in this section.



- Continuing in the topic of parallel computing introduction we will again describe parallel computing in brief with illustration of a pictorial representation. Parallel computing is used to refer the assignment of multiple processors simultaneously to solve a computational problem.
- The Fig. 1.2.1 explains using pictorial representation the task level parallel computing. Here the overall problem to be solved is represented by a big rectangular shape.
- The problem is divided into four subtasks and each is represented by four smaller rectangular shapes.
- Now by the use of four CPUs each part of the sub problems is processed simultaneously. In this way all subtasks of a problem are computed together in different CPUs. In this way the computation time is reduced significantly. ①

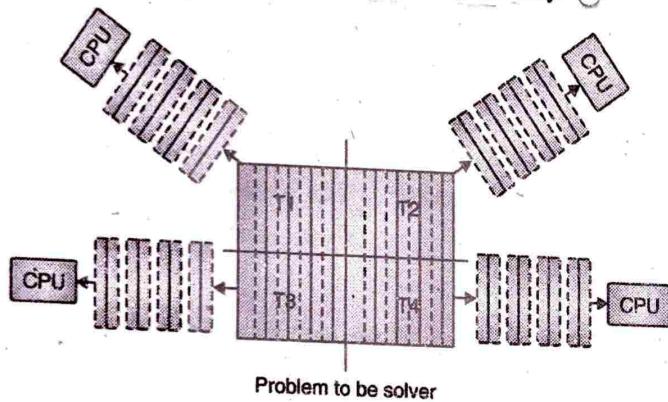
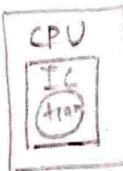


Fig. 1.2.1 : Parallel computation of tasks

- The most influential factor for the adaptation of parallel computing is the availability of powerful hardware devices. ②
- We will discuss various reasons directly or indirectly affects the utilization of parallel computing so that the actual facts behind the use of parallel computing will be explored.

1.2.1 Increased Number of Transistors in Integrated Chips Enhances Computing Power



The power of CPUs increases because of the additional transistors in the ICs. The overall computing power improvements in the modern processors opens the doors for parallel utilization of processing elements.

- The modern processing elements are equipments with the multiple units and these units can be assigned to perform processing tasks individually.

- According to Moore's law the number of transistors would double in every two years. The counts of transistors included in devices are very large but the use of all the transistors must be feasible then only the collective goal to achieve fast processing capability will be met.
- The parallel processing techniques are applied to utilize these devices effectively. The two forms of parallel processing implicit and explicit can be used with these devices. In later chapters these forms of parallelism are discussed in detail.

1.2.2 Improvements In Storage Technology (Memory/Disk)

As we all know that all modern computers work on the basis of Von Neumann architecture. According to this approach the program and data stored in the permanent storage and during executions should be loaded in the primary memory.

This makes clear that processor alone is not responsible for the enhancement in computation speed but primary memory as well as disk speed also plays very significant roles.

The access time of DRAM has increased but not yet increased much as compare to processor clock rates. This showed major performance bottleneck between a processor and a memory unit.

We further elaborate this as the number of instructions executed in per clock cycle by the processor has increased but at the same time memory should be able to feed the required data for executions.

The technique called as locality of reference is used to manage the mismatch between the memory and the processor speed.

In fact additional faster memory termed as cache memory is used to cope up with the situation.

The factors counted in representation of the memory performance are the speed and the latency. In fact the relationship between speed and latency is used to show the performance of the memory system.

The memory latency is the delay occurs in transmitting the data from RAM to the processor.

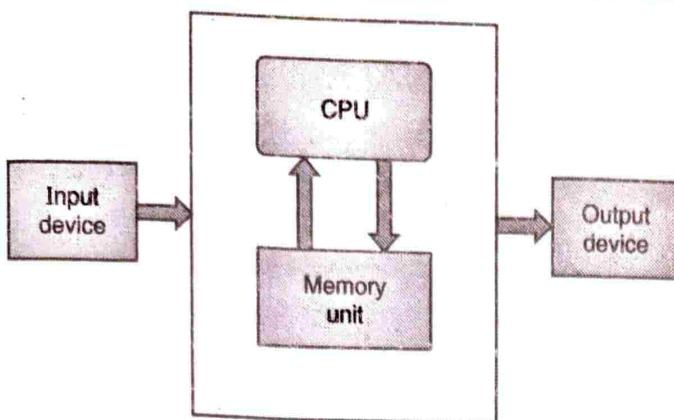


Fig. 1.2.2 : Von Neumann architecture



Fig. 1.2.3 : Cache and main memory

- The collective performance of the memory system is based upon the overall memory requests handle from the cache memory.
- Now when we focus about the caches and bandwidth in a parallel computing platform it is obvious that the memory system performance is better. In parallel systems because of the existence of multiple processors the overall cache size becomes larger.
- In the similar fashion the larger disk capacity is also be achieved for the permanent storage purposes.
- The efficient utilization of parallel systems can be achieved while running parallel algorithms. The performance of such algorithms depend upon the speed at which data transferred to and from the storage systems instead of fully relying on computations performed by the processors.

1.2.3 Improved Networking Devices and Systems for Data Communications

- The recent trend in computing is the use of Internet as a large platform for parallel and distributed computations. This means Internet is used to provide the environment for performing large computations.
- The inventions of many software systems and tools for such a platform started a new field for application developments.

- Many parallel applications suits for Internet based distributed computing platforms. In all such applications the need for travelling of data from one node to other nodes effectively handled by several networking devices. This is made possible because of the growth in the networking devices and technologies.

Syllabus Topic : Scope of Parallel Computing

1.3 Scope of Parallel Computing

Q. 1.3.1 Explain Scope of Parallel Computing.

(Refer section 1.3)

(2 Marks)

- The parallel computing is suitable for the problems require much more time for computation completion. The several areas in which parallel computing are used usually based on high - end engineering and scientific problems.
- This includes computer based simulations and Computational Fluid Dynamics (CFD) as well as other computer based digital image processing and security algorithms. In this section we will cover brief descriptions about such areas.

1.3.1 Applications of Parallel Computing in Engineering

Q. 1.3.2 What are the Applications of Parallel Computing? (Refer section 1.3.1) (3 Marks)

- Parallel computing is used in various engineering problems. These problems typically solved in the form of computer programs in traditional machines.
- These machines are capable of performing the tasks sequentially but designers closely observed and found the subtasks included in the problem can be performed as different processes simultaneously.
- The parallel computations provide the improvements in overall efficiency of the application. The main factor responsible for efficiency improvements is the speedup of the computations.
- This phenomenon is adopted in several engineering application domains such as applications of aerodynamics, optimization algorithms like branch and bound and genetic programming etc.



1.3.2 Scientific Applications using Parallel Computing

- In short scientific applications are created to show the simulated behaviours of real world entities by using mathematics and mathematical formulas.
- This means the object exists in the real world are mapped in the form of mathematical models and the actions present in that object are simulated by executing the formula.
- Simulations are based on very high-end calculations and require the use of powerful computers. The most of the powerful computers are parallel computers and do the computations in parallel form.
- The scientific applications are major candidates for the parallel computations. For example weather forecasting and climate modeling, Oil exploration and energy research, Drug discovery and genomic research etc. are some of the scientific applications in which parallel activities are carried out throughout the completion.
- These activities are performed by the tools and techniques provided under the field of parallel computing.

1.3.3 Commercial Applications Based on Parallel Computing

- The increasing need for more processing power is full - filled by parallel architectures. The parallel architectures continuously evolve to satisfy the processing requirements of the developing trends of the society.
- The commercial applications require more processing power in the current market trends because of performing many activities simultaneously.
- This creates an inclination towards the use of parallel computing applications for commercial problems. We can consider multimedia applications as an example of commercial applications in which parallel computing plays vital role.
- For large commercial applications such as multimedia applications enhance processing power and most of the circumstances the overall application should be divided into different modules. These modules work in a close cooperation to complete the overall required task in less amount of time.

1.3.4 Parallel Computing Applications in Computer Systems

- In the recent developments in computing the techniques of parallel processing becomes easily adoptable because of growth in computer and communication networks.
- The computations perform in parallel, by the use of collection of low power computing devices in the form of clusters. In fact a group of computers configured to solve a problem by means of parallel processing is termed as cluster computing systems.
- The advances in the cluster computing in terms of software frameworks made possible to utilize the computing power of multiple devices for solving of large computing tasks efficiently.
- The parallel computing is effectively applied in the field of computer security. The Internet based parallel computing set-up is suitably used for the implementation of extremely large set of data required for computations.

Syllabus Topic : Parallel Programming Platforms

1.4 Parallel Programming Platforms

- The basic components of the sequential computers are memory units and the processors. The logical representation of sequential computer is based upon the interconnection of memory and processors.
- In such systems the memory unit is connected with the processor through the data path.
- This way we can understand about the overall processing of the computer system is based upon the basic components : memory, processor and data path.
- The processing capability of the system is improved because of the different architectural innovations tried in the computing field.
- The approach called as multiplicity is used for the performance improvement. The multiplicity is achieved by introducing multiple elements of memory unit, processing element and data path.
- There are variations in the use of multiplicity; one is in implicit parallelism where it is not seen by the programmer whereas it can be exposed to the programmer in different forms.



- This section focuses about useful architectural concepts applied in the field of parallel processing. The concepts and techniques covered here are applicable into varieties of platforms so that programmers can write different parallel computing programs.

Syllabus Topic : Implicit Parallelism - Trends in Microprocessor and Architectures

1.4.1 Implicit Parallelism : Trends in Microprocessor and Architectures

- The field of computer architecture suggests designers about the selection and interconnection of hardware components for creation of computers that meet goals have been set for functionality, performance and cost.
- It is obvious thing that a new computer cannot be designed without defining the goals based on performance, power and costs.
- The overall design process is focused on understanding and making trade-offs. This include about the target market or users and what types of applications going to execute on these computers.
- This section describes about the understanding of various approaches of designs and their limitations as well as how it affects the overall performance of program developments.

1.4.1(A) Pipelining and Superscalar Execution

- The pipeline processor is analogically based on pipelining used in the car assembly line. The whole task to be performed in an assembly line is divided into sub-tasks. It was invented by Henry Ford long ago in the car assembly process.
- Henry Ford got an idea that it took much more time to build a car physically but he could actually build a car in a minute. What he did; all the steps required to build a car was divided into different stages on an assembly line. In this way one stage is assigned to put engine, another stage is responsible to assign wheels, next stage is used to put cabinets and so on.

- When this logic was applied it still takes several hours to build first car but because overall assembly process is divided into different stages, the second car was just behind the first car and it was almost done when the first car comes out from the assembly line.
- Then 3rd car followed with the 2nd car and so on. In this way car assembly line was formed and mass production started.
- The basic logic used in the car assembly line is applied in the computer technology also but it is based on the workload instead of producing something physical.
- In computer pipeline the task to be performed is broken down into smaller subtasks being performed into different stages.
- We will understand about pipelining in computer by the use of an example. Suppose there is a requirement to add two numbers N1 and N2.
- When we think this from human brain points of view then we just look at the numbers and perform operation and write down the result. If these numbers are too big then we simply use calculator. The point here is we do not think about the process, we just apply the required operation.
- Now if we consider this thing in computer's points of view then while writing the program we are required to tell about the locations of these two numbers and what operation to be performed.
- In the similar fashion after performing the said operation we have to tell the computer where to store the result.
- The pipeline steps for performing addition operation on two numbers can be described with an example. The pipeline used for this example has four stages as Fetch N1, Fetch N2, Add N1 and N2, Store result.
- Now according to the modern computers, assume each of these operation use one clock cycle to complete. So collectively these take four clock cycles for completion.
- However, the ability to perform operations in parallel while using the pipeline it improves the performance. This is further elaborated as while one instruction is being executed next instruction will be fetched and the overall output is actually improved. This can be illustrated better using pictorial representation as shown in Fig. 1.4.1.



	Unit					
	1	2	3	4		
Clock cycles	1	Fetch-N1	Inst-3	Inst-2	Inst-1	Instruction 1 completed
	2	Inst-4	Fetch-N2	Inst-3	Inst-2	Instruction 2 completed
	3	Inst-5	Inst-4	ADD	Inst-3	Instruction 3 completed
	4	Inst-6	Inst-5	Inst-4	Store result	Our Instruction completed

Fig. 1.4.1 : Instruction pipeline

- Here assume different instructions are represented using the name as Inst-#. Also one has to imagine that each Inst representing a stage involved in processing a computer instruction and each takes four clock cycles for completion.
- This example contains four units and in every clock cycle each unit does something. This can be further demonstrated by dividing the previous diagram shown in Fig. 1.4.1 on the basis of units as shown in Fig. 1.4.2.

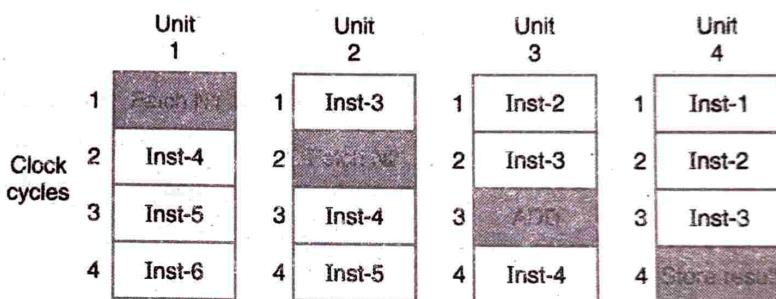


Fig. 1.4.2 : Instruction pipeline

- In every clock cycle each unit does something. This way we can easily understand that when instruction Inst-1 finishes by that time Inst-2 will become available to be finished in the next clock cycle.
- Each unit is processed in every clock cycle and in this way effectively utilization is achieved.
- Because all steps work together, four-step instructions (or tasks) can be completed at a rate of one per clock.
- Now the concept of **superscalar** execution can be covered after having sufficient understanding of pipeline processors.
- A processor designed to execute more than one instruction at a time during a single clock cycle is considered as a superscalar execution.
- A typical superscalar processor fetches and decodes several instructions at a time. The superscalar architecture exploits the potential of Instruction Level Parallelism (ILP).
- The typical behaviour of a superscalar processor is described with the fetching of multiple instructions at a time and then attempt for fetching of nearby independent instructions.
- These independent instructions can be executed parallel. The dependencies among instructions are identified so that instructions may be issued and executed in an order.
- The Fig. 1.4.3 shows the general architecture of the superscalar processor.

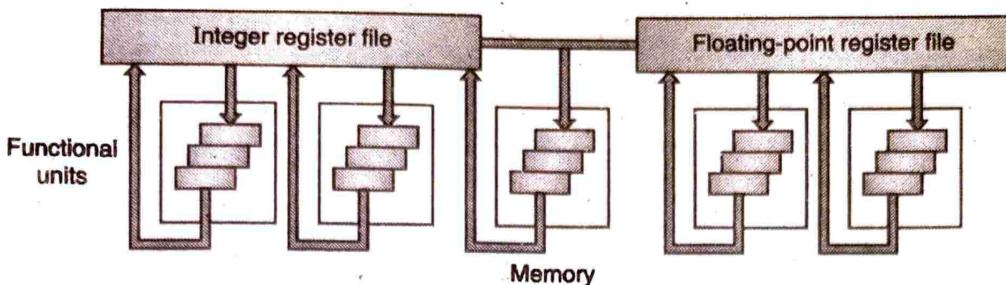


Fig. 1.4.3 : Superscalar Organization

- In Superscalar architecture multiple functional units are included. In this type of organization each unit is implemented with pipeline to provide support for parallel executions of multiple instructions.
- In the above example five pipeline functional units are considered where integer, floating point and memory unit are included. These units are used to perform operations concurrently so that overall performance of the system is effectively improved.

Superscalar execution

- A superscalar machine is based on the Von-Neumann architecture but it can issue more than one instructions per clock cycle.
- A superscalar machine uses multiple pipelines because with a single pipeline it is not possible to issue multiple instructions.
- A superscalar processor can exploit any degree of instruction level parallelism. It depends upon the number of parallel blocks exists in the sequential code.
- This is made possible by increasing the spatial parallelism (building multiple execution units).
- The classifications of the superscalar processors are based upon the maximum number of instructions can be issued at the same time.
- This depends on the pipeline structure used in the processor. One might clearly expect that the number of simultaneous instructions issued depends upon the number of pipelines built into the CPU.

1.4.1(B) Very Long Instruction Word Processors

- VLIW*
- VLIW (Very Long Instruction Word) architectures are considered as one of the suitable alternatives to achieve Instruction Level Parallelism (ILP) in programs.
 - This means VLIW architectures are used for execution of more than one basic instruction at a time in a program.
 - VLIW architecture can store multiple instructions in a single word. In VLIW based systems a parallel compiler is used to generate operations to be executed in parallel in the same word.
 - The compiler is responsible for resolving dependencies among instructions at compile time. The instructions ready to be executed concurrently are managed in a single word so that these can be executed on multiple functional units at the same time.
 - Very large instruction word in VLIW architecture indicates that the program to be executed in such processors is to be recompiled in a way that the instruction runs sequentially without existence of stall in the pipeline.
 - In this architecture it is not required for the hardware to examine the instruction stream about the instructions to be executed in parallel.
 - The compiler determines which operations to be executed in parallel. The execution unit used by the operations is also determined by the compiler.
 - The performance of VLIW architecture is very good when sequential programs written in C or FORTRAN languages are executed after recompilation for such systems.



- The VLIW compiler ensures that all operations in executing unit can perform simultaneously.
- The VLIW processors are used in application area where high-performance is required with less cost. For example, Digital Signal Processing (DSP) applications are suitable for VLIW based architectures.
- VLIW architecture uses a fixed length machine language instruction format. This format is of fixed length but is much larger than 32 - or 64 - bit formats as shown in Fig. 1.4.4.

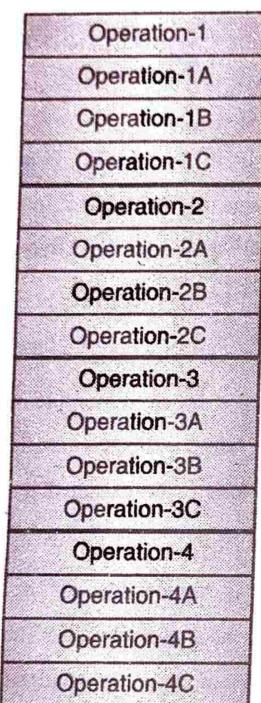


Fig. 1.4.4 : An example VLIW word format

- In VLIW every instruction is considered as a very long instruction. Each very long instruction contains bits to specify several instructions to be executed simultaneously.
- In a VLIW groups of bit fields are called as slots. These slots are used to specify operands and operations for every functional unit present in the machine.
- All of the slots present in the format can be filled if the instruction level parallelisms present in the program are sufficient.
- The superscalar design and VLIW have almost similar effects. The different activity in VLIW is done in case of scheduling in which most or the entire scheduling task is decided statically at compile time by the compiler instead of runtime by the control unit.

- The data and resource dependencies associated in the program is handled and analyzed by the compiler.
- The compiler first analyzes the dependencies associated in the program and then packs the slots of each VLIW with as many concurrently executable operations as possible.
- In this way a VLIW architecture based machine can execute more instructions than a superscalar machine.

1.4.1(C) Basic Working Principle of VLIW Processor

Q. 1.4.1 What is the Basic working principle of VLIW processor? (Refer section 1.4.1(C)) **(3 Marks)**

The following are some of the points can be considered as principles on which VLIW processor works :

- VLIW processor's basic aim is to speeding up computation by exploiting instruction-level parallelism. *L1 F*
- VLIW uses the same hardware core as superscalar processors with multiple execution units (EUs) working in parallel.
- In VLIW processor an instruction consists of multiple operations in which a typical word length considered from 52 bits to 1 Kbits. *52 b to 1Kb*
- In VLIW processor all operations in an instruction are executed in a lock-step mode.
- The VLIW processor relies on compiler to find parallelism and schedule dependency free program code.

1.4.1(D) Advantages of VLIW Processor

Q. 1.4.2 What are the Advantages of VLIW processor ?
(Refer section 1.4.1(D)) **(3 Marks)**

- Pure VLIW machines do not need complicated logic to check for dependencies.
- Eliminates complicated instruction scheduling and parallel dispatch associated with superscalar approach.
- Compiler is critical to performance : better compiler technology can result in improved performance on the same hardware.



1.4.1(E) Disadvantages of VLIW Processor

Q. 1.4.3 What are the Disadvantages of VLIW processor ? (Refer section 1.4.1(E)) (3 Marks)

- Increased code size due to empty "slots".
- Increased memory bandwidth.
- Compiler is critical to performance: must do all dependency resolution.
- Cache misses in one pipeline will force all pipelines to stall in a "pure" VLIW machine.

Syllabus Topic : Limitations of Memory System Performance

1.5 Limitations of Memory System Performance

Q. 1.5.1 What are the Limitations of Memory System Performance ? (Refer section 1.5) (4 Marks)

- The overall performance of a computer system not only depends upon the speed of the processing elements but significant memory unit working is also countable.
- This means processor has the capability for speed execution but how fast memory unit feeds data to the processor is also considered.
- It is obvious that the speed of instructions executions by processors have been increased remarkably. The important fact here is that the execution speed highly depends upon the speed at which instructions and data supplied to the processor by memory components.
- It is really unfortunate that the access time of the memory components has not been improving as fast as the processor performance.
- The use of cache memory shows very useful impact on the overall performance of the system. In most of the modern architectures a memory system consists of different levels of cache.
- A cache is a high - speed memory to be used as the buffer memory. It is logically placed between the CPU and the main memory. The purpose of using cache memory is to hold instructions and data likely to be needed in the near future by the processor.



- It is required to decide what actually one has to measure apart from reading and writing associated with the memory systems.

- The first term needs to be highlighted here is the latency. It is the time elapses between the start of the operation and completion of the operation.

There are enough reasons while saying that latency does not provide complete information about the performance of the memory system.

- At this particular point we need to discuss about the working of hardware. The understanding about hardware working makes clear why latency does not suffice as a measure of memory performance.

- The Fig. 1.5.1 shows a controller resides between the processor and the physical memory.

- The access of memory by the processor is handled by the controller as an intermediate entity. The processor sends its read or write request to the controller.

- The controller performs the translation of memory addresses and requests into appropriate signals for the underlying memory.



Fig. 1.5.1 : Controller between memory and processor

- Finally the controller passes these signals to the memory chips. The latency is minimized here when the controller returns an answer as fast as possible.

- This way we understand that memory latency is not fully sufficient to characterize as a measure for performance because it may need extra time between operations.

- The memory system performance can be assessed by measuring the speed at which a sequence of operations performed by the system.



- The term memory cycle time is used in this regard. The two parameters frequently used are read cycle time and the write cycle time in association with the memory system performance.

1.5.1 Use of Caches for Improvement of Latency

- The cache memory is used to enhance the access speed of any storage devices, for example, disk drives, main memory, tape storage, web servers and even for other cache also.
- The principle upon which cache works is called locality of reference. This principle says that the application refers a predictably small amount of data within a given window of time.
- The caches are divided into two basic groups : hardware and software cache which are shown in Fig. 1.5.2 and Fig. 1.5.3.
- The cache is the memory with low - latency and high - bandwidth properties. During functioning the required data by the processor is first fetched and stored in the cache memory so that further requests for the same data can be served from the cache.
- In this way the latency is reduced because the further fetch operations do not performed from the original locations.

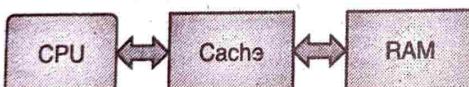


Fig. 1.5.2 : Hardware Cache

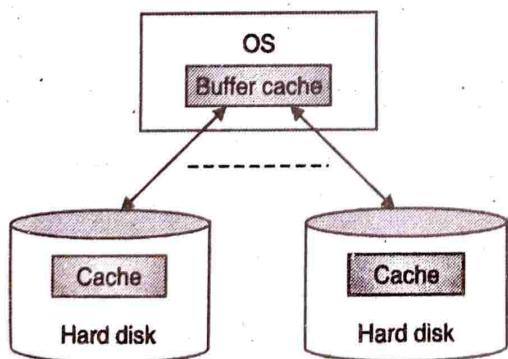


Fig. 1.5.3 : Software Cache

- There are two terms frequently used here are **cache-hit** and **cache-miss**. When the requested data reference is satisfied by the cache is called cache-hit otherwise cache-miss.

- The computation rate of many applications depends upon the rate at which memory can provide data to the processor.
 - Such applications' performances mainly depend on the cache hit and are called as **memory bound**.
 - The cache memory comes with different capacity, actually the amount of data that can be stored in the cache is considered as the capacity of that cache, for example, 32KB cache.
- The following are some of the cache related terms frequently used**
- **Cache block** is the basic unit of cache storage. It can contain multiple bytes/words of data.
 - **Cache set** is the term used to refer a row in the cache. The number of blocks per set is determined by the layout of the cache.
 - **Tag** is used to refer a group of data uniquely. An identifier is assigned for this purpose and is called as a tag. The tag is used to differentiate between different regions of memory mapped into the same block.

1.5.2 Memory Bandwidth

The rate of moving data between processor and memory is called as memory bandwidth. The memory bus and memory units are used to determine memory bandwidth. The memory bandwidth can be improved by increasing the size of the memory block.

1.5.3 Memory Latency Hiding Techniques

The increase in memory latency typically occurs when need arises to access remote memory. For example a distributed shared memory based systems. The description of important latency hiding mechanisms used in the systems is mentioned in the following :

Using pre-fetching techniques

- The prefetching technique is used for latency hiding because it brings instruction or data close to the processor before their actual requirements. Prefetching techniques are classified based upon whether they are controlled by hardware or software.



- The prefetching techniques based on hardware control uses approaches like enhanced size of cache lines so that spatial locality of reference is reduced in multiprocessor applications.
- On other side explicit prefetch instructions are required to issue when software controlled prefetching is used. While prefetching is done using software control the bandwidth requirements is reduced because prefetching is handled selectively.
- The direct effect of this scheme is that the time duration between the issue of instruction and its actual reference is increased. This has very significant impact when latencies are large.

1.5.4 Effects of Multithreading and Prefetching (Tradeoffs of Multithreading and Prefetching)

The problems associated with the performance of the memory system are minimized and in most of the cases eliminated by the use of multithreading and prefetching. The use of multithreading and prefetching affects the overall memory system performances.

Syllabus Topic : Dichotomy of Parallel Computing Platforms

1.6 Dichotomy of Parallel Computing Platforms

- Now we will have a brief discussion about the factors of parallel computing systems required for performance and portability in parallel programming.
- There are two different types of parallel platform organizations in a broader sense. These are physical and logical organizations of platforms.
- When the platform is explored in terms of the parallel program design so that programmer focused on the program design only is referred as a logical organization. On other hand the physical organization deals with the hardware components and their interactions with each other to provide the platform for development.

- The programmer's point of view in parallel program development is based on the representation of parallel activities in terms of tasks and how various tasks communicate with each other to provide inter-task communication.

1.6.1 Control Structure of Parallel Platforms

- In general term granularity refers about how a system is divisible. This means a system is considered as divided into multiple smaller parts is granular in structure. There are two basic types of granularity : course grained and fine grained.
- When a system is divided into large number of small parts we call it as fine grained whereas a course grained is referred as the division of a system into smaller number of large parts.
- The parallel computing field uses the term granularity to describe about the division of a task into number of smaller subtasks.
- The fine grained granularity in parallel computing refers to the division of a task into a large number of subtasks usually of shorter duration whereas course grained is used to refer smaller number of subtasks.
- The granularity in a parallel program is considered at different levels for example program level and instruction level.
- A parallel program may be considered as a collection of programs or collection of instructions to be executed in parallel.
- This means numbers of parallel tasks in a parallel program execute independently in terms of various subtasks or various instructions of a program execute concurrently as separate tasks in parallel.
- The working of different processing units in a parallel computer is characterized by two approaches.
- First a single control unit is used to coordinate all processing units centrally and second approach is based on working of various processing units independently.

SIMD (Single Instruction Stream Multiple Data Stream) Architecture

- In SIMD processors one instruction works on several data items simultaneously by using several processing elements (PEs), all of which carry out the same operation as shown in Fig. 1.6.1.

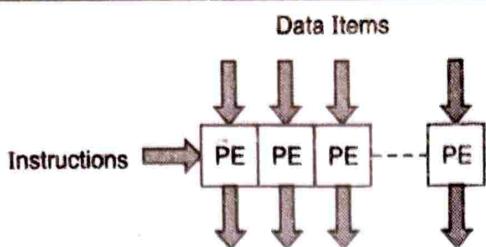


Fig. 1.6.1 : SIMD architecture

- The SIMD model of systems uses a single control unit to dispatch multiple instructions to various processing elements. In the SIMD computing model a single control unit is used to read instructions by a single Program Counter (PC), decode them and send control signals to the PEs. The number of data paths are depends upon the number of processing elements.
- Data are to be supplied to and derived from PEs by the memory. The structure of the system is shown in Fig. 1.6.2 called as array processor.

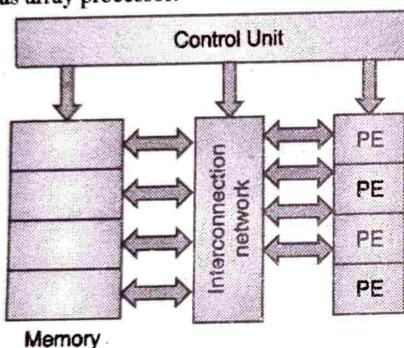


Fig. 1.6.2 : Processor array

- The different units like PEs and memory modules are interconnected by interconnection networks.
- The data transfers from to and from the PEs are managed by interconnected networks.

Example : Execution of conditional statements on a SIMD Architecture

- The following program segment where a conditional statement is included is considered to illustrate the use of SIMD architecture.

```
if (y == 0)
{
    z = x;
}
else
{
    z = x / y;
}
```

Fig. 1.6.3 : Conditional statement

- This statement is executed in two steps. In the first step all the processors where the instruction $z = x$ is present execute this instruction whereas all other processors remain idle. In the second step else part of the segment will execute and the processors involved in the first step remain idle. Fig. 1.6.4 shows the execution of the conditional statement two steps.

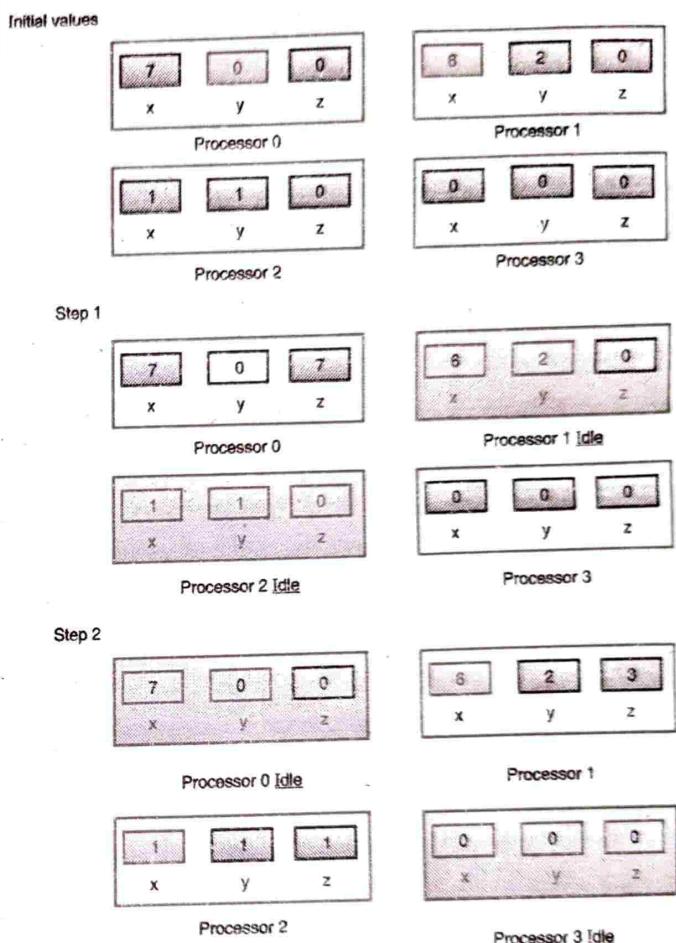


Fig. 1.6.4 : Execution of conditional statement

MIMD (Multiple Instruction Stream Multiple Data Stream) Architecture

- This model represents the system capable of executing multiple instructions on multiple data sets simultaneously.
- Most of the multiprocessing systems come under this category of systems. In fact the MIMD is used to describe a parallel machine able to perform independent computations at the same time.
- The MIMD class of machines can execute independent programs at the same time. The processing elements included in MIMD class of machines execute different programs at a time.



- This means in the MIMD class of machines each processor fetches its own instruction and operates on its own data.

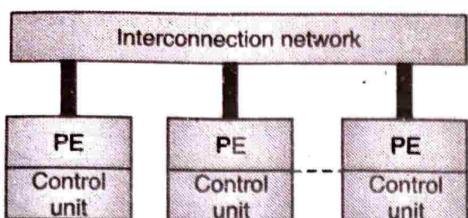


Fig. 1.6.5 : MIMD architecture

1.6.2 Communication Model of Parallel Platforms

There are different forms of data transfer among number of parallel tasks running concurrently. The two basic forms of data exchange among parallel tasks are shared data and message – passing approaches.

1.6.2(A) Shared-Address-Space

- This form of model for data exchange provides a common data space accessible by all processors included in the system.
- This shared space is used to provide interactions among number of processors by modifying data objects.
- The system is also called as multiprocessor systems because it supports the parallel programming approach termed as Single Program Multiple Data (SPMD) programming.
- This type of platform uses separate memory units associated to each processor or a common memory unit globally available for all the processors included in a multiprocessing environment.
- The Shared - Address space platform is classified as : NUMA and UMA.

Non Uniform Memory Access (NUMA) Multicomputer

Q. 1.6.1 Write short note on NUMA Multicomputer.
(Refer section 1.6.2(A)) (3 Marks)

- These categories of machines allow memory access to every processor without any restrictions.
- Fig. 1.6.6 represents the structure of NUMA. A block of is attached to every processor and all blocks of memory can be accessed through the path provided by the use of interconnection network.

- A processor has direct path to the block of memory attached with it. For example, accessing memory block MEM 1 from CPU 1 will be much faster than accessing block MEM 2 from CPU 1.
- This has significant implications : If we map the address space carefully it may be possible to keep most of the information required by a processor in the block attached to it.
- Therefore, the CPU can access that information directly and reducing the contention for the common bus. Since the time to access a memory location depends on whether it is attached to the invoking CPU or not, this model is called NUMA.

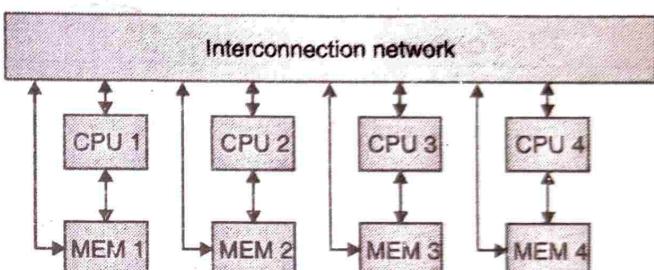


Fig. 1.6.6 : NUMA

Uniform Memory Access (UMA) Multicomputer

Q. 1.6.2 Write short note on UMA multicomputer.

(Refer section 1.6.2(A))

(3 Marks)

- The machines in which each processor gets equal priority to access the main memory of the machine is called as UMA.
- In fact the memory access time by each processor is identical in UMA platforms. Fig. 1.6.7 shows the model for UMA.
- The programming is much easier in such platforms because of the availability of a global memory space in the system.
- The coding for read - only interactions among the number of programs running in different processors is not at all seen by the programmer.

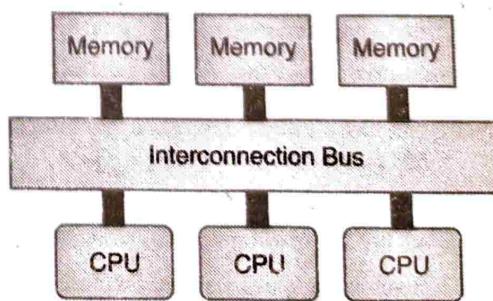


Fig. 1.6.7 : UMA



- This happens because the coding for such platforms is similar to coding usually done in serial programs for a single machine.
- This is very much simplified in terms of complexities associated of writing parallel program. In the situations where interactions among the read/write operations are handled require the use of mutual exclusion or some other tools for synchronizations.

Syllabus Topic : Physical Organization of Parallel Platforms

1.7 Physical Organization of Parallel Platforms

This section focuses about the physical component descriptions of parallel computer systems. The topic starts with the ideal descriptions about the parallel machines. The several things like difficulties associated with parallel systems along with conventional architectures for parallel systems are included in this section.

1.7.1 Architecture of an Ideal Parallel Computer

- As we know about the importance of Von Neumann machine model of computing system. It consists of a CPU connected to the memory for storage.
- The programs stored in the memory are fetched by the processor for executions.

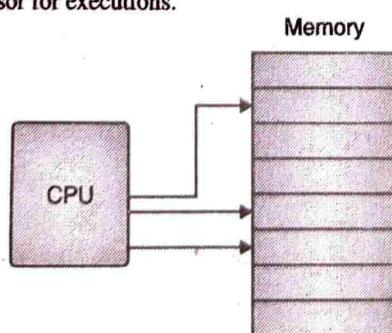


Fig. 1.7.1 : Von Neumann architecture machine model

- The significance of this model is such that the programmers have to trend the art of programming for such an abstract machine instead of a particular architecture of computer system.

- The parallel programming approach uses a parallel machine model. This approach also uses a general model as simple as Von Neumann model of sequential computing.
- The ideal model for parallel computing should have characteristics simple and realistic.
- The simplicity in the model provides the clear understanding of programming for the system and realistic nature of the system gives assurance for program executions in efficient ways on real systems.
- The model of parallel computing suits for the requirements for parallel computations is called the multicomputer.
- Fig. 1.7.2 shows the ideal model of a parallel system in which each node is based on Von Neumann architecture of machines.
- The nodes of this machine are linked by interconnection network. In this architecture each computer executes its own program and data transfer occurs between any two nodes in the form of message passing.
- Here if the interconnection network is assumed to be an ideal network. The ideal network does not depend upon the location of nodes for data transfers but it does affect by length of the message.

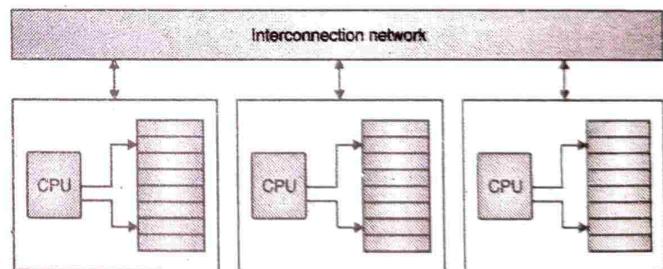


Fig. 1.7.2 : Multicomputer based Parallel computing machine model

- Another category of model for parallel computing is based on the notion of global memory based systems.
- This system consists of number of processors and a common memory accessible to all processors.
- This system contains a global clock to be shared among all the processors. This ideal model is called as a parallel random access machine model (PRAM).
- The diagrammatic representation of such system is shown in Fig. 1.7.3.

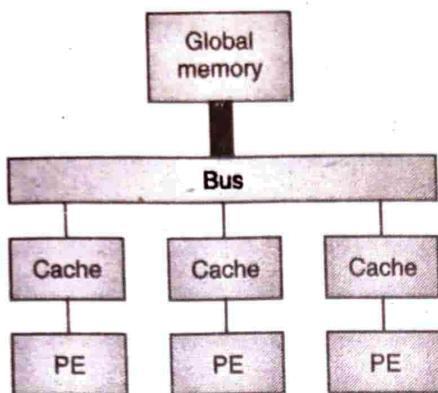


Fig. 1.7.3 : Ideal model for parallel computing system

1.7.2 Interconnection Networks for Parallel Computers

- Typically parallel computing systems consist of more than one processor and these processing elements are connected to memory units directly or indirectly. Interconnection networks are needed to route data when processors need to access a memory structure.
- The data transfer through interconnection networks occurs from processors to memories in case of concurrent access to shared memory structure. This is used to provide a message passing facility also, from one processing element to another. The processing elements are the combination of processor and memory.

Classification of interconnection networks

- The broad level interconnection networks are divided into static and dynamic classes based on connectivity and control.
- The connection between input and output nodes is fixed for the entire duration of communication and not allowed to change in **static network**.
- This means, it is noted that static networks cannot be reconfigured. The examples of this type of network are linear array, ring, tree, star, mesh, tours, systolic arrays, and hypercube etc. Static networks are suitable for building systems where pattern of communication is fixed.
- The interconnection pattern between inputs and outputs is allowed in **dynamic networks**. This means interconnection pattern can be allowed for reconfiguration according to the program demands in dynamic networks.

- Here, instead of fixed connections, the switches or arbiters are used. Examples of such networks are buses, crossbar switches, and multistage networks. The dynamic networks are normally used in Shared Memory(SM) multiprocessors.
- In a static interconnection network nodes are connected using point - to - point communication links. Static network is also called as direct network because of the point to point links among nodes.
- On the other hand, dynamic interconnection network is built up using the switches and communication links.
- Dynamic networks are also referred as indirect networks because of the path selection using switches are handled dynamically.
- The different types of static networks are based upon their node degree. The degree of a node is the number of edges connected to that node. Some of the examples of static network based on node degree is shown in Table 1.7.1.

Table 1.7.1

Node degree	Static network
1	shared bus
2	Ring, linear array
3	binary tree, fat tree, shuffle-exchange
4	two-dimensional mesh
Varying no. of degree	n-cube, n-dimensional mesh, k-ary n-cube

- The relative performance characteristics of different networks are represented by a measurement unit called diameter.
- The largest minimum distance between any pair of nodes is called as diameter of a network.
- The minimum distance between a pair of nodes is the minimum number of communication links (hops) that data from one of the nodes must traverse in order to reach the other node.
- Dynamic networks can be configured dynamically to meet the communication needs of user programs. Examples of dynamic networks include system buses, crossbar switches, multistage networks etc.

1.7.3 Network Topologies

Q. 1.7.1 Write short note on : Bus-Based Networks, Crossbar Networks, Fully-Connected Network, Meshes, Tree-Based Networks, and Fat tree. (Refer section 1.7) (6 Marks)

A graph is used for representing a network with nodes represent switching points and edges are used to represent links between nodes. Network topologies are graphical representations of various arrangements of nodes and the communication links among nodes.

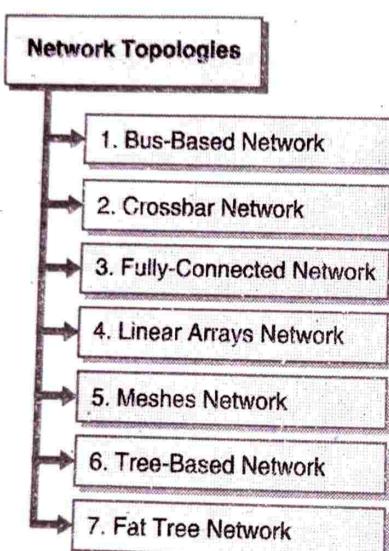


Fig. 1.7.4 : Network Topology

→ 1. Bus-Based Networks

- This is the simplest type of interconnection networks used and static by nature. It has a degree of one. In this network all elements share a common communication link as shown in Fig. 1.7.5.

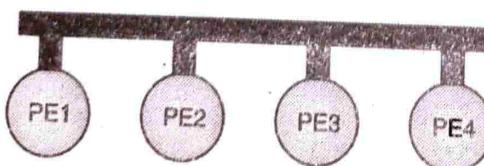


Fig. 1.7.5 : Bus based interconnection network

- From the implementation point of view this is most inexpensive network. The nodes can be easily added and deleted from this network. This network always has a requirement for handling of conflicts when several node elements request for the bus at the same time.

- The bus controller mechanism is used to provide bus access on a first come first serve basis or priority basis.

→ 2. Crossbar Networks

- This is one of the simplest network topology used for interconnection among computing nodes. It consists of two dimensional grid of switches. This network provides non-blocking connectivity between inputs and outputs. This type of interconnection network provides the characteristics in which any of the inputs can join to any of the outputs.
- The Fig. 1.7.6 shows an $N \times M$ crossbar network and the switch connections are also shown on the Fig. 1.7.7.

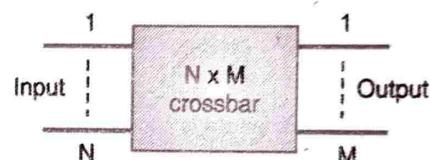


Fig. 1.7.6 : $N \times M$ crossbar

- The connection establishment is done through a cross point for a particular row input and a particular column output.
- The number of switch requirements for a network with N input and N output is N^2 .

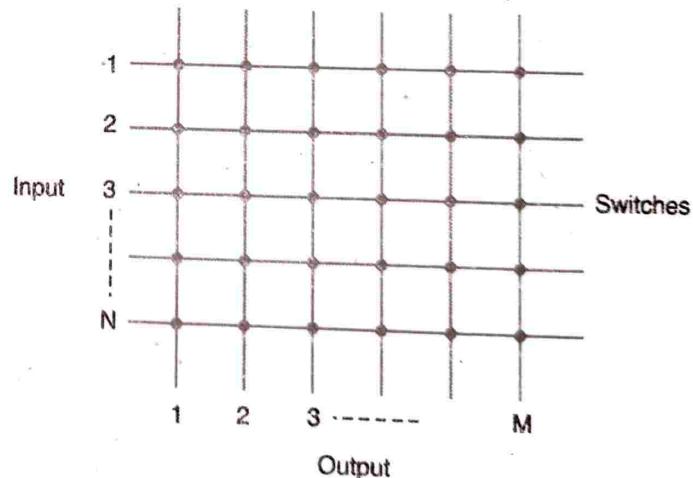


Fig. 1.7.7 : Switch connections

→ 3. Fully-Connected Network

- This is one of the most powerful interconnection topology used for providing connectivity among nodes. In this type of topology each node is directly connected to all other nodes. The shortcoming of this network is that, it requires too many connections as shown in Fig. 1.7.8.

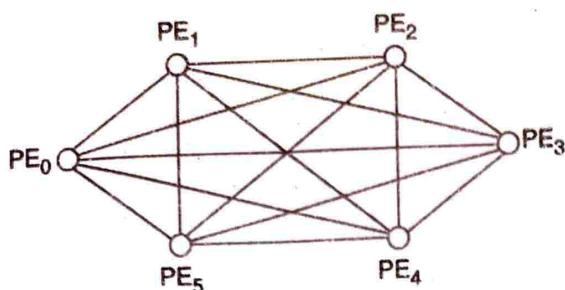


Fig. 1.7.8

→ 4. Linear Array Network

Linear array is one of the fundamental interconnection network patterns. In this type of networks processors are connected in a one dimensional linear array fashion.

This is a one dimensional interconnection network where first and last elements are connected with only one adjacent processor and all intermediate elements are connected with two adjacent processing elements.

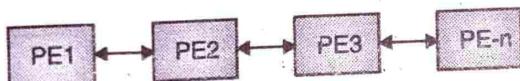


Fig. 1.7.9 : linear array

→ 5. Meshes Network

A mesh is two dimensional network in which processing elements are arranged in a two dimensional grid. The respective row and column positions are used to denote a particular processor in the mesh network. For example $PE_{(i, j)}$ is used to denote the processing element in row i and column j.

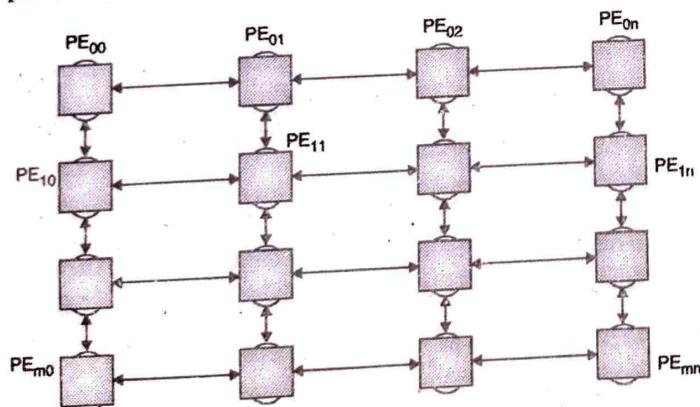


Fig. 1.7.10 : Mesh network

→ 6. Tree-Based Networks

In a tree network only one path is used between any pair of nodes. In this basis the linear array and star connected networks are considered as special cases of tree networks. In

this network processors are arranged in a completely like binary - tree patterns.

Tree interconnection networks are divided into two categories : static tree networks and dynamic tree networks.

When each node of a tree are processing elements we call it as a static tree networks whereas in the case when the nodes included at the intermediate levels are switching nodes the tree network is considered as a dynamic tree interconnection networks.

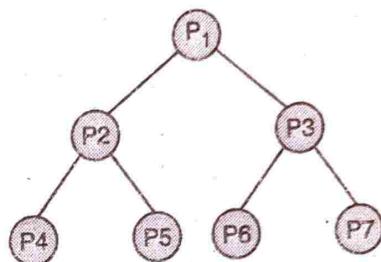


Fig. 1.7.11 : A tree network static in nature

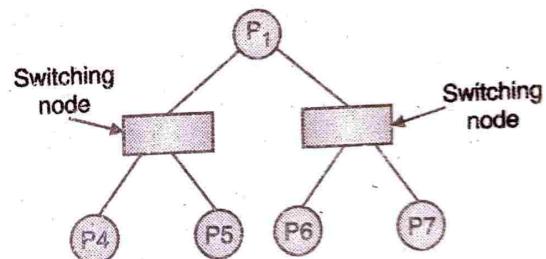


Fig. 1.7.12 : A tree network dynamic in nature

→ 7. Fat Tree

This type of network is modified form of the original tree network. This has increased bandwidth of edges into root direction.

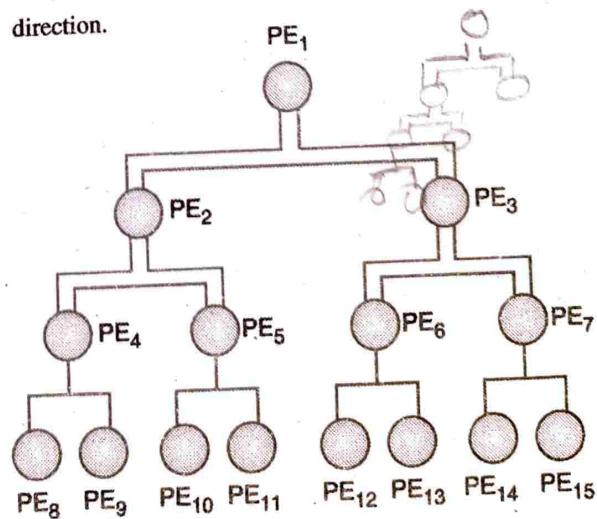


Fig. 1.7.13 : Fat tree



- This is simulated on the basis of actual trees where branches get thicker towards root. This type of network provides more adaptability because practically more traffic occurs towards the root as compared to leaves.
- Therefore the larger bandwidth provided at root side is capable to handle the traffic in effective ways and no bottlenecks can occur.

Syllabus Topic : Communication Costs in Parallel Machines

1.8 Communication Costs in Parallel Machines

Q. 1.8.1 Write short note on Communication Costs in Parallel Machines.

(Refer section 1.8) (5 Marks)

- The parallel computing platform provides the facility for executions of parallel programs.
- The various modules of these parallel programs execute in different nodes of a distributed computing based parallel systems or in different processing elements of a shared memory parallel system.
- The modules need to communicate with each other for completion of computing tasks.
- The communication costs among these modules depends on several factors like interconnection network used for connecting nodes, modes used for parallel programming, protocols used in the communication etc. In this section such issues affects the overall communication are discussed.

1.8.1 Message Passing Costs

- In message passing based systems communication occurs between any two nodes is in the form of message exchange.
- The time required to send a message from one node to another node depends upon the collective time for message preparation and the actual time takes to travel a message from source to destination.
- This means message communication time between any two nodes includes message creation time as well as transfer time of that message from one node to another node in a message passing network.

- As it is define that the communication latency is the time required for a packet to be received by the remote computer.
- There are three basic parameters startup - time, per hop time and per word transfer time are used in the communication system to determine latency of communication between two nodes.
- This means the total time to transfer a message in communication network is comprised of startup - time, per - hop time and per - word transfer time.
- **Startup - time :** This time includes overall message handling time at sender and receiver nodes. As we know a message contains things like header, error correcting codes etc. and the process of handling all these things is called as message preparation. The other parameters considered for counting of startup - time are **selection of the routing algorithm** for finding out the best route to a destination node and the actual time required for **interface establishment** between a and the router. The important point here is this time is considerable only once for a single message transfer.

- **Per-hop time :** As we know in communication networking terminology one portion of the path between the source and destination is called a hop. Several networking devices are used in the network like bridges, routers, gateways and packets pass through these devices on the way. An occurrence of a hop is considered when each time packets are passed to the next device. Per - hop time is the time taken by header of a message between two directly connected nodes. Usually it is represented by T_h .

- **Per-word transfer time :** This time is considered for the time required for handling of all overheads determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.

- The parallel computers basically use two routing techniques termed as **store - and - forward routing** and **Cut-Through Routing**. We will now consider these two techniques in terms of message passing costs associated.

Store-and-forward routing

- This is one of the most common techniques used in message passing based services. In this technique a data is transmitted from sender device to the destination device but passes through an intermediary message center typically called as a server.



- The server stored the message till the period of locating the destination device in the network.
- Once the receiving device located, the message is forwarded to that device and it is deleted from the server.
- A common type of store and forward approach is used in mobile phones.
- The effective time required in store and forward routing is discussed based on an example. In this example a message named as M is considered for transmission using store-and-forward network.
- Also assume that the message M traverses L links. Here we consider the message incurring costs separately for the message header and the rest of the message portions.
- During traversal at each link, the time T_h required for header traversal and T_w m for the rest of the message to traverse the link. This way in this technique the total time becomes :

$$T_{com} = T_s + (m * T_w + T_h) * L \quad \dots(1.8.1)$$

- The Equation (1.9.1) can be further simplified while ignoring T_h because it is very small for the parallel computing systems and less than $T_w * m$ for small values of m for most parallel algorithm. The resultant equation is shown below :

$$T_{com} = T_s + (m * T_w) * L \quad \dots(1.8.2)$$

Packet Routing

- The utilization of communication resources is not effective in store - and - forward routing techniques.
- This happens because a message can only be sent from one node to the next node after receiving in its entirety.
- The packet routing technique offers better utilization of communication resources. In this technique the message is cut into smaller parts.
- Later on these parts are transferred separately through the communication network.
- The packet routing provides various benefits over store - and - forward routing technique. Some of the advantages are listed below :
 - o This approach has lower overhead for errors therefore less requirements for retransmission of messages.

- o Packet passing provides more robust routing facility. These include different paths for packet routing and avoid congestions.
- o Resource utilization is better in packet routing like pipeline processing used in multiprocessor systems.
- o The considerable thing in packet based routing approach is that more complex protocols are required in handling the activities.
- The total communication time required in packet routing is given by the following equation. The cost is based on the message size M and it traversing L links.

$$T_{com} = T_s + (T_h * L) + (T_w * M) \quad \dots(1.8.3)$$

Where, $T_w = T_{w1} + T_{w2}(1 + S/r)$

- The message to be sent is divided into packets, and these packets are assembled with their error, routing and sequencing fields.
- The size of a packet now can be represented using r + s. In this expression r represents the original message and s is some extra bit of entries carried in the packet.
- The total time required to prepare packets is based upon the message length and the packaging time is proportional to the length of the message. This time is denoted by $M * t_{w1}$.
- A delay of T_h is incurred on each hop when the network has the capability of communicating one word in every T_{w2} seconds. In the continuation suppose the packet traverses L hops and reaches the destination the total time required by it, is represented by $(T_h * L) + T_{w2} * (r + s)$.

Cut-Through Routing

- This is an optimized approach used for interconnection networks of parallel machines. This is suitable for parallel machines because of the less error rates occur in interconnection networks used for connection of nodes. The low rates for errors exist because it has dedicated network.
- In this way all packets of a message follows the same path and the overhead of transmitting of routing information associated with each packet can be eliminated.
- No need to keep sequencing information with packets also because of forcing in - sequence delivery of packets.



- The error detection and correction overhead is reduced significantly because of keeping error information at message level instead of packet levels.
- The overall communication time for this routing is represented using the costs equation shown below but we need to explore these parameters.
- Assume a message is M words long, the links traverses by that message is L and T_h is the per hop time.
- Based on these assumptions the header of the message require $L * T_h$ time to reach destination. The message arrives in $T_w * M$ times after header arrived to the destination.
- Therefore, the cut - through routing supports the simplified cost model and is represented by following equation.
- The total cost of message communication among nodes L hops away using this cut - through routing is represented using equation below.

$$T_{\text{com}} = T_s + T_h(T_h * L)(T_w * M)$$

1.8.2 Communication Costs in Shared-Address-Space Machines

- It is really difficult to have accurate models for shared address space machines because of some factors directly associated with such architectures. The following are some of the factors associated :
 - o The overall Memory layout depends on the system.
 - o These machines have limited availability of cache memories.
 - o It is difficult to implement cache coherence protocols.
 - o Difficulty in estimating the spatial locality.
 - o There may be a problem while sharing of items is not handled properly.
- These factors forced the use of same model as before with much smaller T_w for Unified Memory Access machines.

Syllabus Topic : N-wide Superscalar Architectures, Multi-core Architecture

1.9 Architectures : N-wide Superscalar Architectures, Multi-core, Multi-Threaded

This section includes several architectures of the computer. The different architectures include here are multithreaded to superscalar approaches of processing.

1.9.1 Multithreaded Architecture

- The thread in association with the hardware context is different than the software threads in multithreaded operating systems. The hardware supported threads depend on the specific form of multithreaded processor.
- The hardware based multithreaded technology enables the thread – level parallelism at the processor level. This is made up of duplication of architectural state on each processor while sharing only one set of processor execution states.
- The different architectural states are considered as separate logical processors by the operating system during executions of multiple threads.
- The operating system schedules the different threads as units of executions on logical processors.
- The logical processors are created through hardware for sharing the processor's functional units among different threads.
- The sharing of resources is crucial in multithreaded architectures because the more number of sharing of resources between logical processors provides more efficient operations at multithreading levels.

1.9.2 Multi - Core Architecture

- In the recent era of computing multicore processor systems have become the choice of everyone because of the performance of computations. This section contains the description about the multicore processors with overview about the core included in it for processing.



- The computer must have processor for providing computing capability. Every computer must have the processor whether it is a small desktop systems or a very large high - performance computer system.
- The processor or central processing unit is the most important part for functioning of a computer system.
- In the modern days of computing processors are constructed to perform several tasks simultaneously at processor level itself in the form of multiple cores. Multiple core processors are typically designed in the form of dual - core, quad - core, many - cores etc.
- A processing unit in the processor responsible to read instructions to perform specific actions is called a *core*.
- Anything we do using the computer has to be processed by the processor included in that computer. For example, in the modern computer systems equipped with the graphic card.
- The graphic card is used to generate high-end graphics activities and it contains many processors to quickly perform the tasks simultaneously - but graphics card also has to use the central processor as well.

☛ Single - CoreCPU

- The old CPUs utilizes only one core of the processor is called as a single - core processor. The single - core CPUs were very first types of CPUs and not suitable for modern applications.
- Fig. 1.9.1 shows the single - core CPU chip.

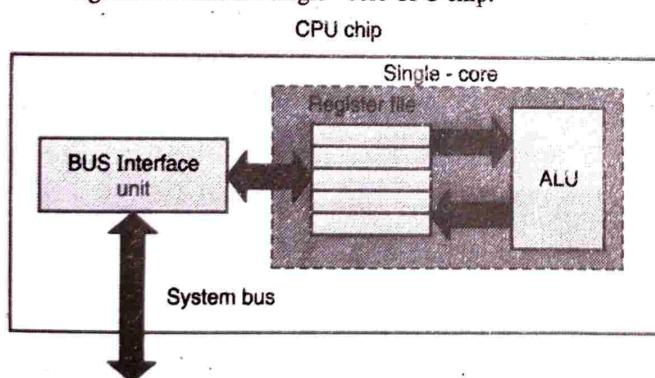


Fig. 1.9.1 : Single-core chip

☛ Multicore CPU

- The multicore processors contain more than one distinct core in a chip. In this way, if a chip has two cores it is named as dual core processor similarly a quad - core processor contains four cores in it.

- The multicore CPU design implements separate execution pipelines for each and every core.
- This means each core can run a thread of execution without requiring any resources from the other cores. Multicore processors are considered as MIMD category of machines because multiple instructions (threads) are executed by different cores and operating on different parts of memory (data).
- It also supports shared memory multiprocessor because all cores can share the same memory.

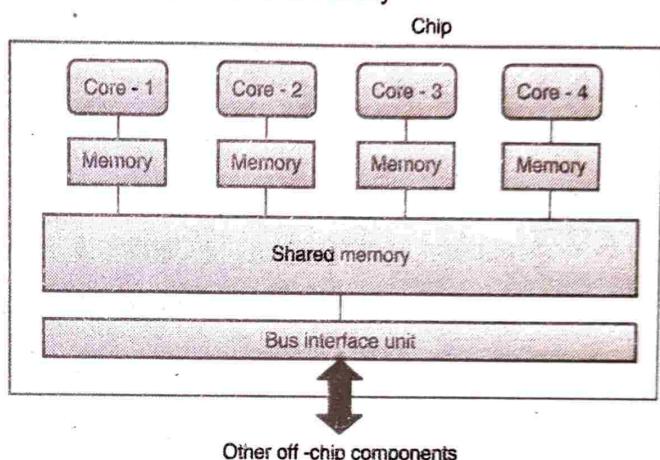


Fig. 1.9.2 : Multicore processor

- The design of multicore processor can be described by considering a practically available Intel Xeon processor. Intel Xeon is a multicore processor packaged with the logic and circuitry for two or more Intel Xeon processors.
- In fact the multicore architecture is based on the packaging of more such processors in a single physical processor.
- The most straight forward advantage of the multicore processor is the improvement in performance by using multiple cores to run multiple tasks simultaneously.

1.9.3 N-wide Superscalar Architectures

Q. 1.9.1 Explain N-wide Superscalar Architectures.

(Refer section 1.9.3)

(6 Marks)

- In this section, we will discuss about the superscalar processor. The superscalar architecture implemented to support issue of one, two, etc. instructions and accordingly 4 - wide, 6 - wide superscalar architectures are designed by various vendors.



- In general a superscalar architecture is called as ***n-wide superscalar architecture*** if conceptually it supports to fetch and dispatch of ***n instructions*** in every cycle.

☞ Superscalar architecture processor

- The superscalar architecture contains multiple execution units for instruction executions. In this architecture a single centralized register file is used to read operands from it and write results into it by each execution unit.
- The result written back to the register file by the execution units becomes visible to all of the execution units on the next cycle. This way it becomes possible to execute on different units from the operations that generate their inputs.

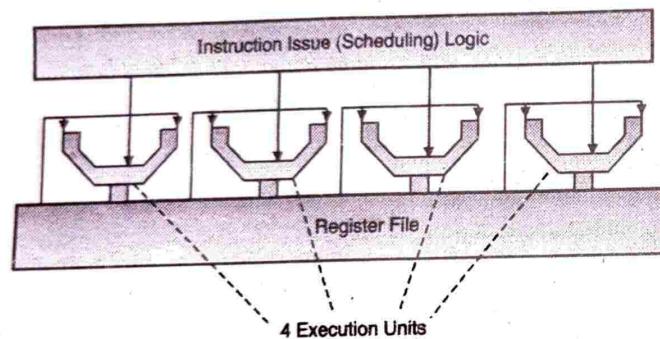


Fig. 1.9.3 : Instruction Issue Logic and Four execution units in a superscalar

☞ ILP (Instruction Level Parallelism) in Superscalar processors

- The superscalar architecture uses a complex bypassing hardware to reduce delay between dependent instructions.
- The delay is reduced because the bypassing hardware forwards results of instruction executions to all execution units.
- The superscalar processor contains the instruction issue logic which is used to store instructions of a program. This instruction issue logic provides instructions to the units in parallel.

☞ Instruction Issue Logic in Superscalar processors

- The changes in the control flow in program execution due to branches occur simultaneously across all of the units.
- The program developments under superscalar processor become much easier because of the occurrences of simultaneous activities across all units.

☞ Hardware used in a superscalar processor

- The instruction level parallelism is extracted from a sequential program from the hardware unit used in a superscalar processor.
- The instruction issue logic of a superscalar processor examines the instructions in the sequential program to determine which instructions may be issued on that cycle.

☞ ILP strengths and weakness

- The superscalar processor executes instructions in parallel and in this way it is possible to achieve significant speedup while executing different types of programs.
- The maximum improvement in the performance depends upon the limitations imposed by instruction dependencies.

☞ Addition of more execution units to a processor

- The addition of execution units decreases the instruction execution times. For example shifting from one execution unit to two execution units provides substantial reductions in execution time.
- However, as the number of execution units is increased to four, eight, or more, the additional execution units spend most of their time idle, particularly if the program has not been compiled to take advantage of the additional execution units.

☞ Multiple-issue Processors

- A multi-stage instruction pipeline and the superscalar processor are generally used in modern microprocessors for achieving instruction level parallelism.
- In fact most of the modern processors support four- or six-issue superscalars.

☞ Pipeline In Superscalar

- Fig. 1.9.4 shows the basic arrangements in the pipeline implemented in superscalar processors.

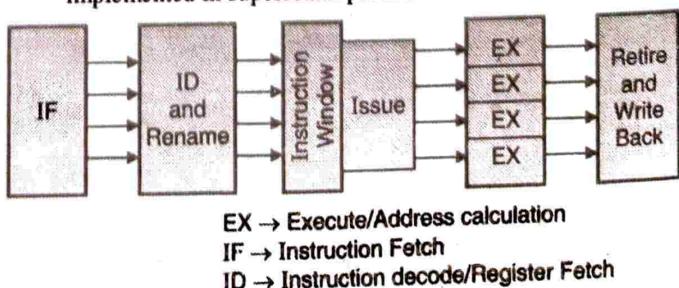


Fig. 1.9.4 : superscalar pipeline

- The branch prediction logic used in the superscalar pipeline provides control independencies for instructions in the instruction window.
- The register renaming is used here to achieve name independencies in the pipeline. This way only true dependency like data dependencies and structural conflicts remain in the architecture to be handled.

☞ Sections of a Superscalar Pipeline

A superscalar pipeline is partitioned in distinct sections based on the ability to issue and execute instructions.

☞ In-order section

With the instruction fetch, decode and rename stages - the issue is also part of the in-order section in case of an in-order issue.

☞ Out - of - order section

Instructions can be executed in an order different from that specified in the program.

1.9.3(A) Dynamically-scheduled Superscalar Processor

- Superscalar processors issue varying numbers of instructions per clock and are either **Statically scheduled** (using compiler techniques) or **Dynamically scheduled**.
- Statically scheduled processors use in order execution, while dynamically scheduled processors use out-of-order execution.
- Dynamic scheduling is one method for improving performance in a multiple instruction issue processor.
- When applied to a superscalar processor, dynamic scheduling has the traditional benefit of boosting performance in the face of data hazards, but it also allows the processors to potentially overcome the issue restrictions.
- Put another way, although the hardware may not be able to initiate execution of more than one integer and one FP operation in a clock cycle, dynamic scheduling can eliminate this restriction at instruction issue, at least until the hardware runs out of reservation stations.

☞ Out-of-order execution

- The Fig. 1.9.5 depicts a dynamically-scheduled superscalar machine. At a high level, the machine comprises three parts: *an instruction supply*, *a data supply* and *an execution core*.
- The task of the instruction supply component is to deliver a stream of instructions into the execution core.
- The latter then processes those instructions according to a dynamically-determined execution plan, at times initiating interaction with the data supply component to read from and write to the memory system. Such a machine's performance is determined by a number of factors.

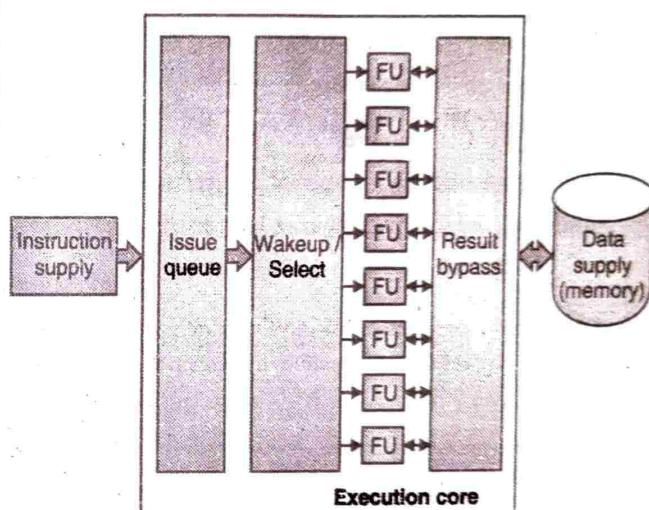


Fig. 1.9.5 : A typical out-of-order superscalar machine

- Among them is the average instruction supply rate, the average latency of memory operations, the amount of ILP available in the instruction stream, and the ability of the execution core to find and exploit that ILP.
- In Fig. 1.9.5 the 8 functional units facilitate 8 - wide superscalar execution. In this each cycle, a dynamic scheduler finds and issues data - ready instructions from the issue queue. Within the core, processing occurs as per the dataflow dependences embedded in the instruction stream.
- Each cycle, a dynamic scheduler (the wakeup/select logic in the diagram) inspects the contents of the issue queue, looking for instructions whose operands are ready, and for which execution resources are available.
- It then issues those instructions to the functional units for execution, in the process rendering their dataflow dependents, if any, eligible for issue in subsequent cycles.



- A result bypass network facilitates rapid issue of those dependent instructions by permitting them to receive their operands directly from the recently-issued producers.
- Within the confines of its execution core, then, the machine is generally able to execute instructions subject only to the constraints imposed by program dataflow.
- The flexibility thereby achieved allows the machine to dynamically respond to unanticipated events like cache misses and control flow misspeculation, and so to partially or wholly hide their effects in the shadow of useful work. It is this capability that underlies the success of the dynamically-scheduled machines.
- Pursuit of more ILP involves scaling an out-of-order design in two, largely orthogonal, dimensions. The first is the machine's *depth* its window size.
- A larger window improves the ability to find ILP by permitting the machine to maintain a larger set of in-flight instructions, and so to expose more independent instructions to the scheduling logic.
- The second is the machine's *width* its issue and execute bandwidth. A wider issue capability improves the machine's ability to exploit ILP when it is available, since it increases the rate at which independent instructions can be extracted from the issue queue.
- Expanding a machine's size in either of these dimensions runs head on into problems on a number of fronts.
- Fig. 1.9.6 shows how the 8-wide machine from previous figure can be partitioned into four 2-wide *processing elements* (PEs).

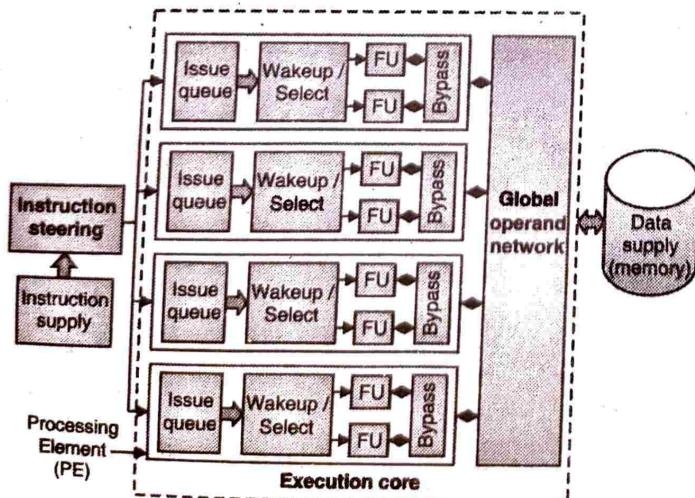


Fig. 1.9.6 : Instruction-level distributed processing

- Such a design eliminates the large, wire-limited and power-hungry parts of the monolithic design, replacing them instead with small, replicated components.
- This permits localization of work at relatively simple execution units that can be engineered to be fast (because communication is local), but it maintains, in aggregate, the large window and wide issue capabilities required for aggressive pursuit of ILP. And the fact that PEs are small and replicated offers opportunity for tackling the power problem, both on the static and the dynamic front.
- This design partitions the execution resources of the machine shown in earlier figure into 4 independent processing elements (PEs), each a self-contained 2-wide dynamically-scheduled execution unit. Instruction steering logic is added to the front-end of the machine to dynamically allocate each fetched instruction to the PE at which it will eventually execute.
- Communication of operands between PEs occurs via a global bypass network, which introduces additional latency between dependent instructions.

1.10 Cache Coherence in Multiprocessor Systems

- The shared memory multiprocessor system equipped with a separate cache memory for each processor.
- Such a system keeps many copies of data and instructions in the fashion that one copy is kept in the memory and one copy in each cache memory.
- When one copy is changed by the processor other copies must also reflect with the changes. Here cache coherency deals with the changes in the shared data should propagate throughout the system in a timely fashion.
- The data transfer amongst nodes is handled by the interconnection networks in the system. The system in which address space is shared needs an additional hardware unit for maintaining of data consistently. For example, when multiple copies are maintained and processors operate on some data it should have the mechanism to maintain that all copies should be managed when changes occurs at any particular copy. In this system all processors share the same address space.



- This creates the possibility for more than one processor to cache an address at the same time. The coherency issue occurs when one processor updates the data item without informing the other processors.
- This causes inconsistency and may create the problem of incorrect executions.

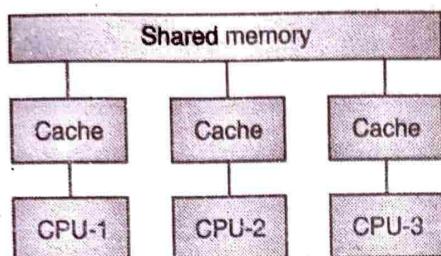


Fig. 1.10.1 : Multiprocessor system with shared memory

☞ Cache coherence problem

- In the shared memory system private cache is associated with each processor. In such situation each cache contains multiple copies of the same data.
- The problem occurs when all processors are allowed to update the data independently. This is called as **cache coherence problem**.
- The system is called cache coherent only when every read operation results in the value which is updated by previous write operation, even by the process at any other processor of that system.
- The system must be able to maintain a coherent view of memory and gives assurance about the program executions on correct version of data.
- It is required to maintain consistency of data in shared resources and individual local cache memory in the system.
- The dealing with this consistency issues is called as **cache coherence**. The problem occurs when the multiple levels of caching with different scopes exist.
- The coherence problem may happen in multiprocessor systems where each processor have local caches as well as can access global shared cache of the system.
- One must look for enforcing of coherency and consistency to solve the coherence problem.

- The behaviour of reads and writes associated with the same memory location is defined by the coherence. On the other hand consistency defines the reads and writes behaviour while accessing different memory locations.
- The basic things included with coherency associated in multiprocessor system are migration and replication. The migration approach is used in coherent caches and it refers the moving of data to the local memory for use.
- The genuine drawbacks while accessing shared data that is not available locally are latency and bandwidth.
- The migration is used to lower the latency and bandwidth here. Another approach to deal with this is replication but it copies the shared data in the local caches.
- Cache coherence protocols are implemented at hardware level instead of the software because of the efficiency of the access.
- A cache coherence protocol is used to keep records about the shared data blocks and their associated states.
- Snooping and directory based are the two basic types of the coherence protocols.

1.10.1 Hardware-based Protocols

- Handling of cache coherence issues at the hardware level are based from various commercial manufacturers provided policies and schemes.
- The hardware manufacturers design and implement suitable cache coherence mechanism in the digital logic.
- After the successful implementation it is included in every node and it becomes transparent to the programmer and compiler.
- The hardware implementation without any types of compiler support shows substantial amount of performance gains.
- There are various hardware methods have been developed depending on the size of the multiprocessors.
- It has been observed that working of snooping protocols is well acceptable when the number of processors is less.
- This way the observation says that the snooping protocols do not scale according to the increased number of processors more than 32.

- The hardware solution for cache coherency problem is based upon some protocols and called as hardware based protocols for cache coherency.
- Hardware based protocols include snoopy cache protocols, directory schemes and cache coherent network architectures.
- Now we will look at different policies based on hardware for providing solutions for the cache coherency problem.
- Hardware based protocols for maintaining cache coherency issues provides memory system coherence guarantee without any assistance from software approaches.
- The inconsistency situations are detected by the hardware mechanisms and appropriate actions are performed according to a protocol implemented in the hardware level itself.
- A unit of data transfer between memory and caches is called **block**. This means a block of data transferred is performed in per unit time. In the system overall data to be transferred is divided into number of blocks.
- An arbitrary number of copies of a block exist at the same time by the use of hardware protocol.
- The basic policies used for maintaining cache consistency are **write - invalidate** and **write - update**.

☞ Write - invalidate

- The consistency of multiple copies is maintained by the write - invalidate policy. According to this policy the read request is performed when a copy of the block exists. The invalidation is applied when a processor updates a block.
- This indicates that the other copies are automatically invalidated when a block is modified by a particular processor.
- The steps handled to perform this process depend upon the interconnection network in use. A subsequent update by the same processor can then be performed locally in the cache, since copies no longer exist.
- Fig. 1.10.2 shows the representation of this policy.

- The Fig. 1.10.2(a) shows copies of block X stored. The consistent copies of X are stored in the memory and in three processor cache. In this way collectively four copies of block X are stored consistently.

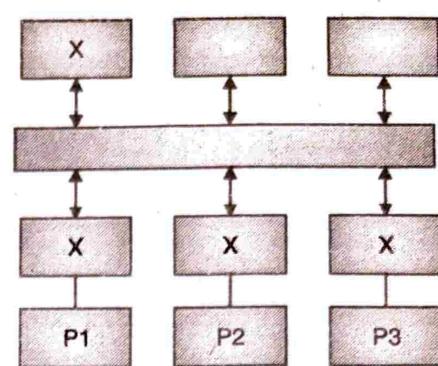


Fig. 1.10.2(a) : Four copy of a block X

- The Fig. 1.10.2(b) shows an update situation. Here processor P1 updated an item in block X. Now the updated block is represented by X'.

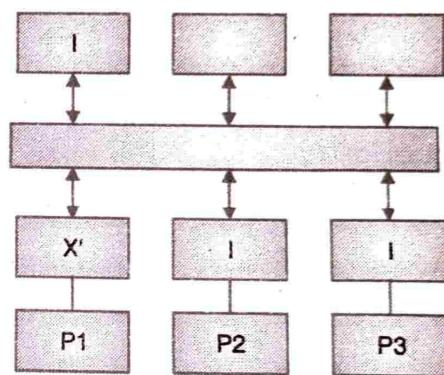


Fig. 1.10.2(b) : Copies of P2 and P3 are invalidated

- After this update operation, indication is handled using invalidation of all other copies and this is denoted by the letter "I".
- When processor P2 requests for the block X' the request is satisfied by the cache attached with processor P1.

☞ write - update

- The processor currently updating or modifying the data in its cache broadcasts the new data over the bus without issuing the invalidation signal.
- The other processor caches are able to update their copy of the data because of the update made by one processor.
- This differs from the write - update in the sense that it does not create only one copy for writes.
- The Fig 1.10.2(c) explains the changes made when write - update policy is applied.

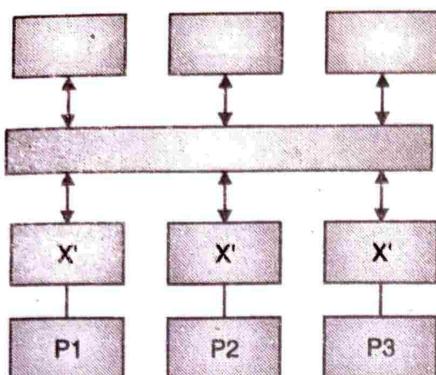


Fig. 1.10.2(c) : write update policy

- This policy has a different approach for consistency maintenance.
- In this case all copies are not invalidated instead of that updates applied to other copies also.
- The memory copy is updated or not it depends upon the implementation of protocols.

Snoopy cache Protocol

- Snoopy cache protocols are used in bus based systems. These systems are basically bus based multiprocessor systems.
- In such systems the memory transactions are carefully observed by all processors. These processors take appropriate actions whenever required in the form of invalidation and update operation of local cache content.
- The term snooping is used because when the data is copied to the local cache every other cache with the data from the same block can track the sharing of the memory block.
- The caches in symmetric multiprocessor systems are usually connected with a bus.
- At this stage the other caches or cache controllers snoop via the bus and watch about their data is being requested by another cache.
- Fig. 1.10.3 describes this operation. Here cache L and cache M are connected via a bus.
- The data transfer occurs between the two caches when cache M receives a snoop hit from cache L, which has actually requested it from memory.
- The snoopy protocol will be covered now in detail. As mentioned earlier this protocol is suited for the multiprocessor system with shared memory and has a shared bus interconnection among nodes.

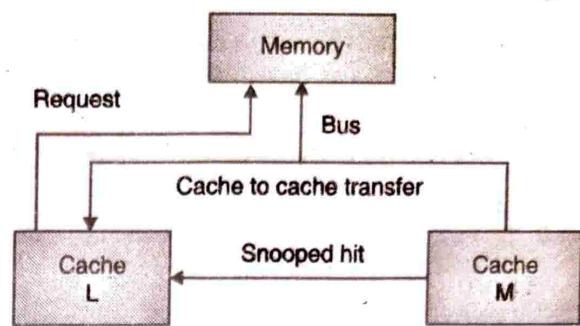


Fig. 1.10.3 : Cache coherence

- The shared bus has a property to broadcast coherent information in very fast manner among all processors connected as nodes in the system.
- This multiprocessor system strictly maintains consistency of data by updating information among all the processors immediately when any of them updates data.
- The shared bus becomes bottleneck when large number of processors is included. This situation can be handled by increasing the bandwidth of the bus but the consequence of this is the increased memory delay in communications.
- The write - invalidate and write - update are considered as consistency commands used in the policies we have discussed previously. The implications of consistency commands not considered for other networks such as shared buses.
- In some of the interconnection networks like bus network it is feasible to broadcast consistency commands to all caches. In this case it is required that every cache should execute consistency commands to find out whether it refers to data in the cache.
- This type of protocol is called as **snoopy cache protocol** because each cache "snoops" on the network for every incoming consistency command.

Write-Invalidate snoopy cache protocols

- The first write - invalidate snoopy cache protocol named as write - once was proposed by Goodman and later on revised by Archibald and Baer. We will have the detail view about this protocol.
- The write - once protocol provides the assumption of a state with each copy of a block. These states are named as **Invalid**, **Valid**, **Reserved** and **Dirty** for a copy of a block.



- **Invalid** : This state indicates when the copy is inconsistent.
 - **Valid** : A copy is consistent with the memory copy.
 - **Reserved** : When the data is written exactly once and the copy is consistent with the memory copy.
 - **Dirty** : Copy is the only one in the system but data is modified more than once.
- A copy - back memory update policy is used in write - once approach. This indicates that the entire copy of the data block must be written back to memory when it is replaced. This operation is performed only when the data has been modified during its cache residence time (i.e. the state is dirty). The normal memory commands read block (Read-Blk) and write block (Write-Blk) are applied and besides these commands other consistency commands required to maintain consistency are used. These commands are as shown below :
- **Write-Inv** : This command invalidates all other copies of a block.
 - **Read-Inv** : This is used to read a block and all other copies will be invalidated.
- State transitions occur either from the local processor read and write commands or the consistency commands. The local processor read and write commands are named as **P-Read** and **P-Write** whereas consistency commands come from the global bus are named as **Read-Blk**, **Write-Blk**, **Write-Inv**, and **Read-Inv**.
- The Fig. 1.10.4 shows the graphical state transition representation for the write - once protocol. In this diagram solid lines mark the actions initiated by the processor. The dashed lines represent consistency actions initiated by other caches and sent over the bus.
- The protocol operation can be specified using the clear actions taken on processor reads and writes.
- The state transitions do not occurs for the read - hit operation because it can always be performed on the local copy of the cache.

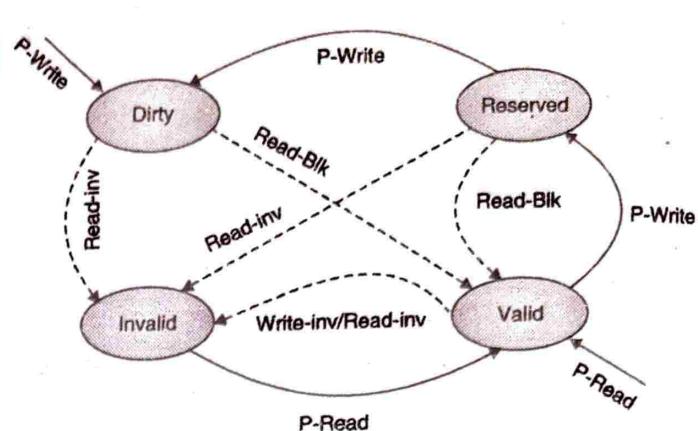


Fig. 1.10.4 : write-once protocol state-transition diagram of cached copies

Write-update snoopy cache protocols

- Firefly protocol has been implemented in Digital Equipment Corporation's Firefly multiprocessors workstation system.
- It is not required to describe the machine in which the protocol was implemented but it is worth to have brief mention about it here.
- The Firefly workstation machine consists of from one to nine VLSI VAX processors. Each processor equipped with the floating point unit and a cache memory attached. In the system coherent caches provides a consistent view of the memory.
- This protocol (Firefly protocol) is one of the examples of a write - update protocol. In this protocol three states are possible with the cached copy of a block.
- These states are listed below :

 - **Valid-exclusive** : This state indicates about the only cached copy and this copy is consistent with the memory copy.
 - **Shared** : The copy of the data to be shared is consistent and there are existences of other consistent copies also.
 - **Dirty** : This is the only available copy of the data and in this state the inconsistency of the memory copy of the data is assumed.

- There are two different policies used by the Firefly protocol for different types of blocks.
- The Firefly protocol has implementation of **copy-back-update** policy for private blocks whereas shared blocks are supported using **write-through policy** options.



- The write - update consistency command is used to update all copies for maintaining of consistency in the system.
- The Fig 1.10.5 summarizes the state transition – diagram of this model for states of cached copies for the Firefly protocol.

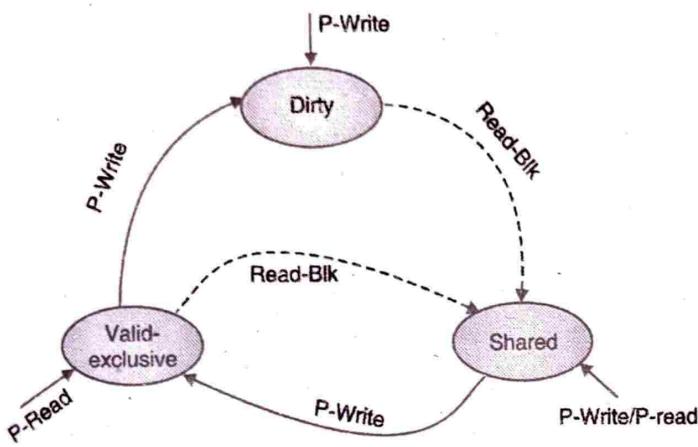


Fig. 1.10.5 : State-transition diagram for the Firefly protocol

Directory based schemes

- This is a cache coherence protocol that is not based on broadcasting of data. This protocol stores the locations of all cached copies of every block of shared data.
- The cached locations can be kept at centralized place or at distributed manner and are called **directories**.
- Used in scalable cache-coherent distributed memory multiprocessor systems where cache directories are used to keep a record on where copies of cache blocks reside.
- The directory based protocol considers the system shown in the Fig. 1.10.6.
- As shown in the Fig. 1.10.6, a directory is included in each node of the system for implementation of cache coherence.

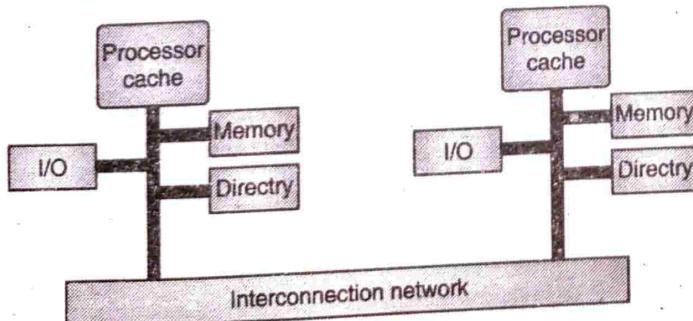


Fig. 1.10.6 : Directory based protocol organization

- The two basic operations called as handling read miss and handling a write to a shared cache block are implemented in this protocol.

- The implementations of these operations are possible if the directory track the state of each cache block. The different states considered here are named as shared, uncached and exclusive.

- o **Shared :** The value in the memory is up to date of the block cached by one or more processors.
- o **Uncached :** Processors do not have the copy of the cache block.
- o **Exclusive :** Exactly one processor owns the copy of the cache block and it is designated as the owner. In this state the memory copy becomes out of date because the owner has updated the block.

Performance issues of Hardware cache coherence protocols

- The outline of the significant issues that affects the performance of hardware cache systems are discussed in brief in this section. The directory based scheme suffers with two main drawbacks.
- The first problem associated with the directory based scheme is the interconnection network suffers from the substantial invalidation or update traffics.
- The second major issue is the memory blocks that have been modified by the processor but not yet updated in the main memory.
- These modified blocks have to refer the directory to the cache containing the referred block.
- One more factor affects the hardware scheme is the size of the cache line. In this regards the software assistance have not yet provided to deal with the cache line size makes an open question about the cache line size.
- This means the question is still open that whether multiple-word cache lines will provide higher performance than single word lines.
- The performance of the directory based protocols is very good when hundreds or thousands of processors are included, but it shows scalability barriers beyond this point.
- The directory based protocols are used in the current commercial systems with very good performance achievements.



1.11 Levels of Parallelism

Q. 1.11.1 Write short note on Levels of Parallelism.

(Refer section 1.11)

(4 Marks)

- There are different categories of parallelism like hardware and software parallelisms.
- The parallelism in hardware level includes the parallel activities provided at architecture level itself. In the other side the software parallelism is broadly divided into parallelism based on task and data levels.
- This section is based on the points to be focused on types of parallelism in applications. The types of parallelism in applications include: instruction - level, Thread - level or Task - level parallelism, Transaction - level parallelism etc.

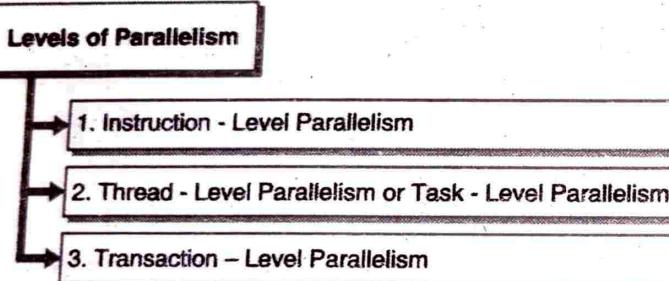


Fig. 1.11.1 : Levels of Parallelism

1.11.1 Instruction - Level Parallelism

- Modern processors are equipped with the ability to execute several operations of a program simultaneously. Instruction - level parallelism (ILP) is the measure about how many operations associated with a program can be performed simultaneously.
- This is handled by overlapping the instructions of a program during executions and is actually termed as Instruction - level parallelism (ILP). This actually characterized by the fact that the multiple instructions from the same instruction stream execute concurrently.
- Such instructions are generated and managed by the hardware level for example superscalar processors or the compiler also generates instructions to be executed concurrently as an example VLIW.

1.11.2 Thread - Level Parallelism or Task - Level Parallelism

- The thread - level parallelism occurs when multiple threads or instruction sequences in a same application executed concurrently.
- In thread - level parallelism a program is divided into independent parts in which each part is called as a **thread** and these threads execute concurrently.
- The parallelism occurs here because threads can have the effects of separate independent programs running together or separate activities are being performed inside the same parallel program.
- This type of parallelism is used in MIMD class of machines defined in the Flynn's taxonomy of the computer systems. There are basically two different types of thread - level parallelism namely *fine-grained* multithreading and *course-grained* multithreading.
- The thread level parallelism comes under the task - level parallelism because it is based upon the organization of a program or computing solution into a set of threads for simultaneous executions on different processors.
- Threads are generated by the compiler and managed by the hardware or also by the compiler in terms of user - level library functions. The drawbacks of thread - level parallelism include the limitations by the communications and synchronization and by the characteristics of algorithm. This means if the algorithm itself has dependency among various operations then the implementation becomes difficult.

1.11.3 Transaction - Level Parallelism

A **transaction** is a sequence of information exchange and related work considered as a unit in programming. The term transaction - level parallelism is used to describe concurrent executions of multiple processes and threads from different transactions.

1.11.4 Memory

This is described in computer architecture in which multiple memory operations are used to perform concurrently. In particular pending memory operations like cache memory and TLB misses operations are handled simultaneously.

1.11.5 Function - Level Parallelism

- The function – level parallelism is based on the steps to be performed by the tasks and functions associated with these tasks.
- This means the functions execute concurrently and completes the tasks defined in it.
- The function - level parallelism focuses on the computational tasks to be performed instead – of the data manipulation done using the computation.
- The problem to be solved using the computer system is divided according to the work that must be performed.
- In this way the solution of the problem becomes the collection of tasks to be performed in parallel and each task handles a portion of the overall work.
- Consider a program to do arithmetic operations as an example to understand function – level parallelism.
- Here the overall program is divided into four functions named as ADD(), SUB(), MULT() and DIV() and these functions execute concurrently by the use of different processing elements. The arrangement of this example is shown in Fig. 1.11.2.

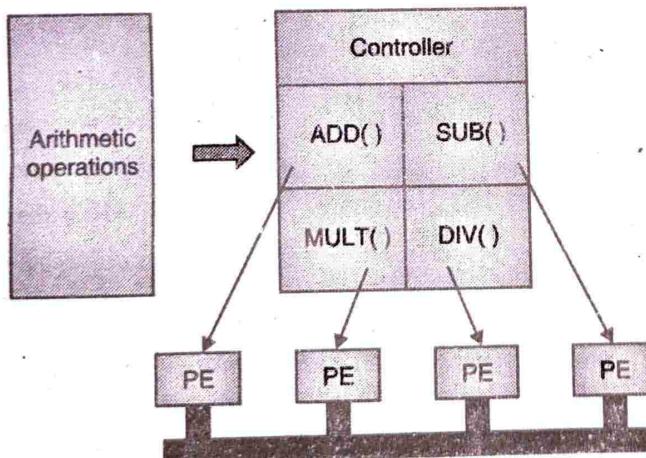


Fig. 1.11.2 : Function level parallelism example

1.12 Models

Q. 1.12.1 Explain SIMD, MIMD and SIMT architecture.
(Refer sections 1.12.1, 1.12.2 and 1.12.3)
(6 Marks)

- The researchers of the computing field suggested basic control mechanisms such as sequential control, dependency control and demand control.
- The **sequential control** is described as the passing of control from one instruction to another. This is applicable in traditional Von Neumann model of computation.
- The **dependency based control** typically called as **control transfer mechanism by dependency**. In this approach the control transfer is based on the availability of arguments of the operation to be performed.
- It is strictly based upon the principle in which an instruction only executes when all of its operands become available.
- The third category of control passing mechanism called as **by demand approach**. It is based upon the approach in which an instruction is executed whenever the result of the instruction execution is required by another instruction.
- In this approach the control returns back to the instruction actually invoked the executed instruction.
- The data mechanism implemented by the researchers are of two types namely by value and by reference.
- According to the approach of by **value** data mechanism an argument carries the actual data used in computations.
- In another approach for passing data to the arguments is **by reference** in which an argument points to the memory location of the data.
- The different computing models suggested by the computer researchers are mentioned on the Table 1.12.1.

Table 1.12.1 : Different models based on control and data mechanisms

Data Mechanism	Control mechanism		
	Sequential	Dependences	Demand
By value		Dataflow	String reduction
By reference	Von Neumann	Parallel control flow	Graph reduction

- In this section, we will discuss about the physical architectures of parallel computing systems. These systems are used to handle parallel activities at different levels

according to the architecture implementations. Parallel activities are implemented from instruction level to thread levels and achieve efficiency of program executions.

1.12.1 SIMD (Single Instruction Stream Multiple Data Stream) Architecture

- In SIMD processors one instruction works on several data items simultaneously by using several processing elements (PEs), all of which carry out the same operation as shown in Fig. 1.12.1.

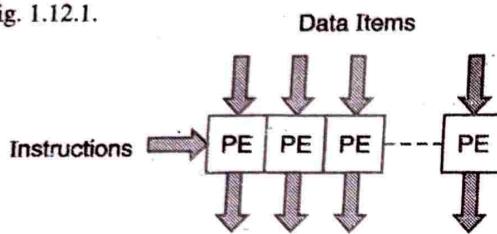


Fig. 1.12.1 : SIMD architecture

- The SIMD model of systems uses a single control unit to dispatch multiple instructions to various processing elements.
- In the SIMD computing model a single control unit is used to read instructions by a single Program Counter (PC), decoding them and sending control signals to the PEs.
- The number of data paths are depends upon the number of processing elements. Data are to be supplied to and derived from PEs by the memory. The structure of the system is shown in Fig. 1.12.2 called as **array processor**.

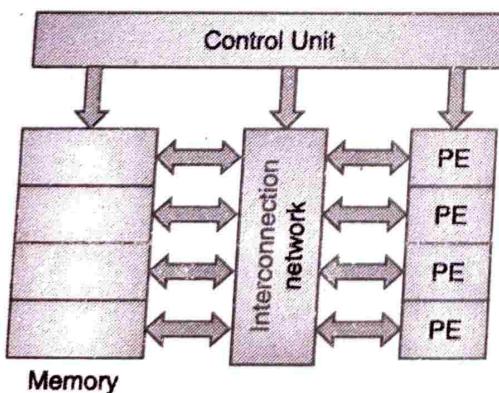


Fig. 1.12.2 : Processor array

- The different units like PEs and memory modules are interconnected by interconnection networks. The data transfers to and from the PEs are managed by interconnected networks.

1.12.2 MIMD (Multiple Instruction Stream Multiple Data Stream) Architecture

- This model represents the system capable of executing multiple instructions on multiple data sets simultaneously.
- Most of the multiprocessing systems come under this category of systems. In fact the MIMD is used to describe a parallel machine able to perform independent computations at the same time.
- The MIMD class of machines can execute independent programs at the same time.

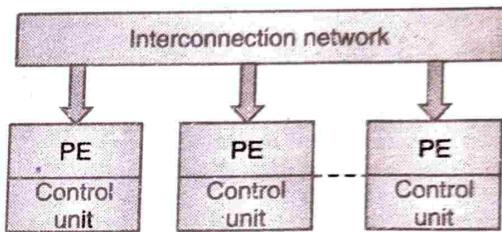


Fig. 1.12.3 : MIMD architecture

- The processing elements included in MIMD class of machines execute different programs at a time.
- This means in the MIMD class of machines each processor fetches its own instruction and operates on its own data.

Types of MIMD Architecture

MIMD machines are classified into two different types based on the use of memory. First type of MIMD architecture is implemented as shared memory systems and called as **Shared Memory MIMD architectures** and second type of machine is distributed memory MIMD architecture.

Shared Memory MIMD Architecture

- In this type of shared memory MIMD architecture a set containing processors and memory units are created.
- The processors can access any memory by the use of interconnection networks.
- Here overall memory is being shared among the processors and collectively forms a global address space. The Fig. 1.12.4 shows this architecture.

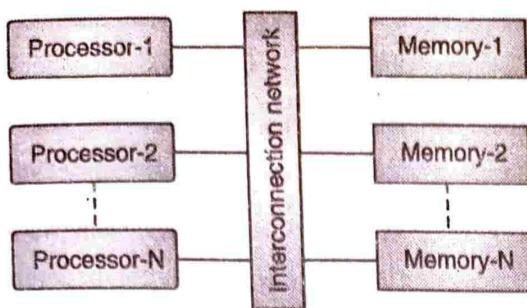


Fig. 1.12.4 : Architecture of shared memory MIMD machine

The following are some of the classes of shared memory MIMD architectures :

☞ **NUMA (Non Uniform Memory Access)**

The overall memory is considered as divided into number of memory blocks and each block is attached with processor.

☞ **COMA (Cache Only Memory Access)**

This type of system is characterized by the fact that each memory block is designed to be worked as a cache memory.

☞ **CC-NUMA (Cache Coherent Non Uniform Memory Access)**

Each cache memory along with the memory blocks are attached with the processor.

1. Distributed Memory MIMD architecture

- This type of MIMD architecture is based on the design in which each processor equipped with its own memory unit.
- The combination of processor and memory is called as processing elements.
- All such processing elements are connected to interconnection network and provide the facility to have communications with each other.

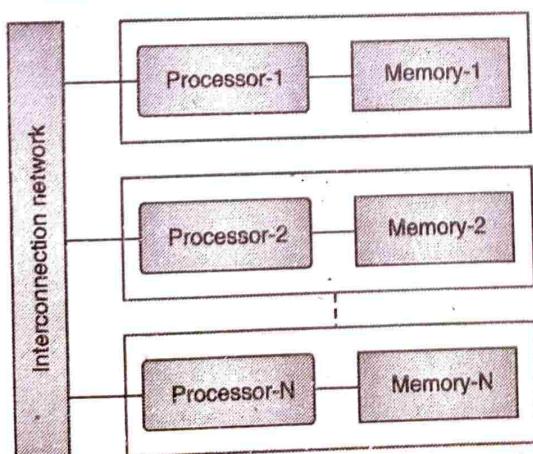


Fig. 1.12.5 : Architecture of Distributed memory MIMD machine

2. Comparison of SIMD and MIMD architectures

- As we have discussed about SIMD and MIMD as SIMD is the single instruction and multiple data whereas MIMD is multiple instruction multiple data class of machines.
- The table outlines the points which are useful parameters to distinguish between SIMD and MIMD.

Property	SIMD	MIMD
Architecture complexity	SIMD architectures are simple	MIMD architectures are Complex
Cost	Low cost incurs using SIMD architectures	Medium cost incurs using MIMD architectures
Program memory requirements	Only one program copy is required to be stored	Number of program copies depends on the number of PE. Each PE stores its own copy of program
Lower instruction cost	The control unit contains only one decoder	Each PE is required to contain one decoder
Size and Performance	Provides scalability in terms of size and performance	MIMD architectures are complex in size but provides good performance
Synchronization	Synchronization is required implicitly at program level	Handling of synchronization requires explicit data structures and operations
PE-to-PE communication	Low overheads because SIMD is based on send and receive primitives for automatic synchronization	Communication among processing elements require explicit synchronization and protocols for identification

1.12.3 SIMT

- The Single Instruction Multiple Thread architecture has been developed to achieve high - throughput computing with high energy efficiency.
- In today's computing field SIMT processors are used most of the times for Graphic Processor Units (GPUs).
- The SIMT architecture is created with the combined supports from the hardware and software sides.



- The hardware architecture side in SIMT provides an approach for the conversion of scalar instructions into vector - style single - instruction multiple data (SIMD) processing for the achievement of energy efficiency.
- On the other hand the software side, the SIMT programming model for example CUDA and OpenCL provide facilities to achieve data parallelism to be implemented as task - level parallelism.

1.12.3(A) Micro Architecture of SIMT

- The Micro architecture of an SIMT core is shown in the Fig. 1.12.6. An instruction is selected and issues from ready

warps by a warp scheduler to the multiple execution pipelines.

- These multiple execution pipelines are also called as streaming processors or processing elements (PEs).
- A warp is considered as a ready warp when all the dependencies require for the execution of the next instruction have been resolved.
- In the architecture each entry of the warp scheduler contains the information of a warp. The instruction from the instruction cache is read using the program counter (PC) field.

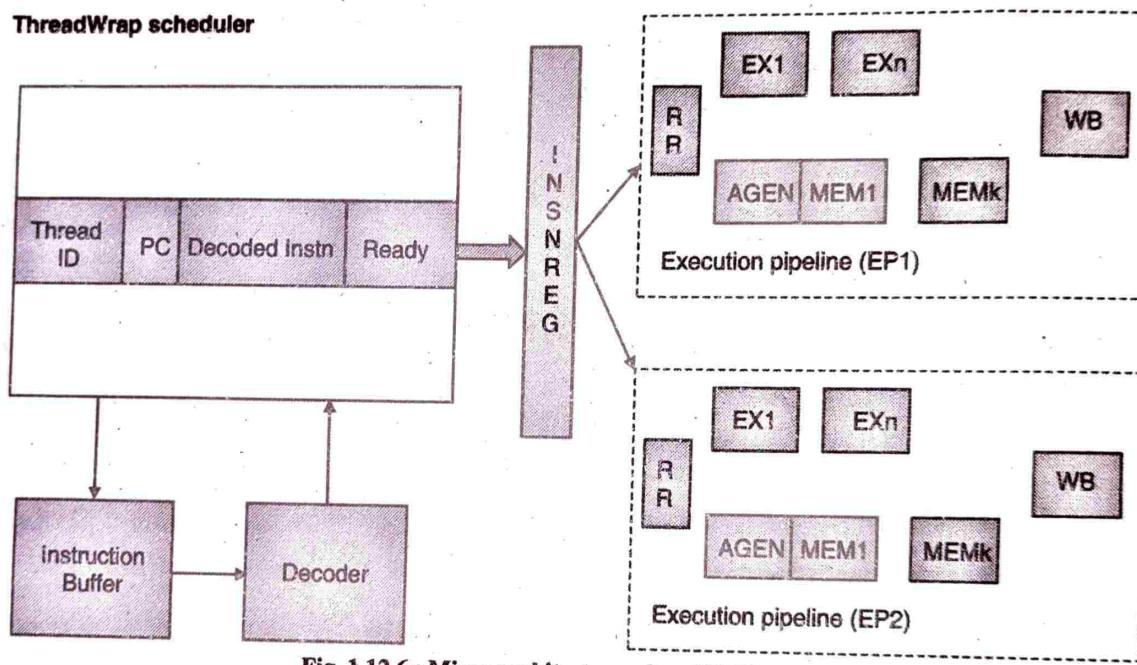


Fig. 1.12.6 : Micro architecture of an SIMT core

1.12.3(B) Mapping SIMT Workloads to SIMT Cores

- There are multiple SIMT cores included on a chip according to the approach in a typical SIMT architecture. For example one of the SIMT core is called as a streaming multiprocessor (SM) based on CUDA terminology and it is referred as a compute unit using the OpenCL terminology.
- Fig. 1.12.7 describes dispatching SIMT workloads to SIMT cores. The thread blocks are mentioned and an SIMT kernel is invoked as a grid of thread blocks (TBs).
- As shown in the figure that the workload dispatcher forwards the TBs to SIMT cores.
- The cores in SIMT check the availability resource so that one more TB can be accommodated.
- It checks register files, shared memory or scratchpad memory, number of threads and number of TBs.
- If the possibility for accommodation exists a TB is assigned to the SIMT core by the workload dispatcher.
- All occupied resources of an SIMT core are released when a TB finishes its execution on that SIMT core and a new TB can be dispatched for the execution.

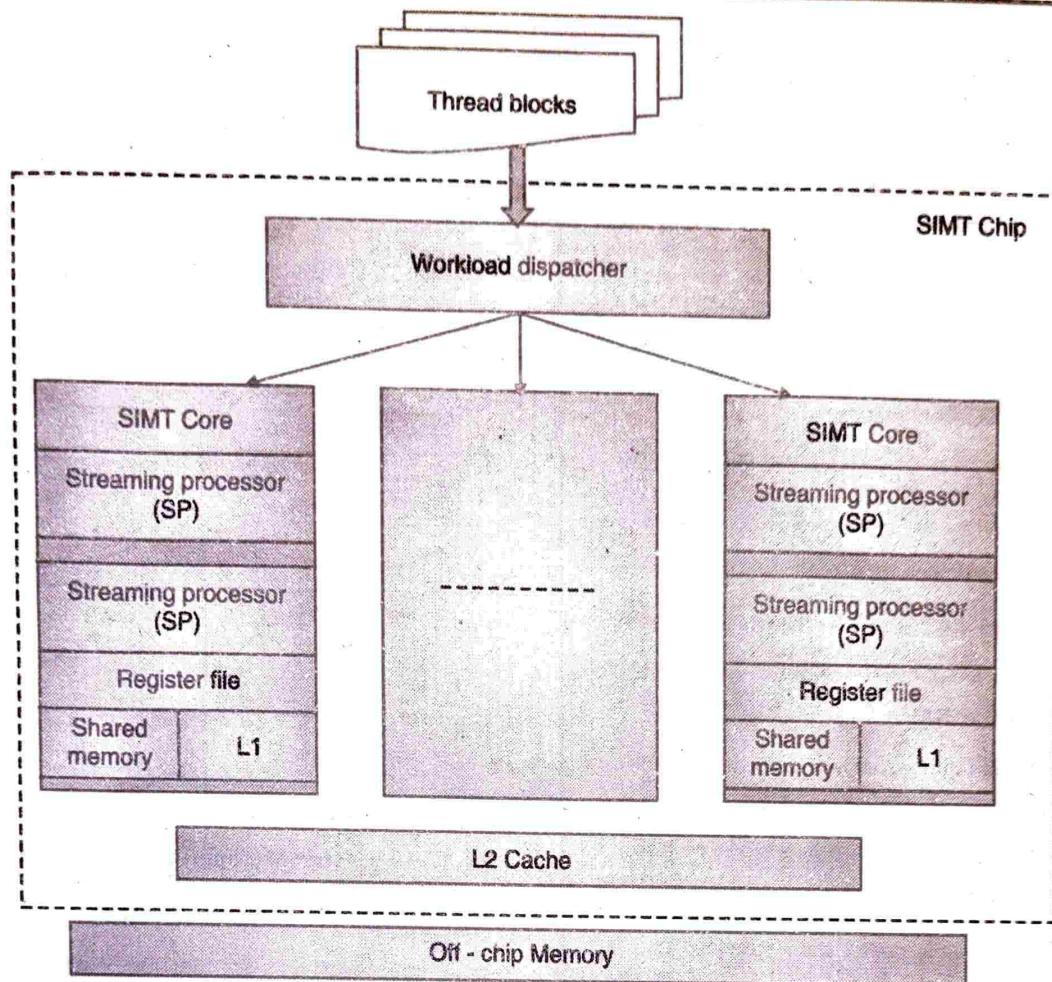


Fig. 1.12.7 : Dispatching of workloads to SIMT cores

1.12.3(C) SIMT Programming Model

- The data parallelism is expressed as task - level according to the SIMT programming model.
- The programming model here follows the single - program multiple thread paradigm, this indicates that all threads share the same program.
- The scalar codes commonly called as kernel code is written by an application developer to utilize the power of SIMT machines.
- All threads execute the same kernel function and are also referred as work items. Each thread identifies the data to be operated upon using the unique identifier.

1.12.4 SPMD (Single Program Multiple Data)

- SPMD is a programming model for parallel computations and it is possible to construct it using the combinations of other parallel programming models.

- In this approach a single program is created by combining multiple tasks (threads or processes) and these tasks can be executed concurrently.
- The SPMD approach of parallel programming has widespread use in the field of massively parallel and scientific computations.
- This model is based on the approach of having multiple processors and mapping of data elements of the data structures into these processors.
- The SPMD model is characterized by the fact that the executable program running on these nodes are same but the data upon which the steps of these programs operate are different.
- This involves the splitting of application data among the available processors. This type of parallelism is also referred to as geometric parallelism, domain decomposition, or data parallelism. Fig. 1.12.8 represents a schematic representation of this paradigm.

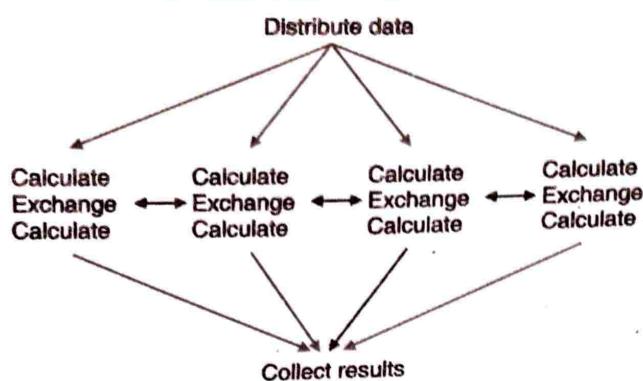


Fig. 1.12.8 : Base structure of SPMD model of computing

- The overall data used by the application is divided and given to a set of processes included to solve the problem.
- One can easily understand SPMD by the use of an example. Consider the matrix multiplication program on four nodes of a parallel computing system.
- In this arrangement three nodes are given exclusive responsibilities to act as slaves whereas one node acts as a master as well as involved in computations as a slave also.
- Also assume two matrices A and B of size 100X100 for multiplications. The whole matrix can be divided into four chunks with 25 elements in each because overall four nodes are available for computations.
- Matrix A is divided into four chunks as $a1[25][25]$, $a2[25][25]$, $a3[25][25]$ and $a4[25][25]$ where $a1$ contains first 25 elements, $a2$ contains second 25 elements, $a3$ contains third part of 25 elements and $a4$ contains last 25 elements. Similarly matrix B is divided into four chunks of 25 elements as $b1$, $b2$, $b3$ and $b4$.
- For such systems the program itself is required to be written in two different programs and popularly known as **master - slave form of programming**.
- In this programming approach the actual computations to be performed is kept in the slave program whereas master program is responsible to accept the input data from the input file or user inputs and divide and distribute the data to all slave programs for processing and waits for the results from all the slaves. Once it gets the results combine all intermediate results and finally produces the final result of the computations.
- In this example the master is responsible to accept whole matrices A and B and divide these into chunks. The

multiplication steps are written in the slave program. The executable copies of these programs are installed in all the nodes before starting the task.

- In this arrangement three copies of slave program are installed in three individual slave nodes and master node will contain master program and it also has one copy of executable slave program.

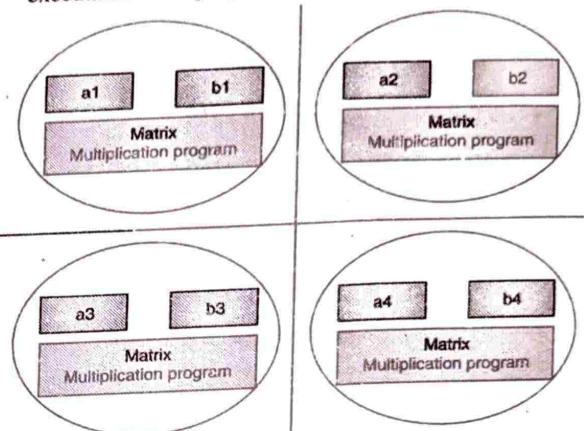


Fig. 1.12.9 : Matrix Multiplication as a SPMD example

- The user starts execution of the master program in master node and master program indirectly starts executions of all four slave programs in four nodes. Once all the slaves have been started, master distributes the chunks of matrices A and B to all the slaves.
- The slave programs in different nodes perform the multiplications of different sets of data simultaneously and produce their part results back to the master.
- Now master is responsible to combine all these intermediate results and finally produces the output of the multiplication.

1.12.5 Dataflow Model

Q. 1.12.2 What are the Types of Dataflow Execution Model? (Refer section 1.12.5) (4 Marks)

- The dataflow model of computation is based on the graphical representation of programs in which operations are represented by nodes and arcs are used to represent data dependencies.
- The parallel processing activities under dataflow model of computing provides various properties to solve the computing problems in efficient ways. Some of the properties associated with dataflow computing model are :



- Asynchronous executions :** The dataflow model of computation is asynchronous by nature. This means an instruction can only execute if all its required operands are available. This provides the nature of implicit synchronization for parallel activities in dataflow computing model.
- No sequencing :** In dataflow model of computing instructions are not necessary to execute in any sequential fashion. The dataflow model only based on data dependencies in the program and other than this no sequencing is expected.
- Dataflow representation :** In dataflow model of computing no sequencing required and it makes the possibility of dataflow representation of a program. The dataflow representation of a program provides the use of all forms of parallel program execution without the assistance of any explicit tools of parallel executions.

1.12.5(A) Dataflow Graph

- A **dataflow graph** is a collection of nodes and arcs. The dataflow computation works on the basis of dataflow graphs. As we all know that the computers understand only machine level language in the similar fashion dataflow computers also works but the machine level language representation of dataflow computers is dataflow graphs.
- Dataflow graph** is a directed graph which shows the data dependencies between a numbers of functions.
- A dataflow graph contains nodes and edges where each node of the dataflow graph contains input and output data ports.
- The connection between output ports and input ports are represented using the edges of dataflow graphs. In this way the dataflow graphs are used to represent programs for data flow computations.
- The Fig. 1.12.10(a) shows an example dataflow graph for a given expression also shown.

Operand-1 Operand-2

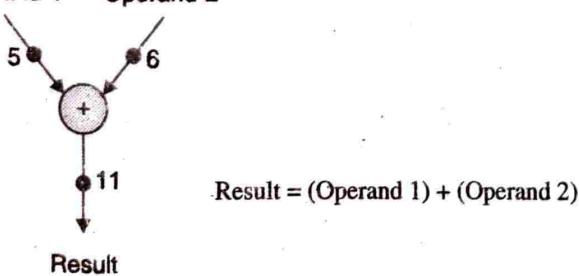


Fig. 1.12.10(a) : Dataflow graph

- The Fig. 1.12.10(b) shows the basic primitives of the dataflow graphs.

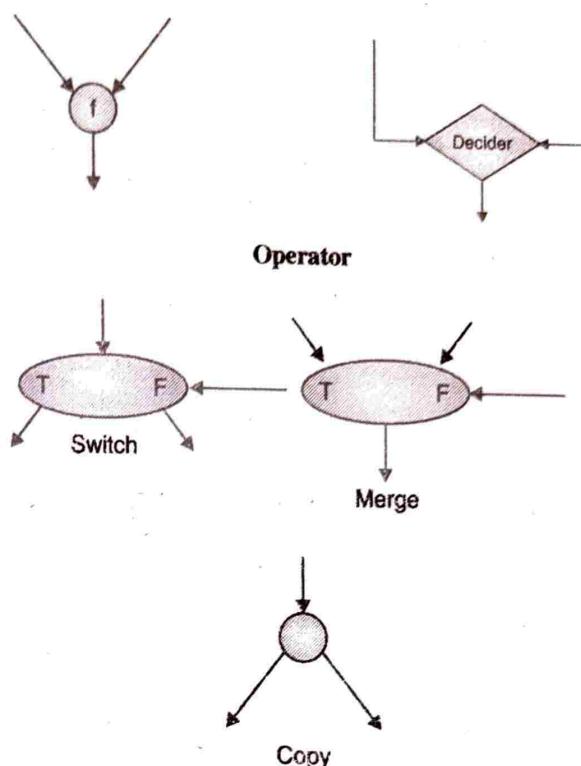


Fig. 1.12.10(b) : Basic primitives of the dataflow graphs

- Operator :** An operator performs some operation (say f) and a data value is produced as the result of the performed operation.
- Decider (Predicate) :** A decider generates a true or false value based on the input tokens provided to it.
- Switch (or Merge) actor :** Switch and merge actors are used for directing data values. It means either a switch or a merge can be used for directing data values depending upon the input. The switch actor directs the input to one of the outputs based on the input control.
- Copy :** An identity operator called as copy is used to duplicate input tokens.

Basic nodes for computations

In dataflow model of computation three basic nodes are used to represent three basic things. These basic nodes are a source node or constant generator, copy or duplicate node and operator or function nodes. The following are the notations used for primitive nodes. These primitive nodes are shown in the Fig. 1.12.11.

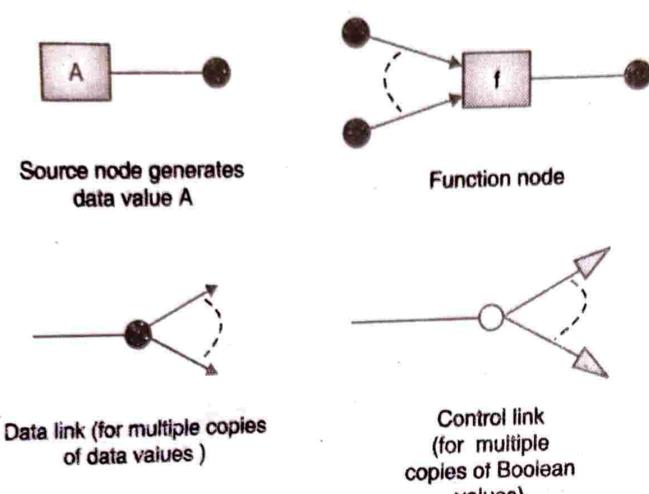


Fig. 1.12.11 : Basic nodes in dataflow model

1.12.5(B) Dataflow Computations

- The traditional computing is based on Von Neumann architecture and all computers work on that basis is termed as control flow machines.
- In control flow machines the computation is based on program counter (PC) and instructions are executed sequentially under the control of program counter. The programs executed sequentially are by nature slow.
- Dataflow computations were invented to utilize the computing power of machines effectively in the form of parallel executions.
- The dataflow computing works on the basis of the principle in which an instruction executes when all required operands become available.
- The availability of all the operands for an instruction execution makes clear idea about the difference between control flow and data flow computations.
- The program counter is not required in dataflow computations because of the data dependency constraints for instruction execution.
- Instructions are not arranged in any order in a dataflow program for dataflow computations. Instead of the ordered executions, dataflow instructions execute based on the data dependency.
- This means an instruction only executes whenever all data required by it are available.

☞ Dataflow computing principles

- There are two basic parameters asynchrony and functionality can be considered on which dataflow computations differs from control flow computations.
- The instructions only get execute when all the required parameters of that instruction become available.
- This is called as enabled instructions this means an instruction becomes ready for execution when it is enabled.
- The important property of the dataflow computation is the concurrent and without interference executions of instructions.
- This means any two enabled instructions are allowed to execute in any order without any interference.
- The basic things such as variables and memory updating used in control flow computing are not exist in dataflow computing.
- In dataflow computing to handle things it supports entities called as objects and actors, instead of variables and memory updating.
- In dataflow computing objects and actors are involved for completion of the defined and expected tasks. Instructions are called as actors whereas data structures or scalar values are called as objects in dataflow computing. For example suppose we have two values going to provide as inputs to an instruction and after execution of the instruction output is passed to another instruction if required.
- This is described in dataflow computing as objects are passed to an actor and output object is passed to next actor(s) for other activities.
- The dataflow computing contains nodes or vertices to perform operations and the operands required are conveyed from one node to another in data packets. These data packets are called as tokens in dataflow computing.
- The conventional computing contains register files for storage but the dataflow computing have token store.
- The operands are conveyed from one node to another in data packets called tokens. Instead of a register file, they have a token store.



Basic data flow mechanism

- A dataflow computer uses template for each instruction and it consists of the operator, receivers for the operands, and result destinations. Incoming and outgoing arcs are used for marking of operands and results respectively.
- The following example shows an expression and its associated template used in dataflow computing. The dataflow program can be represented using dataflow graphs.
- These nodes are usually stored in the form of templates. The inputs are in the form of tokens and these input tokens are stored in the space provided in templates.
- In addition a template also contains description of nodes of a dataflow program. The description contains information regarding the mapping from input values to output values and a list of outgoing arcs typically termed as destination addresses.

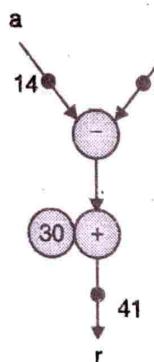


Fig. 1.12.12(a) : Data flow graph for the given expression

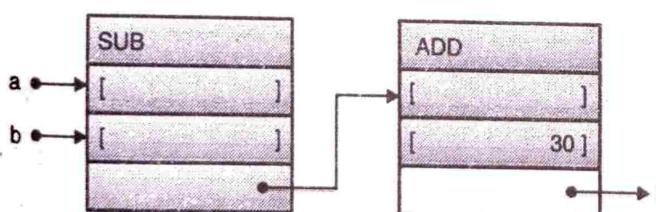


Fig. 1.12.13 : Template for the expression shown in (a)

- The basic and assumed machine components representation is shown in the Fig. 1.12.14. The instruction executes when all required operand values are field in the receivers. The basic working of the dataflow computing is described conceptually here.

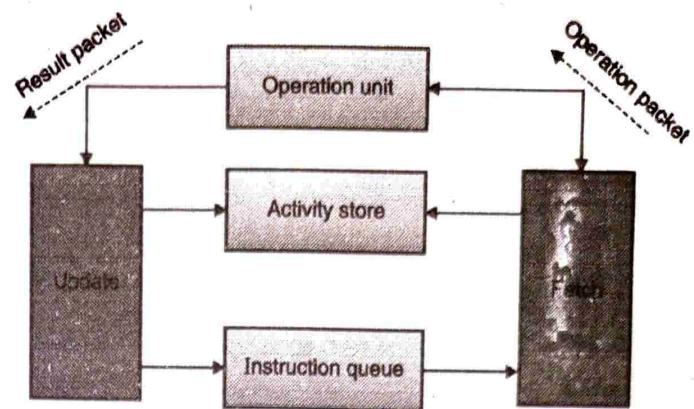


Fig. 1.12.14 : Conceptual working of dataflow machine

- The system has three basic units named as operation unit, update unit and fetch unit. It contains an activity store which holds the template for activity called as **activity template**.
- Each instruction is represented in terms of activity and each activity's unique address is stored in the instruction queue when an instruction becomes ready for the execution.
- The fetch and update units take care about the handling of instruction fetch and data access during the processing.
- The operation specified in the instruction is performed by the operation unit and the resultant value is stored in the destination field of the template.

1.12.5(C) Types of Dataflow Execution Model

- Traditionally dataflow model of computations are classified into two different categories: static and dynamic dataflow.

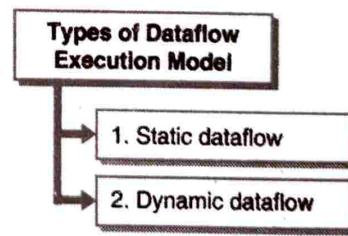


Fig. 1.12.15 : Types of Dataflow Execution Model

1. Static dataflow

- In this approach of dataflow computing at the most one instance of a node is allowed to be enabled for firing.
- The execution of a node is possible only when all input tokens required for that node are available and no token exists in that node's output arcs.



→ 2. Dynamic dataflow

- In this approach of dataflow executions several instances of a node can be activated at the same time during execution time.
- The different instances of a node are distinguished by the use of tags along with the tokens. In this model of computing a tag is associated with each and every token to identify the context for which that token is generated.
- A node is allowed to execute when all of its input arcs contain a set of tokens with identical tags.

1.12.5(D) Dataflow Machine Architecture

The computer architecture actually describes about the functional descriptions, organization and implementation of the computer system. In this section the principle along with the components of the dataflow machines are covered. This includes the processing element, communication among nodes and data structure used by the dataflow computers.

Processing element

- A dataflow machine includes several processing elements and they communicate with each other to accomplish the expected tasks from the machine. The functional description of one processing element is shown in Fig.1.12.16.
- The processing unit in a dataflow machine typically consists of different units and storage for tokens and nodes.

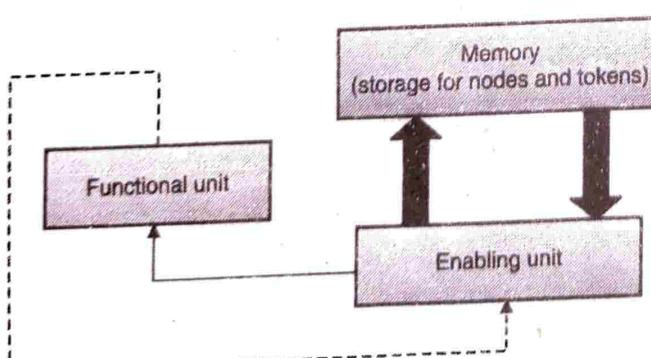


Fig. 1.12.16 : A processing element functional description

- The tokens move from node to node during the progress of activities when processing is carried out.
- The concurrent activities increases when a node produces more tokens than it can consume.

- This situation of concurrent behaviours is handled by the nodes able to consume more than one token.
- It is required to coordinate the activities of those nodes that require more than one input.
- The enabling unit is used in a processing element to implement some rules for enabling of those nodes requires more than one input. A memory is used in a processing element to provide the storage for the tokens.
- Tokens arrive in the enable unit one by one and after acceptance a token is stored in the memory. A token waits in the memory till the node for which it is addressed becomes enabled.
- A node only becomes enabled when all of its required inputs become available. We can understand this by the use of an example.
- Assume we have a node N1 and three input tokens are required for it. Also assume T1, T2 and T3 are three tokens required in three input arcs of the node N1.
- The enabling unit reads these tokens sequentially in the order of their arrival. In this way first token T1 arrives and it is stored in the memory unit.
- In continuation of this next token T2 is arrived and enabling unit found that all tokens of node N1 have not yet arrived, so again T2 is stored in the memory.
- Finally token T3 enters and enabling unit found that all tokens required by N1 have been arrived.
- In the next step enabling unit extracts tokens T1 and T2 from memory and form a packet which contains copy of the node N1 along with tokens T1, T2 and T3. This packet is sent to the functional unit.
- In this case such packets are considered as executable because it contains values of the input tokens, the operand code and a list containing destination addresses. The destination addresses represent the nodes to which output of the current node is transferred.
- The functional unit executes the packet and output values of this execution are combined with the destination addresses into tokens.

- These tokens are again sent back to the enabling unit. The enabling unit may store these tokens for the enabling of other nodes. This approach is termed as a circular pipeline because enabling stage and functional unit works concurrently.
- The template has space for the input values as well as the result of the previous node.
- The large space required for the input tokens in the template because of sharing of nodes between different instances of a graph.
- This creates efficiency problem and after certain stage becomes impractical in handling of storing tokens in the nodes themselves.
- To overcome from this issue for the handling of the tasks the enabling unit is divided into two separate stages namely the matching unit and the fetching unit.
- In the similar fashion two separate memory elements is managed for token storage and node storage. Fig. 1.12.17 shows this arrangement.

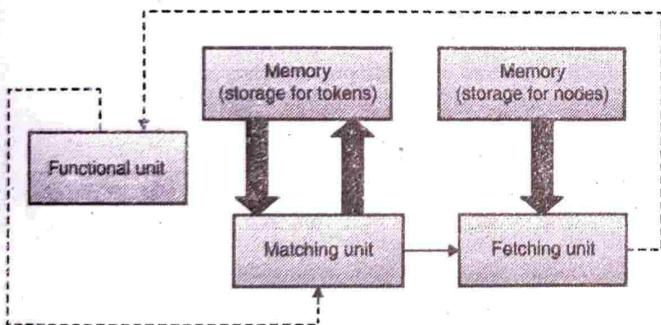


Fig. 1.12.17 : A processing element for a tagged token machine

Communication among nodes

- The different processing elements of a dataflow machine communicate with each other based of some topology. The different topologies like, tree, ring, n-cube and n X n switch can be used.
- In computer network different types of switching like circuit switching and packet switching are used. A dataflow machine can have direct communication or packet based communications among different processing elements.
- The direct communication is possible to handle between connected adjacent nodes of a graph.

- The dataflow machine which support direct communication uses the same processing elements for allocation of adjacent nodes in the graph. In direct communication architecture the tokens are delivered in the same order in which they arrive.
- Packet communication is based on asynchronously operating packet switching modules. The communication unit in packet communication allows handling of load distribution and parallelism.
- The packet switching module accepts a token and forward that token to other module for which it has been destined.

1.12.5(E) Types of Dataflow Machines

- In the general sense dataflow machines are powerful programmable parallel computers in which optimized hardware are included for fine - grained data - driven parallel computations.
 - As we have discussed about the parallel executions of processes in dataflow machines. The fine - grain indicates about these processes that they are similar sizes of machine code instructions usually created after compilation of programs for the conventional machines.
 - The availability of all the input operands are required for the activation of a process is the meaning associated with data - driven computing.
 - At the broad level dataflow machines are classified into two categories namely static dataflow and dynamic dataflow machines.
 - These machines are based upon the models proposed as static and dynamic models of computations.
- ##### 1. Static dataflow machine
- The static dataflow machines do not allow multiple instances of the same routines to be executed simultaneously. It uses the conventional memory system.
 - The static dataflow machine was proposed by Dennis and his research team at Massachusetts Institute of Technology.
 - Fig. 1.12.18 shows the block diagram of this class of machine.

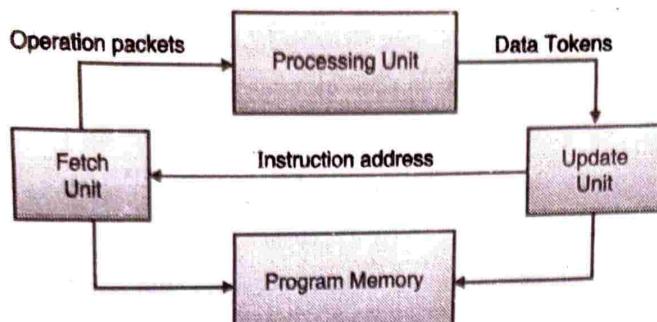


Fig. 1.12.18 : Static dataflow machine

- The program memory contains instruction templates as shown in Fig. 1.12.19. Instruction template represents the nodes in a dataflow graph.
- Instruction template contains an operation, slots for the operands, and destination addresses.
- It uses bits named as PBs is actually short for the presence bits. Presence bits are used to determine the availability of the operands.

Opcode	
PB	Operand 1
PB	Operand 2
Destination 1	
Destination 2	

Fig. 1.12.19 : Instruction template

The activities of the different units of the static machine are described here :

☞ The Update Unit

This unit is responsible for detecting the executability of instructions. This is also responsible for sending the address of the enabled instruction to the fetch unit.

☞ The Fetch Unit

This unit fetches and sends a complete operation packet containing the corresponding opcode, data, and destination list to the processing unit. The fetch unit clears the presence bits.

☞ The Processing Unit

This unit performs the operation, forms a result packets, and sends them to the update unit.

2. Dynamic Dataflow Machine

Proposed by Arvind at Massachusetts Institute of Technology and by Gurd and Watson at the University of Manchester-Tagged-Token Dataflow Architecture. Fig. 1.12.20 shows the block diagram of this class of machine.

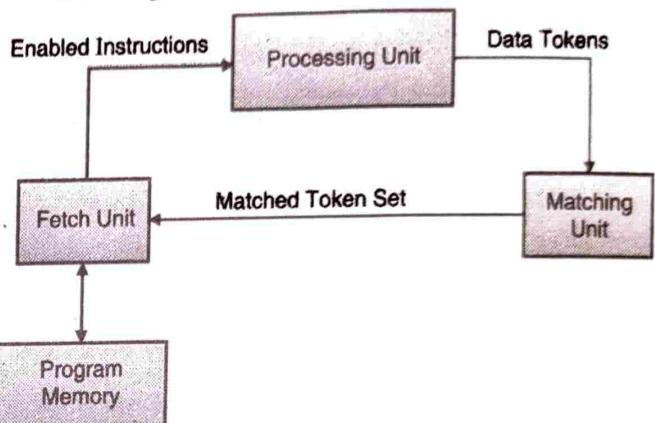


Fig. 1.12.20 : Dynamic dataflow machine

- In this machine the memory contains a pool of waiting tokens. The activities of the different units of the static machine are described here :
 - **The Matching Unit :** this unit receives tokens and brings together tokens with identical tags. If a match exists, the matched token set is passed to the fetch unit. If no match is found then the token will wait for the availability of the token to be matched.
 - **The Fetch Unit :** in this unit the tags of the token pair uniquely identify an instruction to be fetched from the program memory. The instruction together with the token pair forms the enabled instruction and is sent to the Processing Unit.
 - **The Processing Unit :** this unit is responsible for the execution of the enabled instructions and produces result tokens to be sent to the matching unit.

1.12.5(F) Drawbacks in Dataflow Computing

Some of the disadvantages associated with the dataflow computing are listed here :



- **Too fine-grained parallelism :** Incurs more overhead in the execution of an instruction cycle and shows poor performance in applications with low degree of parallelism.
- Inefficiency in representing and handling data structures for example, arrays of data.
- Need for large token stores to handle all operations waiting for execution
- Means to schedule hundreds or thousands of operations that are ready to execute in a limited amount of available hardware.
- The dataflow computing model has not been adopted in mainstream processor design.

1.12.6 Demand-driven Computation

- The traditional model of computations Von Neumann computation is based on program control nature of computing.
- When computer field became matured the main drawbacks of Von Neumann computations became visible and researchers came up with other forms of computations to deal with those deficiencies.
- The other models of computations are based on different principles as compare to Von Neumann model of computing.
- The innovations for new models were based on the goals of providing new execution ordering and different approaches for data manipulations.
- The most apparent models differ from the Von Neumann model of computations are the data – driven and demand – driven models.
- As we have discussed about the dataflow computations and it is based on the data driven model of computations.
- The data driven model of computation based upon the principle of executing the operations only when all the required operands become available.
- Demand driven computation is another model of computation in which the demand for an output triggers the operation that will actually responsible to generate it.
- This means in demand – driven computing the instructions are selected for executions when the value these instructions

are going to produce are required by another already selected instruction.

- The demand driven computation can be described in a better way by the use of an example. For example consider the following expression to be executed using demand driven approach:

$$\text{Result} = (A - B) * (A + B)$$

- During computation of this expression the **result** is first demanded and because of this the other multiplication terms $(A-B)$ and $(A+B)$ will in demand.
- Now the need arises to fetch values of A and B and then the subtraction and addition operations are computed. The outputs of these two operations are passed back and the multiplication is performed.
- Demand driven computation is also called as **reduction** in outermost order or simply reduction.
- This is called as reduction because in this computation the control coincides with reducing expressions following the outermost order of operations.
- In this example this is illustrated as first the multiplication is handled then subtraction and addition. In this way the expressions are reduced as replaced by their values and at the end the whole program is reduced and only the result remains.
- Here one has to note that the expressions can be reduced in the usual order or the innermost order. For example; subtraction and addition and then multiplication operation performed but when control proceeds in this order, it is called as dependence – driven not **demand - driven** computations.
- In terms of reduction used in demand driven computations, two different types are included : **string reduction** and **graph reduction**.

☞ **String reduction**

- In string reduction approach any common sub expressions will be evaluated separately because each operation gets a different copy of the value.



- This has an advantage of reduced complexity because of exclusion of addressing scheme but suffers from the inefficient use of resources.
- The Fig. 1.12.21 shows string reduction of the given expression

result = (A - B)*(A + B).

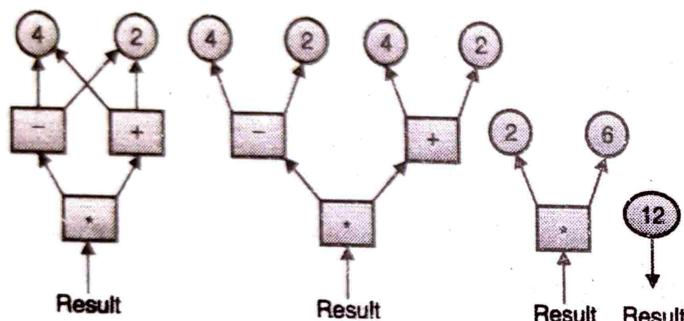


Fig. 1.12.21 : Evaluation of expression result = (A-B)*(A+B) using string reduction

- The process is depicted as the value of the result is demanded. This causes the replacement of identifier result with its value. In the next stage the identifiers A and B are replaced by copies of values.

- This process is termed as the reduction process. In the similar fashion the subtraction and addition are replaced with their corresponding values.
- At the final stage the multiplication is replaced by the value of the variable result.

Graph reduction

- In the graph reduction process instead of providing the different copy pointers are used for the values. The pointers are simply reversed for providing the values of the identifiers. The process for graph reduction is shown in Fig. 1.12.22.

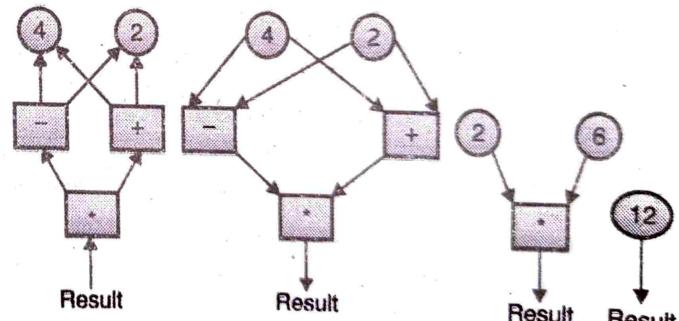


Fig. 1.12.22 : Evaluation of expression result = (A-B)*(A+B) using graph reduction

Parallel Programming

Syllabus Topics

Principles of Parallel Algorithm Design : Preliminaries, Decomposition Techniques, Characteristics of Tasks and Interactions, Mapping Techniques for Load Balancing, Methods for Containing Interaction Overheads, Parallel Algorithm Models, The Age of Parallel Processing, the Rise of GPU Computing, A Brief History of GPUs, Early GPU.

Syllabus Topic : Principles of Parallel Algorithm Design - Preliminaries

2.1 Principles of Parallel Algorithm Design

**Q. 2.1.1 Explain Principles of Parallel Algorithm Design.
(Refer sections 2.1, 2.1.1 and 2.1.2) (5 Marks)**

- An algorithm provides step by step solution of the given problem. An algorithm accepts inputs from the user and based upon the input after performing the defined computations provides the output.
- In the similar fashion particularly a parallel algorithm accepts inputs from the user and execute several instructions simultaneously on different processing units and all separate outputs produced from different units are combined to provide the overall output of the algorithm. In this chapter we will discuss several techniques required to deal with the design of parallel algorithms.

2.1.1 Preliminaries

- The basic steps in the design of parallel algorithm are :
- (i) Partitioning of overall computation into smaller computations and

(ii) Assignments of these smaller computations into different processors.

- These two key terminologies are explored in this section and some of the basic terminologies associated with the parallel algorithm design are also focused at significant levels.

2.1.2 Decomposition, Tasks, and Dependency Graphs

**Q. 2.1.2 Explain Decomposition, Tasks, and Dependency Graphs.
(Refer section 2.1.2) (7 Marks)**

1. Decomposition

- The computer system is used to solve a problem by performing some computations based on input data and provides output data for further activities.
- The overall computation can be partitioned into number of small size computations so that these computations execute in parallel.
- The decomposition deals with the approaches of partitioning the overall computation into sub parts.
- When a computation is divided into many small tasks, it is referred as fine - grained decomposition. On the other hand the course - grained decomposition contains a small number of large tasks.

*fine - grained
course - grained*



2. Tasks

- As we know that the operating system provides an environment for program developments and executions.
- In fact it is considered as an overall controller of the program. In programming the basic unit of computation is referred as a task and is controlled by an operating system.
- In the context of parallel programming the tasks are units of computation based upon that the overall computations is decomposed.
- The problem to be solved using the parallel programming approach is divided into arbitrary sized multiple tasks for simultaneous executions so that the computation time is minimized effectively.

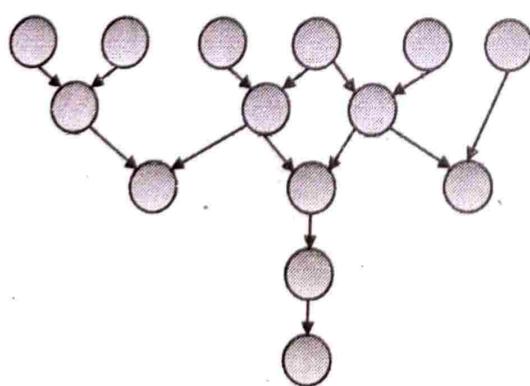
3. Task - dependency graph

- Is a directed acyclic graph. Typically a graph is a collection of nodes and edges, the task - dependency graph also contains nodes and edges.
- The nodes in the task - dependency graph represent tasks whereas edges between any two nodes represent dependency between them. For example, there is an edge exists between two nodes T1 and T2, if T2 must be executed after T1.

Ex. 2.1.1

Consider the task - dependency graph shown in Fig. P. 2.1.1 and find out the following :

- (i) Maximum degree of concurrency
- (ii) Critical path length
- (iii) Total amount of work
- (iv) Average degree of concurrency



- The parallel computations can be performed using dividing the overall computation into number of subtasks.
- In this case each subtask should contain a row of the matrix A and a column of the matrix B.
- The total number of subtasks included to perform the multiplication appears to be equal to N^2 according to the elements in resultant matrix C.
- The necessary computations of the basic subtasks can be performed when the required data sets are available.
- The required data sets are a row of the matrix A and all the columns of matrix B for basic subtasks.
- The simple solution in this case is duplicating the matrix B in all the considered subtasks for multiplications.
- This solution is unacceptable because of the extra memory requirements for extra data storage. The solution for this problem should have the data availability only required for the computations.
- The algorithm for the matrix multiplication with the solution of the problem mentioned here is an iterative procedure.
- In this procedure the number of iterations is equal to the number of subtasks. In this case at each iteration of the algorithm a row of matrix A and a column of matrix B are contained in each subtask.
- The subtasks containing rows and columns computes scalar products at each iteration and the corresponding elements of the matrix C are generated.
- After completing of all iteration computations the columns of matrix B must be transmitted so that subtasks should have new columns of the matrix B and new elements of the matrix C could be calculated.
- This transmission of columns among the subtasks must be executed in such a way that all the columns of matrix B should have appeared in each subtask sequentially.
- Fig. 2.1.1 shows the iterations of the matrix multiplication algorithm where four rows and four columns are assumed.
- In this scenario at the beginning of the computations each subtask contains i^{th} row of A matrix and j^{th} column of B.
- As a result the subtask i can compute the element C_{ij} of the result matrix C. Further each subtask transmits its column of matrix B to the following subtask in accordance with the ring structure. These actions should be repeated until all the iterations of the parallel algorithm are completed.

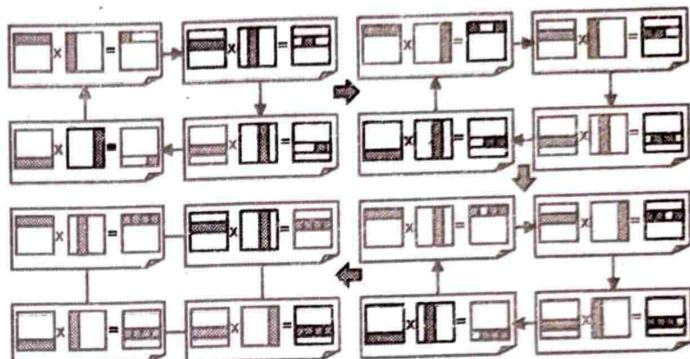


Fig. 2.1.1 : General scheme of data communications for the parallel matrix multiplication algorithm using block striped decomposition

2.1.3 Granularity, Concurrency and Task-Interaction

~~2.1.3(A) Granularity~~

- Granularity is the descriptions about a system in terms of how divisible it is. There are two types of granularity namely **fine - grained** and **course - grained**.
- Fine - grained system has a **high - granularity**. This means in fine - grained an object (or system) is divided into larger numbers of smaller parts.
- In the similar ways a course grained refers to the system divided into a smaller number of larger parts.
- For example, the use of grams to represent weight of an object is more granular than using kilograms to represent the weight of the same object.
- In general parallelism is categorized with instruction and data stream in parallel computing. **Granularity in parallel computing** is an additional way of categorizing parallel computation.
- Term granularity is used in parallel computing to refer about the division of overall task into number of subtasks.
- The **fine - grained** in parallel computing refers the division of a task into a large number of smaller subtasks.



- In the similar fashion course - grained describes the division of a task into larger number of longer subtasks. In parallel computing, technically a measure of the computational work is called as a grain. In fact a grain represents the ratio of computation work to communication work.

↓W → ↓C

How to measure granularity ?

- There are three relative values actually used to specify the granularity : **fine, medium, coarse**. In the parallel algorithm the granularity is generally used to describe about a parallel section of algorithm.
- There are three characteristics of the algorithm used for the determination of granularity. It includes the hardware used for the running of the algorithm.
- The first characteristic is the **structure of the problem**. The data parallel programming works in the form where specified operations are performed on many pieces of data.
- These operations are executed by different processing elements and communications amongst the PEs required.
- In association with granularity the task in data parallel computation is considered to have small granularity or **fine - grained**.
- On other cases like the executions of large subroutines independently with little communications with each other is considered as **coarse - grained**.
- The second characteristic is the size of the problem. This can be described using an example where 10 numbers are to be incremented. The number of processing elements in this case is assumed as 10 then the algorithm requires 1 clock cycle for computation.
- Later on assume the increase in the problem size and therefore 100 numbers are to be incremented. This way the task size has been increased and this is considered as a **coarser granularity**.
- The third characteristic to deal with the granularity in the parallel algorithm is the number of processors available. The number of processors is reduced for the problem size the task size increases and the granularity becomes coarser.

2.1.3(B) Concurrency

- The tendency for the events in real world to happen at the same time is called as concurrency.

- The concurrency is one of the natural phenomena in the real world because at a particular instance of time many things are happening simultaneously. The concurrency is required to deal with for designing the software system for real world problems.
- There are generally two important aspects when dealing with concurrency for real world problems : ability of dealing and responding external events occur in random order, and required to respond these events in some minimum required interval.
- It is really simple to handle when different activities happen in a truly parallel fashion by creating separate programs to deal with each activity.
- The real challenge occurs when required to design concurrent system in which interactions among the concurrent activities need to be co-ordinated.
- The concept termed as degree of concurrency is related to granularity is used in the algorithm design.
- The maximum degree of concurrency is the maximum number of concurrent tasks can be executed simultaneously in a program at any given instance of time.
- Another term related to concurrency is the average degree of concurrency which is used to refer the average number of tasks can be processed in parallel during the execution of the program.

2.1.3(C) Task-Interaction

- The parallel executions of tasks provide efficiency in terms of speedup because tasks in a parallel algorithm simultaneously execute in different processors.
- There are basically three factors granularity, concurrency and interaction are responsible to affects the speedup of the parallelization.
- The interaction provides the communication among the tasks running in different processors.
- The interactions among the tasks included for providing the solution of a problem is required because the tasks share input, output, or intermediate data.
- The task - dependency graph shows the dependencies because output of one task can be the input of another task. This means the second task cannot proceed unless the output of the first task becomes available.



2.1.4 Processes and Mapping

2.1.4(A) Processes

- A problem to be solved using the parallel programming approach is considered to be divided into number of parts. Each part can be performed independently and is called as a task.
- In the context of this discussion a process is an entity responsible for performing the assigned task.
- This process is an abstract idea about doing the computations based upon the steps included in a task and input data available for it for producing the output.
- The environment in which a problem is being solved by means of parallel computations usually has several running process simultaneously.
- These processes communicate with each other for the need of synchronization and exchanging of information.
- The programs written for such platforms are called as parallel programs and one parallel program can have several processes for handling different tasks simultaneously.
- The Fig. 2.1.2 describes a problem divided into several tasks and assigned to separate processors for simultaneous executions.

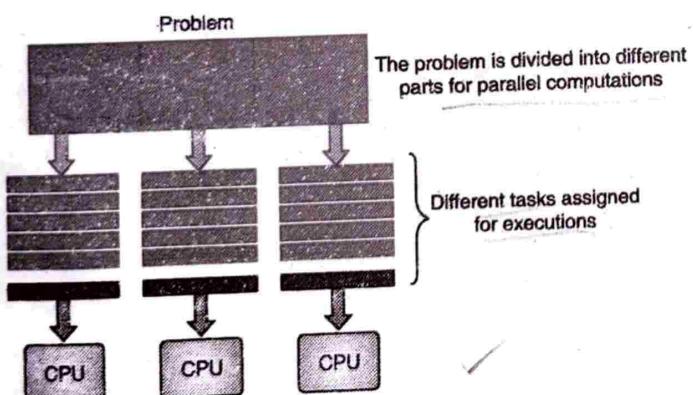


Fig. 2.1.2 : Parallel execution of tasks

2.1.4(B) Mapping

- In parallel program design it is required to specify where each task is to execute.
- As we have discussed in the previous section that a process performs a task. In this context the approach used for assignment of tasks to processes is called mapping.

- In fact mapping deals with two goals and these are referred as maximization of processor utilization and minimization of communication costs amongst processes.

2.1.5 Processes Versus Processors

- A program in execution is called as a process and in the design of parallel algorithms it is considered as the entity responsible for performing tasks.
- The computations defined to be performed in the tasks are executed by the physical unit called as processor. In this chapter the processes are used to describe parallel algorithms and programs.
- In the field of parallel computing it is assumed that the number of processes depends upon the total number of processors exist and most of the time it is considered as each process executes in a separate processor.
- In the parallel algorithm design if the algorithm is very complex and requires to have many processes so it is assumed to have running in the form of pseudo parallel processing so that the availability of processors should not resist in design.

Syllabus Topic : Decomposition Techniques

2.2 Decomposition Techniques

Q. 2.2.1 Write short note on Data-decomposition and Recursive decomposition.
(Refer sections 2.2.1 and 2.2.2) (7 Marks)

- The problem required much more time for computations are possible to divide multiple parts and each and every part is the implementation of a particular task.
- The task - dependency graph used to define a set of tasks and these tasks are the candidate for concurrent executions.
- There are various techniques commonly used for decomposition and in this section some of the techniques are covered.

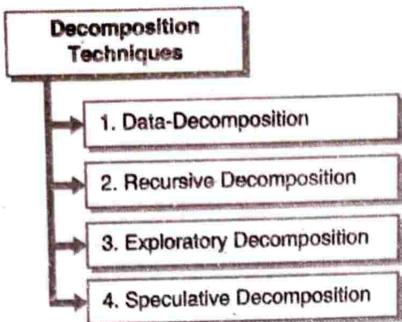


Fig. 2.2.1 : Decomposition techniques

2.2.1 Data-Decomposition

- The large data sets involved in the problems can be divided into smaller parts and processed independently by several computers concurrently.
- The computations of partitioned data on computers are usually used for some kinds of analysis.
- The problem is classified as an embarrassingly parallel if the subset of the data can be analyzed independent of the rest of the data. The data - decomposition is a common technique used for concurrent processing of the data.
- There are basically two steps used in this technique.
- In the first step the overall input data required for performing the defined computation is divided into multiple parts.
- The second step is based upon the division of overall computation into number of tasks handled on the portioned data.
- The actual operations implemented on these tasks are similar but the data upon which these operations work are different.

Example : Matrix multiplication

Consider the problem of multiplying two $n \times n$ matrices A and B to yield matrix C. The output matrix C can be partitioned into four tasks as given below. Here each task computes one element of result matrix.

$$\begin{array}{l}
 \text{Matrix A} \rightarrow \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] \quad \text{Matrix B} \rightarrow \left[\begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] \\
 \\
 \text{Matrix C} \rightarrow \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] \times \left[\begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] \\
 \\
 \text{Matrix C} \rightarrow \left[\begin{array}{cc} c_{11} & c_{12} \\ c_{21} & c_{22} \end{array} \right] \\
 \\
 \left[\begin{array}{cc} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{array} \right]
 \end{array}$$

2.2.2 Recursive Decomposition

- As we know that the Divide and conquer is one of the strategy of solving the computational problems.
- The divide and conquer strategy is based on two ideas : first approach says to solve the problem directly if it is trivial and in the second approach, decompose the problem into smaller parts if it cannot be solved as it is and solve the smaller parts.
- The recursive decomposition is based on providing concurrency in problems that can be handled in divide and conquer strategy.
- The problem to be solved using recursion is divided into multiple sub problems provided that each of these sub problems is independent.
- Each sub problem can be solved individually by dividing further into other sub problems and solved using recursion.
- In programming a function or procedure calls it-self is called as recursion. For example in the following program the function demo is a recursive procedure.

```

main()
{
    int i;
    i = 10;
    demo(i);
}

demo(int count)
{
    count--;
    printf("The value of the count is %d\n",count);
    if (count > 0)
        demo(count);
    printf("The count is %d\n",count);
}
  
```

Example : Merge sort

- The example of merge sort as shown below is implemented in sequential and parallel version thereafter. In this example the array is first divided into two parts then sorted these two parts recursively. The sorted parts are finally merges to produce the final output. The overall computations are organized here in the binary tree.



The parent process provides array to each process for sorting of elements. The process divides the array into two halves and sends to the children. The children perform their part of computations and send the sorted array back to the parent. The merging of these elements is performed by the parent and sends the array back up in the tree.

Merge Sort : Pseudo code for sorting of an array using merge sort sequentially

```
void mergeSort(int* a, int first, int last, int* aux)
{
    if (last <= first)
        return;
    int mid = (first+last)/2;
    mergeSort(a, first, mid, aux);
    mergeSort(a, mid+1, last, aux);
    mergeArrays(a, first, mid, a, mid+1, last, aux, first,
    last);
    for (int i=first; i<=last; i++)
        a[i] = aux[i];
}

void mergeArrays(int* a, int afirst, int alast, int* b, int bfirst,
intblast, int* c, int cfirrst, int clast)
{
    int i=afirst, j=bfirst, k=cfirrst;
    while (i<=alast&& j<=blast)
    {
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }
    while (i<=alast)
        c[k++] = a[i++];
    while (j<=blast)
        c[k++] = b[j++];
}
```

Merge Sort : Pseudo code for sorting of an array using merge sort in parallel

```
void parallel_mergeSort()
{
    if (proc_id> 0)
    {
```

```
        Recv(size, parent);
        Recv(a, size, parent);
    }
    mid = size/2;
    if (both children)
    {
        Send(mid, child1);
        Send(size-mid, child2);
        Send(a, mid, child1);
        Send(a+mid, size-mid, child2);
        Recv(a, mid, child1);
        Recv(a+mid, size-mid, child2);
        mergeArrays(a, 0, mid, a, mid+1, size, aux, 0, size);
        // declare aux local
        for (int i=first; i<=last; i++)
            a[i] = aux[i];
    }
    else
        mergeSort(a, 0, size);
    if (proc_id> 0)
        Send(a, size, parent);
}
```

2.2.3 Exploratory Decomposition

- There are situations where the problem decomposition goes hand in hand with its executions.
- Such problems typically involve the exploration or search of a state space of solutions. The search space of the problem is divided into smaller parts and each smaller part is searched concurrently till the point at which the expected solution is found.

Example of exploratory decomposition

Consider a state space searching problem such as finding a solution to a puzzle problem. The steps of solving such problems using the exploratory decomposition are as follows :

- The computations required for the decomposition can be divided into multiple tasks where each task is searching for a different portion of the search space.



1 2 3 4	1 2 3 4	1 2 3 4	1 2 3 4
5 6 7 8	5 6 7 8	5 6 7 8	5 6 7 8
9 10 7 11	9 10 7 11	9 10 11 12	9 10 11 12
13 14 15 12	13 14 15 12	13 14 15 12	13 14 15 12

Fig. 2.2.2

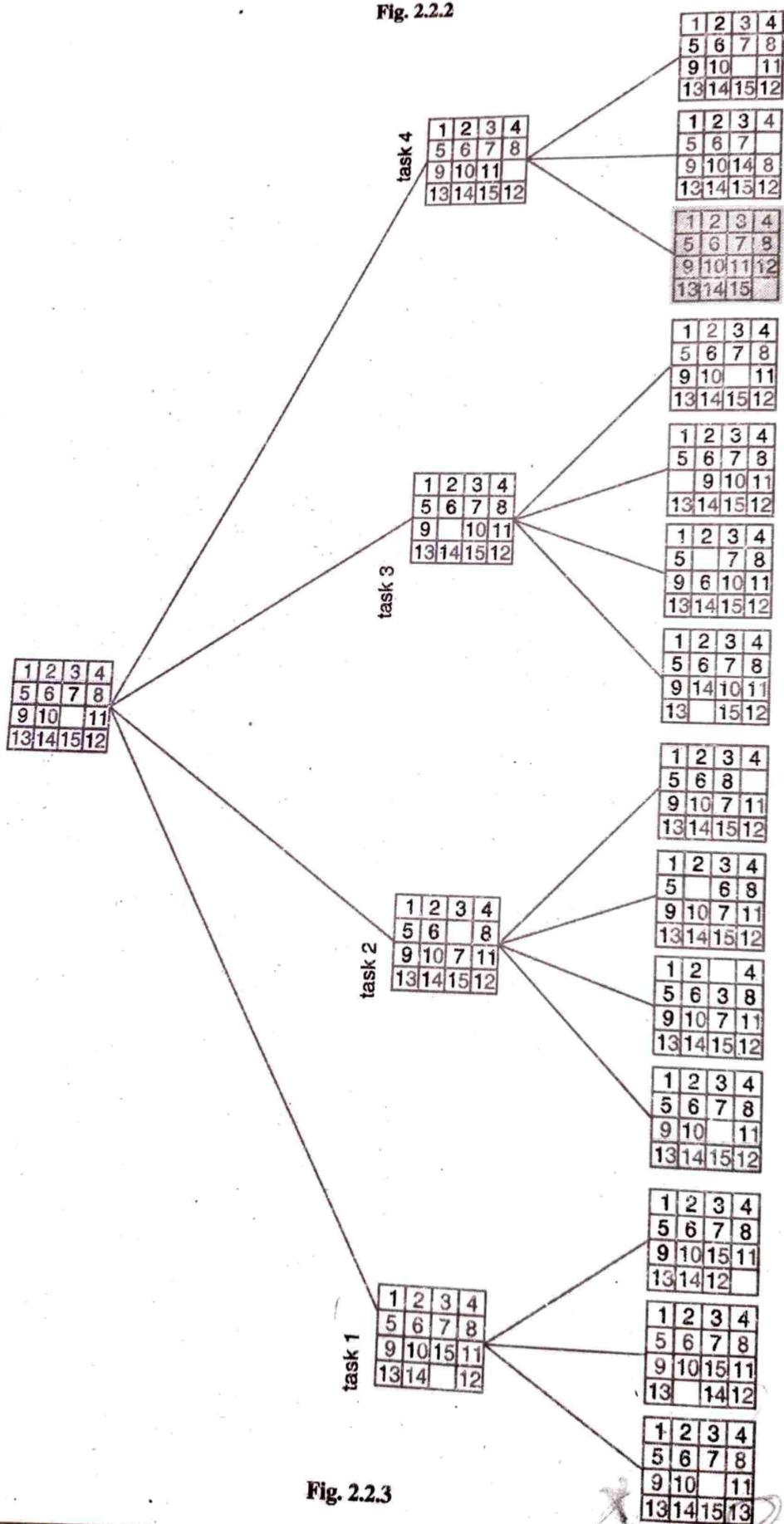


Fig. 2.2.3



- The task of finding the shortest path from initial to final configuration now translates to finding a path from one of these newly generated nodes to final configuration.
- The 15 puzzle problem is typically solved using tree search techniques. Starting from initial configuration, all possible successors are generated.
- In the 15 puzzle problem there are 15 tiles numbered 1 through 15 are arranged in a 4×4 grid as shown in Fig. 2.2.2. One tile is left blank so that moves can be made.
- The four possible moves here are represented as moves up, down, left and right. The initial as well as the final configurations are specified.
- This problem is characterized by the objective of determining any sequence of moves or a shortest sequence of moves.
- The solution of the problem must be searched from an arbitrary state.
- The solution of the 15 puzzle problem is provided using the tree - search techniques. Once we start from the initial configuration all the configurations possible as successor configurations are generated.
- This is handled using two possibilities: one says about to occupy empty slot by any one of the neighbour present, and second is to find out a path from one of the new configurations to the final one.

A state space graph

- Fig. 2.2.3 shows the configuration space generated by the tree search. As we know that any graph is a collection of nodes and edges the state space graph also contains nodes and edges.
- In this graph the nodes are used to represent configurations and edges represent connections of configurations. Here every edge of the graph connects configurations that can be reached from one another by a single move of a tile.

2.2.4 Speculative Decomposition

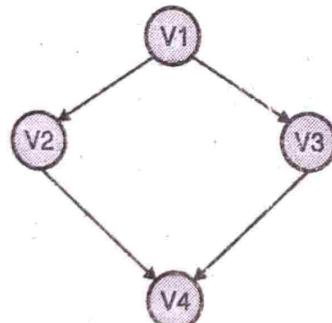
- The speculation decomposition technique is used in a situation when a program has many options to take in terms of branches based upon the outputs of other parts that preceded it.

- The situation can be described in terms of specific task, consider a task T1 involves in doing the computation C1 and going to produce the output O1.
- The next computation to be performed may be decided based upon the output of task T1. In the similar fashion other tasks can start executions concurrently in the next stage.
- The situation can be better compared with the switch statement in C programming.
- The switch statement works based upon the value of the expression on which it is based and only the corresponding case statement executes. Some or all of the cases execute in advance in speculative decomposition.
- The situation in which value of the expression is known then the results from the computation to be executed in that case will be kept.
- The anticipation about the possible computations causes the performance gain in the statement executions. For example consider the following switch statement in sequential and parallel versions.

Sequential	Parallel version
<pre>compute expr; switch (expr) { case 1 : compute-e1; break; case 2 : compute-e2; break; case 3 : compute-e3; break; }</pre>	<pre>Slave(i) { compute ei; Wait(request); if (request) Send(ei,0); } Master() { compute expr; switch (expr) { case i: Send(request, 1); Receive(a1,i); ... } }</pre>

Example : Topological sorting

- In a directed acyclic graph topological sorting is used to provide an ordering of the vertices. For example consider two vertices U and V of a graph G.
- In this graph when a path exists between U and V, then V appears after u in the ordering.
- It is a requirement that the graph should have acyclic property, otherwise for an edge represented as (U, V) there would be a path from U to V and also from V to U, and therefore the ordering cannot be obtained.
- Now let's assume there are a number of tasks we need to be performed in which some of the tasks depend on the others and it is possible to do only one at one.
- These tasks can be organized in the dependency graph. Here one must be able to find out an ordering of the tasks based on the dependencies. Consider Fig. 2.2.4 for understanding of topological sorting.



The legal orderings are →
V1, V2, V3, V4
and
V1, V3, V2, V4

Fig. 2.2.4

Steps of the topological sorting algorithm

- Step I :** Initially compute the in - degrees of all the vertices in the graph.
- Step II :** Find U as a vertex with degree 0 and store it in the list for ordering.
- Step III :** At this stage if no such vertex is detected then there is a cycle found and the algorithm stops
- Step IV :** Remove the vertex U and all the edges (U, V) it belongs from the graph
- Step V :** At this step need to update in - degrees of the vertices remains
- Step VI :** Repeat step 2 through 4 till the vertices present for processing

- In the topological sorting algorithm we can get the idea about the next vertex in the order before the next trace for the vertex is actually performed.

Example based on Topological sorting

Ex. 2.2.1

Now we will discuss another example to understand topological sorting. Consider the given directed graph $G = (V, E)$, find a linear ordering of vertices such that : for all edges (v, w) in E , v precedes w in the ordering.

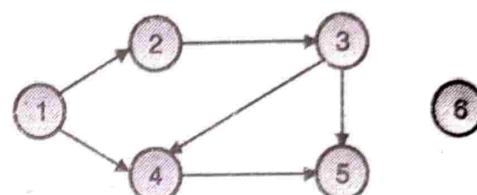


Fig. P. 2.2.1

Soln. :

For the topological sorting any linear ordering in which all the arrows go to the right is a valid solution.

This way the linear ordering shown in Fig. P. 2.2.1(a) is a valid solution.

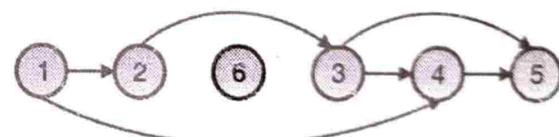


Fig. P. 2.2.1(a) : Valid topological sorted graph

The graph shown in Fig. P. 2.2.1(b) is not a valid topological sorted order.

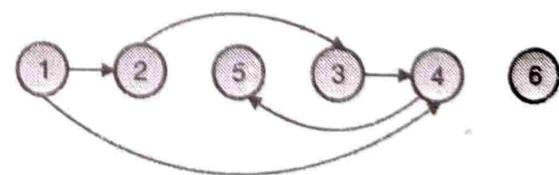


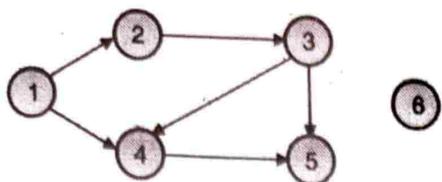
Fig. P. 2.2.1(b) : Invalid topological sorted graph

Topological sort algorithm : The steps of the topological sorting algorithm are given below.

- Step 1 :** Identify vertices that have no incoming edges (select one vertex)
- Step 2 :** Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.
- Step 3 :** Repeat Steps 1 and Step 2 until graph is empty.

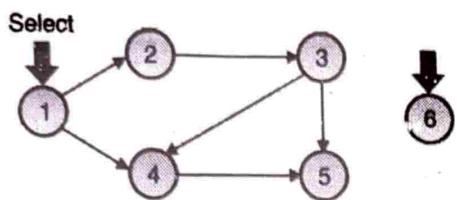


Now the steps of the algorithm are performed on the given graph.

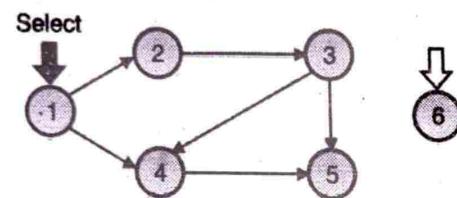


Perform Step 1 : Identify vertices that have no incoming edges.

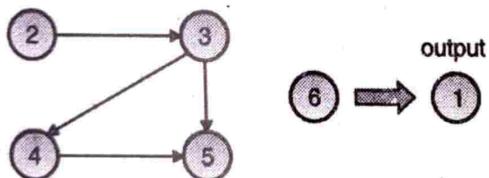
This means select vertices with in-degree zero.



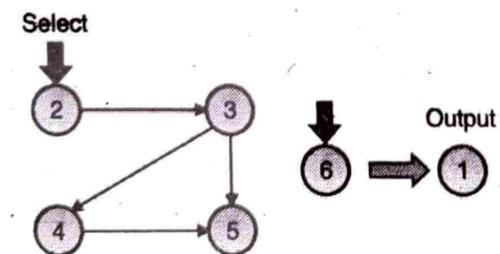
Perform Step 1 : Identify vertices that have no incoming edges. (Select one vertex)



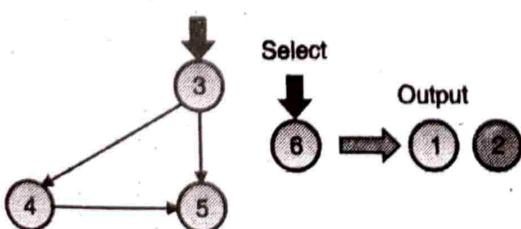
Perform Step 2 : Delete this vertex of in-degree 0 and all its outgoing edges from the graph. Place it in the output.



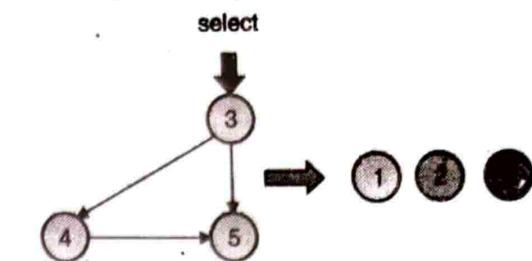
Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty



Repeat Steps 1 and Step 2 until graph is empty

Final Result



Finally the result of the topological sorting is shown here.

Syllabus Topic : Characteristics of Tasks and Interactions

2.3 Characteristics of Tasks and Interactions

- The different decomposition techniques are used to divide the overall problem into several concurrent tasks so that they can be executed in parallel.
- The further step of the parallel algorithm design deals with the mapping of these tasks into available processes.
- The decomposition approach provides the possibilities of writing good parallel programs to be performed in efficient time.
- The communications amongst tasks play very crucial roles while mapping the tasks into processes for parallel executions.
- This part describes properties of tasks and inter-task communications among them for good mapping for better performance oriented parallel algorithm designs.

2.3.1 Tasks Characteristics

Q. 2.3.1 What are the Characteristics of Tasks?

(Refer section 2.3.1)

(5 Marks)

- There are some key characteristics used to influence the choice of mapping of tasks onto available processes and at the same time the performance of the parallel algorithm is also maintained.

- The four basic characteristics are Generation of task, Size of tasks, Knowledge of Task Sizes and Data size associated with Tasks.

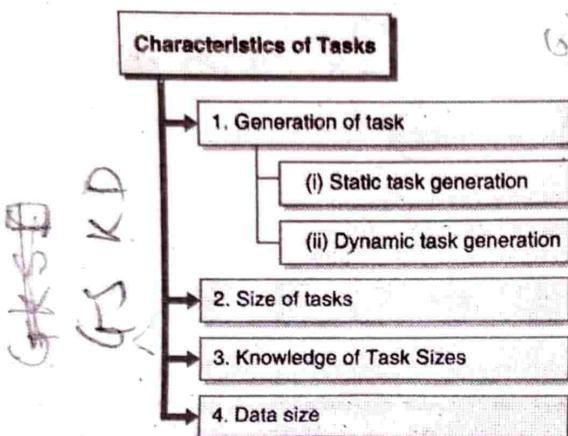


Fig. 2.3.1 : Characteristics of Tasks

→ 1. Generation of task

As we know an algorithm consists of tasks and in a parallel algorithm these tasks are considered to be executed in parallel. The tasks included in a parallel algorithm are possible to generate either on a prior basis termed as static or created whenever required termed as dynamically.

→ (i) Static task generation

- The approach in which the work(s) to be performed by an algorithm is known in advance on prior basis is called as static task generation.
- The tasks are defined before starting the execution of the algorithm and the specified order is to be followed by the algorithm in execution.
- For example consider the matrix multiplication algorithm where overall matrices divided into different chunks and distributed to various processors in a parallel machine. In each processor, the task for matrix multiplication executes on different sets of data considered for operations.

→ (ii) Dynamic task generation

- The tasks to be performed are created dynamically based upon the decomposition of the data in certain situations. In these cases the tasks to be performed are not available before algorithm executions.
- The recursive and exploratory decomposition techniques are considered as examples of dynamic task generation.

→ 2. Size of tasks

- Every task takes some amount of time for its completion. The size of a task is represented by the time required for its completion.
- There are two different categories in which programming tasks can be divided : uniform and non-uniform.
- The tasks are required to map into available processes so that it can be computed and it depends upon the type of the tasks.
- There are different mapping schemes and uniform tasks takes almost same amounts of time in mapping. In the case of non uniform tasks, the mapping times required varies in different schemes.

→ 3. Knowledge of Task Sizes

- The task size knowledge is used for the choice of mapping scheme. The tasks are mapped into processes and the prior knowledge of the tasks used in mapping.
- As an example the knowledge of the computation time required for each task is used to apply different decomposition techniques.
- In the similar fashion in some of the examples where dynamic decision are required during processing the prior knowledge of the task size is not known.

→ 4. Data size

- The data size is one more crucial property associated with the task. The required data for performing a task should be made available when mapping it into processes.
- The overheads associated with the data movement can be reduced when the size of the data as well as its memory location are available.

2.3.2 Characteristics of Inter-Task Interactions

- As we know a parallel algorithm contains many tasks and these tasks may have communications with each other to share some information.
- The exchange of information amongst tasks is done by the use of mechanism called as inter - tasks communications.
- The interactions amongst the tasks depend upon the types of interactions in different algorithms.



- The different programming paradigms and mapping schemes may support different natures of interaction schemes.
- This indicates that some of the interactions are suited to some techniques and other works most efficient with other techniques.

2.3.3 Static Versus Dynamic Pattern

There are two classes of interactions amongst the concurrent tasks based upon the types of pattern used in interactions. These patterns are static and dynamic patterns.

1. Static Interaction pattern

- The static pattern specifies that the times at which interaction among tasks occur is predetermined. The set of tasks to have interactions with each other is also known before the execution of algorithms in static pattern.
- The message - passing paradigm like MPI(Message Passing Interface) and PVM (Parallel Virtual Machine) can be easily used to create applications with static interaction of tasks.
- The message - passing approach provides the specifications for having association of sender and receiver tasks in message exchange.

2. Dynamic interaction pattern

- The interaction times of the tasks to be interacted as well as a set of tasks to have interactions decided at the algorithm execution time only in dynamic interaction pattern.
- It is always a daunting task to program dynamic interaction of tasks because of unpredictable nature of dynamic communications.
- The synchronization between a sender and receiver is one of the challenging issues. This may create problem if not handled properly particularly in a situation where both sender and receiver communicate at the same time.

2.3.4 Regular Versus Irregular

- The interactions are also placed in different classes based upon their spatial structure.
- The interactions are kept in regular and irregular classes in consideration with spatial structures.

- The interaction pattern is considered as **regular** if it is possible to exploit some of the structure of the interaction pattern for efficient implementation.
- In the case where no such pattern exist called as **irregular** pattern. Like dynamic pattern of interactions the irregular pattern is also difficult to implement particularly in message - passing environment.

2.3.5 Read-Only Versus Read-Write

- Inter-task communication commonly called as task-interaction is used for data sharing among tasks. The choice of mapping of tasks into processes is affected by the type of sharing data between tasks.
- There are two different categories of data sharing can be considered in task - interactions : **read-only** and **read-write**.

1. Read - only interactions

- This is used in a situation where the data to be shared among concurrent tasks is required for read only purpose.
- The interactions among tasks in such situations are read - only interactions.
- Consider an array A with 10000 elements and these elements are added in parallel. In this example A is divided into multiple parts and the elements in each part are added concurrently.
- Here the input array is required for reading purpose only. In this example the tasks require to read array A for input purpose only.

2. Read - write interactions

This is required in a situation where many tasks have to perform read and write operations on some shared data.

2.3.6 One-way Versus Two-way

- The communications among tasks can also be categorized in terms of one way and two way communications. Suppose there is a pair of task in which one is the producer and another is the consumer.
- The interaction between producer and consumer task is termed as two - way because the data required by the consumer task is supplied by the producer task.



- The interaction between two tasks is typically called as **two - way interaction** when data required by one task is supplied by another task explicitly.
- The interaction is termed as **one - way interaction** when one task in a pair of tasks starts the communication and completes the interaction without interrupting other task.
- The read - only interactions in any form is considered as one - way whereas all read - write interactions can be considered as one - way or two-way depending upon the context.

Syllabus Topic : Mapping Techniques for Load Balancing

2.4 Mapping Techniques for Load Balancing

- The mapping techniques are used to map tasks into processes so that parallel executions can be performed.
- The overall computation associated in a problem to be solved divided into number of tasks. These tasks are mapped onto processes with the goal of completing executions in the minimum amount of time.
- This goal can only be achieved when the overheads associated with the parallel executions of tasks are minimized.
- There are two factors involved with the overheads in the parallel executions of tasks. The first factor due to which overhead occurs is the total time required in inter - process interactions during communications.
- One more factor responsible for overhead is the amount of time some of the processes wait without doing any significant works typically called as idleness. These factors provide the objectives for considering mapping as a good mapping.
- The mapping is assumed as a good mapping when the computations and interactions are well balanced at each stage of the parallel algorithm.

2.4.1 Mapping Techniques

Q. 2.4.1 What is Mapping Techniques?

(Refer section 2.4.1)

(4 Marks)

There are two different categories of mapping techniques are used in parallel algorithms : **static** and **dynamic**. The suitability of these two mapping techniques is decided on the basis of tasks characteristics and how they interact with each other.

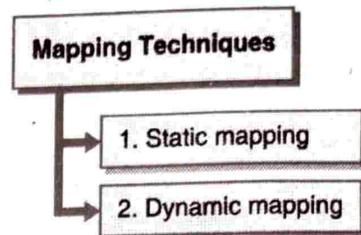


Fig. 2.4.1 : Mapping Techniques

→ 1. Static mapping

- In this technique, the mapping of tasks onto processes is performed before execution of algorithms. This indicates the tasks are distributed among available processes prior to execution of algorithms.
- The factors like knowledge of the task size, data size and inter - task interaction characteristics are used in deciding about the mapping technique to be used.
- For example, these two techniques static and dynamic mapping can be used when tasks are generated statically but which mapping will perform better is decided on the basis of task factors and parallel programming paradigm.

→ 2. Dynamic mapping

- In this technique, the mapping of tasks onto processes is performed during the execution of algorithms.
- This indicates the tasks are distributed among available processes when algorithm actually executes.
- There are basic situations where dynamic mapping is applied. The first and straightforward case is if tasks are generated dynamically then this mapping technique is used.
- Dynamic technique is applied in a situation where the large amount of data is associated with tasks. In this situation dynamic mapping moves data among available processes.



2.4.2 Schemes for Static Mapping

- The static mapping is usually used along with some partitioning techniques naturally. There are three different schemes suggested for static mapping :
- **Mapping Based on Data Partitioning, Task Graph Partitioning and Hybrid Strategies.**

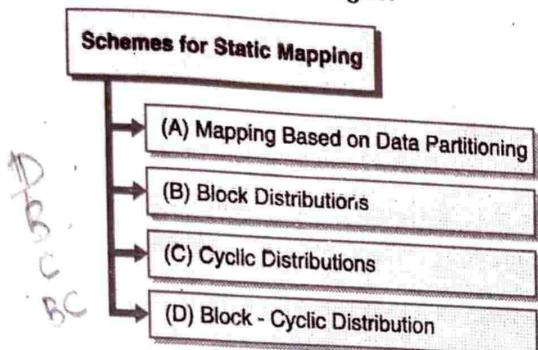


Fig. 2.4.2 : Schemes for static mapping

2.4.2(A) Mapping Based on Data Partitioning

- The mapping is used for associating tasks into processes. One of the popular rules used in High - Performance Computing called as owner computes rule and according to this rule mapping of tasks onto processes is similar to mapping of data to processes.
- The data associated with the algorithms are commonly represented by arrays and graphs. In this section we will discuss mapping of tasks onto processes on the basis of arrays and graphs.

Array Distribution Schemes

- The purpose of this scheme is to distribute array elements across local memories of a parallel computer.
- The direct benefit of this distribution is that the elements can be accessed in parallel for processing.
- There are three standard distributions of the dense arrays : **block, cyclic and block cyclic.**
- The rule commonly used and implemented in High Performance Fortran is named as Owner Computes rule.
- The owner computes rule is applied in a decomposition techniques based on the partitioning of data.
- According to this rule mapping of tasks is similar to mapping of required data onto the processes. At this stage we will

discuss commonly used techniques of distributing arrays among processes.

- The owner computes rule is most often used in High - Performance Fortran compilation systems.
- According to this rule the required calculation will be performed by the processor that owns the left hand side element. This is described using an example loop shown below in FORTRAN language.

```

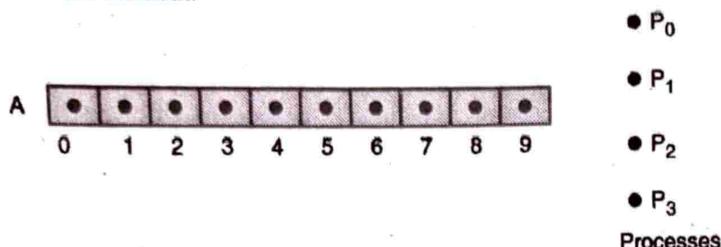
DO i = 1,n
  a(i-3) = b(i*2)/c(i+j)-a(i**i)
END DO
  
```

- In this example the expression $a(i-3)$ is the left hand side of the assignment statement. This expression is owned by a processor in a multiprocessor system.
- This induces that the assignment will be performed by the processor owns $a(i-3)$ expression. Therefore all the components present at the right hand side should be made available to the processor for performing computation without any hassle.

2.4.2(B) Block Distributions

- The block distribution deals with the homogeneous distribution of computational load over a regular data structure such as a Cartesian grid.
- According to block distribution blocks of size S of the vector is assigned to processes for mapping of elements during processing.
- This distribution is based on the approach of assigning contiguous portions of the array to different processes.
- In this distribution technique an array with m - dimension is distributed among a set of processes in the fashion such as each process gets contiguous block of array elements.
- The block distribution of array elements is most of the time suitable when the computation performs on the nearby elements of an array called as locality of interaction.
- The block distribution is described in Fig. 2.4.3 with pictorial representation.
- In Fig. 2.4.3 an array named as A of size 10 is considered. Also assume four processes P_0, P_1, P_2 and P_3 execute in parallel to compute on array elements.

- In the first case the array elements are distributed to four processes. Here, the block size is assumed three.
- This means processes P_0 , P_1 and P_2 will get three elements each of the array A whereas process P_3 will get remaining one element.



• Distribute array elements into processes with block size as three :



Fig. 2.4.3

- In the first case, the array elements are distributed to four processes. Here, in Fig. 2.4.4, the block size is assumed four.
- This means processes P_0 , P_1 will get four elements each of the array A whereas process P_2 will get two elements and all elements have been distributed. The last process P_3 will not get any element.

Distribute array elements into process with block size as four :

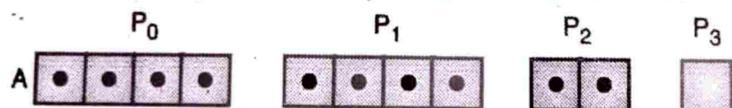


Fig. 2.4.4

- In this scheme, the $A[i]$ element is mapped to the processor $[i/b]$ if distribution is BLOCK(b).
- Now, we will consider the distribution of an array A of M elements over P processes. The mapping of the global index m where it lies between 0 to M represented as $(0 \leq m < M)$ is handled. The global index m of the data object will be mapped to an index pair (P, I) .
- In the current discussion p is used to specify processes to which the elements are mapped.
- The value of p is represented by $0 \leq p < P$ with indication of its range. The value of variable i indicates the location of the element in the array. The mapping represented as $m \rightarrow (p, i)$ for block distribution is defined as:

$$m \rightarrow (\text{floor}(m/L), m \bmod L)$$

where, L is defined as $L = \text{ceiling}(M/P)$.

- The block distribution is described using an example where the parameters p, m etc. are used. Let's assume we have M = 23 data elements to be distributed over three processes ($P = 3$). Also consider the block size S = 8.

m	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22
p	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	2 2 2 2 2 2 2 2
I	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6
B	0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1	2 2 2 2 2 2 2 2

- It is clear in the example that uneven distribution occurs. Here because of uneven distribution the last block becomes smaller than the others. A global block number B is also shown here for indication of block distribution.

☞ Distribution of $M \times N$ matrix using block distribution

- In block distribution of any array each process gets a contiguous part of the array data. An $M \times N$ matrix is distributed using two different approaches : **row - wise** and **column - wise distribution**.
- Each process receives N/P rows of a matrix; here P indicates the number of processes used for mapping of tasks when elements are distributed along with row - wise distribution approach.
- On the other hand each process gets M/P columns of the matrix when column - wise distribution approach is applied. The block distribution approach is suitable for some computations like matrix - vector multiplication.

☞ 2-D distribution on processes

- We have to assume a process grid for the distribution of array elements on processes.
- In general one has to assume the size of the process grid, assume now as $P_1 \times P_2$. The number of processes for which this grid can be used is $P = P_1 * P_2$.
- For example consider the distribution of $N \times N$ matrix. The matrix can be divided into $N/P_1 \times N/P_2$ sub matrices. Fig. 2.4.5 shows the 4×4 process grid.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

Fig. 2.4.5



- This 4×4 process grid is used to execute processes in parallel with distribution of data in the form of the grid.
- Once the data required by a process is provided that process can start execution. In this way all processes operate on their part of the data concurrently.

2.4.2(C) Cyclic Distributions

- In the situation where computational load is distributed in inhomogeneous fashion, cyclic distribution is used. The cyclic distribution is used in such case to improve load balancing.
- The consecutive entries of the global vector are used to assign in successive processes.
- The cyclic distribution of data among processes does not use any concept of block numbers. The mapping $m \rightarrow (p, i)$ is defined as shown below for cyclic distribution :

$$m \rightarrow (m \bmod P, \text{floor}(m/P))$$

- This scheme of distribution can be used to reduce the problems like the load - imbalance and idling.
- The load imbalances occur when the amount of work is different for different part of a matrix. This can be avoided by using the cyclic or block cyclic distributions.
- The cyclic distribution is described in Fig. 2.4.6 using an example where the parameters p , m etc. are used. Let's assume we have $m = 23$ data elements to be distributed over three processes ($p = 3$).

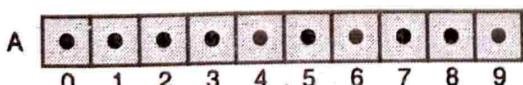
m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1
i	0	0	0	1	1	1	2	2	2	3	3	3	4	4	4	5	5	5	6	6	6	7	7

Fig. 2.4.6

Example

1-D Cyclic distribution of array 'A' on four processes

- The Cyclic distribution is described using an example shown in Fig. 2.4.7. Assume there are four processes P_0, P_1, P_2 and P_3 are available so that elements of an array can be mapped onto these for doing computations.



Processes : P_0, P_1, P_2, P_3

Fig. 2.4.7(a)

- Now the cyclic distribution of array 'A' becomes as shown in Fig. 2.4.7(a)

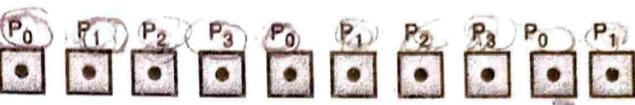


Fig. 2.4.7(b)

In Fig. 2.4.8 the distribution of elements to processes are shown according to cyclic distribution approach. In this example 1-D cyclic distribution on four processes is described.

P ₀												
P ₁												
P ₂												
P ₃												
P ₀												
P ₁												
P ₂												
P ₃												

Fig. 2.4.8

2.4.2(D) Block - Cyclic Distribution

- This type of data distribution onto processes is another technique which is slight variation of the block distribution.
- The block - cyclic distribution is the generalization of the block and cyclic distribution.
- In this form of distribution a block is considered which consists of number of consecutive data objects represented by r . These consecutive data objects are distributed cyclically onto number of processes represented by p .
- The parameters used here are the global index m and index triplet (p, b, i) . In this case the global index m lies between 0 and M and represented by $m(0 \leq m < M)$.
- In the index triplet (p, b, i) the value of p lies between 0 and P and represented as $p(0 \leq p < P)$.
- The global index of the data object is mapped into index triplet where p specifies processes onto which elements are mapped.
- The triplet (p, b, i) has p to be specified the process to which data element is mapped. The block number is represented by b in process p and i is the location in the block. The mapping $m \rightarrow (p, b, i)$ in block - cyclic distribution is defined as shown below:

$$M \rightarrow (\text{floor}((m \bmod T)/r), \text{floor}(m/T), m \bmod r)$$



- It is required to know about the block - cyclic distribution is that it reverts to the cyclic distribution in which $r = 1$ and a block distribution where $r = L$. Here $T = rP$
- Consider an example of block - cyclic distribution where $M = 23$ data objects. These data objects are distributed over three processes ($p = 3$). The block size represented by r is 2.
- The uneven distribution of data objects are shown in Fig. 2.4.9. This happens because the last block is smaller than others.

m	0 1 2 3 4 5	6 7 8 9 10 11	12 13 14 15 16 17	18 19 20 21 22
p	0 0 1 1 2 2	0 0 1 1 2 2	0 0 1 1 2 2	0 0 1 1 2
b	0 0 0 0 0 0	1 1 1 1 1 1	1 2 2 2 2 2	3 3 3 3 3 3
i	0 1 0 1 0 1	0 1 0 1 0 1	0 1 0 1 0 1	0 1 0 1 0
B	0 0 1 1 2 2	3 3 4 4 5 5	6 6 7 7 8 8	9 9 10 10 11

Fig. 2.4.9

2.4.3 Randomized Block Distributions

- This type of distribution is considered as a more general form of block distribution. The load balancing is achieved by dividing the array into more number of blocks than the total number of processes currently active for processing the data.
- The load balancing in randomized distribution is similar to a block - cyclic distribution.
- In this technique blocks are distributed randomly among processes but the block distribution is handled uniformly.
- Now consider an example of a one dimensional block distribution in randomized fashion shown in Fig. 2.4.10.
- The vector V with 15 elements included is distributed using randomized block distribution. The assumed block size is 3 and five processes are available.

$$V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$$

$$\text{Random}(V) = \{11, 5, 13, 8, 14, 1, 6, 7, 3, 10, 0, 12, 2, 4, 9\}$$

Now perform mapping of blocks onto processes.

{11, 5, 13}	{8, 14, 1}	{6, 7, 3}	{10, 0, 12}	{2, 4, 9}
P ₀	P ₁	P ₂	P ₃	P ₄

Fig. 2.4.10

2.4.4 Hierarchical Mappings

- There are some algorithms can be represented using Task dependency graphs.
- They are represented in the straight forward manner because of the task implemented on those are naturally suited to represent using Task dependency graphs.
- The mapping in this approach also has some problems like load-imbalance or inadequate concurrency.
- For example hierarchical mapping can be described for the binary tree task dependency graph.
- One such binary tree is shown in Fig. 2.4.11. Here at the top of the tree only few concurrent tasks can be performed because of availability of less number of tasks.
- The decomposition is expected at further level when the level at which number of tasks available are large so that mapping can be handled efficiently.
- In such cases a large task is divided into smaller sub - tasks at further levels.

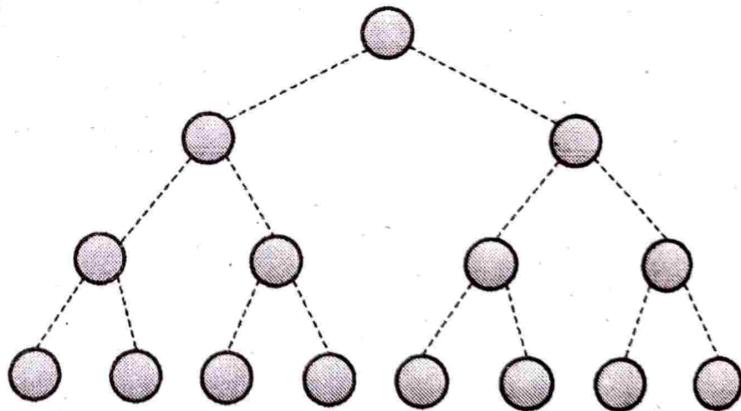


Fig. 2.4.11

- In Fig. 2.4.11, the binary tree is shown with level size as 4.
- Here initially the root task can be divided and assigned into eight processes. In the similar fashion task at the level next to root can be divided and assigned into four processes.
- These divisions at different levels are used for mapping of tasks onto processes and finally at leaf level tasks are mapped with processes in the form of one to one mapping.
- The Fig. 2.4.11 shows the hierarchical mapping based on the descriptions we have just made.



2.4.5 Schemes for Dynamic Mapping

- Dynamic mapping is applied in the situation where static mapping is not efficient because it shows very highly imbalanced distribution of works.
- The dynamic mapping of work distribution to processes used in another situation where task - dependency graph itself is dynamic by nature.
- The dynamic mapping is also called as dynamic load - balancing because it is basically used for balancing of workloads among processes. There are two different classes of dynamic mapping **centralized and distributed**.

1. Dynamic mapping with Centralized Schemes

- In this scheme a common data structure is used centrally to maintain all executable tasks. These tasks can also be maintained centrally by a special process.
- The processes involved in such a system are managed in the master - slave fashion. The master process is a special process responsible to manage the pool of available tasks.
- The other processes referred as slave processes get their work assigned from the master process.
- Any newly generated tasks are stored in the centralized data structure and master process assigns tasks from this structure to slave processes.
- The main advantage of the centralized scheme is its simplicity in implementation.
- It is really easier in terms of implementation as compare to distributed scheme. The limited scalability is the disadvantage of this scheme.
- For example in the case when number of processes has to access central data structure increases; the efficiency of the master process to assign tasks may degrade the overall performance.
- Consider sorting of an example matrix A of size $M \times N$ using the centralized mapping approach. The program segment shown below is used to do sorting of entries in each row of A in serial order.

```
for ( i = 0; i < M; i++ )  
    sort( A[i], M );
```

- The sorting of initial elements of an array affects the total sorting time of a particular sorting algorithm.
- This way the time required for sorting of an array elements vary because of dependency of sorting of initial elements.
- This concludes that each iterations of the '**for loop**' used in the program segment considered in this example may take different amount of time.
- In such type of situations load - imbalance may occur when mapping of the task (sorting of an equal number of rows) onto processes.

There are two techniques have been suggested for handling load - imbalance problem are : **self-scheduling** and **chunk scheduling**.

(A) **Self-scheduling** : This method is used for scheduling of independent iterations of a loop on number of available processes in parallel. This way it may happen that separate iteration of the loop is assigned to different processes for executions.

 - All such processes then execute iterations of the loop in the parallel fashion. The approach is based on the use of a central pool of row indices of not yet sorted elements.
 - When a process becomes available it picks up an index from the pool and performs sorting on that row. After sorting successfully that index is deleted from the pool of indices. This way all rows are sorted till the pool becomes empty.

(B) **Chunk scheduling** : The overall task is divided into number of subtasks. A chunk refers to a group of such tasks. In this scheme chunks are created and assigned to different processes for parallel executions. The problem of load - imbalance occurs when the number of tasks assigned in a single step is large.

 - The problem becomes more severe when the chunk size increases during program progress. This problem is solved by reducing the chunk size when program progresses further.
 - This means when the program execution starts the chunk size remains large but when program further progresses the chunk size also reduced.



2. Dynamic mapping with Distributed Schemes

- In this scheme the set of available executable tasks are distributed among processes for executions.
- The work balance is handled at runtime with exchange of tasks among processes.
- In this scheme it is assumed that communications among processes for sending and receiving work-loads is possible. The main advantage of this scheme is that these methods do not encounter any type of bottleneck.

Syllabus Topic : Methods for Containing Interaction Overheads

2.5 Methods for Containing Interaction Overheads

Q. 2.5.1 Explain Methods for Containing Interaction Overheads. (Refer sections 2.5 to 2.5.4)
(8 Marks)

- A parallel program performs efficiently when the interactions among the concurrent tasks are efficiently handled. There are many factors associated due to which interaction overhead increases in concurrent tasks.
- Some of the factors which are most important and require attention are: amount of data exchanged during interactions, the frequency of interactions and the spatial and temporal pattern of interactions.
- The general techniques used to reduce the interaction overheads are applicable during the stages when the decomposition and mapping schemes for the algorithms are devised. Some techniques are useful algorithm is being programmed in a given paradigm.

2.5.1 Maximizing Data Locality

- The tasks executed by different processes have to access some common data in most of the parallel programs.
- A particular task is able to compute efficiently when the required data upon which it operates are available in time.
- The data locality technique is used here to make the required data available to the tasks so that task interactions can be reduced.

- The various techniques like increase the reuse of recently accessed data and reduce the data access frequency up to some level so that data locality can be maximized.
- The techniques used in the modern processors for improvement of the data availability through cache, work almost similar like the schemes implemented for improving the data availability.

Minimize Volume of Data-Exchange

- The interaction overhead is reduced using the basic technique in which the overall volume of shared data is minimized.
- In this basic approach the shared data that needs to be accessed by concurrent tasks are targeted to be minimized.
- This approach is based on the facility where the locality of the temporal data is maximized. This way the consecutive references to the same data increases.
- The requirements for bringing more data into local memory or cache are minimized because of performing most of the computations on local data.
- One more approach is considerable here to decrease the amount of shared data required to be accessed by multiple processes is use of local data for intermediate results. This way shared data access is performed only for storing of the final results of the computations.

Minimize Frequency of Interactions

- The overheads occur in the interaction of tasks in parallel programming is reduced by minimizing the frequency of interaction.
- The use of shared data in large pieces through restricting of algorithm may affect the reduction in interaction frequency.
- This way the overall interaction overhead is reduced, even if such restructuring does not necessarily reduce the overall volume of shared data that need to be accessed.

2.5.2 Overlapping Computations with Interactions

- In the parallel program executions different processes involved in the computations have to cooperate and communicate with each other. These processes have to wait shared data to be arrived from other processes.



- The overlapping of computation with the interaction can be handled when a process has to wait for interaction at that time it can be assigned to do other independent computations. There are various approaches used to overlap computations with interactions.
- We can consider an example where overlapping is possible, simply identify the parts of the program code can be executed before initiating the interaction.
- Thereafter the parallel program is required to be structured in such a way that the interaction is to be initiated at an earlier point in the execution than it is needed in the original algorithm.

2.5.3 Replicating Data or Computations

- The interaction overheads amongst the concurrent tasks can also be reduced using the replication of data and computations.
- The explicit programmer intervention is not required for the read - only data needed for frequent access in the shared - address - space paradigm because such data very often affected by the caches.
- The architectures and programming paradigms where read - only access to shared data is expensive and harder as compare to local data access the obvious solution is based on replication of the data. This way parallel programming using message - passing paradigm benefits from the replication of data.
- The message - passing significantly reduces the interaction overheads and simplifies the programming.
- The downside of the data replication is increased memory requirements of a parallel program.
- The overall memory requirement for replicated data storage increases linearly with the number of concurrent processes.
- The direct restriction imposed because of the memory limitation is the limitations on the size of the problem to be solved on a particular parallel machine.

2.5.4 Overlapping Interactions with Other Interactions

- The overlapping of interactions with other interactions amongst several pairs of processes can reduce the effective volume of communications.

- We can consider four processes P_0 , P_1 , P_2 and P_3 as an example of overlapping interactions in a message - passing paradigm.
 - In this paradigm broadcasting of data from process P_0 to all other processes works in the fashion as follows.
 - Process P_0 sends data to be sent to process P_2 . In the next step the data is transferred to process P_1 and concurrently P_2 sends the data it had received from P_0 to P_3 . In this way the operation is completed in two steps instead of multiple steps.

Syllabus Topic : Parallel Algorithm Models

2.6 Parallel Algorithm Models

Q. 2.6.1 Write short note on Parallel Algorithm Models.
(Refer section 2.6.1) **(6 Marks)**

- The parallel algorithm models are used to describe the strategy for partitioning data and how these data are processed.
- A model is used to provide appropriate structuring of parallel algorithms on the basis of two techniques : **selection of a partitioning and mapping technique, proper use of strategy for interaction minimization**.
- There are basically six different types of models for parallel algorithms have been suggested by researchers.

2.6.1 The Data-Parallel Model

- This model of algorithm design is considered as one of the simplest models of parallel algorithm design.
- The various tasks identified in the problem to be solved are mapped to processes. The mapping of tasks onto processes are handled statically or semi - statically.
- In this strategy once the data have been distributed parallel operations can be performed. In this way the operations performed by each task are similar but the data on which these operations work are different.
- The parallelism occurs here and it is a form of data parallelism because identical operations are being performed on the different data sets.
- The overall task to be performed may be divided into different phases. The data upon which the tasks work in different phases may be different.



- In this approach different computation phases are interspersed with interactions to synchronize the tasks.
- The problem being solved using this approach is divided among number of tasks on the basis of data partitioning.
- The data partitioning is used because all tasks perform similar computations and a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.
- The two popular paradigms shared - address space and message - passing can be used for implementations of data - parallel algorithms.
- The interaction overheads occur in data parallel model but it can be minimized by adopting some strategy on implementations.
- A locality preserving decomposition is used for the interaction overhead minimization and in the similar way computation and interaction can be overlapped in the design.
- In the data parallelism, one of the key characteristics for most of the problem is that the degree of data parallelism increases with the size of the problem.
- This effectively provides the options for the use of more processes to solve larger problems.
- The parallel matrix multiplication is one of the examples of the data - parallel computations.

2.6.2 The Task Graph Model

- We have already discussed that the task dependency graph is used to describe the computations in any parallel algorithms. The task - dependency graph is explicitly used in mapping of some algorithms.
- The interaction costs among the tasks can be reduced by the use of interrelationships among the tasks in task dependency graph.
- The problems in which tasks are associated with large amount of data as compare to the actual computations can be solved effectively using this model.
The data movement cost among task is optimized by the use of mapping of tasks to processes statically.
- The paradigm with globally addressable space can be used to share work in more easier ways, but disjoint address space

paradigm can also be used because mechanisms are available to share the work.

- The parallelism described with the task - dependency graph where each task is an independent task is called as **task - parallelism**.
- There are several examples of parallel algorithms based on the task - graph model and some of them are **parallel quick - sort**, parallel implementation of algorithms based on divide and conquer etc.

2.6.3 The Work Pool Model (or Task Pool Model)

- The mapping of tasks onto processes for parallel execution is described by the dynamic mapping approach so that load balancing can be handled.
- The dynamic mapping is used because according to that any task can be mapped with any process provided that it should have the availability of task assignment.
- The work pool model uses the dynamic mapping approach for task assignments.
- In the work pool model of algorithm design centralized or decentralized mapping can be adopted.
- The work to be performed sometimes available in advance but can be generated at runtime also. The executing processes generated the work to be performed and these are added into the globally available work pool.
- The work pool model can be used in the message - passing paradigm when the data associated with the tasks is smaller than the computation associated with the tasks.
- This way the tasks can be moved without causing too much overhead due to interactions. The examples of the work - pool model are loop parallelization using chunk scheduling, parallel tree search etc.

2.6.4 The Master Slave Model (Manager - Worker Model)

- This model of parallel algorithm design designates two different types of processes: master and slave.



- In this scheme one or more processes are designated as master and are responsible to coordinate the activities among all slave processes.
- The master process generates work to be performed and assigns it to the number of worker/slave processes.
- The allocation of the tasks depends upon the size of the task. If the size of the task can be estimated on prior basis the master can allocate the tasks to the required processes.
- In another scenario the slave processes can get the pieces of work at different times.
- The assignment of work to the slaves at different times is preferable when the situation where all slaves cannot wait for the availability of tasks from the master on prior basis.
- In some of the scenarios where work have to be done in different phases this model can be used effectively by assigning different slaves to perform tasks at different phases. In this case the master is responsible to synchronize the activities of the slaves after each phase.
- The master - slave model is generally suitable for shared address space and message passing paradigms.
- This is possible because in the master - slave model interaction among the processes are always in two - way fashion.
- The two - fashion is described as the master always knows that it has to provide things to slaves and in the similar way slaves know that they get things from the master.

2.6.5 The Pipeline or Producer Consumer Model

- This model is based on the passing on stream of data through processes arranged in a succession.
- The data flows through successive processes and at that time each process does some operations on it. This is called stream parallelism because simultaneous executions of different programs on data streams are performed.
- Whenever a new data arrive a new task execution initiates by a process in the pipeline. This model is also called as producer - consumer model because a pipeline acts as a chain of producers and consumers. Suppose P_1 and P_2 are two processes associated with a pipeline and P_1 precedes P_2 .

- In this situation process P_2 is the consumer process and P_1 becomes the producer because it provides the input stream for process P_2 .
- The pipeline can be arranged in a directed graph fashion also instead of strictly on a linear chain fashion.
- The mapping of tasks onto processes in the pipeline model is usually handled using a static mapping approach.

2.6.6 Hybrid Models

The combination of more than one model of parallel algorithm design is applicable in some cases for solving the problem at hand. These models are used at different phases of the parallel algorithm design. The suitability of a particular model in a phase of the algorithm is taken care and accordingly it is used.

Syllabus Topic : The Age of Parallel Processing

2.7 The Age of Parallel Processing

- The parallel computing activities have been adopted by many industries to fulfil their computing requirements of tasks. In modern machines almost all the processors are equipped with the in-built multicore processors.
- The parallel processing facility is provided into many handheld electronic devices such has mobile phones and portable music players.
- The software development field uses many tools for the developments and the design of parallel software applications because of the availability of multiple processing elements in electronic devices.
- The incorporation of parallel processing in various devices makes the devices like cell phones not only used for call making but also other activities like playing music, Internet access etc. are widely adopted.

2.7.1 Central Processing Units and GPUs

- All the program and instructions are executed in order to derive the necessary data in case of CPUs. The advancement in modern day CPUs have allowed it to crunch more numbers than ever before, but the advancement in software technology meant that CPUs are still trying to catch up.



- The load of the CPU is alleviated by the GPU by handling all the advanced computations necessary to project the final display on the monitor.
- Originally, CPUs handle all of the computations and instructions in the whole computer, thus the use of the word 'central'.
- But as technology progressed, it became more advantageous to take out some of the responsibilities from the CPU and have it performed by other microprocessors.
- In the days before GUIs, the screen was simply a small grid with each box having an 8 bit value that corresponds to a character. This was relatively very easy to do for the CPU, but GUIs have greater resolutions with each pixel having a 16 bit or 32 bit color value.
- GPUs were originally developed to render 2D graphics; specifically, to accelerate the drawing of windows in a GUI. But as the need for 3D and faster graphics acceleration grew, the GPU became faster and more specialized in its task.
- GPUs are now generally floating point processors that can easily crunch geometric computations along with texture mapping tasks.
- Most GPUs have implemented MPEG primitives to make enhance the playback of videos; some even have the capability to directly decode HD video data, taking another task away from the CPU.
- Hardware wise, GPUs and CPUs are similar but not identical. If we looked at the very building block of each, the transistors, we can see that most GPUs already rival CPUs in transistor count.
- The specialized nature of GPUs means that it can do its task much faster than a CPU ever can, but it is not able to cover all of the capabilities of the CPU.
- Multiple GPUs can also be employed to achieve a single goal much like the dual core CPUs currently available. ATI's Crossfire and NVidia's SLI allow users to connect two identical GPU's and make them work as one.

Syllabus Topic : Brief History of GPUs, Early GPU

2.7.2 The Rise of GPU Computing

The GPU is used now a day to perform general purpose computations and because of that the computing tasks are performed in efficient ways. The idea of graphics processors is not new in the field of computing but GPU is comparatively a new concepts in computing.

A brief history of GPUs

- The graphic processing field also evolved like clock speed changes in central processing units.
- The graphically driven operating systems like Microsoft Windows entered in the software field in early 1990s and since then the graphics processing became prime concern for processor manufacturing companies.
- The hardware devices require for handling graphics based systems were developed and adopted by the users and programmers.
- At round the same time the company named as Silicon Graphics popularized the use of three dimensional graphics in various markets like government sector and defense applications.
- The OpenGL library is released by Silicon Graphics in 1992 and the programming interfaces for their hardware were opened.
- The intention of releasing OpenGL was based upon the focus that it should be used as a standardized, platform-independent method for writing 3D graphics applications.
- In the mid-90s the demand for 3D graphics based applications increased because of the significant developments in the 3D gaming industry.
- At the same time companies like NVIDIA, ATI Technologies and others started releasing graphics accelerators in affordable price.
- The capabilities of the consumer graphics hardware further pushed after release of NVIDIA's GeForce 256.
- In such systems, the graphics processors were first time used to perform the computations directly and because of that the scope for the developments for visually interesting applications were enhanced.



- | | |
|--|--|
| <ul style="list-style-type: none">- The most important achievement in the GPU computing technology is the release of NVIDIA's GeForce 3 series in 2001.- The GeForce 3 series was the computing industry's first chip to implement Microsoft's then-new DirectX 8.0 standard. | <p>This standard required that compliant hardware contain both programmable vertex and programmable pixel shading stages.</p> <ul style="list-style-type: none">- For the first time, developers had some control over the exact computations that would be performed on their GPUs. |
|--|--|

