Unit I

1 Introduction to Parallel Computing

Syllabus

Introduction to Parallel Computing *: Motivating Parallelism,* ***Modern Processor*** *: Stored-program computer architecture, General-purpose Cache-based Microprocessor architecture.* ***Parallel Programming Platforms*** *: Implicit Parallelism, Dichotomy of Parallel Computing Platforms, Physical Organization of Parallel Platforms, Communication Costs in Parallel Machines. Levels of parallelism,* ***Models*** *: SIMD, MIMD, SIMT, SPMD, Data Flow Models, Demand-driven Computation,* ***Architectures*** *: N-wide superscalar architectures, multi-core, multi-threaded.*

Contents

1.0 What is HPC ?

SPPU : March-17, May-19, Oct.-19, Dec.-19

- The term High Performance Computing (HPC) has an abstract level understanding. It refers to performing computational operations collaboratively on multiple computers that have higher level performance in terms of throughput.
- One may wonder , why there is a need of an HPC when there are already similar concepts like Parallel computers, Supercomputers and even Mainframes.
- With the growth of higher processing capabilities, information flood, superspeed network connectivity and big data, various research institutes and universities have acknowledged the need of fast and accurate computing to -
  1. Perform a high number of operations per seconds (FLOPS)
  2. Complete a time-consuming operation in less time
  3. Complete an operation under a tight deadline
  4. Handle huge amount of data

HPC brings in the solution to these issues.

- High-performance computing is a mechanism of fast computations in parallel over lots of computing nodes (like CPU, GPU) interconnected on a very fast network (System interconnects). To explain this concept let's have a look at below fish tank, that helps us in understanding how HPC is different than other computational systems.
- The HPC facilitates parallel computing on a large number of smaller capacity computational nodes with higher efficiency than using high end systems like Supercomputers, Mainframes or vectored parallel computer systems that uses specialized, high capacity few computational nodes.

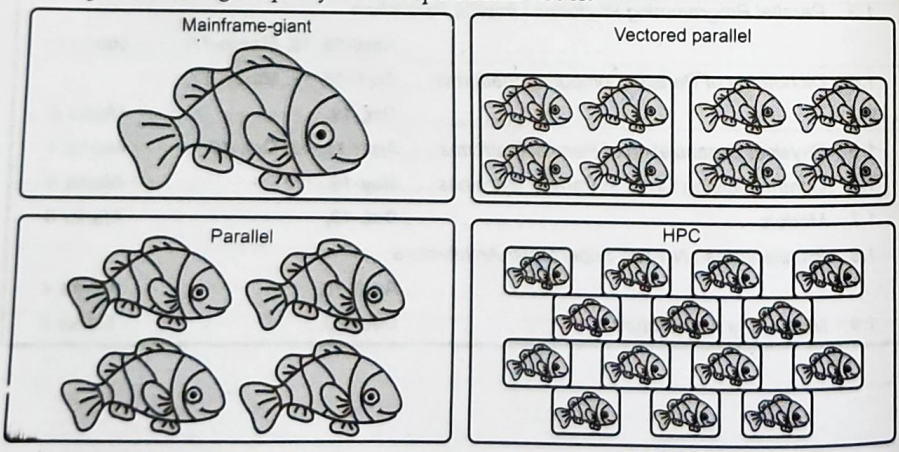


Fig. 1.0.1 Pictorial representation of different high performance system

- High-performance computing, known as HPC, refers to the use of aggregated computing power for handling compute and data-intensive tasks - including simulation, modeling and rendering that standard workstations are unable to address.
- The problems which cannot be solved on a commodity computer (standard-issue PC that is widely available for purchase) within a reasonable amount of time (too many operations are required) or the execution is impossible, due to limited available resources (too much data is required) are targeted.
- HPC is the approach to overcome these limitations by using specialized or high-end hardware or by accumulating computational power from several units.

1.0.1 Who Uses HPC Today ?

- The HPC has been traditionally used by research institutes, universities and government Institutions like meteorological departments to solve complex computational problems related to weather, using computer modeling, simulation and analysis. With recent developments in technology , even mainstream businesses started using HPC to enhance their business models. E.g. Financial institutes use HPC for economic and financial market analysis, faster and more secure financial transactions , fraud detection in credit/debit cards using specialized algorithms, etc. In the life sciences sector including pharmaceuticals , HPC is used to design molecular chemistry models, to identify genetic patterns and disorders using gnomes, to mine clinical data. In the energy industry, the HPC is used to analyze site data to develop geological models to simulate drilling for energy stations like oil and gas deposits in the earth's crust.

Here is a brief look at who uses HPC

1. Financial institutions : Transactions and card fraud detection.
2. Bio-sciences and the human genome : Drug discovery, disease detection / prevention.
3. Computer Aided Engineering (CAE) : Automotive design and testing, transportation, structural, mechanical design.
4. Chemical engineering : Process and molecular design.
5. Digital Content Creation (DCC) and distribution : Computer aided graphics in film and media.
6. Economics / financial : Wall Street risk analysis, portfolio management, automated trading.
7. Electronic Design and Automation (EDA) : Electronic component design and verification.

8. Geosciences and geo-engineering : Oil and gas exploration and reservoir modeling.
9. Mechanical design and drafting : 2D and 3D design and verification, mechanical modeling.
10. Defense and energy : Nuclear stewardship, basic and applied research.
11. Government labs, University/academic : Basic and applied research.
12. Meteorological Departments : Weather forecasting.

Some of the prominent areas of application are

1) **Engineering and design** : Parallel computing has traditionally been employed with great success in the design of airfoils, internal combustion engines, high speed circuits and structures.
Other applications in engineering and design focus on optimization of processes using algorithms like simplex, Interior Point Method for linear optimization, branch and bound etc.

2) **Scientific applications** : Bioinformatics and astrophysics has some challenging areas to deal with large datasets. Protein and gene databases, SKY survey databases requires tremendous computational powers. Analyzing biological sequences in protein and genome databases to view developing new drugs and cures require power of parallel computing.

3) **Commercial applications** : Parallel platforms are used as web and database servers. The sheer volume and geographically distributed nature of data require the use of effective parallel algorithms for data association rule for mining, clustering, classification and time series analysis.

1.0.2 HPC as a System

HPC systems provides computational clusters which are more affordable, efficient and scalable in nature. A typical HPC cluster has following components

- **Cluster** : *Cluster is a widely used term meaning independent computers combined into a unified system through software and networking. Each Cluster Node is an SMP Server , Workstation or a PC. All Cluster Nodes must be in a position to work together as a Single Integrated Computing Resource Clusters are typically used for :*
  - *High Performance Computing (HPC) to provide greater computational Power than a single computer can provide*
  - *High Availability (HA) for greater reliability*

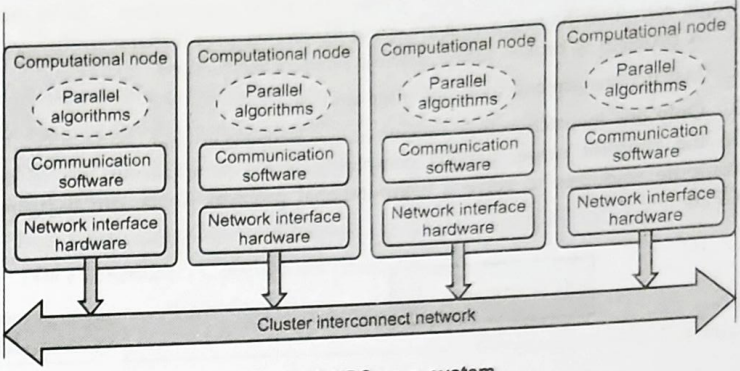


Fig. 1.0.2 HPC as a system

- Some of the industry leaders in HPC are Intel for HPC enabled processors, Fujitsu for network clusters, Hewlett Packard hardware, AWS for data services, etc.
- In the later part of this book, we will learn more about each component of the HPC system.

Review Questions

1. *Describe HPC as a system.*
2. *Define - a) Runtime b) Flops c) Efficiency d) Scalability e) Throughput.*
3. *Explain the following algorithmic functions*
   a) The θ notation b) The big O notation c) The O notation.
4. *What are applications of parallel Computing ?* **SPPU : March-17, Oct.-19, Marks 4**
5. *Discuss the applications that benifit from multi-core architecture.* **SPPU : May-19, Marks 6**
6. *List application of parallel programming.* **SPPU : Dec.-19, Marks 6**

1.1 Introduction to Parallel Computing

- A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem. Parallel computers are interesting because they offer the potential to concentrate computational resources like processors, memory, or I/O bandwidth on important computational problems.
- Parallel computing is a form of computation in which many instructions are carried out simultaneously operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel).

- Traditionally, software has been written for serial computation : To be run on a single computer having a single Central Processing Unit (CPU) :
  1. A problem is broken into a discrete series of instructions.
  2. Instructions are executed one after another.
  3. Only one instruction may execute at any moment in time.
- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. To be run using multiple CPUs

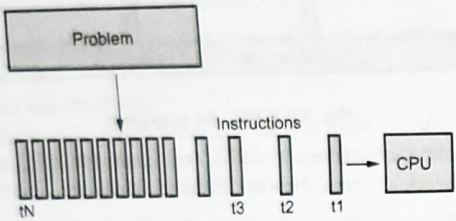


Fig. 1.1.1 Serial execution

1. A problem is broken into discrete parts that can be solved concurrently.
2. Each part is further broken down to a series of instructions.
3. Instructions from each part execute simultaneously on different CPUs.

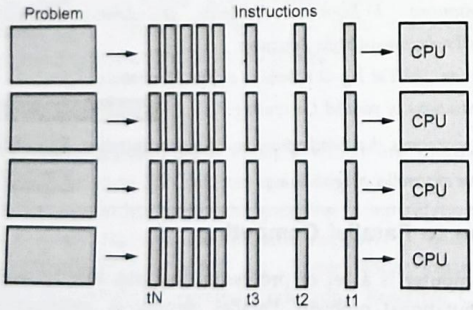


Fig. 1.1.2 Parallel execution

- In the parallel computing the computational node can include :
  1. A single computer with multiple cores.
  2. A single computer with (multiple) processor(s) and some specialized computer resources (like GPU)

3. An arbitrary number of computers connected by a network (using various topologies)
4. A combination of all the above.

- The types or the levels of the parallelism are -
  1. Bit-Level parallelism
  2. Instructional parallelism
  3. Data Level / Loop level parallelism
  4. Task / Functional / Control level parallelism

Review Questions

1. *Write a short note on parallel computing.*
2. *List out the various levels of parallelism.*

1.2 Motivating Parallelism

- Recent years have experienced a significant development of parallel processing paradigm. This is primarily due to advancements in specifying and coordinating complex concurrent tasks, a portable algorithms, specialized execution environments and software development toolkits. These advancements are based on some past arguments in the favor of parallel computing platforms. The influential arguments are
  1. The computational power argument
  2. The memory / disk speed argument
  3. The data communication argument

1) The significant growth in the CMOS chip based processors and networking paradigm has motivated the parallelism in application development.
2) Standardized hardware interfaces have reduced the turnaround time from the development of a microprocessor to a parallel machine based on the microprocessor.
3) Considerable progress has been made in standardization of programming environments to ensure a longer life-cycle for parallel applications.

Review Question

1. *Write a short note on factors motivating parallelism.*

1.3 Parallel Programming Platforms : Implicit Parallelism

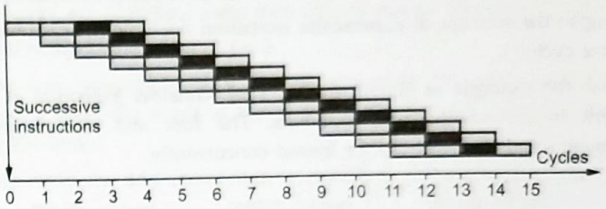
SPPU : April-16, 18, March-17

- Implicit parallelism allows programmers to write their programs without any concern about the exploitation of parallelism.
- The compiler, runtime system and the underlying hardware play an important role in exploiting the parallelism implicitly.
- The parallelism is transparent to the programmer so the programmer will write the standard sequential program without adapting any special parallel constructs.
- It is the job of underlying systems to figure out the parallelism from the sequential code, with the help of different techniques and later to implement it.
- In this section, various implicit parallel mechanisms used in Pipelining and superscalar execution and VLIW processing will be discussed.

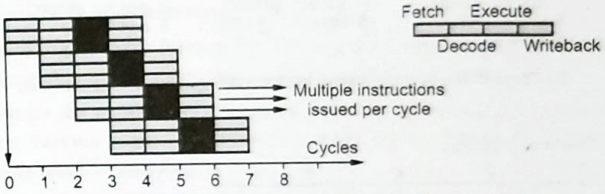
1.3.1 Pipelining and Superscalar Execution

- Let's first revise some basic terminologies related to pipelining :
  - **Clock cycle :**
    - The workload is generally expressed in terms of the number of processor clock cycles.
    - Any high level program contains sequence of instructions, which are later translated to sequence of instructions in binary code.
    - These instructions are executed by the processor as a sequence of basic steps called machine cycle.
    - Each machine cycle consists of one or more processor clock cycles or clock periods or clocks, which are the reciprocal of processor clock rate.
    - The sequence of machine cycles executed for an instruction is called an instruction cycle.
  - **Basics of instruction pipeline :**
    - A typical instruction can be divided in four phases : fetch, decode, execute and write back.
    - These phases are executed by a pipelined processor with multiple stages called the instruction pipeline.
    - The pipeline is a hardware structure which executes the sequential instructions like an industrial assembly line.
    - By overlapping various phases or stages in instruction execution, pipelining enables faster execution.
    - For example, the Pentium 4, which operates at 2.0 GHz, has a 20 stage pipeline.

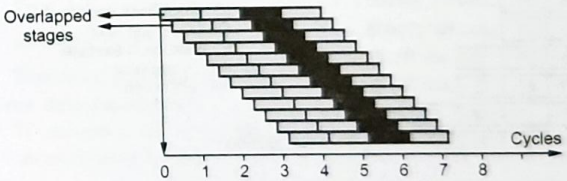
- To enhance the processor performance there are two techniques :
  - To exploit higher Instruction Level Parallelism (ILP)
  - To subdivide the pipeline into simpler stages.
- In a **superscalar processor**, multiple instructions are issued per cycle and multiple results are generated by multiple pipelines per cycle.



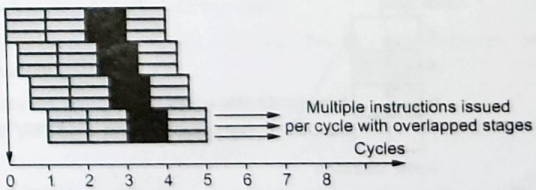
(a) Single-issue base pipeline



(b) Three-issue superscalar pipeline



(c) Single-issue superpipeline



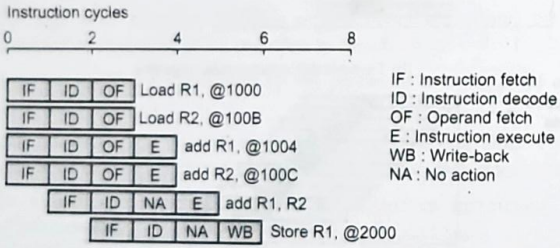
(d) Three-issue superscalar superpipeline

Fig. 1.3.1 Pipelining operations in a base pipeline, superscalar and superpipelined processors

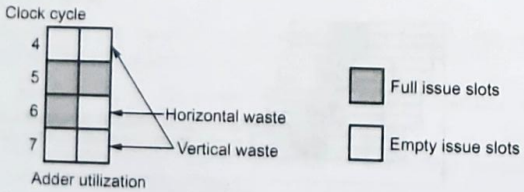
- Superscalar processors are designed to exploit **Instruction Level Parallelism** in user programs.
- As shown in the Fig. 1.3.1, instruction pipelines can be designed in four ways. (See Fig. 1.3.1 on previous page.)
- To understand superscalar execution let's consider a processor with two pipelines and the ability to simultaneously issue two instructions (two issue superscalar).
- Catering to the concept of superscalar execution multiple instructions are issued in the same cycle.
- Consider the example as shown in Fig. 1.3.2, consider execution of the first code fragment in for adding four numbers. The first and second instructions are independent and therefore can be issued concurrently.

1. load R1, @1000	1. load R1, @1000	1. load R1, @1000
2. load R2, @1000	2. add R1, @1004	2. add R1, @1004
3. add R1, @1004	3. add R1, @1008	3. load R2, @1008
4. add R2, @100C	4. add R1, @100C	4. add R2, @100C
5. add R1, R2	5. store R1, @ 2000	5. add R1, R2
6. store R1, @ 2000		6. store R1, @2000

(a) Three different code fragments for addling a list of four number.



(b) Execution schedule for code fragment (i) above



(c) Hardware utilization trance for schedule in (b)

Fig. 1.3.2 Example of a two-way superscalar execution of instructions

- At t = 0, simultaneous issue of the instructions load R1, @1000 and load R2, @1008 will take place.
- The instructions are fetched, decoded and the operands are fetched.
- After execution of first two instructions, the next two mutually independent instructions, add R1, @1004 and add R2,@100C are executed.
- As processors are pipelined these instructions can be issued concurrently at t = 1.
- These instructions terminate at t = 5.
- The next two instructions, add R1, R2 and store R1, @2000 cannot be executed concurrently since the result of the earlier instruction i.e. contents of register R1 is used by the latter.
- Therefore, only the add instruction is issued at t = 2 and the store instruction at t = 3.
- The instruction add R1, R2 can be executed only after the previous two instructions have been executed.
- To achieve higher performance by simultaneous execution of the instructions, the instructions should be completely independent of each other.
- Even though the superscalar execution seems to be simple there are certain issues, posed by various dependencies which exist while dealing with instruction level parallelism need to resolved.
- Some of the dependencies which can be listed are :
  - True data dependency
  - Resource dependency
  - Branch or procedural dependency
- **True data dependency :**
  - If execution of a particular instruction depends on the result of previous instruction in a program then it is known as **true data dependency.**
  - Consider the example in Fig. 1.3.2, true data dependency exists between the instructions load R1, @1000 and add R1,@1004, as without getting the contents of R1 it is not possible to compute add operation.
  - True data dependency must be resolved before simultaneous issue of instructions.
  - Note that two important aspects are involved in this :
    - There must be proper hardware support as dependency is to be resolved at runtime.
    - There are limitations to ILP in a program and it depends on the coding technique. Many a times just by reordering the instructions more parallelism can be exploited.

- **Resource dependency :**
  - To understand resource dependency consider the example of co-scheduling of two floating point operations on a two issue superscalar processor with a single floating point unit.
  - Note that no data dependency exists in these instructions.
  - Here the dependency is posed by the use of finite resources, which are shared by different pipelines.
  - As a result even though these instructions are independent both cannot be scheduled together as there is a single floating point unit which is needed by both the instructions at a time.
  - This form of dependency in which two instructions compete for a single processor resource is referred to as **resource dependency.**
- **Branch or Procedural dependency :**
  - Branch dependencies exist due to the flow of the program.
  - Consider the execution of a conditional branch instruction,
  - As after computing the branch instruction it can be decided that which path is to be executed if the instructions are scheduled apriori, it may lead to errors.
  - These dependencies are referred to as **branch dependencies** or **procedural dependencies**.
  - Accurate branch prediction is very important for efficient superscalar execution as in a code generally branching instructions are present between every five to six instructions.
  - To handle branch dependencies **speculative scheduling** is done, i.e. the instructions which are control independent are moved before the execution of the control instructions (branches).
- The most important concept involved in superscalar execution is to detect and schedule concurrent instructions in a program.
- If in a program the instructions are executed in the order in which they are written then it is called as **in order execution** of the program.
- Consider the example 1.3.2 (iii), in this piece of code we can observe that in the first two instructions - load R1, @1000 and add R1, @1004 data dependency exists so they cannot be executed simultaneously if we follow in order execution of the program.
- Now if the processor has capability to look ahead, it can reschedule the third instruction load R2, @1008 with the first instruction for simultaneous execution.
- This ability of a processor to reschedule the instructions to exploit the parallelism is known as, **out of order execution.**

- It is also called as **dynamic instruction issue** by which maximum ILP can be exploited.
- Most current microprocessors are capable of out of-order issue and completion.
- Apart from pipelining aspects the performance of superscalar execution also depends on the execution aspects of a program.
- Consider the example in Fig. 1.3.2. Assume two execution units (multiply-add units), it is observed that there are several cycles in which the floating point unit is idle.
- These are called as zero-issue cycles which are the wasted cycles with respect to execution.
- If, during a particular cycle, no instructions are issued on execution units, it is referred to as **vertical waste**; if only part of the execution units are used during a cycle, it is termed **horizontal waste.**
- In the example, there are two cycles of vertical waste and one cycle with horizontal waste.
- In this case only three of the eight available cycles are used for computation.
- Due to these limitations posed by various factors like resource dependencies, limited parallelism, etc. the resources in superscalar processors remain underutilized.

1.3.2 Very Long Instruction Word Processors

- Apart from superscalar processors one more approach to exploit instruction-level parallelism is by **Very Long Instruction Word(VLIW) processors.**
- In VLIW processors compiler is the key.
- Various techniques like **branch predication, speculative decomposition, loop unrolling,** etc. are used in VLIW processors to exploit parallelism.
- We will learn some of them in later units.
- During compile time, dependencies are resolved and resource availability is checked.
- As shown in Fig. 1.3.3, instructions that can be executed concurrently are packed into bundles or groups and parceled off to the processor as a single long instruction word.

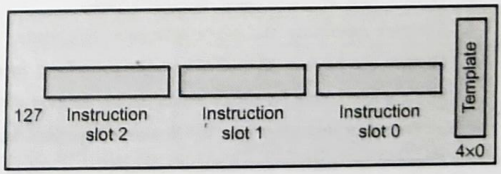


Fig. 1.3.3 (a) Bundle Structure

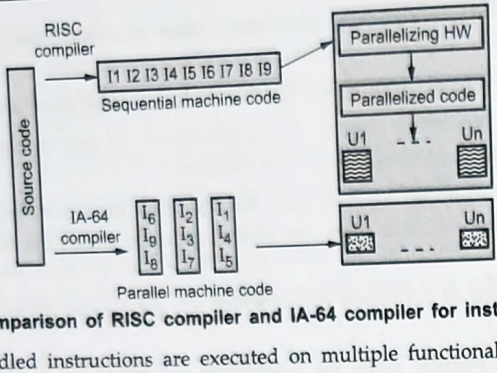


Fig. 1.3.3 (b) Comparison of RISC compiler and IA-64 compiler for instruction processing

- These bundled instructions are executed on multiple functional units at the same time.
- A very long instruction word can be as long as 256 B or 1024 B as per the design of **Multiflow computer (1980).**
- **IA-64 architecture** is the another variant of VLIW concept.
- VLIW as well as IA-64 both the processors has both advantages and disadvantages compared to superscalar processors.
- The decoding and instruction issue mechanisms are simpler in VLIW processors, due to software scheduling
- The compiler can take up the additional parallel instructions to control parallel execution.
- The drawbacks of VLIW processor :
  - Compilers do not have the dynamic program state available to make scheduling decisions which reduces the accuracy of branch and memory prediction.
  - It is very difficult to predict stalls on data fetch due to cache misses.

Review Questions

1. *What is implicit parallelism ?*
2. *Describe the pipelining execution mode.*
3. *Write a note on superscalar execution mode.*
4. *Compare pipelining and superscalar execution mode.*
5. *Discuss the various dependencies to be considered in the superscalar execution.*
6. *Write a short note on the following in the context of superscalar execution on*
   a) True data dependencies b) Resource dependencies c) Branch or procedural dependencies.
7. *Describe speculative dependancy concept.*
8. *Explain with suitable example : Very Long Instruction Word Processors.*
9. *What are the drawbacks of VLIW processors ?*

10. *Explain basic working principal of Superscalar processor.* **SPPU : April-16, Marks 6**
11. *Explain basic working principal of VLIW processor.* **SPPU : March-17, Marks 6**
12. *Explain Superscalar execution in terms of horizontal waste and vertical waste with example.* **SPPU : April-18, Marks 6**

1.4 Dichotomy of Parallel Computing Platforms

SPPU : April-16, 18, March-17, Oct.-19

- There are several parallel platforms which facilitates parallel computing.
- In this section the division based on logical and and physical organization of parallel platforms will be discussed.
- Physical organization is the actual hardware organization of a platform whereas logical organization refers to a programmer's view of the platform.
- From programmer's perspective the two important components of parallel computing are :
  - **Control structure :** The various ways of expressing parallel tasks is known as control structure
  - **The communication model :** The mechanisms for specifying interaction between the parallel tasks is called as communication model.

1.4.1 Control Structure of Parallel Platforms

- Depending on the application, the parallel tasks can have different levels of granularity.
- In case of coarse grain granularity each program in a set of programs can act as a parallel program whereas in case of fine grain granularity each instruction of a program can be considered as a parallel task.
- Based on this diversity in formation of the parallel tasks, control structure models with appropriate architectural support can be specified.
- In parallel machines either there can be single control unit under the centralized control of which all the processing units will work or the processing units work independently.
- Based on this the parallel computers can be classified based on Flynn's taxonomy.

Flynn's taxonomy was first proposed by Michael J. Flynn in 1966. It gives the specific classification of parallel computer architectures that are based on the number of concurrent instruction (single or multiple) and data streams (single or multiple) available in the architecture.

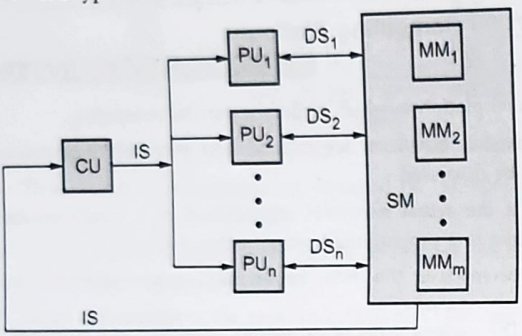
SISD Single Instruction, Single Data	**SIMD** Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	**MIMD** Multiple Instruction, Multiple data

Fig. 1.4.1

- Parallel Processing computers falls under SIMD and MIMD category according to flynn's classification.

1.4.1.1 Single Instruction Stream Multiple Data Stream (SIMD)

- The typical structure of SIMD architecture is shown as :



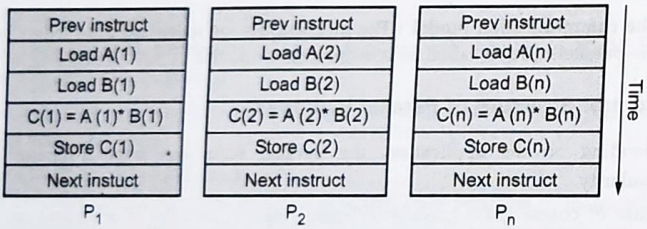


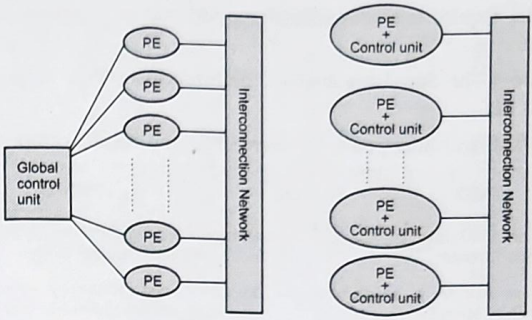
Fig. 1.4.2 SIMD architecture

- As shown in the example single control unit dispatches instructions to each processing unit. Fig. 1.4.3 (a) illustrates a typical SIMD architecture. (See Fig. 1.4.3 on next page.)

- The examples of SIMD computers are: the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1,co-processing, units such as the MMX units in Intel processors and DSP chips such as the Sharc. The Intel Pentium processor with its SSE (Streaming SIMD Extensions) provides a number of instructions that execute the same instruction on multiple data items.

1.4.1.2 Multiple Instruction Stream Multiple Data Stream (MIMD)

- Parallel computers in which each processing element is capable of executing a different program independent of the other processing elements are called multiple instruction stream, multiple data stream (MIMD) computers.

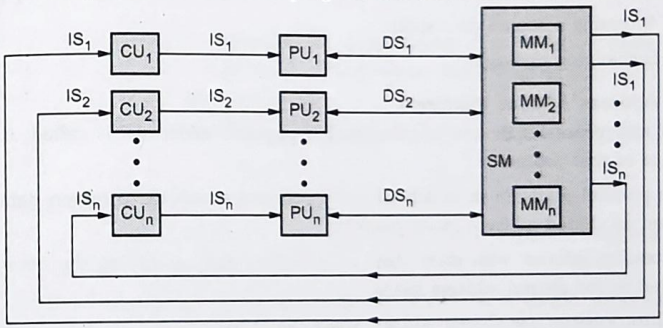


(a) A typical SIMD architecture (b) A typical MIMD architecture

PE - Processing Element

Fig. 1.4.3 Parallel system architectures

- One more variant of this model is **Single Program Multiple Data (SPMD)**, in which multiple instances of same program execute different data items.



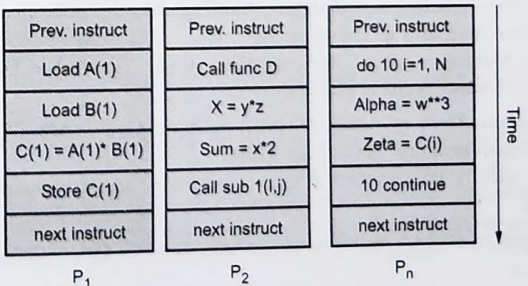


Fig. 1.4.4 MIMD architecture

- SPMD model require less architectural support and is popularly used by various parallel platforms.
- Examples are : The Sun Ultra Servers, multiprocessor PCs, workstation clusters, and the IBM SP.
- Some more distinguishing points between SIMD and MIMD computers are :

SIMD	MIMD
Computers need less hardware as only one global control unit is present	Computers need more hardware as all the nodes are independent nodes
Machins require less memory as only one copy of the program will be stored	Machines need more memory as program and operating system should be present at each processor
Computers require specialized extensive hardware architecture	Computers can be built from inexpensive components in less efforts

1.4.2 Communication Model of Parallel Platforms

- Two different ways by which data can be exchanged between parallel tasks are :
  - Accessing a shared data space
  - Exchanging messages.

Shared - Address - Space Platforms

- For any processor,the set of all possible physical addresses is called as **address space** of that processor.
- The parallel platform in which all the processors access the common data space is called as shared address space platform.
- Processors interact with each other by accessing and modifying the data elements stored in the shared address space.
- Multiprocessors use shared address space platforms.
- Based on the memory access time following are the classifications of shared address space machines as shown in Fig. 1.4.5.
  - Uniform Memory Access (UMA) multicomputer : Time taken by the processor to access memory word is identical.
  - Non Uniform Memory Access (NUMA) multicomputer : More time is taken to access certain memory words than others.
- UMA and NUMA are specified in terms of memory access times rather than cache access time.
- The examples of NUMA multiprocessors are : The SGI Origin 2000 and Sun Ultra HPC servers.

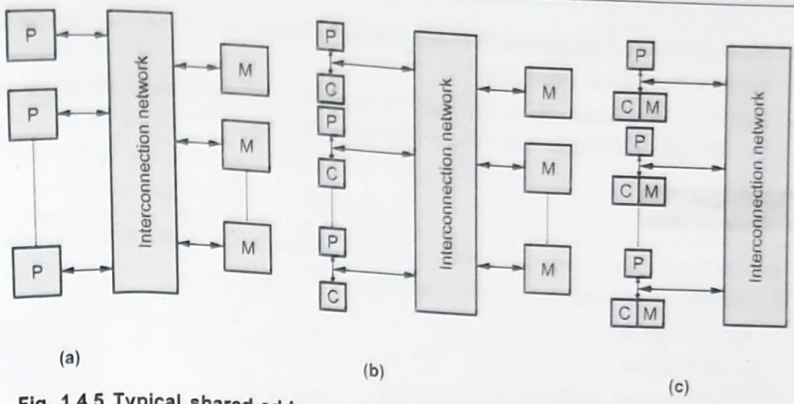


Fig. 1.4.5 Typical shared-address-space architectures: (a) Uniformmemory-access shared-address-space computer; (b) Uniformmemory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access

- To write programs on shared address space machines is simpler as transparency is provided to the programmer in reading operation and it is similar to the serial program.
- While performing write operation the programmer has to incorporate mutual exclusion for concurrent access.Also interprocess synchronization is very crucial which can be included using locks, etc.
- The examples of shared address space programming standards are: POSIX,NT and Open MP.

Message - Passing Platforms

- In message passing platform p processors with separate address space communicate with each other.
- Each node is a complete node in its sense. It can be a single processor or a shared-address-space multiprocessor.
- The processors interact with each other through messages so it is called as message passing model.
- Through message data, work, and to synchronize actions are transferred between the processors.
- To write any basic message passing program there are four basic opeartions: the basic operations in message passing platform are sending (send) and receiving (receive) so there must be a mechanism to assign a unique identification or ID to each process for specifying the target address. The fourth important function specifies the number of processes participating in the group.

- Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the APIs support these basic operations.
- Examples of parallel platforms that support the message-passing paradigm are the IBM SP, SGI Origin 2000, and workstation clusters.

Review Questions

1. *What are the two important components of parallel systems from a programmers perspective ?*
2. *Describe control structure of parallel platforms.*
3. *Write a short note on*
   a) SIMD stream b) MIMD stream
4. *Compare between SIMD and MIMD streams.*
5. *What are the communication models of parallel computing platforms ?*
6. *Write a short note on*
   a) Shared address space platforms b) Message passing platforms.
7. *Explain control structure of parallel platforms in detail.* **SPPU : April-16, Marks 4**
8. *Explain SIMD, MIMD and SIMT architecture.* **SPPU : March-17, Marks 6**
9. *Explain following models : i) MIMD ii) SIMD* **SPPU : April-18, Marks 6**
10. *Define latency and bandwidth of memory.* **SPPU : Oct.-19, Marks 4**

1.5 Physical Organization of Parallel Platforms SPPU : April-16,18, Oct.-19

- To understand physical architecture of a parallel computer, Let's understand the architecture of a ideal parallel computer and the practical difficulties faced.

1.5.1 Architecture of an Ideal Parallel Computer

- A Random Access Machine (RAM) is a simple model of computation. Its memory consists of an unbounded sequence of registers. Each of the registers may hold an integer value. The control unit of a RAM holds a program, i.e. a numbered list of statements. The program counter determines which statement is to be executed next.
- This simple model of RAM can be extended to parallel model by adding p processors and a global memory of unbounded size that is uniformly accessible to all processors.
- This ideal parallel model is known as Parallel Random Access Machine (PRAM).
- All the processes work on the same clock but may execute different instructions in each cycle.
- The processors in PRAM share the same address space.

- As concurrent access to memory is permitted, PRAM can be divided in following four subclasses based on the patterns of memory access :
  - Exclusive-read, exclusive-write (EREW) PRAM
  - Concurrent-read, exclusive-write (CREW) PRAM
  - Exclusive-read, concurrent-write (ERCW) PRAM
  - Concurrent-read, concurrent-write (CRCW) PRAM
- These subclasses can be compared as follows :

PRAM Class	Memory access pattern Read/Write access	Additional features
EREW	• Access to a memory location is exclusive • Concurrent read or write operations are not allowed	Weakest PRAM model which affords minimum concurrency in memory access
CRCW	• Multiple read accesses to a memory location are allowed. • However, multiple write accesses to a memory location are arranged in a order.	
ERCW	• Multiple write accesses are allowed to a memory location, but multiple read accesses are arranged in a order .	
CRCW	• Multiple read and write accesses to a common memory location are allowed	Most powerful PRAM model

- Note that concurrent read access does not lead to any inconsistency in the program, but concurrent write access should be managed properly.
- Different types of protocols can be used to manage concurrent writes.
- Some of them are :

Name of the protocol	Common	Arbitrary	Priority	Sum
Features	Concurrent write is allowed if all the processors are attempting to write the same value	Arbitrary processor is chosen and allowed to write, rest of the processors will fail	Priority list of Processors is generated. Processor with highest priority can write, others will fail.	Sum of all quantities is written. This protocol can be further extended to any associative operator.

- To understand the practical difficulties in **architectural complexity of the ideal model**, consider the example of EREW PRAM with shared memory model, having p processors and shared global memory of m words.
- Set of switches connect processors to memory.

- As the processors share the memory, each of p processors can access any memory word from the global memory.
- Note that the word cannot be access by more than one processor at a time.
- To accomplish this the total number of switches should be Θ (mp), which is not practically possible as it is very expensive to construct such a network.
- Due to this constraint it is not possible to implement the PRAM models in practice.

1.5.2 Interconnection Networks for Parallel Computers

- In parallel computers, data transfer between processors and memory modules is provided by establishing interconnection network.
- Typically a interconnection network consists of n inputs and m outputs as shown in Fig. 1.5.1 (a).

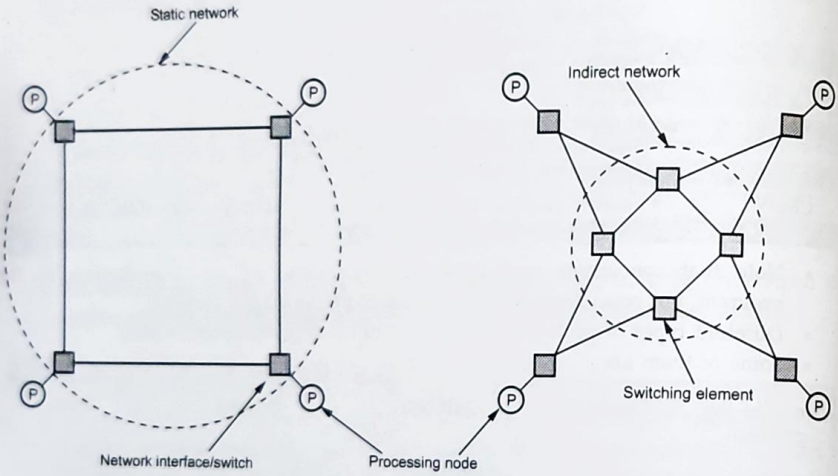


Fig. 1.5.1 Classification of interconnection networks : (a) a static network; and (b) a dynamic network

- Interconnection networks are built using links and switches.
- A link is a physical media such as a set of wires or fibers capable of carrying information.
- Note that if link is formed by conducting medium, the capacitive coupling between wires limits the speed of signal propagation, here capacitive coupling depends on length of the link.

- Two types of interconnection networks can be established :
  - **Static or direct network** : The network contain point-to-point communication links among processors. Fig. 1.5.1 (a) shows a simple static network of four processors.
  - **Dynamic or indirect network** : The network is built using switches and communication links. To provide the path between processors and memory modules, the links can be connected to each other dynamically. Fig. 1.5.1 (b) shows a dynamic network of four processors connected via a network of switches to other processors.
- Following are some characteristics of a switch :
  - Switches provide support for
    - Internal buffering
    - Routing
    - Multicast
- Mapping input to output ports is the basic functionality provided by a switch.
- This mapping is provided by the mechanisms like :
  - Crossbars
  - Multi port memories
  - Multiplexor-Demultiplexors
  - Multiplexed buses
- Cost of the mapping hardware (which typically grows as the square of the degree of the switch) , the peripheral hardware (grows linearly as the degree) and packaging costs (grows linearly as the number of pins) decide the total cost of a switch.
- The connectivity between the nodes and the network is provided by a network interface.
- The network interface has input and output ports to send the data into and out of the network, whose position is very important in the netwok.
- Network interface is responsible for the following tasks :
  - Packetizing data
  - Computing routing information
  - Buffering incoming and outgoing data
  - Error checking

1.5.3 Network Topologies

- Network topology refers to the physical or logical layout of a network.
- It defines the way different nodes are placed and interconnected with each other.
- Network topology can also describe how the data is transferred between these nodes.
- Interconnection network uses variety of network topologies
- Some of the topologies which will be explained in this section are :
  - Bus-Based Networks
  - Crossbar Networks
  - Multistage Networks
  - Completely Connected Networks
  - Star-Connected Networks
  - Linear-Arrays, Meshes and k-d meshes
  - Tree-Based Networks

1. Bus-Based Networks

- Bus topology is a specific kind of network topology in which all of the various devices in the network are connected to a single cable or line.
- It is a simplest topology, containing a shared medium which is common to all the nodes.

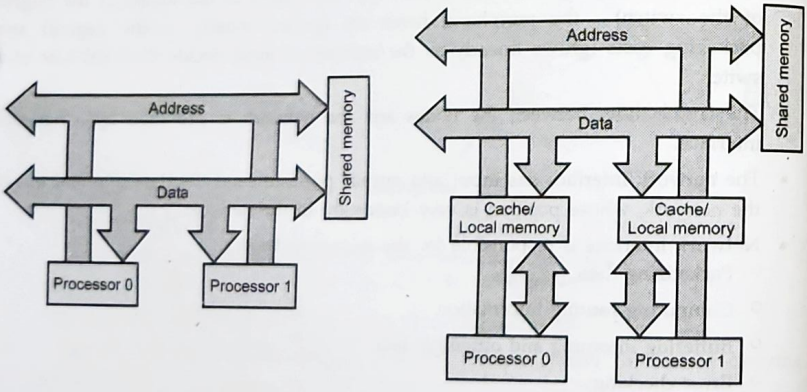


Fig. 1.5.2 Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

- The cost of bus based network increases as the number of nodes in the network increase.
- The cost also depends on the bus interfaces.
- By making use of bus topology,the information can be broadcasted effectively among nodes.
- Some of the limitations of bus topology are :
  - The overall performance of the network is restricted by the limited bandwidth associated with the bus structure.
  - Scalability is the problem with bus topology as nodes cannot be added dynamically without the availability of the physical resources.
- If the data to be accessed is local to the node then cache memory can be provided with each node, by this arrangement the bus bandwidth can be utilized properly, as bus will be used for accessing remote data only as shown in Fig. 1.5.2.
- Examples of bus based structures are : **Sun Enterprise servers** and **Intel Pentium based shared-bus multiprocessors.**

2. Cross Bar Networks

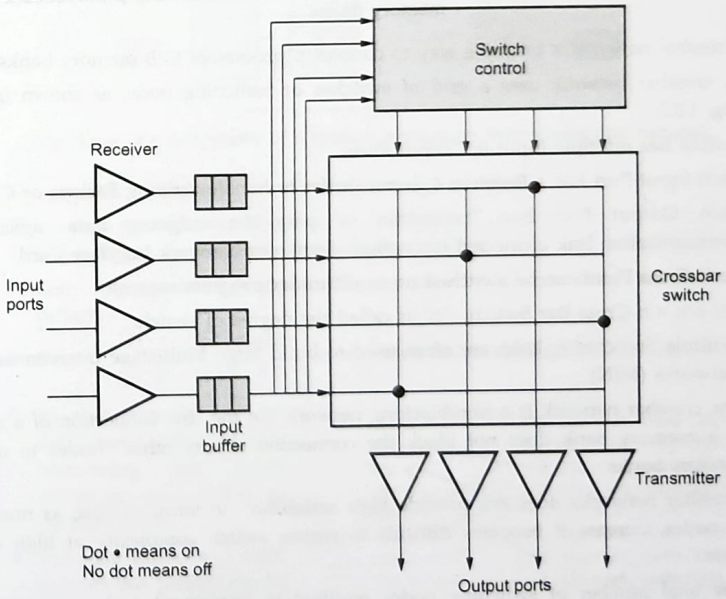


Fig. 1.5.3 (a) A four port crossbar switch providing connection between 4 inputs and 4 outputs

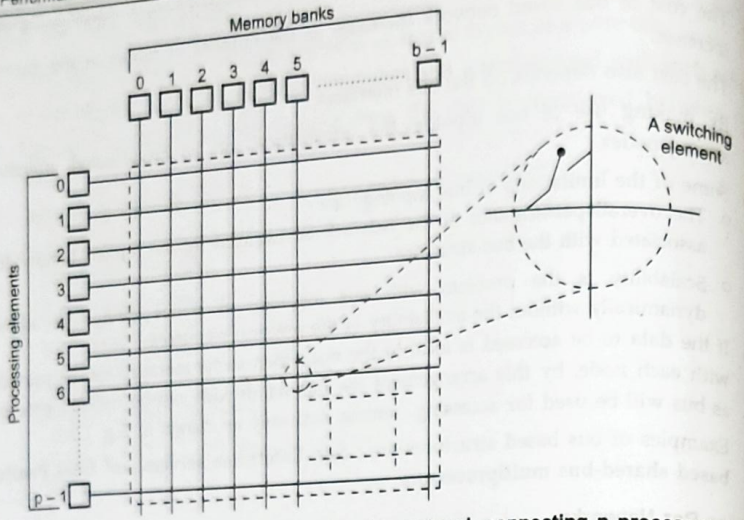


Fig. 1.5.3 (b) A completely non-blocking crossbar network connecting p processors to b memory banks.

- Crossbar network is a simple way to connect p processors to b memory banks.
- A crossbar network uses a grid of switches or switching node, as shown in the Fig. 1.5.3.
- Switch has multiple Input & Output Ports
- Each Input Port has a Receiver & Input Buffer to handle arriving Packets or Cells
- Each Output Port has Transmitter to pass the outgoing data signal to communication link connected to another Switch or Network Interface Card.
- Each Cross Point can be switched on or off under program control.
- For a n × n Cross Bar Switch, "n" is called the degree of switch.
- Multiple Switches & links are often used to build large Multistage Interconnection Networks (MIN)
- The crossbar network is a non-blocking network i.e the the connection of a node to a memory bank does not block the connection of any other nodes to other memory banks.
- Crossbar networks does not provide high scalability in terms of cost, as number of nodes increase it becomes difficult to realize switch complexity at high data rates.
- The total number of switching nodes required to implement such a network is $\Theta(p * b)$.

3. Multistage Networks

- Among the two networks discussed above, shared bus structure can be scaled in terms of cost but gives restricted performance, whereas crossbar network is scalable in terms of performance but unscalable in terms of cost.
- The intermediate network called as multistage network provides scalability in terms of performance as compared to bus structure and in terms of cost as compared with crossbar network.
- As shown in Fig. 1.5.4, a multistage network consists of p processors and b memory modules.

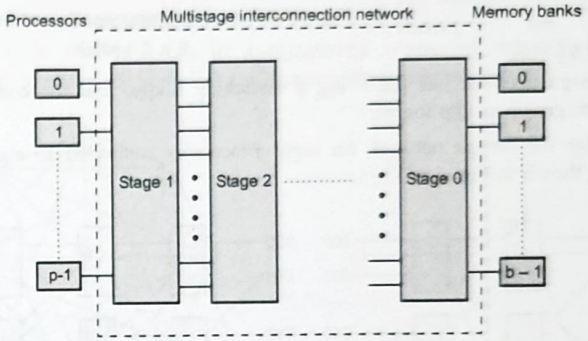


Fig. 1.5.4 The schematic of a typical multistage interconnection network

- The popular multistage network is the **omega network.**
- The omega network consists of :
  - Log p stages, p = Number of inputs and outputs.
  - Each stage connects p inputs to p outputs by means of an interconnection pattern.
  - If $j = \begin{cases} 2i & 0 \le i \le p/2-1 \\ 2i+1-p, & p/2 \le i \le p-1 \end{cases}$ is true then link is established between i and j.
  - By this equation we get the binary value of j,by performing left rotate operation on i, called as perfect shuffle.
  - For eight inputs and eight outputs this pattern can be calculated as shown in Fig. 1.5.5.

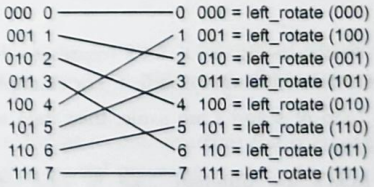


Fig. 1.5.5 A perfect shuffle interconnection for eight inputs and outputs.

- The number of switches will be p/2.
- There are two possible connection modes of a switch :
  - Pass-through connection : In this the inputs are sent straight through to the outputs,through a switch. (Fig. 1.5.6 (a))
  - Cross-over connection : In this the inputs are crossed over and sent out of the switch (Fig. 1.5.6 (b)).

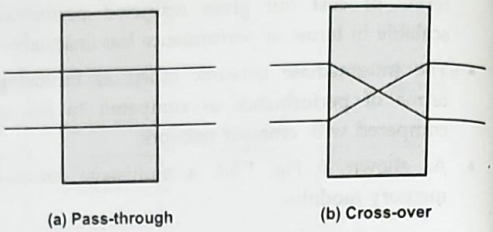


Fig. 1.5.6 Two switching configurations of the 2 x 2 switch

- An omega network has p/2 x log p switching nodes, and the cost of such a network grows as Q(p log p).
- Consider the omega network for eight processors connected to eight memory banks, shown in Fig. 1.5.7.

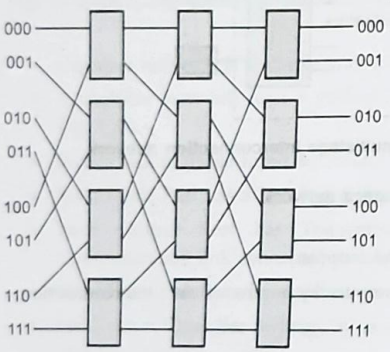


Fig. 1.5.7 A complete omega network connecting eight inputs and eight outputs.

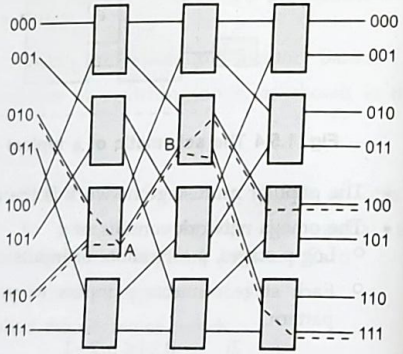


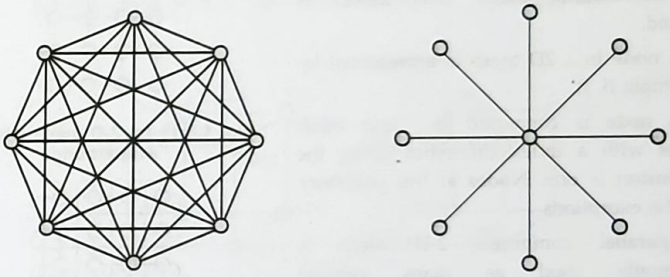
Fig. 1.5.8 An example of blocking in omega network : one of the messages (010 to 111 or 110 to 100) is blocked at link AB

- As shown in Fig. 1.5.8, let s (represented in binary) is the processor that wants to write data to memory bank t. For example consider s = 110(six) and t = 100(four)
- As MSB of s and t are same, then data will be sent in pass through mode by the switch.
- If MSB's are different ,like in case of data routing from node 010(two) to 111(seven) then crossover mode is chosen.

- In the next switching stage same pattern of communication is followed using next MSB.
- To traverse log p stages all log p bits are used.
- Now if processor 010 (two) wants to communicate to memory bank 111(seven), the path from processor 110(six) to memory bank 100(four) is blocked, as it uses the same communication link AB.
- It shows that, in an omega network, access to a memory bank by a processor may disallow access to another memory bank by another processor. Networks with this property are referred to as **blocking networks**.

4. Completely - Connected Network

- As shown in Fig. 1.5.9 (a), in a completely connected network each node is connected to every other node by a direct communication link.



(a) A completely-connected network of eight nodes

(b) A star connected network of nine nodes

Fig. 1.5.9 Examples of connected networks

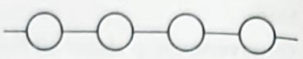
- A message can be sent from one node to another in a single step, due to communication link between them.
- The communication happens independently between the pair of nodes so there will not be blocking of communication of other pairs.

5. Star-Connected Network

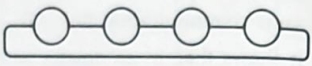
- As shown in Fig. 1.5.9 (b), unlike completely connected network, communication links are established between one central processor and every other processor in a network.
- It is similar to bus-based networks, as similar to bus structure the communication between the nodes will happen through a central processor.
- At times central processor can prove bottleneck in star topology.

6. Linear Arrays, Meshes, and k-d Meshes

- A **linear array** as shown in Fig. 1.5.10 (a) is a static network in which each node has two neighbors, one each to its left and right.
- In a linear array the start and end node does not have the connection. If this connection is established, which is called as wrap around connection, then the structure which is formed is called as a **ring** or **1-D torus** (Fig. 1.5.10 (b))
- As shown in Fig. 1.5.11 (a), if linear array is extended to two dimensions **two-dimensional mesh (2-D mesh)** is formed.
- Each node in a 2D mesh is represented by two-tuple (i, j).
- Each node is connected to four other nodes with a index difference along the dimension is one. Nodes at the periphery are the exceptions.
- In parallel computers 2-D mesh is commonly used as many parallel computations map naturally to 2D mesh.
- If wraparound links are established between the periphery nodes 2 D meshes form **two dimensional tori** (Fig. 1.5.11 (b))
- If 2D mesh is extended to three dimensions, 3-D cube (Fig. 1.5.11 (c)) is formed.
- In 3D cube each node is connected to 6 other nodes,two along each of the three dimensions.
- 3 - D cubes are also used widely in parallel machines like Cray T3E, as they map directly to some real life applications like 3 - D weather modeling, structural modeling, etc.



(a) with no wraparound links



(b) with wraparound link

Fig. 1.5.10 Linear arrays

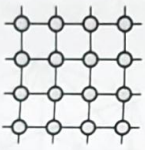


Fig. 1.5.11 (a) 2-D mesh with no wraparound

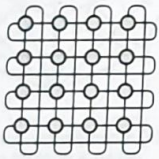


Fig. 1.5.11 (b) 2-D mesh with wraparound link (2-D torus)

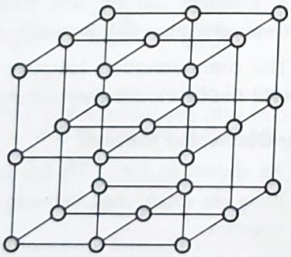
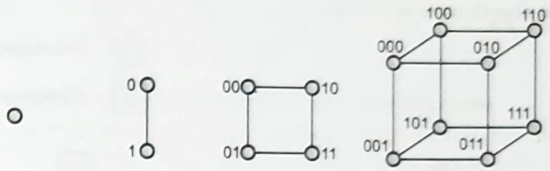
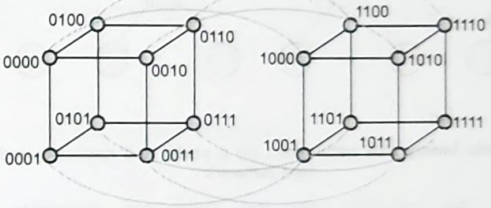


Fig. 1.5.11 (c) a 3-D mesh with no wraparound

- If we generalize the mesh structure, a class of topologies called as **k - d meshes** is formed.
- In k - d meshes, d represents the number of dimensions and k is the number of nodes along each dimension.
- One extreme of k - d meshes is a linear array and other extreme is called as **hypercube** (Fig. 1.5.12).



0-D hypercube **1-D hypercube** **2-D hypercube** **3-D hypercube**



4-D hypercube

Fig. 1.5.12 Construction of hypercubes from hypercubes of lower dimension.

- The hypercube has log p dimensions with two nodes along each dimension.
- A zero-dimensional hypercube consists of one node.
- A one-dimensional hypercube is constructed from two zero-dimensional hypercubes by connecting them.
- A two-dimensional hypercube of four nodes is constructed from two one dimensional hypercubes by connecting corresponding nodes.
- In general a d-dimensional hypercube is constructed by connecting corresponding nodes of two (d – 1) dimensional hypercubes.
- A 4-D hypercube contain 16 nodes as shown in Fig. 1.5.12.
- It is useful to derive a numbering scheme for nodes in a hypercube.

- To understand the numbering system in hypercube, consider the example of two nodes labeled as 0110 and 0101. These two nodes differ by two bit positions as they are two links apart.

7. Tree-Based Networks

- In tree based networks there will be a single path between any pair of nodes.
- As shown in Fig. 1.5.13 (a) ,in static tree network there will be a processing element at each node of the tree.

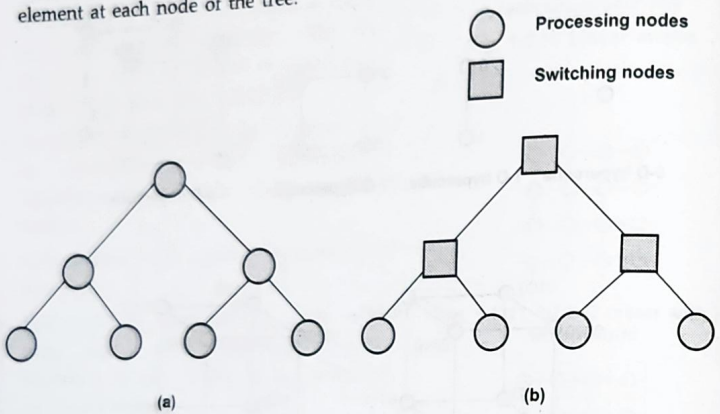


Fig. 1.5.13 Complete binary tree networks : (a) a static tree network; and (b) a dynamic tree network

- In dynamic tree network, only leaf nodes are processing elements, and intermediate levels are formed by switching nodes. (Fig. 1.5.13 (b))
- The communication will take place by the following process in the tree :
  - The source node first sends the message up the tree.
  - This process will be continued till the message reaches a smallest subtree which contains both source and destination node.
  - The message will be then sent to the destination node.
- Consider the case when many nodes in the left subtree of a node communicate with nodes in the right subtree.
- As per the scheme root node has to handle all the messages, which leads to the bottleneck in the network.
- Fig. 1.5.14 shows a tree network of 16 processing nodes. This network is called as **a fat tree network.**

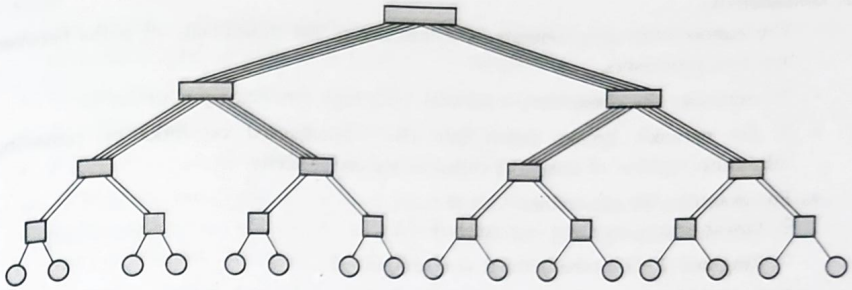


Fig. 1.5.14 A fat tree network of 16 processing nodes

- In case of dynamic tree networks this problem can be resolved by increasing the number of communication links. Also switching nodes can be kept closer to the root.

1.5.4 Evaluating Static Interconnection Networks

- The cost and performance of static interconnection network depends on various criterias like :
  - Diameter
  - Connectivity
  - Bisection Width and Bisection Bandwidth
  - Cost

1. Diameter

- The maximum distance between any two processors in the network is called as diameter of the network.
- The diameter of :
  - Completely-connected network = 1
  - Star-connected network = 2
  - Ring network = $\lfloor p/2 \rfloor$
  - 2D mesh without wrap around connections(for the two nodes at diagonally opposed corners) = $2(\sqrt{p}-1)$.
  - 2D mesh with wrap around connections = $2\lfloor \sqrt{p}/2 \rfloor$.
  - Hypercube connected network = $\log p$.
  - A complete binary tree = $2\log((p+1)/2)$.

2. Connectivity

- The connectivity of a network is a measure of the multiplicity of paths between any two processors.
- To minimize the contention, a network with high connectivity is desirable.
- If the network breaks down into two disconnected networks by removing minimum number of arcs, it is called as **arc connectivity.**
- For example, the arc connectivity of :
  - Linear arrays, tree and star network = 1
  - Ring and 2 - D meshe without wraparound = 2
  - 2 - D wraparound mesh = 4
  - d-dimensional hypercube = d

3. Bisection Width and Bisection Bandwidth

- The minimum number of communication links that must be removed to divide the network in two equal parts is called as **bisection width of a network.**
- The bisection width of :
  - Ring = 2
  - 2D p node mesh without wraparound connection = $\sqrt{p}$
  - 2D p node mesh with wraparound connection = $2\sqrt{p}$
  - Tree and Star = 1
  - Completely connected network of p nodes = $p^2/4$
  - D-dimensional Hypercube = p/2
- The number of bits that can be communicated simultaneously over a link connecting two nodes is called the **channel width.**
- Channel width is equal to the number of physical wires in each communication link.
- The peak rate at which a single physical wire can deliver bits is called the **channel rate.**
- The peak rate at which data can be communicated between the ends of a communication link is called **channel bandwidth.**
- Channel bandwidth is the product of channel rate and channel width.
- Minimum volume of communication allowed between any two halves of the network is called as **bisection bandwidth or cross section bandwidth.**
- Bisection bandwidth = bisection width * channel bandwidth.

4. Cost

- The number of communication links or the number of wires required by the network is used to evaluate the cost of a network.
- Linear arrays and trees use only p - 1 links to connect p nodes.
- A d-dimensional wraparound mesh has dp links.
- A hypercube-connected network has (p log p)/2 links.
- As bisection bandwidth provides a lower bound on the area in a two -dimensional packaging or the volume in a three-dimensional packaging, it can be used to measure the cost of a network.
- If bisection width = w, lower bound on the area in a two dimensional packaging $= \Theta(w^2)$ and the volume in a three-dimensional packaging = $\Theta(w^{3/2})$.
- Based on this criteria it is observed that hypercubes and completely connected networks are more expensive than the other networks.
- Table 1.5.1 enlists different cost performance characteristics of various static networks.

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely - connected	1	$p^2/4$	p – 1	p (p – 1)/2
Star	2	1	1	p – 1
Complete binary tree	$2\log((p+1)/2)$	1	1	p – 1
Linear array	p – 1	1	1	p – 1
2 - D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2 - D wraparound mesh	$2[\sqrt{p}/2]$	$2\sqrt{p}$	4	2p
Hypercube	log p	p/2	log p	(p log p)/2
Wraparound K - arry d - cube	d [k/2]	$2k^{d-1}$	2d	dp

Table 1.5.1 A summary of the characteristics of various static network topologies connecting p nodes.

1.5.5 Evaluating Dynamic Interconnection Networks

- In dynamic interconnection network, as the links connecting any two nodes are decided dynamically, so overhead is incurred by every message routed through a switch.
- Therefore in addition to the processing nodes each switch must also be considered as as a node in the network.

- With this context diameter of the network is the maximum distance between any two nodes in the network.
- The connectivity of a dynamic network can be defined in terms of node or edge connectivity.
- The node connectivity is the minimum number of nodes that must be removed from the network, to divide the network into two parts.
- The arc connectivity of the network can be defined as the minimum number of edges that must be removed from the network to divide the network into two unreachable parts.
- The bisection width partition the p nodes into two equal parts.
- The bisection bandwidth can also be considered as the minimum number of edges crossing the partition or it can also be considered as the minimum number of edges to be removed from the network to partition it into two equal parts with same number of nodes.
- For example, As shown in the Fig. 1.5.15, there can be three bisections A,B and C, partitioning the network into two groups of two processing nodes each.
- As each partition consists of four nodes, so the bisection width of network is four.

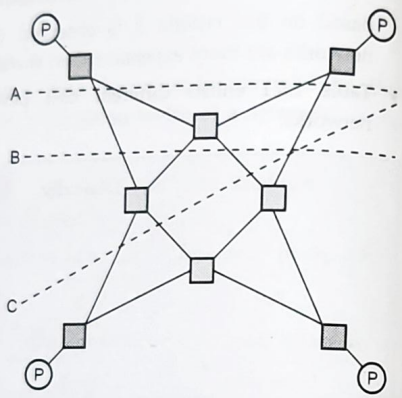


Fig. 1.5.15 Bisection width of a dynamic network is computed by examining various equi-partitions of the processing nodes and selecting the minimum number of edges crossing the partition. In this case, each partition yields an edge cut of four.

- Therefore, the bisection width of this graph is four.
- The cost of dynamic network is sum of link cost and switch cost.
- As in a dynamic network, degree of a switch is constant, number of links and switches is same.
- So cost of dynamic networks is determined by number of switching nodes in the network.

1.5.6 Cache Coherence in Multiprocessor Systems

- The cache coherency problem refers to inconsistency of distributed cached copies of the same cache line addressed from the shared memory.
- In shared address space machines it is very critical to maintain consistency in the copies of data in cache memory as well as in the shared memory.
- To maintain cache coherence additional hardware and protocols are needed.
- In multiprocessor architecture multiple processors modify the cache copies, making it complex to maintain the consistency between them.
- Let's consider the example shown in Fig. 1.5.16 to understand the various protocols used to maintain cache coherency in shared address space machines.
- As shown in the Fig. 1.5.16 two processors P0 and P1 share the global memory.

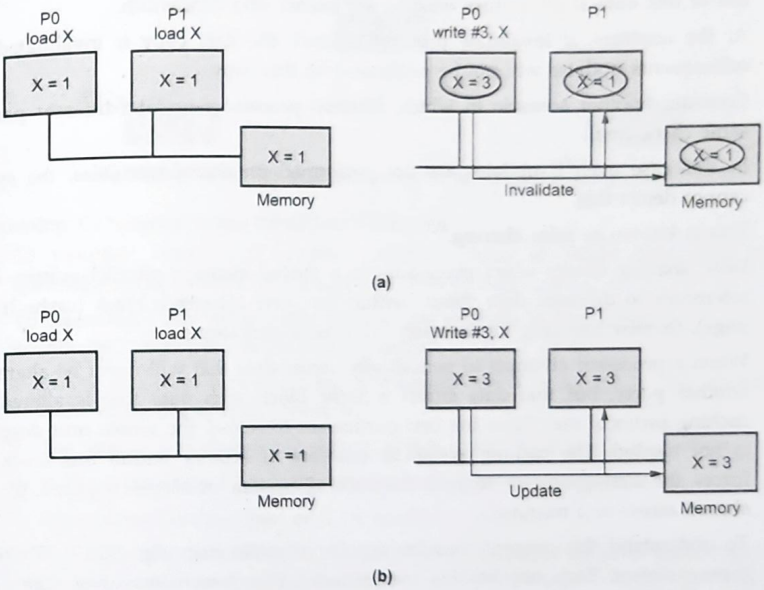


Fig. 1.5.16 Cache coherence in multiprocessor systems :

(a) Invalidate protocol (b) Update protocol for shared variables

- Both the processors execute load x instruction which fetches the the variable x from main memory and keep the copies of variable X in the local cache of each processor.
- Now there are three copies of variable X .

- To maintain the consistency in cache as well as global memory, when any processor attempts to modify the value of variable X, two actions can be taken :
  - All the copies of variable X present with all the other processors as well as global memory must be marked and considered as invalid. This protocol is known as **invalidate protocol.**

    OR
  - All the copies of variable X present with all the other processors as well as global memory must be updated. This protocol is known as **update protocol.**
- If one of these actions is not taken then other processors will use incorrect copy of X for computation.
- In some cases, the update protocol may cause additional overhead in terms of latency and bandwidth, if a processor loads the data and never uses it again. In such situation every time the data is modified by some processor, it has to modified in all the cache copies of all the other processors which are not making use of this data at all, in turn wasting the latency and bandwidth.
- At the contrary, if invalidate protocol is used, the data copy is invalidated and subsequents updates will not be performed on this copy.
- Consider another scenario in which different processors update different parts of same cache line.
- In this case even if updates are not performed on shared variables, the system cannot detect this.
- This is known as **false sharing.**
- False sharing occurs when processors in a shared-memory parallel system make references to different data objects within the same coherence block (cache line or page), thereby inducing "unnecessary" coherence operations.
- When a processor attempts to periodically access data that will never be altered by another party, but that data shares a cache block with data that is altered, the caching protocol may force the first participant to reload the whole unit despite it is not needed. The caching system is unaware of activity within this block and forces the first processor to bear the caching system overhead required by true shared access of a resource.
- To understand this concept consider real life example from Fig. 1.5.17 : There are three painters. Each one has his own wooden board on which they paint, each board has three divisions , say division 1, division 2 and division 3. A painter can only paint one of these three divisions. When a painter paints one division of his wooden board, the other two boards must also be changed to reflect what the first painter has done.Here the wooden boards are analogous to cache blocks, painters are analogous to parallel threads and painting is analogous to write activity.
- Note that recent cache coherent machines rely on invalidate protocol.

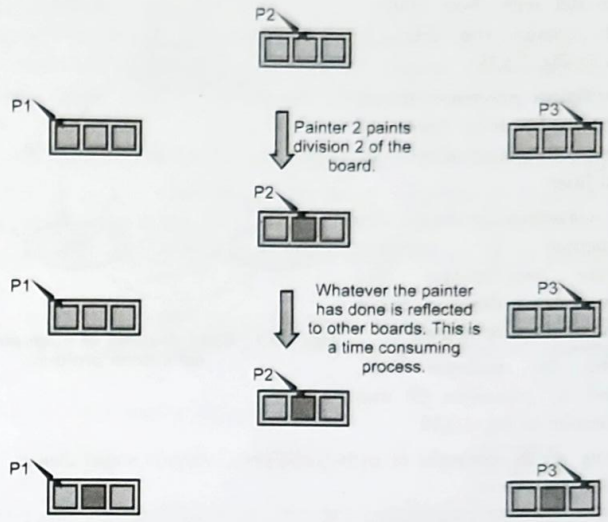


Fig. 1.5.17 Real life example to understand cache coherency

Maintaining Coherence using Invalidate Protocols

- To maintain consistency between multiple copies of single data item proper tracking should be done of number and state of these copies.
- To understand this process let's consider the example in Fig. 1.5.16.
  - Initially X is present in the shared memory.
  - In the first step, load operation is executed and X is loaded in the respective cache memories of each processor.
  - The state of X is changed to shared, as it is shared by multiple processors.
  - In the second step P0 executes a write instruction on this variable.
  - Immediately all the copies of X are marked as invalid.
  - The copy of X owned by P0 is marked as modified or dirty, to ensure that all subsequent accesses to this variable at other processors will be serviced by processor P0 and not from the memory.
  - Now if processor P1 execute second load operation on X, it will attempt to fetch X.
  - But as X was marked dirty by P0, this request is attended by P0.
  - Copies of X at global memory and with P1 are updated and X is again marked as shared.

- This model with three states - shared, invalid and dirty, is shown in Fig. 1.5.18.
- In the figure processor actions are shown by solid lines and coherence actions are shown by dashed lines.
- These coherency protocols are implemented by applying hardware mechanisms like snoopy systems, directory based systems or there combination
- Consider the example code, executed by processor P0 and P1 as shown in Fig. 1.5.19.

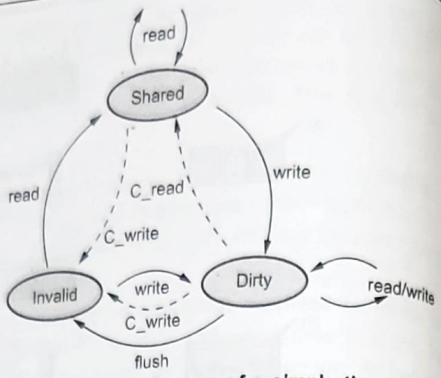


Fig. 1.5.18 State diagram of a simple three-state coherence protocol.

- Applying all the concepts of cache coherency, various states at each time step in execution are shown.

Time ↓	Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
					x = 5, D
					y = 12, D
	read x		x = 5, S		x = 5, S
		read y		y = 12, S	y = 12, S
	x = x + 1		x = 6, D		x = 5, I
		y = y + 1		y = 13, D	y = 12, I
	read y		y = 13, S	y = 13, S	y = 13, S
		read x	x = 6, S	x = 6, S	x = 6, S
	x = x + y		x = 19, D	x = 6, I	x = 6, I
		y = x + y	y = 13, I	y = 19, D	y = 13, I
	x = x + 1		x = 20, D		x = 6, I
		y = y + 1		y = 20, D	y = 13, I

Fig. 1.5.19 Example of parallel program execution with the simple three-state coherence protocol

Snoopy Cache Systems

- As discussed in section 1.5.6 the invalidate protocol invalidates all other cached copies when a local cached copy is updated whereas update protocol broadcasts the newly cached copy to update all other cached copies with the same line address.
- These cache coherency protocols are implemented by the use of snoopy buses.
- Bus snooping is a scheme that a coherency controller in a cache monitors or snoops the bus transactions and its goal is to maintain a cache coherency.
- Snoopy protocols require a broadcast mechanism, which can be provided by a bus or ring.
- The bus is designed such that it constantly monitor the caching events between processor and memory modules, so they are also called as **snoopy coherency protocols.**
- Fig. 1.5.20 shows a snoopy bus system.

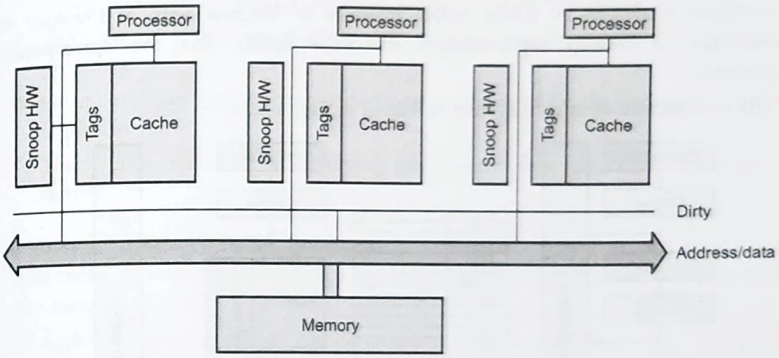


Fig. 1.5.20 A simple snoopy bus based cache coherence system.

- As shown in the Fig. 1.5.20 each processor's cache has a set of tag bits associated with it that determine the state of the cache blocks.
- The value of a tag bit depends on the coherence protocol state diagram.
- For example, if the snoop hardware detects that there is a read request to a cache block having a dirty copy, it puts the data out of the bus.
- If it detects the write operation request on the cache block that it has a copy of, it invalidates the block.
- Snoopy bus protocols are used extensively in commercial systems as they are simple and existing bus based systems can be upgraded to accommodate snoopy protocols.

- The advantage of snoopy protocols is as different processors work on different data items, once these items are marked as dirty all the operations are performed locally on the cache.
- The bottleneck for the snoopy protocol is the shared bus, with a finite bandwidth by wich only limited number of coherence operations can be carried out.
- In case of snoopy protocols all the memory operations performed by all the processors is broadcasted to all the other processors.
- Instead if of these if we keep track of processors having copies of different data items along with status of their state, then only the processors which must take part in the operations can be sent the coherency operations.
- Such an information can be stored in a directory.
- If coherency operations are based on such mechanism, then the system is called as directory based system.

Directory Based Systems

- Consider a system in which global memory is attached with a directory that maintains a bitmap representation of cache blocks and the corresponding processors.
- The architecture of such a system is shown in Fig. 1.5.21.

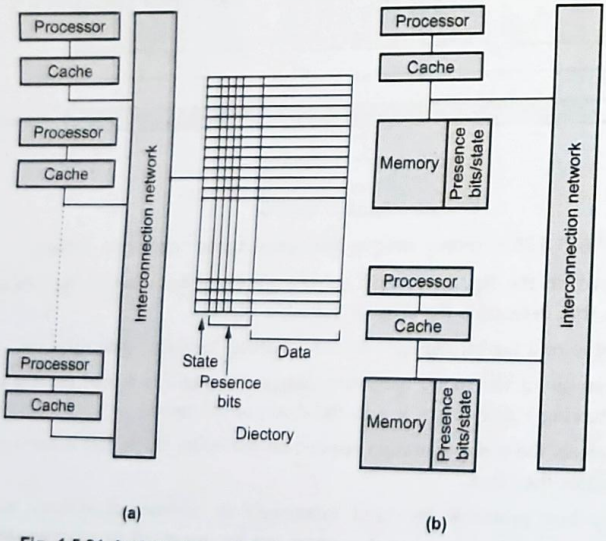


Fig. 1.5.21 Architecture of typical directory based systems : (a) a centralized directory; and (b) a distributed directory.

- These bitmap entries are called as presence bits.
- The performance of directory based system depends on the fact that only the processors holding a particular block can participate cache coherence transition operations.
- For example, consider the example in Fig. 1.5.16, the flow can be explained as below
  - Processors P0 and P1 access the block corresponding to variable x.
  - The state of the block will be changed to shared.
  - Presence bits are updated to indicate that processors P0 and P1 have shared the block.
  - When P0 executes store instruction, state in the directory is changed to dirty.
  - Presence bit of P1 will be resetted.
  - P0 performs all the operations on this variable locally.
  - If any processor attempts to read the value, directory notices the dirty tag.
  - Then the processor uses presence bits to direct the request to the appropriate processor.
  - P0 updates the block in memory and sends the block to the processor who has requested for it.
  - This will be reflected by modifying the presence bits and state transitions to shared.

Performance of Directory Based Schemes

- The cache coherency protocols are applied when multiple processors try to update the same data item.
- In such a situation overhead can be caused due to two factors :
  - Movement of the data between the processors and memory via bus structure
  - Sending status updates : invalidate or update (leads to communication overhead)
  - Generation of state information from the directory (leads to contention)
- The communication overhead includes : the number of processors requiring state updates and the algorithm for propagating state information.
- The contention overhead includes the limitations posed by the directory. Since the directory is in memory and memory can support only limited number of read and write operations in unit time, directory will provide limited parallel performance if large number of coherence actions are requested.
- If we consider the cost, as the number of processors increases the amount of memory required to store the directory becomes a bottleneck

- Consider m is the number of memory blocks and p is the number of processors, then directory size grows as $O(mp)$.

Distributed Directory Schemes

- In case of directory based cache coherence, directory becomes a central point of contention
- If each processor could maintain the coherence of it's own memory block then the task of maintaining the coherence is distributed among processors.
- This is the basic principle of a distributed directory system.
- As shown in Fig. 1.5.21 (b), each memory block will have an owner, whose location in directory is known to all processors
- When a processor attempts to read a block for the first time, it requests the owner for the block.
- The owner directs this request based on presence and state information available with it.
- When a processor writes into a memory block, it sends an invalidate to the owner, which in turn forwards the invalidate to all processors that have a cached copy of the block.
- By this contention caused due to central directory can be avoided.
- Note that the communication overhead associated with state update messages is not reduced.

Performance of Distributed Directory Schemes

- Distributed directories are more scalable than snoopy systems or centralized directory systems.
- Distributed directories allow $O(p)$ simultaneous coherence operations.
- In such systems the bottleneck is created by latency and bandwidth of a network.

Review Questions

1. *Describe the architecture of an ideal parallel computer.*
2. *Write a note on interconnection network for parallel computers.*
3. *Describe and differentiate between static and dynamic network.*
4. *Write a note on with suitable diagrams.*
   - *a) Bus based networks*
   - *b) Cross bar switch*
   - *c) Multistage networks*
   - *d) Completely connected networks*
   - *e) Star - connected networks*
   - *f) Linear arrays*
   - *g) Mesh network*
   - *h) Tree - based networks.*

5. *Discuss the parameters on which the performance of static network depends.*
6. *How the performance of dynamic network is evaluated ?*
7. *Describe in detail, the cache coherence in multiprocessor systems.*
8. *What is invalidate / update protocol ?*
9. *Discuss in detail, maintaining coherence using invalidate protocol.*
10. *Write a short note on snoopy cache systems.*
11. *Draw and explain the state transition diagram of a simple three state coherence protocol.*
12. *Explain in detail, the architecture of a directory based system.*
13. *Identify the overheads associated with directory based schemes.*
14. *Write a short note on distributed directory based schemes.*
15. *Compare the performance of snoopy, simple directory based and distributed directory based systems.*
16. *Describe the merits of Multi-threading over Pre-fetching techniques.* **SPPU : April-16, Marks 4**
17. *Explain memory hierarchy and thread organization.* **SPPU : April-18, Marks 4**
18. *Explain cache coherence in multiprocessor system.* **SPPU : Oct.-19, Marks 6**

1.6 Communication Costs in Parallel Machines

SPPU : Oct.-19

- Communication and computation are two important integer operations in a parallel program execution.
- Communication of the processing elements leads to major overhead in parallel programming.
- In general the cost of communication depends on the following features.
  - Programming modes semantic
  - Network topology
  - Data handling and routing
  - Software protocols associated to a program
- In this section the focus will be on various factors in separate as well as shared address space machines, that contributes to communis in parallel machines.

1.6.1 Message Passing Costs in Parallel Computers

- In distributed address space machines, arc nodes communicate with each other to exchange data and information, through message passing.
- This communication time between the nodes is characterized by sum of
  1) Time to prepare message for transmission.
  2) Time required by the message to traverse the network to its destination.

Following parameters determine the delay or latency in communication.

1) Startup time (t_s) :

- It is the time required to handle a message at sending and receiving nodes.
- Note that it is applicable for only one time single message transfer.
  1) Prepare a message by adding header, trailer and error correction information.
  2) To execute routing algorithm.
  3) to establish an interface between the local node and the routes.

2) Per-hop time (t_h) :

- It is a time taken by the header of a message to travel between two directly connected nodes in the network.
- Per hop time is also called as node latency.
- t_h depends on delay in the routing switch where as switch determines that message should be forwarded to which channel or buffer.

3) Per word transfer time (t_w) :

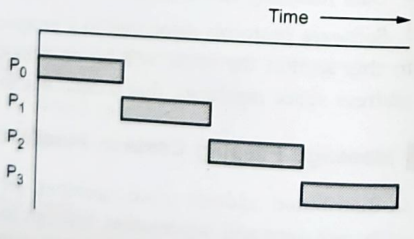
- It is the time required by each word to traverse the link.
- It r words can be transferred through a channel i.e. if channel bandwidth is r words/sec. then per word transfer time t_w is given as

$$t_w = \frac{1}{r}$$

- Network and buffering overheads are included in this time.
- In parallel machines typically two routing techniques are used :
  1. Store-and-forward
  2. Cut-through routing.

1.6.1.1 Store-and-Forward Routing

- In communication of a message between nodes, a message traverses a path with multiple links.
- As shown in the Fig. 1.6.1 (a), once a node receives and stores a complete message from the earlier node, it forwards the message to the next node.



(a) Single message sent over a store-and-forward network

Fig. 1.6.1 Passing a message from node P0 to P3 (a) through a store-and-forward communication

- Consider message of size m is traversing l links in a network.

- At each link t_h will be the time spent for hopping to the next node and $t_w m$ will be the time taken by the message to traverse the link.
- As number of links = l, total time taken by a message to travel between the nodes is $(t_h + t_w m) l$.
- Therefore in Store and forward routing, the total communication cost for a message of size m words to traverse l communication links is

$$t_{comm} = t_s + (mt_w + t_h) l$$

- In recent parallel computers, the per-hop time t_h is very small.
- Also it has been observed that is less than t_w for even small values of m and thus can be ignored.
- Catering to this, the communication cost further reduces to $t_{comm} = t_s + mlt_w$

1.6.1.2 Packet Routing

- In store and forward routing a message will be sent to the next node only when entier message has been received by the earlier node, so in this case communication resources are wasted.
- To overcome this drawback, packet routing mechanism is used in which the original message is broken into two equal sized parts before it is sent.
- As shown in the Fig. 1.6.1 (b), as soon as half of the original message is received by the node, the message is passed on to the next node.
- By this the communication time is reduced and there will be increase in the utilization of communication resources.
- As shown in Fig. 1.6.1 (c), this mechanism can be further enhanced by breaking the message into four parts.

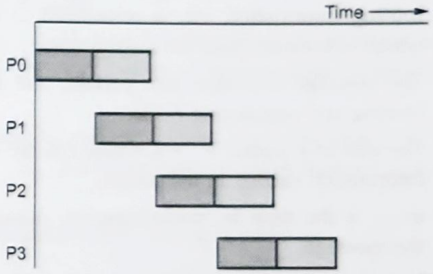


Fig. 1.6.1 (b) The same message broken into two parts and sent over the network

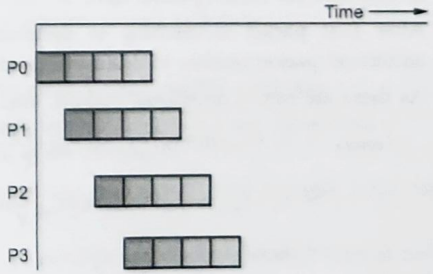


Fig. 1.6.1 (c) The same message broken into four parts and sent over the network

- The advantaged of this scheme is :
  1. Utilization of the resources is enhanced.
  2. Overhead due to packet loss are minimised.
  3. Possibility of packets taking different paths will increase.
  4. Error correction capability increases.
- All the above mentioned advantages makes this scheme suitable for long distance communication network like internet where error rates, number of hops, and variation in network state can be higher.
- The overhead associated with this scheme is : each packet must carry routing, error correction, and sequencing information.
- Packet routing is suitable to networks with highly dynamic states and higher error rates, such as local- and wide-area networks as individual packets may take different routes and retransmissions can be localized to lost packets.
- Consider the example of transferring m word message through the network, where it is assumed that all the packets are taking the same path.
- The time taken for programming the network interfaces and computing the routing information, etc. is considered to be independent of the message length, which is included into the startup time t_s of the message.
- The message is broken into packets, and packets are assembled with their error, routing and sequencing fields.
- The size of a packet = r + s, where r is the original message and s is the additional information carried in the packet.
- mt_{w1} is the time for packetizing the message, which is proportional to length of the message.
- Let's consider that network communicates one word in every t_{w2} second, with delay of t_h per hop and if first packet traverses l hops.
- In this case the time a packet takes to reach to destination is $t_h l + t_{w2}(r+s)$.
- After first packet is reaching to destination, in every next $t_{w2}(r+s)$ seconds, additional packet reaches to destination node.
- As there are $m/r - 1$ additional packets, total communication time is given as :

$$t_{comm} = t_s + t_{w1} m + t_h l + t_{w2}(r+s) + \left(\frac{m}{r} - 1\right) t_{w2}(r+s)$$

$$= t_s + t_{w1} m + t_h l + t_{w2} m + t_{w2} \frac{s}{r} m$$

$$= t_s + t_h l + t_w m$$

where, $t_w = t_{w1} + t_{w2}\left(1 + \frac{s}{r}\right)$

1.6.1.3 Cut-Through Routing

- In parallel machines, the overhead of transmission in the packet switching in communication networks can be reduced by :
  - Forcing all packets to take the same path, by this the sequencing information can be eliminated.
  - Inclusion of error information at message level rather than packet level, the overhead associated with error detection and correction can be reduced.
  - As error rates in interconnection networks are very low, instead of expensive error detection schemes a simple one can be used.
- Cut through routing scheme takes care of all the above mentioned factors.
- The message is broken into fixed size units called **flow control digits** or **flits.**
- Flits are smaller than packets.
- To establish a connection a tracer is first sent from the source to the destination node.
- After establishing the connection flits are sent one by one on the same path.
- As soon as a flit is received at an intermediate node,it is passed on to the next node without waiting for the entire message.
- It is not necessary that each node should have a buffer space to store the entire message, resulting in less memory and bandwidth at intermediate nodes, which makes it faster.
- Consider the m word long message traversing l links, per hop time = t_h, the header of the message takes time = lt_h, the entire message takes time = $t_w m$, after the header arrives.
- So the total communication time for cut-through routing is given as :

$$t_{comm} = t_s + lt_h + t_w m$$

- The communication time for store and forward scheme and cut-through routing scheme is similar if :
  - l = 1 i.e. Communication happens between nearest neighbors.
  - Message size is small .
- Cut through routing is supported by most current parallel machines.
- For deciding the size of flit following network parameters are considered :
  - The control circuitry must operate at the flit rate.
  - If a very small flit size is formed, the required flit rate becomes large for a given link bandwidth.
  - If flit sizes become large latency of message transfer increases as internal buffer size increase.

- Most current parallel computers and many local area networks support cut-through routing.
- In recent cut-through interconnection networks flit size range from four bits to 32 bytes.
- In some parallel models the latency of message is very important.
- Sometimes a long message traversing a link can hold the short message.
- This issue can be solved by making use of multiline cut through routing, in which a physical channel is split into number of virtual channels.
- Let's understand how deadlock can occur in cut through routing.
- Consider the example shown in Fig. 1.6.2.

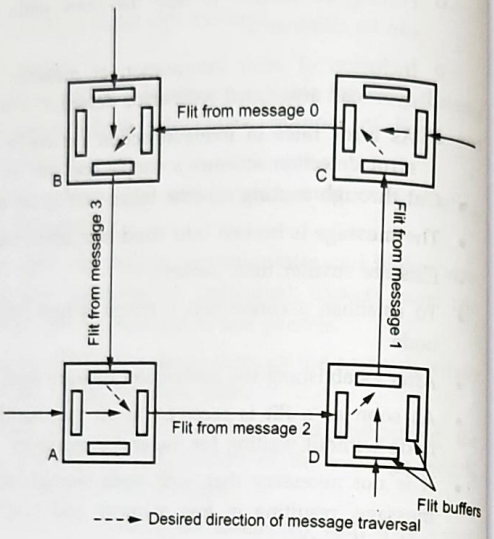


Fig. 1.6.2 An example of deadlock in a cut-through routing network.

- Note that if the message has to use the link which some other processor is using then the message is blocked, resulting in a deadlock.
- As shown in the diagram destinations of messages 0,1,2 and 3 are A,B,C and D respectively.
- A flit from message 0 occupies the link CB . However, since link BA is occupied by a flit from message 3, the flit from message 0 is blocked.
- In this case no message can move further and deadlock is caused.

1.6.1.4 A Simplified Cost Model for Communicating Messages

- To communicate a message between two nodes l hops away using cut-through routing, the total cost needed is :

$$t_{comm} = t_s + lt_h + t_w m$$

- To optimize the cost, following points are to be taken care of :
  - **Communicate in bulk :**
    - Combine small messages into a single large message to reduce the startup cost t_s.

- This is beneficial for the parallel platforms like message passing machines and clusters in which t_s is much more than t_h or t_w.

- **Minimize the volume of data :**
  - If volume of the data is minimized then the cost of per word transfer time t_w can be reduced.
- **Minimize distance of data transfer :**
  - Minimize the number of hops l that a message must traverse.

- To minimize the distance of data transfer is sometimes difficult due to following reasons :
  - In case of message passing standards like MPI, mapping of the processes onto actual physical processors is a challenge, as programmer is unaware of and does not have any control on this mapping.
  - In case of architectures which follows two step routing, the message is sent from source node to destination via intermediate node. So it is not beneficial if the number of hops are minimized.
  - The per-hop time (t_h) is dominated by :
  - The startup latency (t_s)for small messages
  - Per-word component (t_wm) for large messages.
  - Since the maximum number of hops (l) in many networks is relatively small, the per-hop time can be ignored.
- Taking into consideration all the above points message transfer cost between two nodes is given as :

$$t_{comm} = t_s + t_w m$$

- Note that communication cost depends on architecture as well as the communication pattern.
- For communication patterns that do not congest the network, the effective bandwidth is identical to the link bandwidth.
- For communication operations that congest the network, the effective bandwidth is the link bandwidth scaled down by the degree of congestion on the most congested link.

1.6.2 Communication Costs in Shared - Address - Space Machines

- To achieve the efficiency in terms of cost is a difficult task in shared address space machines because of following factors :
  - Memory layout is determined by the system, so it is difficult to know the local and remote accesses.Thus if access times for local and remote data is different cost varies depending on data layout.

- Finite cache sizes can result in cache thrashing. If the size of the portion of data needed for computation is greater than the available cache, certain portion may get overwritten and accessed many times, degrading the performance due to increased problem size. It proves costly in shared address space machines as each cache miss may involve coherence operations and interprocessor communication.
- In case of cache coherency operations overheads can prove costly as in case of invalidate protocol the data item must pay a remote access latency cost again in case of the read operation after invalidation. In case of update operation, programmer does not have control on the number of copies of a data item and the schedule of instruction execution.
- Spatial locality is difficult to model as cache line are longer than words and there are variations in the latency of words.
- By making use of prefetching techniques, the compiler can prefetch the loads in turn reducing the cost incurred in the overhead associated with data access. Programmer does not have control on this. But it is difficult to achieve as it depends on the compiler, program and available resources.
- To minimize the overhead caused by false sharing, the programmer must use the data structures used by various processors to minimize it.
- The overhead caused due to contention in shared accesses will affect the performance of shared address space machines. It depends on execution schedule and difficult to model.

Review Questions

1. *Write a short note on communication costs in parallel computing.*
2. *Define the parameters that determine the delay / latency in communication.*
3. *Describe briefly*
   a) Startup time b) Per hop time c) per word transfer time.
4. *What is store and forward routing ?*
5. *What is packet routing ?*
6. *Define cut - through routing.*
7. *Discuss the simplified cost model for message communication.*
8. *What are the communication costs in the shared - address space machines ?*
9. *Describe in detail, the scalable design principles.*
10. *Discuss the applications that have incorporated independence principle.*
11. *Explain Store - and - Forward and packet routing with its communication cost.*

SPPU : May-19, Marks 6

1.7 Models

SPPU : Oct.-19

- HPC architecture can be implemented using different models by various organizations based on their requirements.
- HPC aims at providing computing infrastructures capable of fulfilling the increasing performance requirements of modern applications

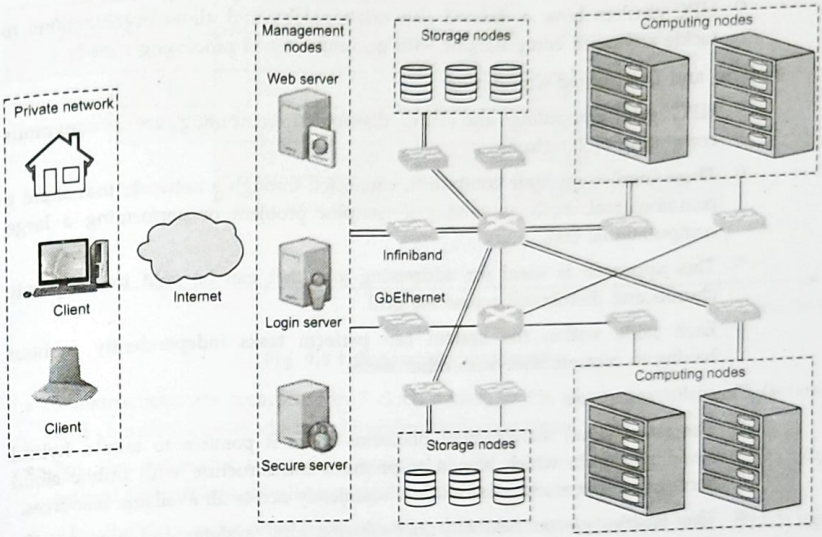


Fig. 1.7.1 Example of HPC model

- The popular HPC models are listed below
  1) Parallel Computing Across Multiple Architectures
     - Parallel computing allows HPC clusters to execute large workloads and splits them into separate computational tasks that are carried out at the same time.
     - These systems can be designed to either scale up or scale out.
     - Scale-up designs involve taking a job within a single system and breaking it up so that individual cores can perform the work, using as much of the server as possible.
     - In contrast, scale-out designs involve taking that same job, splitting it into manageable parts, and distributing those parts to multiple servers or computers with all work performed in parallel.

2) Cluster Computing

- High performance computing clusters link multiple computers, or nodes, through a Local Area Network (LAN).
- These interconnected nodes act as a single computer-one with cutting-edge computational power.
- HPC clusters are uniquely designed to solve one problem or execute one complex computational task by spanning it across the nodes in a system.
- HPC clusters have a defined network topology and allow organizations to tackle advanced computations with uncompromised processing speeds.

3) Grid and Distributed Computing

- HPC grid computing and HPC distributed computing are synonymous computing architectures.
- These involve multiple computers, connected through a network, that share a common goal, such as solving a complex problem or performing a large computational task.
- This approach is ideal for addressing jobs that can be split into separate chunks and distributed across the grid.
- Each node within the system can perform tasks independently without having to communicate with other nodes.

4) Cloud Infrastructure

- The latest cloud management platforms make it possible to take a hybrid cloud approach, which blends on-premises infrastructure with public cloud services so that workloads can flow seamlessly across all available resources.
- This enables greater flexibility in deploying HPC systems and how quickly they can scale up, along with the opportunity to optimize Total Cost of Ownership (TCO).
- Typically, an on-premises HPC system offers a lower TCO than the equivalent HPC system reserved 24/7 in the cloud.

Review Question

1. *What are types of dataflow execution model ?* **SPPU : Oct.-19, Marks 6**

1.8 Architectures : N-Wide Superscalar Architecture

SPPU : April-18, Dec.-19

- The combination of temporal parallelism (used in pipeline processing) and data parallelism by issuing several instructions simultaneously in each cycle, to improve the speed of a processor, is called as superscalar processing.

- In an n-issue superscalar, n instructions are fetched, decoded, executed and committed per cycle.
- As shown in Fig. 1.8.1 assume pipeline execution with 2 instructions is issued simultaneously.

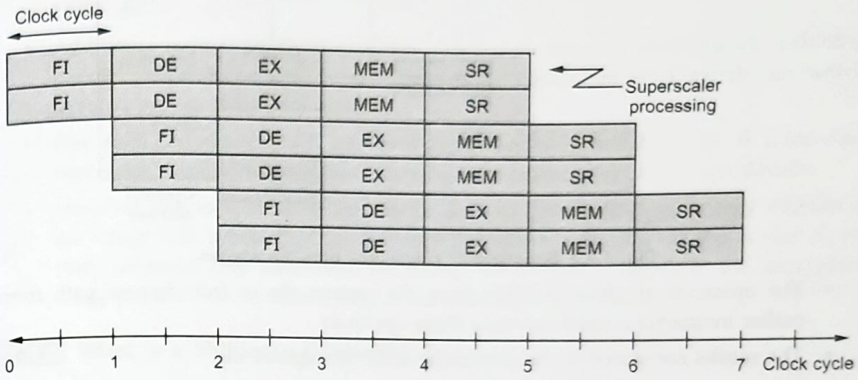


Fig. 1.8.1 Superscalar processing

- 6 instructions are completed in 7 clock cycles, in the steady state two instructions will completed every clock cycle under ideal conditions.
- For the successful superscalar processing the hardware should permit fetching several instructions simultaneously from the instruction memory.
- Also the data cache must have several independent ports for read/write which can be used simultaneously.
- If the instruction is a 32 bit instruction and we fetch 2 instructions, 64 bit data path from memory is required and 2 instruction registers are needed.
- Executing multiple instructions simultaneously would require multiple execution units to avoid resource conflicts.
- A block diagram shown in Fig. 1.8.2 shows the pipeline stages of superscalar processor.
- The fetch unit fetches several instructions in each clock cycle from the instruction cache.
- The decode unit decodes the instructions and renames registers so that false dependencies are eliminated and only true dependencies remain.
- The instructions are sent to an instruction buffer where they are kept temporarily till the source operands are identified and become available.
- Ready instructions are dispatched based on the availability of functional units.

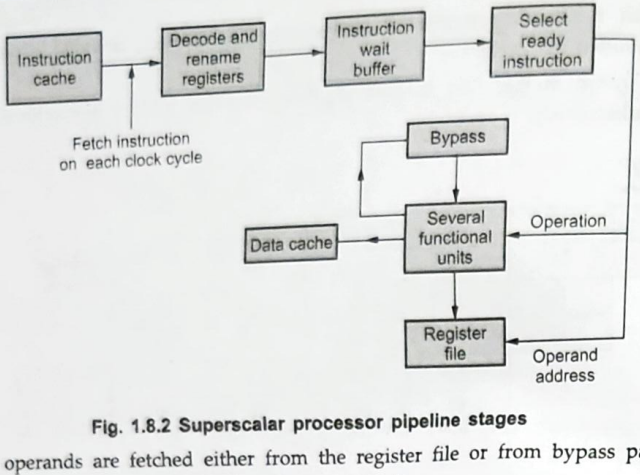


Fig. 1.8.2 Superscalar processor pipeline stages

- The operands are fetched either from the register file or from bypass path from earlier instructions which produce these operands.
- The results are stored in the data cache or in the register file.
- Superscalar processing is based on the available parallelism in groups of instructions of programs.
- If available parallelism is more than several instructions can be issued in the same cycle.
- It has been observed that processors can issue 4 to 5 instructions in each cycle.

Review Questions

1. *What is n - wide superscalar architecture ?*
2. *Explain N wide superscalar architecture in detail.* **SPPU : April-18, Marks 4**

1.9 Multi Cone Architecture

SPPU : Dec.-19

1.9.1 Setting the Context

- Since their existence in 1970, The microprocessor industry continues to have great importance in the course of technological advancements.
- A number of techniques such as data level parallelism, instruction level parallelism and hyper threading (Intel's HT) already exists which have dramatically improved the performance of microprocessor cores.

- The growing market and the demand for faster performance drove the industry to manufacture faster and smarter chips.
- One of the most classic and proven techniques to improve performance is to clock the chip at higher frequency which enables the processor to execute the programs in a much quicker time and the industry has been following this trend from 1983 - 2002
- Additional techniques have also been devised to improve performance including parallel processing, data level parallelism and instruction level parallelism which have all proven to be very effective.
- One such technique which improves significant performance boost is multi-core processors. Multi-core processors have been in existence since the past decade.
- However due to technology limitations such as battery life and energy efficiency, the usage and production of multicore processors were low in the earlier days. With advanced chip fabrication technology for integrated circuit in the later years made possible the manufacturing of the multicore processors.

1.9.2 What is a Multicore Processor ?

- A Multi-core processor is typically a single processor which contains several cores on a chip.
- The cores are functional units made up of computation units and caches. These multiple cores on a single chip combine to replicate the performance of a single faster processor.
- The individual cores on a multi-core processor does not necessarily run as fast as the highest performing single-core processors, but they improve overall performance by handling more tasks in parallel.
- The idea of multicore technology is to use multiple cores instead of one (like single processor) at a comparatively lower frequency, but an overall improvement in the performance is delivered through multiple cores operating simultaneously on multiple instructions.
- Multicore processors work on multiple instructions and multiple data. Multiple cores execute multiple threads (multiple processes/instructions) while using different parts of memory (multiple data). This enhances Thread Level Parallelism (TLP).
- The main memory is shared by all cores. Each core is associated with its own cache and they all share the system bus.
- A typical multi-core CPU chip is illustrated in the below Fig. 1.9.1.

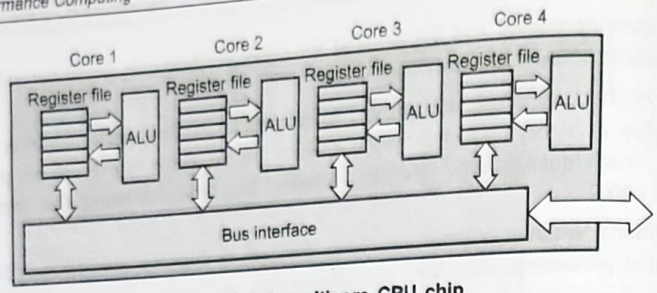


Fig. 1.9.1 A multicore CPU chip

- Multicore CPU is different than traditionally known SMT (Simultaneous Multithreading). SMT Permits multiple independent threads with independent functionality to execute simultaneously on the same core. But it can't simultaneously use the same functional unit on the same core. Hence SMT is not a "true" parallel processor.
- On the other hand, in the case of multi-core processors if there are multiple tasks that can be run in parallel at the same time with same functional unit, each of them will be executed by separate core in parallel as shown in the below Fig. 1.9.2.

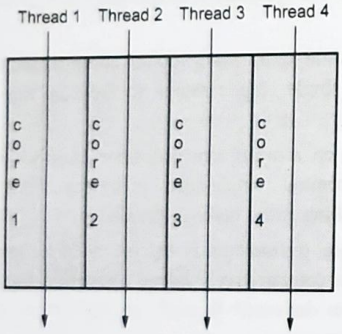


Fig. 1.9.2 Multithreaded execution in multicore architecture

- Also within each core of multicore CPU, threads are time-sliced (just like on a uniprocessor)
- Hence the performance improvement is significant with multicore CPUs as shown in below Fig. 1.9.4.
- The multiple cores inside the chip are not clocked at a higher frequency, but instead their capability to execute programs in parallel is what ultimately contributes to the overall performance making them more energy efficient.

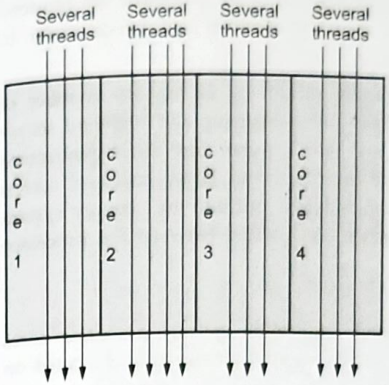


Fig. 1.9.3 Parallel threads in multicore architecture

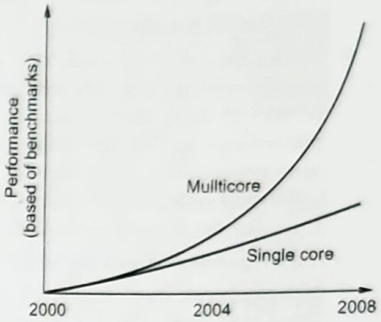


Fig. 1.9.4 Performance improvement in multicore architecture in recent years

- Multi-core processors are generally designed partitioned so that the unused cores can be powered down or powered up as and when needed by the application contributing to overall power dissipation savings.
- Here is an example of 64 core processor by Tilera Corporation. It is is the Tile64 CPU with the cores communicating via a mesh architecture, called iMesh, intended to scale to hundreds of cores on a single chip.
- For example : Dual core processor at 20 % reduced clock frequency effectively delivers 73 % more performance while approximately using the same power as a single-core processor at maximum frequency.
- Other popular processor manufactures namely Intel, AMD, IBM and TENSILICA all have started developing multi-core processors.

1.9.3 Multicore Architectures

- Multi-core processors could be implemented in many ways based on the application requirement. It could be implemented either as a group of heterogeneous cores or as a group of homogeneous cores or a combination of both.
  - **Homogeneous core architecture :**
    - In homogeneous core architecture, all the cores in the CPU are identical and they apply divide and conquer approach to improve the overall processor performance by breaking up a high computationally intensive application into less computationally intensive applications and execute them in parallel.

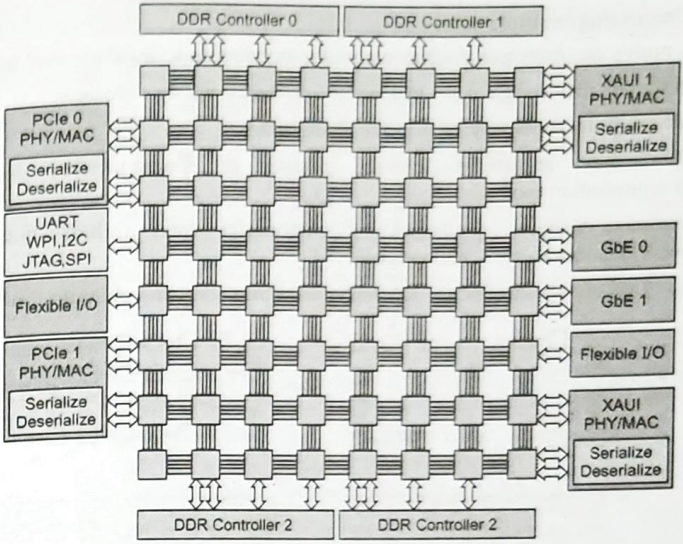
- Other major benefits of using a homogeneous multi-core processor are reduced design complexity, reusability, reduced verification effort and hence easier to meet time to market criterion
- During the last three technology generations (45 nm to 22 nm) the number of on-chip cores has not changed dramatically for mainstream and high-end server systems by Intel, IBM, Fujitsu, Oracle, and AMD. Again core microarchitecture performance and energy efficiency were improved and larger last-level caches were implemented. Much effort by all contenders is put into the memory system bandwidth optimization. Fast buffer caches are inserted between the processor cores and the memory controllers.

- **Heterogeneous core architecture :**

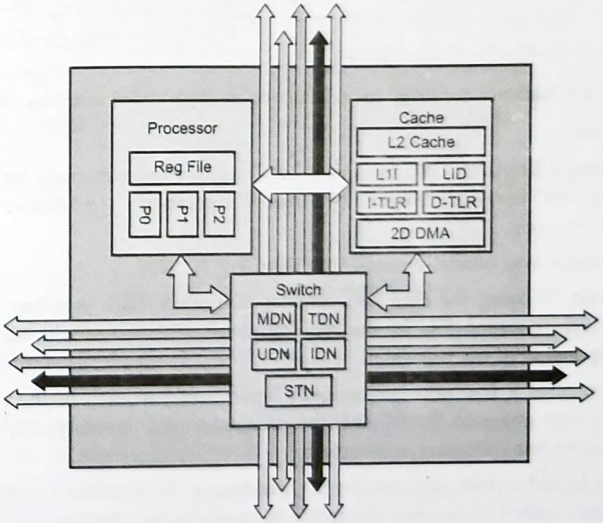
- Heterogeneous cores consist of dedicated application specific processor cores that would target the issue of running variety of applications to be executed on a computer.
- An example could be a DSP core addressing multimedia applications that require heavy mathematical calculations, a complex core addressing computationally intensive application and a remedial core which addresses less computationally intensive applications.
- Many multicore products are offered as IP cores that can be used as building blocks for designing comple custom or FPGA-based heterogeneous multicore systems.
- ARM, Texas Instruments, MIPS, Freescale, Altera, Xilinx and other vendors offer solutions for various target markets that include mobile, IT, automotive, manufacturing, and other areas.
- In the following, out of a very rich landscape, we only give three examples of typical heterogeneous designs.
- Freescale QorlQ T Series
- Altera Stratix 10
- Intel with four ARM Cortex-A53 cores on the chip

- **Combination of homogeneous and heterogeneous core architectures :**

- Multi-core processors could also be implemented as a combination of both homogeneous and heterogeneous cores to improve performance taking advantages of both implementations.
- CELL multi-core processor from IBM follows this approach and contains a single general purpose microprocessor and eight similar area and power efficient accelerators targeting for specific applications has proven to be performance efficient



(a) Schematic of the TILE64 Processor



(b) Schematic of a TILE of the TILE 64 Processor

Fig. 1.9.5 TILE 64 processors

- **Connecting multiple cores :**
  - A multi-core processor implements multiprocessing in a single physical package.
  - Designers may couple cores in a multi-core device tightly or loosely.
  - For example, cores may or may not share caches.
  - They may implement message passing or shared-memory inter-core communication methods.
  - Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar.
  - Fig. 1.9.6 shows some of the topologies used for connecting multiple cores

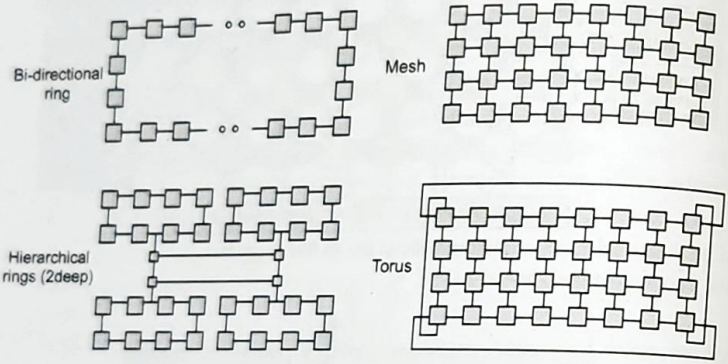


Fig. 1.9.6 Network topology for connecting multiple cores within a chip

- **Intel's Polaris :**
- The Teraflops Research Chip (also called Polaris) is a research manycore processor, containing 80 cores developed by Intel Corporation's Tera-Scale Computing Research Program.
- The processor was officially announced February 11, 2007.
- Along with 80 cores, the chip also contains 80 routers. Each core has a dedicated router which is responsible for the communication of that core with all other cores and components of the processor.
- The router uses a five port system with 1 port going to each of the surrounding cores and one going to the DRAM (the processors local memory). The individual tile of Polaris has following architecture.
- The chip is laid out in an 8 core by 10 core format. Each of the 8 cores in any of the 10 rows, called nodes, has the ability to communicate directly with other cores within the same node.

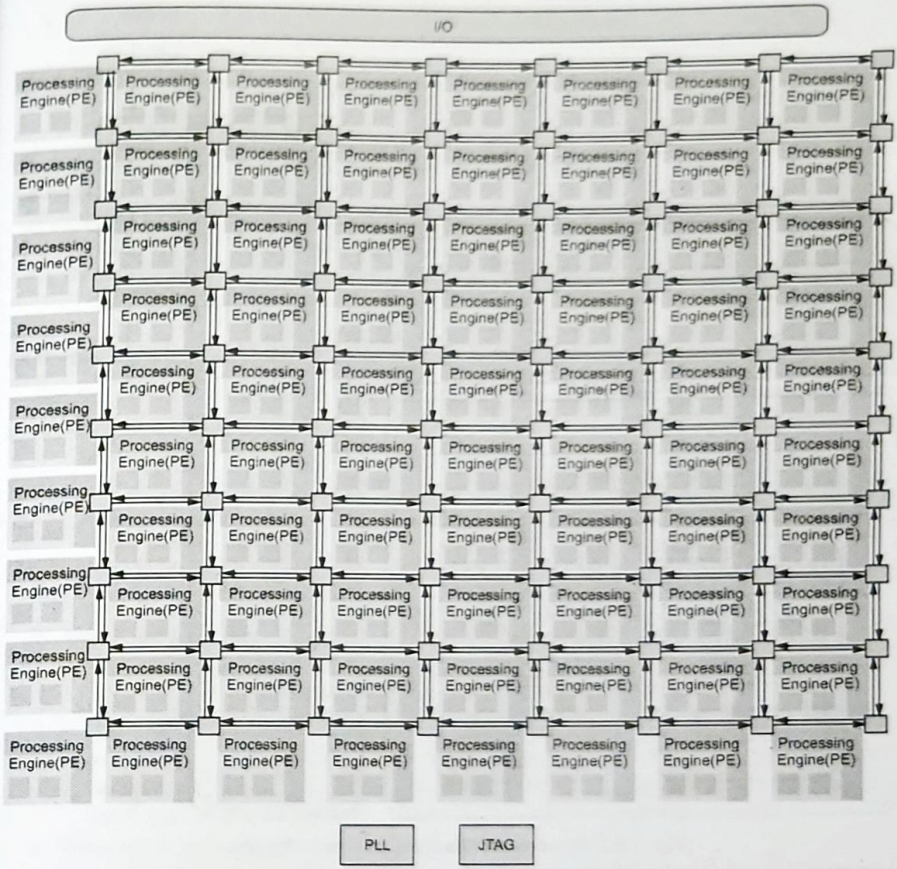


Fig. 1.9.7 Intel's Polaris architecture

- Communication between nodes and to other processor components is directed through a routing system.
- The entire on-chip network features a bisectional bandwidth of 256 GB/s. The router interface block (RIB) interfaces between the core and the router and performs the packet encapsulations.
- The architecture allows any core to send or receive instructions and data packets from and to any other core.

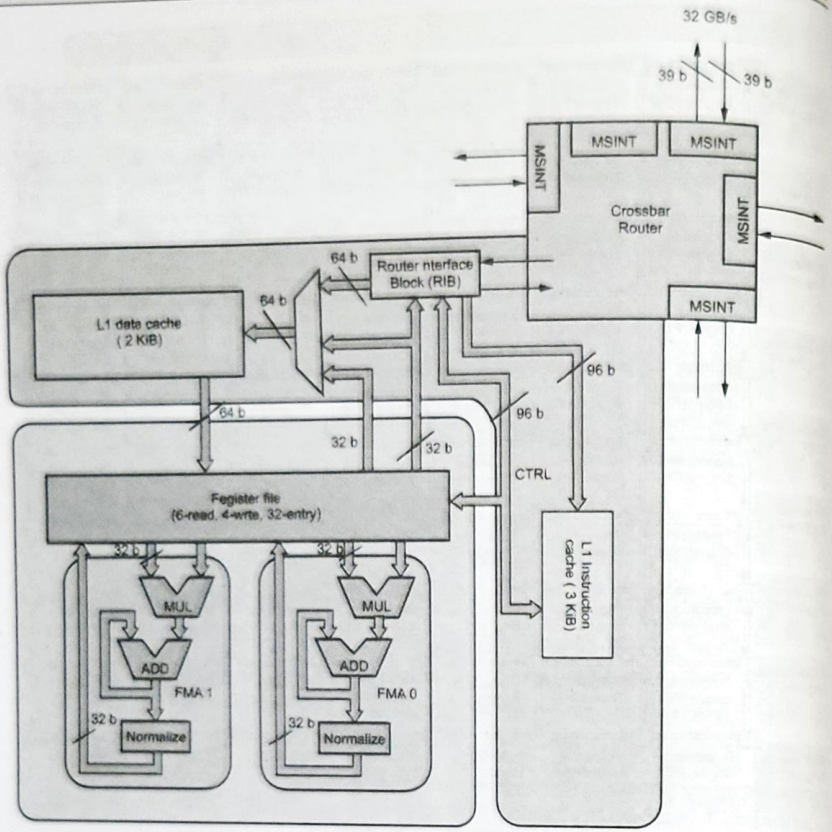


Fig. 1.9.8 In core architecture of Intel's Polaris

- Each tile is connected to a 5-port (East, West, North, South and Up) wormhole-switched router with mesochronous interfaces as shown in below Fig. 1.9.9.
- The on-die interconnect fabric which the cores use to communicate with each other is currently being researched.

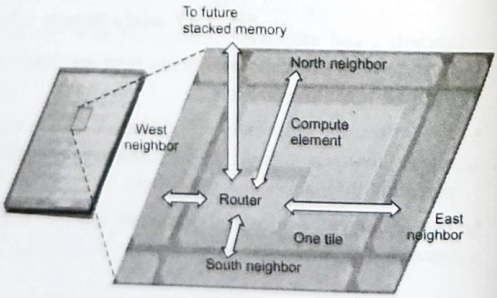


Fig. 1.9.9 TILE connectivity in Intel's polaris

- **Applications that benefit from multi-core architectures**
  - Database servers
  - Web servers (Web commerce)
  - Embedded, network,
  - Digital signal processing (DSP), and graphics (GPU).
  - Multimedia applications
  - Scientific applications
  - In general, applications with Thread-level parallelism (as opposed to instruction level parallelism)

Multicore architectural challenges

So far, we have seen the benefits of multicore technology but there are some problems that arise when more cores are added.

- Thermal Issues (Power and temperature):
  - To reduce the unnecessary power consumption, the multicore design also has to make use of a separate power management unit that can manage or control unnecessary wastage of power.
  - The architecture of the core must be such that the amount of heat generated in the chip is well distributed across the chip.
- **Level of Parallelism :**
  - One of the biggest factors affecting the performance of a multicore processor is the level of parallelism of the process/ application.
  - The lesser the time required to complete a process, better will be the performance. Performance is directly related to the amount of parallelism because more the number of processes that can be executed simultaneously more will be the parallelism.
- **Interconnect Issues :**
  - Since there are so many components on chip in a multicore processor like cores, caches, network controllers etc., the interaction between them can affect the performance if the interconnection issues are not resolved properly.
  - In the initial processors, bus was used for communication between the components. In order to reduce the latency, crossbar and mesh topologies are used for interconnection of components.
  - Also, as the parallelism increases at the thread level, communication also increases off-chip for memory access, I/O etc.
- Research is constantly going on in the areas like developing more efficient applications/ algorithms for multicore environment and also in other areas in order to get the maximum performance throughput from multicore processors.

- Industries are constantly working towards achieving better and better performance from multicore processors.

Review Questions

1. *Write a short note on multicore processors.*
2. *Discuss Tile64 as a multicore processor.*
3. *What are the different types of multicore architectures ?*
4. *Discuss Intel's Polaris as an example of multicore processor.*
5. *What are the challenges in multicore architectures ?*
6. *Discuss the applications that benefit from multicore architecture.*
7. *Explain N-wide superscalar architecture.* **SPPU : Dec.-19, Marks 6**

University Questions with Answers

Oct. - 2019

Q.1 *What are applications of parallel computing ?* **(Refer section 1.0.1)** **[4]**

Q.2 *What are types of dataflow execution model ?* **(Refer section 1.7)** **[6]**

Q.3 *Explain cache coherence in multiprocessor system.* **(Refer section 1.5.6)** **[6]**

May - 2019

Q.4 *Explain Store - and - Forward and packet routing with its communication cost.* **(Refer section 1.6.11)** **[6]**

Q.5 *Discuss the applications that benefit from multi - core architecture.* **(Refer section 1.0.1)** **[6]**

Dec. - 2019

Q.6 *Explain N-wide superscalar architecture.* **(Refer section 1.8)** **[6]**

Q.7 *List application of parallel programming.* **(Refer section 1.0.1)** **[6]**

❑❑❑

Unit II

2 Parallel Algorithm Design

Syllabus

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*** : *Data, Task, Work Pool and Master Slave Model,* ***Complexities*** : *Sequential and Parallel Computational Complexity, Anomalies in Parallel Algorithms.*

Contents

SPPU : April-17

2.0 Some Basics

- A sequential algorithm gives a sequence of steps for solving a given problem on a serial computer.
- A parallel algorithm tells us how to solve a problem using multiple processors.
- Parallelism can be achieved by two ways,
  1. **Implicit parallelism** : Parallelism is exploited by underlying advanced hardware and compiler techniques.
  2. **Explicit parallelism** : Programmer has to play a major role. Parallelism is explicitly specified in the source code by the programmer using special language constructs, compiler directives or library function calls.
- In this unit the focus will be on Explicit parallelism.
- Note that a sequential algorithm focuses only on computation whereas a parallel algorithm should take care of computation as well as communication between the processors.
- In parallel algorithm it is very important to specify the steps that can be executed simultaneously to achieve performance enhancement.
- **A parallel algorithm should include some/all of the below :**
  - Mechanism to identify the parts of work which can be concurrently executed.
  - Technique to map these identified concurrent parts of work onto multiple processes running in parallel.
  - Method to distribute the input,output and intermediate data associated with the program.
  - In case of distributed architecture a parallel algorithm should manage the access of data which is shared by multiple processors.
  - Consideration on Processor synchronization at various stages of the parallel program execution.

Review Questions

1. *Explain granularity, con-currency and dependency graph.* SPPU : April-17, Marks 5
2. *Explain data dependency.*
3. *Explain task dependency graph.*
4. *What are the characteristics of parallel algorithm ?*

2.1 Preliminaries

SPPU : May-19, Oct.-19

- In this unit we are going to discuss how to design and implement parallel algorithm.

- Two important steps in design of a parallel algorithm are,
  1. Divide computation into smaller computation.
  2. Assign these computation to different processors to execute them parallely.

2.1.1 Decomposition, Tasks and Dependency Graphs

- **Decomposition** : To divide the computation into sub computations to execute them parallely is called as decomposition.
- Note that checking various kinds of dependencies within the sub computation is the major criteria of dividing the computation.
- **Task** : Task is a programmer-defined unit of computation. Tasks are generated by subdividing the main computation by decomposition.
- To reduce the computation time for solving the problem simultaneous execution of tasks is done.
- Tasks are inseparable or indivisible parts of computation (cannot be further splited).
- It is not necessary to decompose the problem into tasks of same size, tasks can be of arbitrary size.
- To illustrate this consider the example of dense matrix-vector multiplication.
- Let's understand how multiplication of matrix with vector can be solved parallely.
- As shown in the Fig. 2.1.1, consider the problem of multiplying dense n x n matrix A and vector b to generate resultant vector y.
- The ith element y[i] of the product vector is obtained by dot-product of the ith row of A with the entire input vector b.
- In this case computation of each y[i] can be considered as a task.
- In total n tasks can be generated where n is number of rows in the matrix.

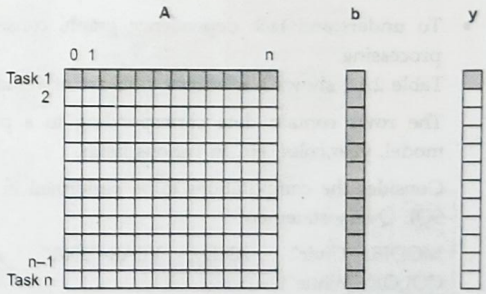


Fig. 2.1.1 Decomposition of dense matrix-vector multiplication into n tasks, where n is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted

- Note the following important points in this problem :
  1. All n tasks are independent of each other and are not interlinked.

2. No task has to wait for execution of other task, so they can be executed together.
3. Any kind of dependency does not exist between them so they can be executed in any sequence.

Data dependancy :

- In some applications data is shared among the processes. In this case, some tasks will need the data produced by other tasks for their execution and they have to wait till they receive this data from other tasks.

Task dependency graph :

- All such possible dependencies among tasks and order of execution of tasks is shown pictorially by **task dependency graph.**
- A task-dependency graph can be defined as a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them.
- A task at a particular node is executed only when all the tasks connected to this node by incoming edges have finished their execution.
- It is not compulsory to have connected task- dependency graph and also some of edges may be empty.
- For example, in case of matrix-vector multiplication each task computes a subset of the entries of the product vector.
- To understand task dependency graph, consider the example of database query processing.

  Table 2.1.1 shows a relational database of vehicles.

  The rows contain data corresponding to a particular vehicle, such as its ID, model, year,color, etc. in various fields.

  Consider the computations to be performed in processing the where clause of a SQL Query statement :

  MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000

9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

Table 2.1.1 : A database storing information about used vehicles

- The computation required to find out result of this query can be divided in sub computations or subtasks.
- As shown in Fig. 2.1.2, initially four intermediate independent tables are to be created.
  1. Table containing all models = Civic
  2. Table containing all 2001-model cars
  3. Table containing all green-colored cars
  4. Table containing all white-colored cars.

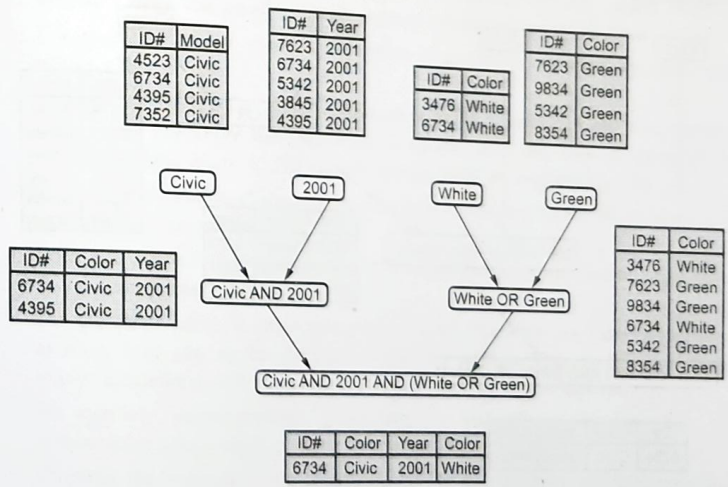


Fig. 2.1.2 The different tables and their dependencies in a query processing operation

- The possible subqueries which can be computed independently, based on the created tables are :
- Compute :
  1. MODEL="Civic"
  2. YEAR="2001"
  3. COLOR="Green"
  4. COLOR="White"
- To get the final result the intersection of the table containing all the 2001 Civics with the table containing all the green or white vehicles is computed.
- The computations, which can be concurrently executed are shown by the task-dependency graph shown in Fig. 2.1.2.
- Each node represents one task corresponding to an intermediate table that needs to be computed.
- The arrows between nodes indicate dependencies between the tasks.
- For example, to compute the table corresponding to the 2001 Civic, first the table of all the Civic and a table of all the 2001-model cars is to be computed.

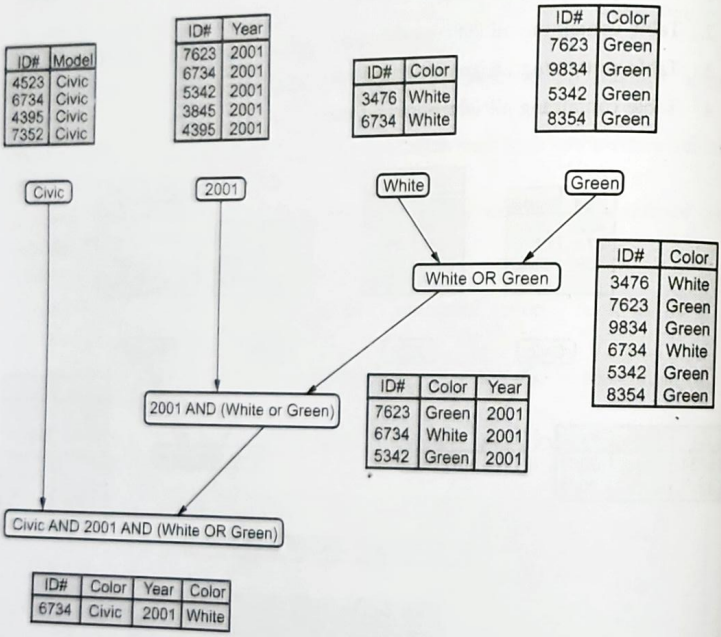


Fig. 2.1.3 An alternate data-dependency graph for the query processing operation

- Note that there can be many ways of computations of the same result, due to involvement of associative operators such as addition, multiplication and logical AND or OR.
- For the same problem, based on the different combinations of computations, different task dependency graphs can be drawn with different characteristics.
- Fig. 2.1.3 shows different version of task-dependency graph, for the same problem of database query, with variation in the combination of associative operators. Refer to the Fig. 2.1.3 on previous page.

2.1.2 Granularity, Concurrency and Task-Interaction

- In parallel algorithm,
  - Every active processor is assigned a specific task.
  - The task may be as simple as incrementing a counter or it may be a subroutine that involves many operations.

2.1.2.1 Granularity

- The size of these tasks is expressed as the **granularity** of the parallelism.
- The number and size of tasks into which a problem is decomposed determines the granularity of the decomposition.
- The grain size of a parallel instruction is a measure of how much work each processor does, compared to an elementary instruction execution time.
- The granularity in a parallel algorithm is generally classified by two relative values : **Fine** or **coarse**
- In some applications a single operation is to be performed on many pieces of data. These operations are performed in parallel over the data set, generally with each Processing Element (PE) communicating with its neighboring PEs.
- As per the definition of granularity, this task would be considered to have a small granularity (fine-grained)
- With this context decomposing a computation into large number of small tasks is called **fine-grained granularity**.
- On the other hand, if large subroutines of an algorithm are independent of one another, they can all be executed in parallel fashion. These subroutines require many calculations with little communication and are **coarse-grained**.
- So formally **coarse-grained granularity** is defined as decomposition of a computation into a small number of large tasks.
- Consider the example of the decomposition for matrix-vector multiplication. In this problem each independent task performs a single dot product, so as per Fig. 2.1.1

matrix can be divided in such a way that each row will part of one task. So large number of small tasks are generated making it fine grained.

- The same problem can be decomposed as a coarse grained as shown in Fig. 2.1.4
- In this 4 rows form one task. So there will be less number of tasks with increased size.
- In this case each task computes n/4 of the entries of the output vector of length n.

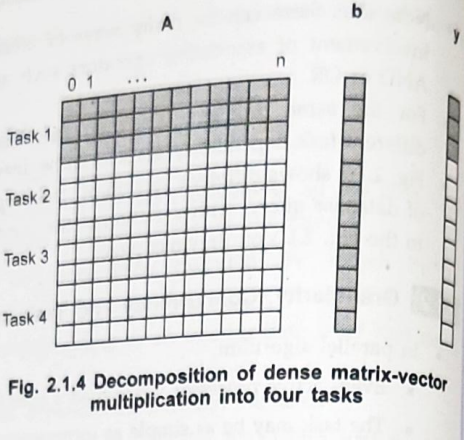
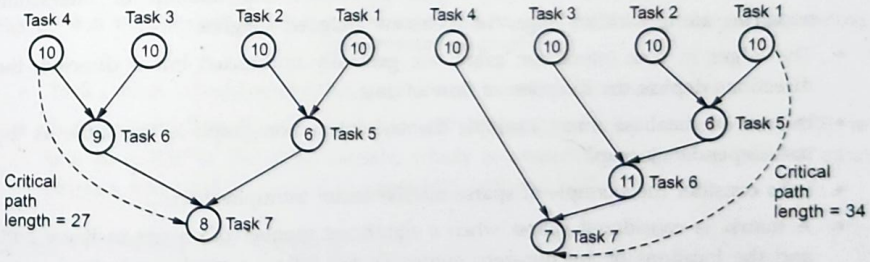


Fig. 2.1.4 Decomposition of dense matrix-vector multiplication into four tasks

Degree of concurrency

- The number of tasks that can be executed in parallel is called as **degree of concurrency.**
- The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its **maximum degree of concurrency.**
- In many problems tasks are interdependent so the maximum degree of concurrency is generally less than the total number of tasks.
- The maximum degree of concurrency in the task-graphs of database query example (Fig. 2.1.2) is 4, as at a given time maximum 4 tasks can be executed.
- Note that when task dependency graphs are trees, the maximum degree of concurrency is always equal to the number of leaves in the tree.
- The average number of tasks that can be executed concurrently over the entire duration of execution of the program is known as **average degree of concurrency.**
- Average degree of concurrency is a more useful measure of a parallel program's performance.
- There is a inverse relation between degree of concurrency and task granularity.
- If large number of small tasks (fine granularity) are executed concurrently then degree of concurrency will increase.
- The maximum and the average degrees of concurrency increase as the granularity increases.
- For example in matrix-vector multiplication shown in Fig. 2.1.1 there is small granularity and a large degree of concurrency, whereas in in Fig. 2.1.4 larger granularity and a smaller degree of concurrency is observed.

- Consider the example of two task graphs shown in Fig. 2.1.5. These graphs represents the conceptualization of the task dependency graphs, shown in Fig. 2.1.2 and 2.1.3.
- Each node contains some amount of work needed for completion of the task.
- The degree of concurrency also depends on the shape of the task-dependency graph and the granularity.
- As shown in the Fig. 2.1.5, The number inside each node represents the amount of work required to complete the task corresponding to that node.
- The **critical path** determines average degree of concurrency for the given granularity.
- The longest directed path between any pair of start and finish nodes is known as the critical path.
- The sum of the weights of nodes along this path is known as the **critical path length**, where the weight of a node is the size or the amount of work associated with the corresponding task.
- The ratio of the total amount of work to the critical-path length is the average degree of concurrency.
- Thus a shorter critical path leads to higher degree of concurrency.
- For example, the critical path length is 27 [10(task 4) + 9(task 6) + 8(task 7)] in the task-dependency graph shown in Fig. 2.1.5 (a) and it is 34 [10(task 1) + 6(task 5) + 11(task 6) + 7(task 7)] in the task-dependency graph shown in Fig. 2.1.5 (b).
- Even if same decomposition is used, the average degree of concurrency of the task graph in Fig. 2.1.5 (a) is 2.33 and of the task graph in Fig. 2.1.5 (b) is 1.88.



(a) **Average degree of concurrency** $= \frac{\text{Total amount of work}}{\text{Critical path length}}$

Average degree of concurrency $= \frac{63}{27} = 2.33$

(b) **Average degree of concurrency** $= \frac{64}{34} = 1.88$

Fig. 2.1.5 Abstractions of the task graphs of Fig. 2.1.2 and 2.1.3

- Since the total amount of work required to solve the problems using the two decompositions is 63 and 64, respectively, the average degree of concurrency of the two task-dependency graphs is 2.33 and 1.88, respectively.
- In addition to granularity and degree of concurrency, the important factor that affects speedup of a parallel execution, is the interaction between the tasks, which are running on different processors.

2.1.2.2 Task Interaction Graph

- The tasks need to interact with each other to share the data.
- Generally output of one task is given as input to other.
- All such dependencies are reflected in the task dependency graph.
- For example, in the database query example discussed earlier, the intermediate data is shared among the processes, to generate a final query.
- Even if there is no dependence between the tasks, and even if they appear to be independent in task dependency graph, there can be interactions between them.
- For example, in matrix-vector multiplication, all the tasks are independent of each other but still each of them need entire vector b for their execution.
- In this case all the tasks need to interact through send and receive operations, in message passing interface in distributed memory model.
- The pattern in which the tasks interact with each other is shown in **task interaction graph.**
- In task interaction graph, the nodes represent tasks and edges show there interaction with each other.
- Based on the amount of computation performed and amount of interaction occurring along with the edge, the nodes are assigned weights.
- The edges in task interaction graph are generally undirected but if directed, the directions depicts the direction of flow of data.
- In case of database query example the task-interaction graph is the same as the task-dependency graph.
- Let's consider the example of sparse matrix- vector multiplication.
- A matrix is considered sparse when a significant number of entries in it are zero and the locations of the non-zero entries do not follow a predefined structure or pattern.
- Given a sparse n x n matrix A and a dense n x 1 vector b, the problem is to calculate product y = Ab.
- In this case computations related to zero entries of the matrix can be avoided.

- So i th entry of the product vector can be computed as : y[i] = A[i j] × b[j], where A[i, j] ≠ 0.
- For example, y[0] = A[0, 0].b[0] + A[0, 1].b[1] + A[0, 4].b[4] + A[0, 8].b[8].... and ...y[11] = A[11, 6].b[6] + A[11, 11].b[11]
- In this case the computation can be decomposed by partitioning the output vector y.
- Each task now can compute each entry in it.
- Each element of vector b is assigned to respective task.
- Output y[i], for task i becomes owner of row A[i, *] of the matrix and the element b[i] of the input vector.
- As shown in Fig. 2.1.6, individual computation y[i] may require the access to elements of b, owned by other tasks.

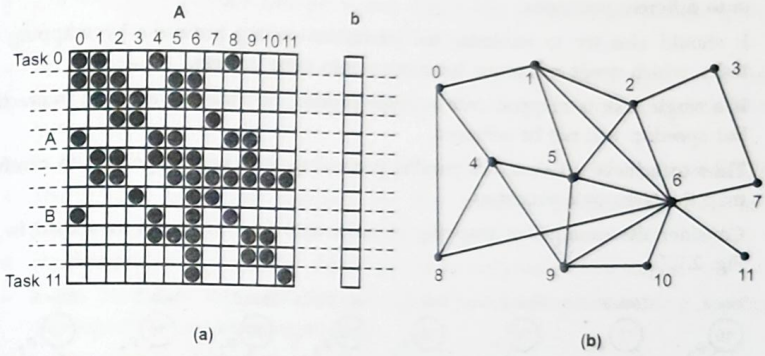


Fig. 2.1.6 A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph

- Task i must interact with other processes to access these elements of b.
- For example, Task 0 need to work on b[0],b[1],b[4] and b[8] for computing y[0]. In this case b[0] is the only element which is owned by task 0, so task 0 must interact with task 1, 4 and 8 to access b[1], b[4] and b[8].

2.1.3 Processes and Mapping

- A very important point to note in this unit is the term **process**. It refers to a **processing or computing agent that performs tasks,** unlike the definition of process in operating system.
- As discussed earlier,the given problem is decomposed into tasks and all the tasks are assigned to physical processors for execution.

- The process use code and data corresponding to a task to produce the output of that task within a time limit.
- In case of need for exchange of data the process may communicate with other processes.
- Speedup in parallel formulation can be achieved if more than one process remains active at a time, performing multiple tasks.
- The assignment of the tasks to processes is called as **mapping**.
- A good mapping scheme for a parallel algorithm is based on task-dependency and task-interaction graphs generated by the decomposition technique for a given problem.
- A mapping scheme should ensure and exploit **maximum concurrency and minimize the execution time** of a parallel program by mapping independent tasks onto different processes.
- It should also try to minimize the interaction among processes by mapping the tasks, which needs maximum interaction onto same process.
- If a single task is mapped onto a single process, no time is wasted in interaction, but speedup will not be achieved.
- Thus to achieve efficiency in parallel processing, it is very important to carefully map the tasks onto processes.
- Consider the example of mapping of tasks onto four processes as shown in the Fig. 2.1.7.

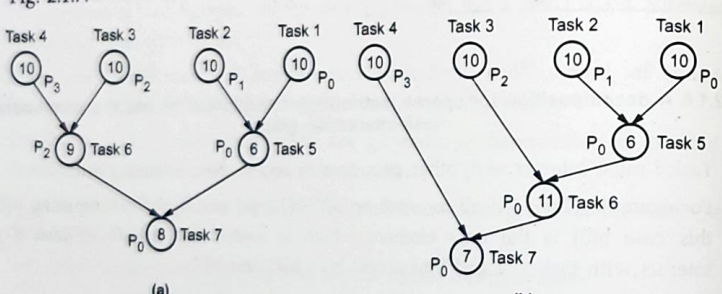


Fig. 2.1.7 Mappings of the task graphs of Fig. 2.1.5 onto four processes

- As shown in the figure total number of tasks is seven, but only four processes can be used to execute them as maximum degree of concurrency is four.
- It will be always beneficial to map the tasks connected by an edge onto the same process because this prevents an inter-task interaction from becoming an inter-processes interaction.

- For example, as shown in Fig. 2.1.7 (b), if task 5 is mapped onto process P_2, then both processes P_0 and P_1 will need to interact with P_2, in contrast with the current mapping of the tasks, having only a single interaction between P_0 and P_1.

2.1.4 Processes versus Processors

- Unlike the terminology used in operating system, in parallel algorithm design processes are considered as logical computing agents that perform tasks.
- Note that parallel algorithms and programs can be expressed in terms of processes.
- Processors are the hardware units that physically perform computations.
- Generally it is assumed that number of processes is equal to number of physical CPUs on the parallel computer i.e. processors.
- Considering processes and processors differently is useful in parallel programs running on different programming models.
- To understand this consider the example of matrix multiplication on a parallel computer which consists of multiple nodes that communicate with each other via message passing.
- In this case each node can also have shared-address-space with multiple CPU's
- The parallel algorithm designed for this scenario should take care of two programming models, message passing and shared address space.
- For this problem the parallel algorithm can be designed in two stages.
- In the first stage the parallelism among the nodes is exploited using appropriate decomposition and mapping strategy.
- In the second stage decomposition and mapping strategy is chosen which is suitable for the shared-memory paradigm which can be implemented individually on each node.

Review Questions

1. *Explain matrix - vector multiplication with reference to decomposition of task.*
2. *Write a short note on :*
   a) Decomposition of tasks b) Dependency graph
   c) Granularity d) Concurrency and task interaction
3. *Explain with examples - Fine grained and coarse grained granularity of a task computational problem.*
4. *Explain importance of degree of concurrency in parallel algorithms.*
5. *Explain the critical path in the task graphs with suitable example.*

6. What is a task interaction graph ?
7. Write a short note on processes and mapping.
8. Define and explain the following term - Degree of concurrency. **SPPU : May-19, Marks 2**
9. Explain decomposition, task and dependancy graph. **SPPU : Oct.-19, Marks 6**
10. What are limitations of parallelization of any algorithm ? **SPPU : Oct.-19, Marks 4**

2.2 Decomposition Techniques

SPPU : April-16, Oct.-19

- As discussed in the introduction, in explicit parallelism programmer has to play the role of exploiting parallelism while addressing any problem.
- Consider the applications like sorting of large amount of elements or the data centric application like market basket analysis, which is to be performed on huge data or consider the application for development of any game.
- All these problems are of different nature and needs different problem solving approach, in turn different algorithmic techniques to solve them.
- To solve above mentioned and many such computationally or data centric complex problems parallely, it is very important to identify how to divide them in smaller parts. These small subtasks can be later executed concurrently without any conflicts posed by dependencies between them.
- The job of dividing the problem into smaller subproblems or subtasks is called as **decomposition**.
- As mentioned above to solve problems of different nature from various domains different decomposition techniques are to be used.
- Some of these techniques are :
  - Recursive decomposition
  - Data-decomposition
  - Exploratory decomposition
  - Speculative decomposition
  - Hybrid decomposition.

2.2.1 Recursive Decomposition

- Recursive decomposition is suitable for the problems that can be solved using divide and conquer strategy.
- A problem is divided into independent sub-problems.
- Each of the sub -problem can be divided further recursively using the same strategy.
- The results of all the sub-problems are combined to get a final solution.

2.2.1.1 Quicksort

- Consider the example of quick sort.
- Sequence A of n elements is to be sorted.
- As shown in the task dependency graph in the Fig. 2.2.1, the first step is to select the pivot element x. This is a divide step.

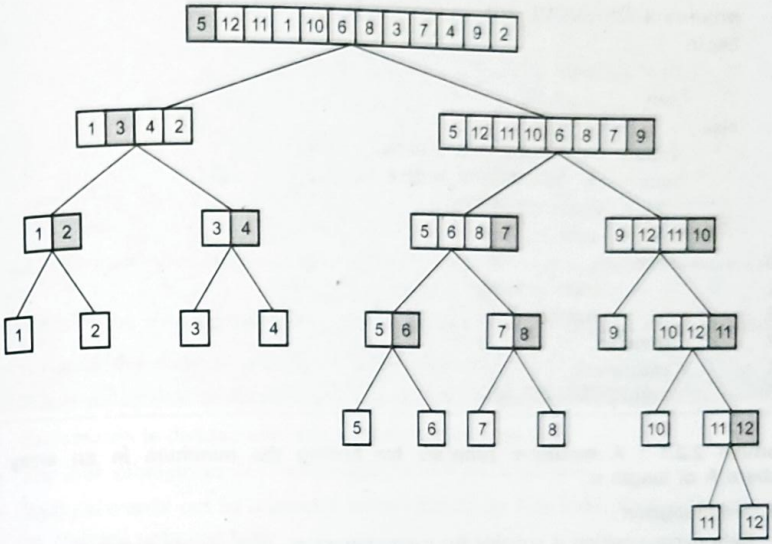


Fig. 2.2.1. The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers

- After selection of pivot x, divide and conquer strategy is applied to partition A into two sub-parts, A0 and A1 where elements in A0 are smaller than x and elements in A1 are greater than or equal to x.
- The same technique is applied in recursion, to partition A0 and A1 further, till each sub-part has only one element.
- In the tree shown in the Fig. 2.2.1, as we move down the concurrency increases.

2.2.1.2 Find Minimum Element

- Consider the second example of finding out minimum element in an unordered sequence A of n elements.
- Even if the problem can be solved by different approach of an algorithm, we can also use divide and conquer strategy to solve it.

- As per the algorithm 2.2.1, the sequence A of n elements is partitioned into two sub-sequences of size n/2.
- From each partition minimum number is found.
- As shown in Fig. 2.2.2 The process is continued till only one element is left in the sub-partition.

```
1.  procedure RECURSIVE_MIN (A, n)
2.  begin
3.  If    (n = 1) then
4.        min     :=  A [0];
5.  else
6.        1min    :=  RECURSIVE_MIN (A,      n/2) :
7.        rmin    :=  RECURSIVE_MIN (& (A, ⌈n/2⌉), n - n/2) :
8.        If      (1min < rmin) then
9.                min  := 1min;
10.       else
11.               1min  := rmin ;
12.               endelse
13.       endelse;
14.       return min;
15.       end RECURSIVE_MIN
```

Algorithm 2.2.1 : A recursive program for finding the minimum in an array of numbers A of length n.

Data Decomposition :

- Data decomposition is suitable for computations on data centric problems.
- Decomposition is done in two steps.
  1. The data is partitioned.
  2. The computation is performed on data partitions.
- Generally same operation is performed on these partitions.
- There are various ways to partition the data,
  1. Partitioning output data.
  2. Partitioning input data.
  3. Partitioning both input and output data.
  4. Partitioning intermediate data.

1. Partitioning output data :

- In some problems output can be obtained in pieces.
- Each piece can be independently computed from the given input.

- The total result or output can be computed by combining results from these independent pieces.
- Each piece of output can be mapped to one task, where each task does the work of computing portion of the output.
- So the decomposition is based on partitioning output data.

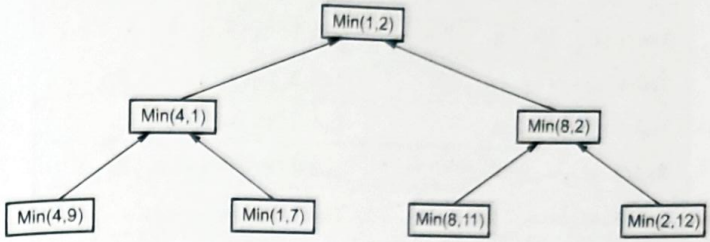


Fig. 2.2.2. The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}.

Each node in the tree represents the task of finding the minimum of a pair of numbers

- Consider the example of matrix multiplication shown in Fig. 2.2.3.
- The problem is to multiply two n x n matrices A and to obtain matrix C
- Each matrix is divided into four submatrices of same size.
- The four submatrices of C are roughly of size n/2 x n/2 each.
- Each submatrix can be computed independently by four tasks. Each submatrix will be mapped onto one task.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1 : $C_{1,1} = A_{1,1} B_{1,1} + A_{1,2} B_{2,1}$

Task 2 : $C_{1,2} = A_{1,1} B_{1,2} + A_{1,2} B_{2,2}$

Task 3 : $C_{2,1} = A_{2,1} B_{1,1} + A_{2,2} B_{2,1}$

Task 2 : $C_{2,2} = A_{2,1} B_{1,2} + A_{2,2} B_{2,2}$

(b)

Fig. 2.2.3. (a) Partitioning of input and output matrices into 2 x 2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a)

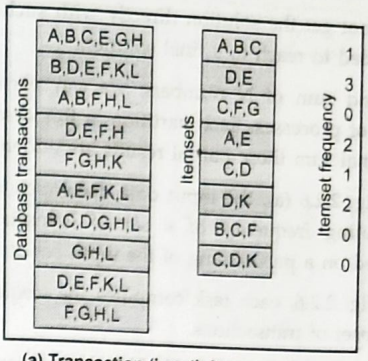
- As shown in the Fig. 2.2.3, output matrix C can be decomposed into four sub-matrices, which intern can be computed independently.

- A very important point to be noted in this case is : data decomposition is different from the decomposition of computation into tasks.
- Rather once data is decomposed, decomposition of computation can be done. This can be understood from Fig. 2.2.4.

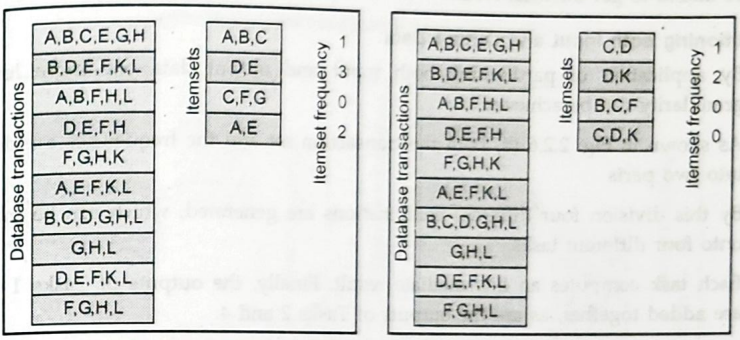
Decomposition I	Decomposition II
Task 1 : $C_{1,1} = A_{1,1} B_{1,1}$	Task 1 : $C_{1,1} = A_{1,1} B_{1,1}$
Task 2 : $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	Task 2 : $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Task 3 : $C_{1,2} = A_{1,1} B_{1,2}$	Task 3 : $C_{1,2} = A_{1,2} B_{2,2}$
Task 4 : $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	Task 4 : $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Task 5 : $C_{2,1} = A_{2,1} B_{1,1}$	Task 5 : $C_{2,1} = A_{2,2} B_{2,1}$
Task 6 : $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	Task 6 : $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Task 7 : $C_{2,2} = A_{2,1} B_{1,2}$	Task 7 : $C_{2,2} = A_{2,1} B_{1,2}$
Task 8 : $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	Task 8 : $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

Fig. 2.2.4. Two examples of decomposition of matrix multiplication into eight tasks

- To understand partitioning of output data, input data, both input and output data and intermediate data, consider the problem of computing the frequency of a set of itemsets in a transaction database.
- Let T be a grocery stores database of customer sales with each transaction being an individual grocery list of a customer and each itemset is a group of items in the store.
- The aim is to find out how many customers bought each of the groups of items, and frequency of buying a particular itemset.
- As shown in Fig. 2.2.5 (a), there are total 10 transactions shown in the first column and 8 itemsets shown in second column. The output, shown in the third column, is the frequencies of these itemsets in each transactions performed.
- Now as the computations are independent of each other they can be computed concurrently, and as the application is data centric we have to apply data decomposition to compute it.
- One way to solve this is to apply output data decomposition as shown in Fig. 2.2.5 (b).
- In this, the computation of frequencies of the itemsets can be decomposed into two tasks by partitioning the output into two parts and having each task compute its half of the frequencies.



(a) Transaction (input), itemsets (input), and frequencies (output)



(b) Partitioning the frequencies (and itemsets) among the tasks

Fig. 2.2.5 Computing itemset frequencies in a transaction database

2. Partitioning Input Data

- Output data decomposition is possible only when, in a set of outputs to be generated, each output is a function of input and can be computed independent of each other.
- There are some computations in which output is just a single value, which is to be computed from set of inputs, for example finding sum of the numbers is one of such problems.
- In such cases it is not desirable to partition the output data.
- One way of solving such problems is to partition the input data to exploit the concurrency.

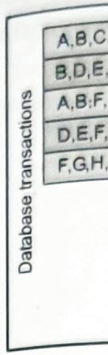
- Note that we may not get the solution directly with such partitioning, a follow up computation is needed to reach to a final solution.
- For example, to find sum of N numbers, we can partition them in N/p parts, where p = number of processes. Each partition in this case will generate the partial result. To get the final sum these partial results are added.
- As shown in the Fig. 2.2.6 (a), the input data decomposition can be applied to the problem of computing frequency of a set of itemsets in a database by the decomposition based on a partitioning of the input set of transactions.
- As shown in the Fig. 2.2.6, each task computes the frequencies of all the itemsets in its respective subset of transactions.
- As a result, two independent set of frequencies are generated per task, which can be added to get the final result.

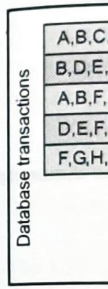
3. Partitioning both Input and Output Data

- By application of partitioning both input and output data one more level of granularity can be achieved.
- As shown in Fig. 2.2.6 (b) both the transaction set and the frequencies are divided into two parts
- By this division four different combinations are generated, which can be mapped onto four different tasks.
- Each task computes an intermediate result. Finally, the outputs of Tasks 1 and 3 are added together, as are the outputs of Tasks 2 and 4.

4. Partitioning Intermediate Data

- In most of the algorithms, the final output is the result of multiple intermediate stages.
- Generally output of one stage is the input to immediate next stage.
- In such problems, input and output data decomposition can be applied on the intermediate stages.
- By partitioning intermediate data higher degree of concurrency can be achieved as compared to input or out data decomposition.
- Consider the example of matrix multiplication shown in Fig. 2.2.3.
- In this the maximum degree of concurrency which can be achieved is 4.





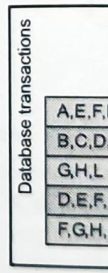
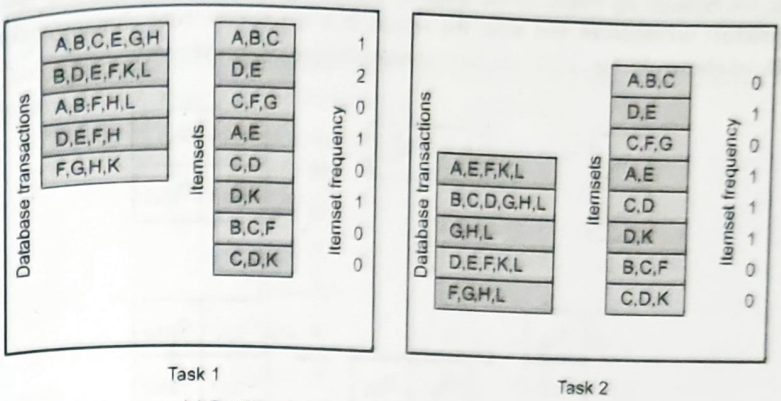
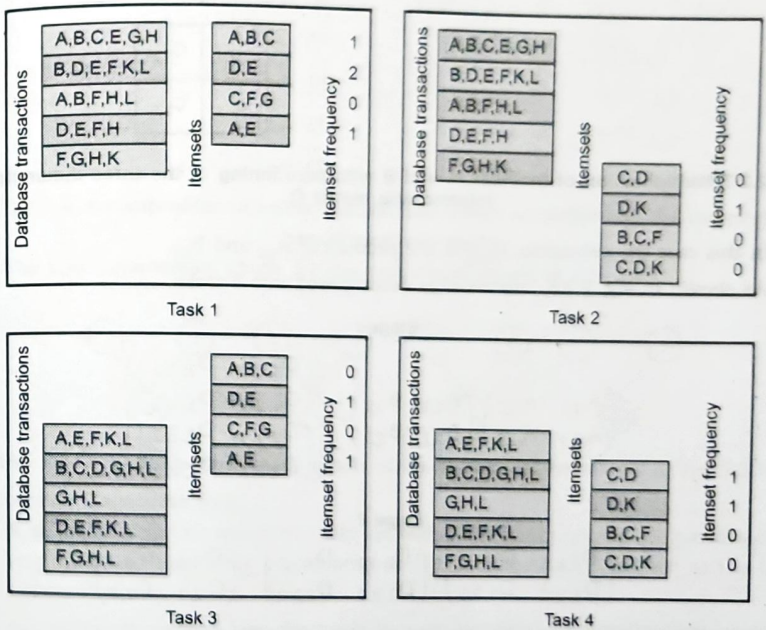


Fig. 2.2.6 Inp



(a) Partitioning the transactions among the tasks



(b) Partitioning both transactions and frequencies among the tasks

Fig. 2.2.6 Input output data decomposition for computing itemset frequencies in a transaction database

- If we include an intermediate stage in which eight tasks compute their respective product submatrices and store the results in a temporary three dimensional matrix D, as shown in Fig. 2.2.7, we can increase degree of concurrency.

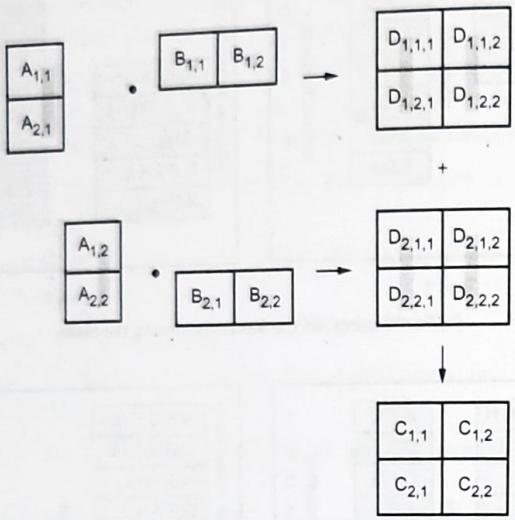


Fig. 2.2.7 Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D

- In this case the submatrix $D_{k,i,j}$ is the product of $A_{i,k}$ and $B_{k,j}$.
- As shown in Fig. 2.2.8, matrix D is decomposed into 8 tasks.

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} \\ \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{1,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

- First phase is multiplication phase, followed by addition step to compute final matrix C.
- All submatrices $D_{*,i,j}$ with the same second and third dimensions i and j are added to obtain $C_{i,j}$.

- The first eight tasks multiply $n/2 \times n/2$ submatrices of A and B and perform $O(n^3/8)$ work.
- The next four tasks 9 through 12 do addition of $n/2 \times n/2$ submatrices of the intermediate matrix D and perform $O(n^2/4)$ work each.

A decomposition induced by a partition of D

Task 01 : $D_{1,1,1} = A_{1,1} B_{1,1}$

Task 02 : $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 03 : $D_{1,1,2} = A_{1,1} B_{1,2}$

Task 04 : $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 05 : $D_{1,2,1} = A_{2,1} B_{1,1}$

Task 06 : $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 07 : $D_{1,2,2} = A_{2,1} B_{1,2}$

Task 08 : $D_{2,2,2} = A_{2,2} B_{2,2}$

Task 09 : $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 10 : $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 11 : $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 12 : $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Fig. 2.2.8 A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix

- The task dependency graph for this problem is shown in Fig. 2.2.9.

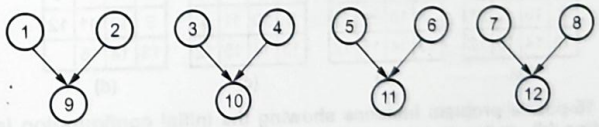


Fig. 2.2.9 The task-dependency graph of the decomposition shown in Fig. 2.2.8

The Owner - Computes Rule :

- A decomposition in which the data on which the computations are performed can be partitioned based one partitioning output or input data is known as the owner - computes rule.
- According to owner - computes rule in each partition computations are performed with the data owned by that partition.
- In case of input data partitioning owner computes rule means that a task performes all the computations that can be done using this data.

- In case of output data partitioning, it means that a task computes all the data in a respective partition.

2.2.2 Exploratory Decomposition

- Exploratory decomposition is suitable of search space problems.
- State space search is a process used in the field of artificial intelligence in which successive configurations or states of an instance are considered with the goal of finding a "goal" state with a desired property.
- Search space is partitioned into smaller parts.
- Each part will be searched concurrently till the solution is obtained.
- To understand exploratory decomposition consider the example of 15 puzzle problem.
- The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 × 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration.
- As shown in Fig. 2.2.10, sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration can be obtained.

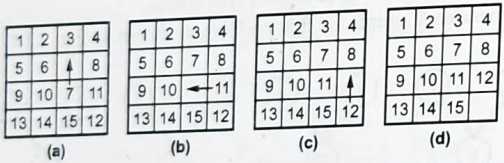


Fig. 2.2.10 A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final

- In this case, a tree search technique is used to obtain the final configuration.
- To solve this problem parallelly, at the beginning initial configuration generates few levels of configuration serially.
- Each of these configurations is assigned to a task for further exploration.
- This process is terminated when one of the tasks finds a solution.
- After finding the solution, the task informs other tasks to terminate their searches.
- The decomposition of four tasks is shown the Fig. 2.2.11. Among these task 4 finds the solution.

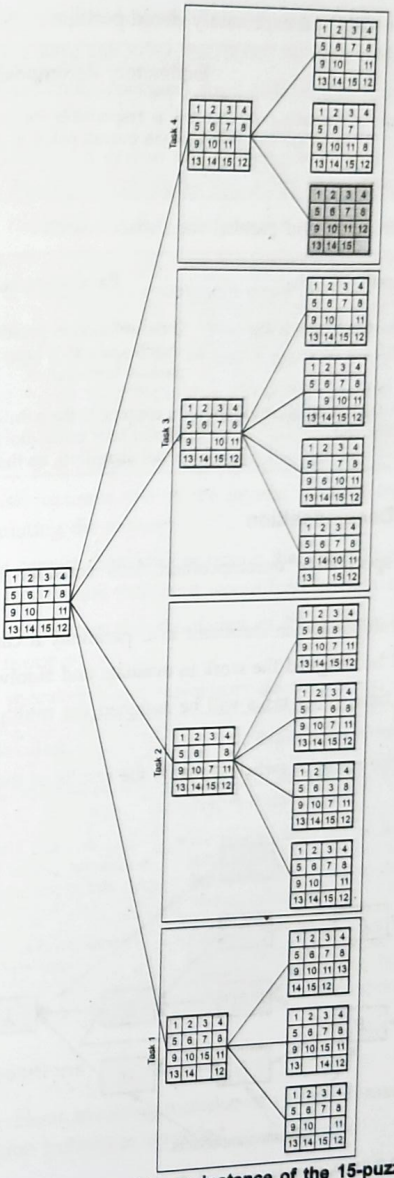


Fig. 2.2.11 The states generated by an instance of the 15-puzzle problem

- Difference between data and exploratory decomposition :

Data decomposition	Exploratory decomposition
Each task performs useful computation which contributes to final solution	Typically only one task is responsible for finding the solution. Other tasks are terminated when overall solution is found.

- Difference between serial and parallel state space search :

Serial state space search	Parallel state space search
Entire space search is searched even if the solution exists at the beginning.	If the solution is present at the beginning of search space it is found immediately due to parallel formulation.
It gives better speed up than parallel algorithm if the solution exists at the end.	In contrast if the solution exists at the end almost four times more work is needed than serial algorithm, so the speed up will decrease.

2.2.3 Speculative Decomposition

- To understand speculative decomposition, let's consider the example of switch statement in C.
- If we want to evaluate switch statement in C parallelly it can be done as follows -
  - One task will be assigned the work to evaluate and resolve the switch.
  - At the same time, other tasks will be assigned the multiple branches of switch in parallel.
  - By the time the result of switch is ready, the results of the branches will also be ready.

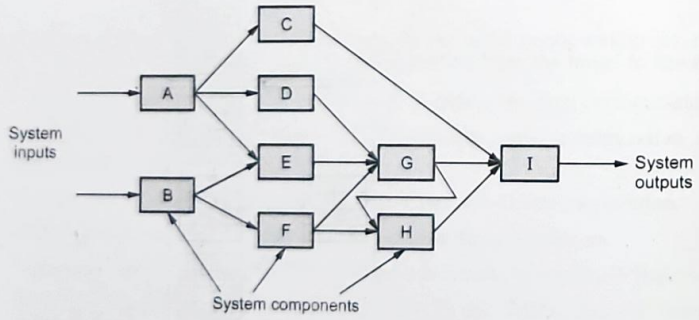


Fig. 2.2.12 A simple network for discrete event simulation

- One of the result computed by the branch, which is the desired result of the switch statement is kept and other results will be discarded.

- To further illustrate this, consider the example of parallel discrete event simulation.
- Consider the simulation of a system that is represented as a network or a directed graph as shown in the Fig. 2.2.12. (Refer Fig. 2.2.12 on previous page)
- The nodes of this network are the components where each component has an input buffer of jobs.
- Initially each node is idle.An idle component picks up a job from its input queue.
- A component has to wait if the input buffer of one of its outgoing neighbors if full, until that neighbor picks up a job to create space in the buffer.
- The problem is to simulate the functioning of the network for a given sequence or a set of sequences of input jobs and compute the total completion time.
- In this case the input of a component is output of another.
- Each speculative task assumes one of the several possible inputs to a particular stage and start simulating the network.
- If the speculation is correct then after receiving the result from the previous task, the work required to simulate this input would have already been finished.
- If the speculation is incorrect the simulation of this stage is restarted with the most recent correct input
- Difference between speculative and exploratory decomposition :

Data decomposition	Exploratory decomposition
• Input at a branch leading to multiple parallel tasks is unknown	• Output of the multiple tasks originating at a branch is unknown
• Serial algorithm perform only one task at a speculative stage, as which branch should be chosen is known at the beginning of this stage.	• Serial algorithm can explore different alternatives one after the other, because the branch that may lead to the solution is not known beforehand.
• Parallel algorithm performs more or same amount of work as serial algorithm.	• Parallel algorithm performs more, less or same amount of work compared to the serial algorithm depending on the location of the solution in the search space

2.2.4 Hybrid Decompositions

- In some complex problems involving inclusion of various domains, combination of different decomposition techniques is needed.

- Generally such computations can be divided into multiple stages and different decompositions are applied in different stages.
- Consider the example of finding out minimum number from a large set of numbers as shown in Fig. 2.2.13.

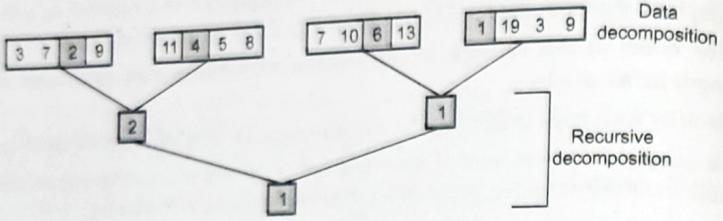


Fig. 2.2.13 Hybrid decomposition for finding the minimum of an array of size 16 using four tasks

- Instead of applying recursive decomposition directly, first the data is partitioned into equal parts using data decomposition.
- Later recursive decomposition can be applied on the partitions and result is obtained.

Review Questions

1. *Explain recursive decomposition with suitable example.* **SPPU : April-16, Marks 5**
2. *Explain data decomposition with a suitable example.*
3. *What are the different types of data decomposition. Explain any one of them with suitable example.*
4. *What is exploratory decomposition ?*
5. *Compare/Write the difference between data decomposition and speculative decomposition.*
6. *What is speculative decomposition ?*
7. *What is hybrid decomposition ?*
8. *Explain with suitable example - Exploratory decomposition.* **SPPU : Oct.-19, Marks 6**

2.3 Characteristics of Tasks and Interactions

SPPU : March-18, Oct.-19

- As described in the introduction of this unit once a problem is decomposed into parallelly executable tasks using suitable decomposition technique, the next step is to map these tasks onto available processes.
- The choice of good mapping is based on the properties of tasks and interactions among the tasks.

- In this section two important parameters for effective mapping of tasks are considered :
  1. Characteristics of tasks.
  2. Characteristics of inter task interactions.

Characteristics of tasks

- The following four characteristics of tasks are important for choosing suitable mapping scheme :
  1. Task generation.
  2. Task sizes.
  3. Knowledge of task sizes.
  4. Size of data associated with tasks.

1. Task generation

Static task generation

- To understand the concept of task generation consider following examples.
- If we decompose the data centric application using data decomposition the size of data and the operation to be performed on it is known apriori.
- In this case we can formulate and generate the tasks before execution of the algorithm.
- Consider the example of finding minimum of the numbers using recursive decomposition.
- In this the data is divided in fixed partitions where each partition can be assigned to a task for execution.
- In both the examples explained above it is observed that the tasks are known before starting the execution of the algorithm.
- Such a scenario is known as **static task generation.**

Dynamic task generation

- In case of the problems of state space search input is expanded in predefined number of steps and then again new tasks are generated to perform same computation on each resulting state.
- Next consider the example of quick sort, In this case the partitions are decided by the values in input array which we need to sort.
- We can observe in both the above examples that the tasks are not known a priori and they are generated dynamically as the algorithm grows.
- This is called as dynamic task generation.
- Note that in case of exploratory decomposition if we expand the tree using breadth first manner it will lead to static task generation.

2. Task sizes

- The time needed for completing the task determines the size of the task.
- Consider the example of matrix multiplication.
- The matrix is partitioned such that each task corresponds to one row of a matrix.
- So in this case each task can be completed in same amount of time.
- Such tasks are known as **uniform tasks.**
- Now let's see the second example of quick sort.
- In quick sort the partitions of the numbers to be sorted are done based on the selection of pivot number.
- So partitions of varied sizes are formed, where each partition contributes to one task.
- In this case the size of each task may be different and they cannot be finished in same amount of time.
- Such tasks are known as **non-uniform tasks.**

3. Knowledge of task sizes

- Consider the example of matrix multiplication.
- In all the decompositions applied for finding matrix multiplication the time required for computation of each task is known, that means knowledge of task size is there before execution of the parallel algorithm.
- In case of 15 puzzle problem, knowledge of tasks is unknown as the number of moves to reach up to solution cannot be determined apriori.

4. Size of data associated with tasks

- If the knowledge of the data associated with task is present i.e. if size and location of data is known, then the task can be performed without the overhead of excessive data movement.
- The size of data is associated with the input as well as output data of the task can be of different sizes.
- For example, the input of 15 puzzle problem is just one state which is a very small input. But there is complex compilation involved to find sequence of moves to reach to final state.
- Another example is finding out minimum of a number. In this, size of input data is proportional to the compilation but output is just a number.

2.3.1 Characteristics of Inter-Task Interactions

- As discussed earlier computation and interaction are two important factors of any parallel algorithm.

- Tasks interact to share data, work or synchronization information.
- Types of interactions vary according to need of the parallel algorithm.
- Following comparison tables explain various types of interactions in a parallel algorithm.

Static Interactions	Dynamic Interactions
• For each task, the interactions happen at predetermined times.	• Timing of interaction is not known prior to execution of the program.
• Task interaction graph and Interaction will happen in which stage is known.	• The stage at which interaction is needed is decided dynamically.
• Can be programmed easily in the message-passing paradigm	• Harder to program in the message-passing paradigm as interactions in message-passing require active involvement of sender and receiver. The unpredictable nature of dynamic iterations makes it hard for both the sender and the receiver to participate in the interaction at the same time. In this case an additional synchronization parameter is to be included.
• Easy to code in shared address space model	• Easy to code in shared address space model.
• Example : Parallel matrix multiplication	• Example : 15 puzzle problem

Regular Interactions	Irregular Interactions
• An interaction pattern is considered to be regular if it has some structure that can be exploited for efficient implementation.	• An interaction pattern is called irregular if no such regular pattern exists.
• Easy to handle	• Irregular and dynamic communications are difficult to handle especially in message passing model.
• Example : Problem of image dithering. In image dithering, the color of each pixel in the image is determined as the weighted average of its original value and the values of its neighboring pixels. This problem can be easily decomposed by breaking the image into square regions and using a different task to dither each one of these regions.	• Example : Sparse matrix-vector multiplication. In this problem a task cannot know which entries of vector it requires.

Read only Interactions	Read write Interactions
• Tasks require only a read-access to the data shared among many concurrent tasks.	• Multiple tasks need to read and write on some shared data.
• **Example** : The decomposition for parallel matrix multiplication. In this problem, the tasks only need to read the shared input matrices A and B.	• **Example** : 15-puzzle problem. In this problem, the priority queue constitutes shared data and tasks need both read and write access to it; they need to put the states resulting from an expansion into the queue and they need to pick up the next most promising state for the next expansion.

One way Interactions	Two way Interactions
• Only one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one.	• The data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks.
• Cannot be programmed directly in message passing model, as information of sender as well as receiver must be known. So all one way interactions should be converted to two way by programming constructs.	• Suitable for message passing model.
• Easy to handle in shared address space model.	• Easy to handle in shared address space model.

Review Questions

1. *Explain characteristics of tasked.* **SPPU : March-18, Marks 5**
2. *Write a short note on task generation.*
3. *Differentiate between static and dynamic task generation.*
4. *Discuss the impact of task size on task generation.*
5. *Compare between -*
   a) *Static interaction and dynamic task interaction*
   b) *Regular and irregular task interaction*
   c) *Read only and read write task interaction.*
   d) *One way and two way interactions.*
6. *What are characteristics of task and interactions ?* **SPPU : Oct.-19, Marks 4**

2.4 Mapping Technique for Load Balancing

SPPU : May-17, 19, Dec.-19

- After decomposition of a problem into sub-tasks the mapping of those sub-tasks into processes should ensure that the subtasks should be executed in less time.
- To achieve small execution time overheads must be reduced.

- Overheads are caused in any decomposition due to -
  1. Inter-process interaction time.
  2. Time for which the processes are idle.
- A good mapping should aim at reducing above mentioned overheads.
- Due to load imbalancing, some processes finish the work early.
- Based on the constraints in task dependency graph some processes may have to wait for other processes to finish their work.
- To reduce overhead caused due to interaction, one way is to assign the tasks which need interaction onto same process.
- But this way leads to imbalance of the workload among processes, processes with less load will be idle and processes with heavy load will be always busy trying to finish their tasks.
- To overcome this problem proper assignment of the tasks to processes and intern good mapping strategy is very important.
- A good mapping scheme must ensure the balance between computations and interactions among processes.
- If synchronization among the interacting tasks is improper then waiting time for sending and receiving data will increase.
- As shown in Fig. 2.4.1, due to dependencies among tasks, both the mappings will have different completion time.

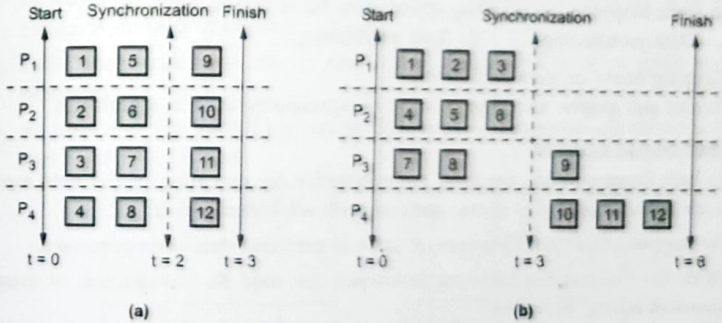


Fig. 2.4.1 Two mappings of a hypothetical decomposition with a synchronization

- There are two broad categories of mapping techniques :
  1. Static
  2. Dynamic.

Static Mapping :

- In static mapping the tasks are distributed to process before execution of the algorithm.
- The task size is known before execution of algorithm.
- The algorithms with static mapping scheme are easy to design and program.

Dynamic Mapping :

- In dynamic mapping the tasks are distributed to processes during execution of the algorithm.
- Task generation and mapping is done dynamically.
- As task sizes are not known before dynamic mapping is beneficial than static mapping as static mapping can lead to load imbalancing in this case.
- If the data associated with the task is large compared to computation, then catering to movement of the data among processes static mapping may prove more suitable.
- However, in shared address space model for read only operations on data, dynamic mapping will still work well.
- The algorithms following dynamic mapping scheme are more complicated, especially in message passing programming model.

2.4.1 Schemes for Static Mapping

- In static mapping the mapping schemes the focus will be more on -
  1. Data partitioning 2. Task partitioning

1. Mapping based on data partitioning :

- Arrays and graphs are common ways of representing data in algorithms.

Array distribution schemes :

- In data decomposition the tasks are responsible for execution of the data associate with them according to owner computes rule explained earlier.
- So mapping tasks onto processes is same as mapping data onto processes.
- With this context, the following techniques are used for distribution of arrays or matrices among processes.

Block distributions :

- In block distributions the uniform contiguous portions of the array are distributed to different processes.
- As a example consider d-dimensional array in which each process will receive contiguous block of array along array dimensions.

- Let's consider n × n two dimensional array with n rows and n columns as shown in Fig. 2.4.2.

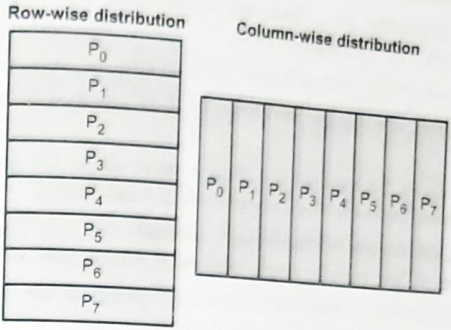


Fig. 2.4.2 : Examples of one-dimensional partitioning of an array among eight processes

- We can partition the array in p parts such that each partition will be a block of n/p consecutive rows of A, if the row is considered as the first dimension.
- Same is the case with the column as a second dimension where each partition contains block of n/p consecutive columns.
- Now consider the case in which multiple dimensions are considered instead of a single dimension partition.
- In this case both the dimensions (row and column) are selected at a time and matrix is divided in number of blocks.
- Each block will have size of $n/p_1 \times n/p_2$ where $p = p_1 \times p_2$ (total number of processes)
- As shown in Fig. 2.4.3 there can be, different two dimension distributions i.e. 4×4 and 2×8 process grid.

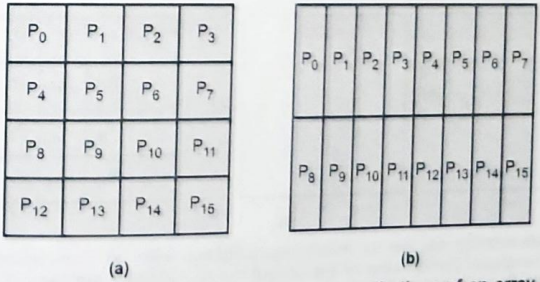


Fig. 2.4.3 Examples of two-dimensional distributions of an array, (a) on a 4 × 4 process grid, and (b) on a 2 × 8 process grid

- For d-dimensional array we can have block distribution up to d-dimensions.
- Consider the example of nxn matrix multiplication C = A × B from 2.2.2.
- If number of processes are p, then one dimensional block distribution of C will give block of n/p rows or columns of C.
- Two dimensional distribution will give block of size $n/\sqrt{p} \times n/\sqrt{p}$.
- In both the cases one of the partitions of C is assigned to one process which computes it.
- Note that in higher dimensional distribution more blocks are generated, so we can make use of more processes for computation of those blocks.
- For example, in matrix multiplication problem we can use n processes for computation of all the rows in single dimension distribution.
- Whereas with two dimensional distribution n^2 processes can be utilized as single element of C is assigned to each process.
- The advantages of use of higher dimensional are :
  1. Higher degree of concurrency. 2. Reduced interactions among the processes.
- To understand how interactions are reduced, consider the example shown in Fig. 2.4.4 for process P5.

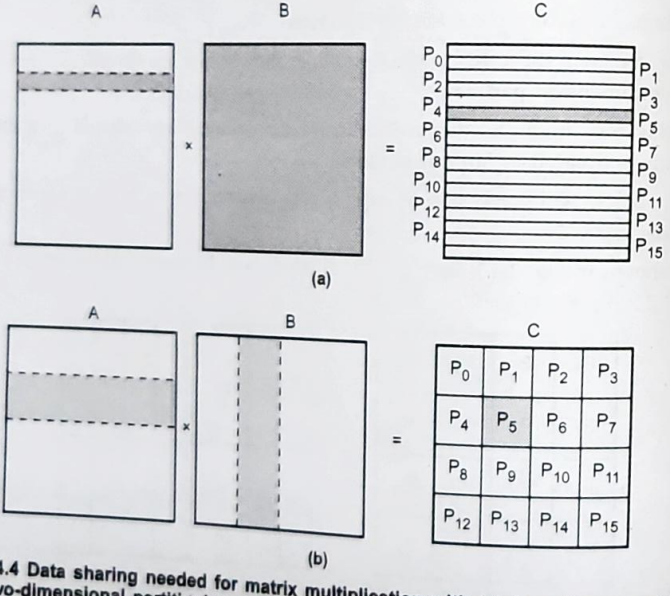


Fig. 2.4.4 Data sharing needed for matrix multiplication with (a) One-dimensional and (b) Two-dimensional partitioning of the output matrix. Shaded portions of the input matrices A and B are required by the process that computes the shaded portion of the output matrix C

One dimensional distribution along rows	Two dimensional distribution
• Each process access n/p rows of A and complete matrix B.	• Each process access $n/\sqrt{p}$ rows of A and $n/\sqrt{p}$ rows of B.
• Total data to be accessed by each process is $\frac{n^2}{p} + n^2$ i.e $O(n^2)$	• Total data to be accessed by each process is $O(n^2/\sqrt{p})$.

- Block distribution is useful if potentially same work is performed on each element.
- If amount of work is different for different elements block distribution results in load imbalance.
- Consider the example of Dennse LU factorization.
- As shown in Fig. 2.4.5, LU factorization algorithm factors a nonsingular square matrix A into the product of a lower triangular matrix L with a unit diagonal and an upper triangular matrix U.
- Let A be an n x n matrix with rows and columns numbered from 1 to n.
- As shown in Fig. 2.4.5 a possible decomposition of LU factorization can be done into 14 tasks using a 3×3 block partitioning of the matrix and using a block version of Algorithm 2.4.1.

```
1.   procedure COL_LU (A)
2.   begin
3.         for k:= 1 to n do
4.               for j:= k to n do
5.                     A (j, k) := A(j, k) / A (k, k);
6.               endfor ;
7.               for j:= k + 1 to n do
8.                     for i := k + 1 to n do
9.                           A [i, j] := A [i, j] – A [i, k] × A [k, j];
10.                    endfor ;
11.              endfor ;
     /*
     After this iteration, cloumn A [k+1 : n, k] is logically the kth column of L and
row A [k, k : n] is logically the kth row of U.
     */
12.              endfor ;
13.  end COL_LU
```

Algorithm 2.4.1 : A serial column-based algorithm to factor a nonsingular matrix A into a lower-triangular matrix L and an upper-triangular matrix U. Matrices L and U share space with A. On Line 9, A[i, j] on the left side of the assignment is equivalent to L [i, j] if i > j; otherwise, it is equivalent to U [i, j].

- As shown in the algorithm, for each iteration of the outer loop k := 1 to n, the next nested loop in k + 1 to n.
- As the computation progresses, the active part of the matrix shrinks towards the bottom right corner of the matrix
- Thus as block distribution is applied, the processes which are computing the values for beginning rows and columns perform less work.
- This can be understood from example in Fig. 2.4.5 with a 3 x 3 two-dimensional block partitioning of the matrix.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1 : $A_{1,1} \rightarrow L_{1,1} U_{1,1}$

2 : $L_{2,1} = A_{2,1} U_{1,1}^{-1}$

3 : $L_{3,1} = A_{3,1} U_{1,1}^{-1}$

4 : $U_{1,2} = L_{1,1}^{-1} A_{1,2}$

5 : $U_{1,3} = L_{1,1}^{-1} A_{1,3}$

6 : $A_{2,2} = A_{2,2} - L_{2,1} U_{1,2}$

7 : $A_{3,2} = A_{3,2} - L_{3,1} U_{1,2}$

8: $A_{2,3} = A_{2,3} - L_{2,1} U_{1,3}$

9 : $A_{3,3} = A_{3,3} - L_{3,1} U_{1,3}$

10 : $A_{2,2} \rightarrow L_{2,2} U_{2,2}$

11 : $L_{3,2} = A_{3,2} U_{2,2}^{-1}$

12 : $U_{2,3} = L_{2,2}^{-1} A_{2,3}$

13 : $A_{3,3} = A_{3,3} - L_{3,2} U_{2,3}$

14 : $A_{3,3} \rightarrow L_{3,3} U_{3,3}$

Fig. 2.4.5 A decomposition of LU factorization into 14 tasks

- In case of group of 9 processes, if mapping is done for a block as shown in Fig. 2.4.6, it results in idle time as different blocks of the matrix need different amount of time.
- For example, only one task is needed to compute $A_{1,1}$, whereas three tasks Task 9, 13 and 14 are needed for computation of $A_{3,3}$

P_0 T_1	P_3 T_4	P_6 T_5
P_1 T_2	P_4 $T_6 T_{10}$	P_7 $T_8 T_{12}$
P_2 T_3	P_5 $T_7 T_{11}$	P_8 $T_9 T_{13} T_{14}$

Fig. 2.4.6 Mapping of LU factorization tasks onto processes based on a two-dimensional block distribution

Cyclic and Block Cyclic Distributions :

- By using block cyclic distribution the problem of load balancing and idling can be eliminated.
- The basic idea is, instead of making partitions of an array=number of processes, partition it in many more blocks than number of available processes.

- Consider the example shown in Fig. 2.4.7 for block cyclic distribution.

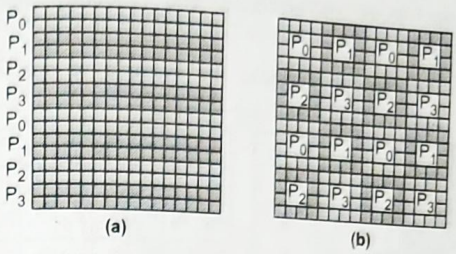
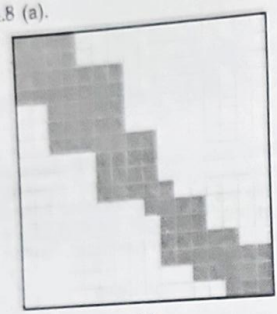


Fig. 2.4.7 Examples of one- and two-dimensional block-cyclic distributions among four processes. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wraparound fashion. (b) The matrix is blocked into 16 blocks each of size 4 × 4, and it is mapped onto a 2 × 2 grid of processes in a wraparound fashion

- Each process is assigned the partition in round robin manner.
- So each process receives many non-adjacent blocks.
- In a one-dimensional block-cyclic distribution of a matrix among p processes, the rows or columns of an n × n matrix are divided into α p groups of n/(α p) consecutive rows or column, where $1 \leq \alpha \leq n/p$.
- Each block b_i is assigned to process $p_{(i \% p)}$ in a wraparound fashion.
- A two-dimensional block-cyclic distribution of an n × n array is obtained by partitioning it into square blocks of size $\alpha\sqrt{p} \times \alpha\sqrt{p}$ and distributing them on a hypothetical array of processes $\sqrt{p} \times \sqrt{p}$ in a wraparound fashion.
- The idling is reduced as the processes have a sampling of tasks from all parts of the matrix.
- So even if work requirement of different parts of matrix is different there will be balancing of work on each process.
- If α is increased to n/p (its upper limit) then each block = single row in 1D in block cyclic distribution and each block = single element in 2D block cyclic distribution.
- The above distribution is called as **cyclic distribution**.
- As decomposition in cyclic distribution is fine grained, perfect load balance is achieved.
- Some of the limitations of this scheme are :
  1. Performance may be degraded as contiguous data is not available to each process for working, resulting in lack of locality.
  2. Amount of interaction is more than the amount of computation in each task.

Randomized block distribution :

- Randomized block distribution is similar to block cyclic distribution with the addition that the blocks are uniformly and randomly distributed among the processes.
- To understand the concept, consider the example of sparse matrix shown in Fig. 2.4.8 (a).



(a)

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

(b)

Fig. 2.4.8 Using the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances

- If we apply two dimensional block cyclic distribution more non zero blocks are assigned to the diagonal processes P_0, P_5, P_{10} and P_{15} than on any other processes, creating load imbalance.In this case some processes like P_{12} will not get any work.
- To address this problem of load imbalancing randomized block distribution is used.
- Similar to block-cyclic distribution array is partitioned into many more blocks than the number of available processes, with the difference that the blocks are uniformly and randomly distributed among the processes.
- In case of one dimensional block distribution A vector V of length p is used.
- V[j] is set to j for $0 \leq j < p$.
- V is randomly permuted and process P_i is assigned the blocks stored in $V[i\alpha \ldots (i+1)\alpha - 1]$
- This is shown in Fig. 2.4.9 for p = 4 and a = 3

V = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

random (V) = [8, 2, 6, 0, 3, 7, 11, 1, 9, 5, 4, 10]

mapping = 8 2 6 0 3 7 11 1 9 5 4 10
P_0 P_1 P_2 P_3

Fig. 2.4.9. A one-dimensional randomized block mapping of 12 blocks onto four process (i.e., a = 3)

- On the similar lines a two-dimensional randomized block distribution of an n x n array can be computed by randomly permuting two vectors of length $\alpha.\sqrt{p}$ each and using them to choose the row and column indices of the blocks to be assigned to each process.

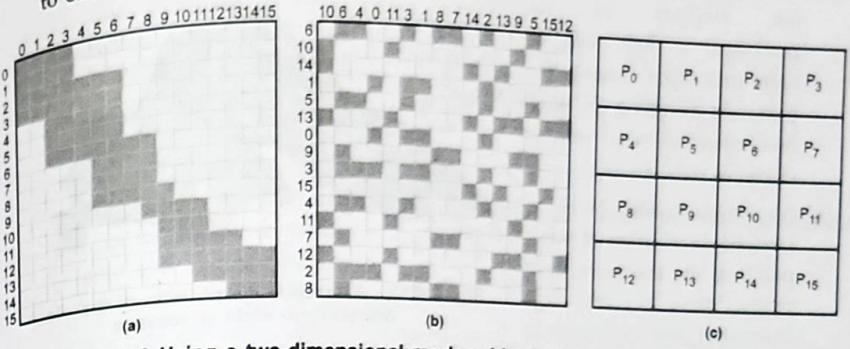


Fig. 2.4.10 Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c)

Graph Partitioning

- The algorithms following array based distribution schemes have structured and regular interaction pattern. They also ensure load balancing with minimum interaction overhead.
- Graph partitioning will be useful for the algorithms, which operate on sparse data structures with highly irregular interaction pattern of data elements.
- Simulations of many real time physical phenomenon falls under this category.
- For computation, the physical domain is discretized and represented by a mesh of elements. Certain physical quantities are then computed at each mesh point.
- The computations at a mesh point takes into consideration the data corresponding to that mesh point and the data associated with mesh points near to it.
- To understand graph partitioning, let's consider the physical phenomenon of simulating dispersion of a water contaminant in the lake.
- As shown in the Fig. 2.4.11, the level of contamination at each vertex of the mesh, imposed on lake is computed at various intervals of time.

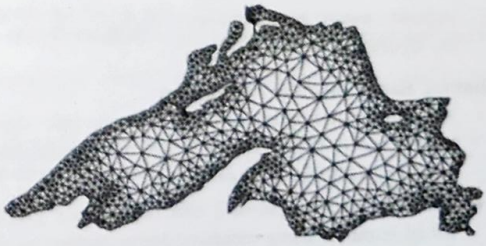


Fig. 2.4.11 : A mesh used to model Lake Superior

- Even if same computation is to be performed at each point, load balancing problem can be eliminated by assigning equal number of mesh points to the processes. But it is very important to assign the mesh points properly.
- If nearby mesh points are not assigned to the processes, it will lead to high interaction among the processes resulting in high interaction overhead due to extensive data sharing.
- As shown in Fig. 2.4.12, in random distribution of mesh points to the processes, each process needs to access large amount of data, belonging to other processes for it's computation.

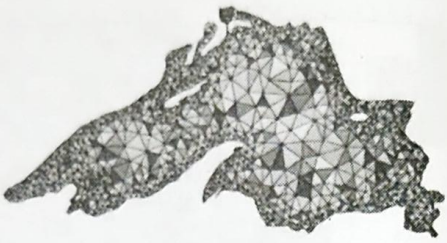


Fig. 2.4.12 : A random distribution of the mesh elements to eight processes

- To overcome this problem of interaction overhead, the mesh points should be such distributed that load will be balanced and data access from other mesh points should be minimum.
- To achieve this the mesh is partitioned into p roughly equal parts, and the number of edges that cross partition boundaries is minimized.
- Later, each of these p partitions is assigned to one of the p processes, such that each process receives contiguous region of mesh.
- A typical graph partitioning software would generate the partition of the Lake Superior mesh as shown in the figure Fig. 2.4.13.

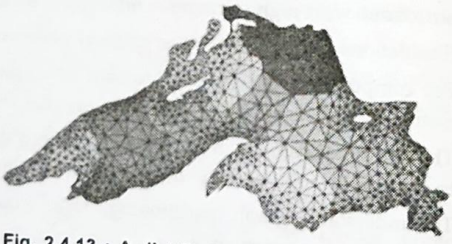


Fig. 2.4.13 : A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm

Mappings Based on Task Partitioning

- When computation is expressed by static task- dependency graph, and size of each task is known then a mapping based on partitioning a task-dependency graph and mapping its nodes onto processes is used.
- Consider the example of a task dependency graph for finding the minimum of a list of numbers using recursive decomposition as shown in Fig. 2.4.10.

- In Fig. 2.4.14 binary tree representation of the task dependency graph in eight processes is shown.
- In this mapping scheme, interaction overhead is minimized by mapping interdependent tasks, along a straight branch of tree onto same process.
- The remaining tasks are mapped onto process which is only one communication link away from each other.

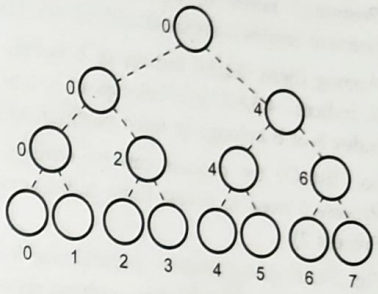


Fig. 2.4.14 Mapping of a binary tree task-dependency graph onto a hypercube of processes

- In this case minimum idling is ensured, as all the concurrently executing tasks are mapped onto different processes.
- To understand task partitioning consider another example of sparse matrix-vector multiplication discussed in Section 2.1.2.
- Fig. 2.4.15 shows mapping of the task-interaction graph of Fig. 2.1.6, along with the partitions.

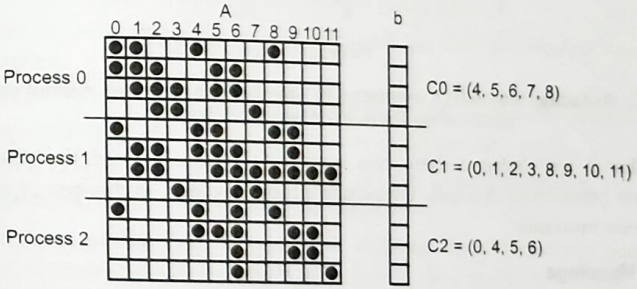


Fig. 2.4.15 A mapping for sparse matrix-vector multiplication onto three processes. The list Ci contains the indices of b that Process i needs to access from other processes

- In each partition, four consecutive entries of b are assigned to tasks of each process.
- The list Ci contains the indices of b that the tasks on Process i need to access from tasks mapped onto other processes.
- For example, for process P_0, the corresponding assigned entries of vector b are (0,1,2,3).

- Process 0 needs to access (0,1,2,3,4,5,6,7,8) indices of b corresponding to the nonzero entries associated with all the four tasks associated with Process 0.
- Among these, 0,1,3,4 indices of b belong to first partition associated with Process 0. Indices 4,5,6,7 of b belongs to second partition associated with process 1 and index 8 of b belongs to third partition associated with process 2.
- So list C0 for process can be written as C0 = (4,5,6,7,8,) which indicates that Process 0 need to access these indices mapped onto other processes(Process 1 and Process 2)
- Fig. 2.4.16 shows another partitioning for the task interaction graph of the sparse matrix vector multiplication problem shown in Fig. 2.1.6 for three processes.

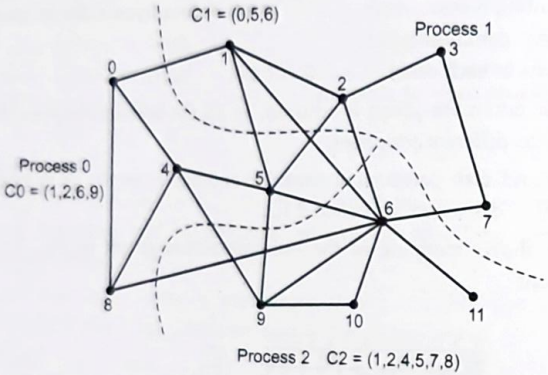


Fig. 2.4.16. Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task-interaction graph

- Note that if we compare these two mappings, it is observed that the mapping based on partitioning the task interaction graph has less exchanges of elements of b between processes.

Hierarchical Mappings

- To understand Hierarchical Mappings consider the example of the binary-tree task-dependency graph of Fig. 2.4.14.
- From the graph it can be observed that only few tasks can be executed parallelly in the top part of the tree.
- This may lead to load imbalance, if the tasks are large.
- A better mapping can be obtained by a further decomposition of the tasks into smaller subtasks.

- As shown in Fig. 2.4.17, the root task can be partitioned among eight processes, the tasks at the next level can divided in four processes, and partition of two processes each is done at the third level.

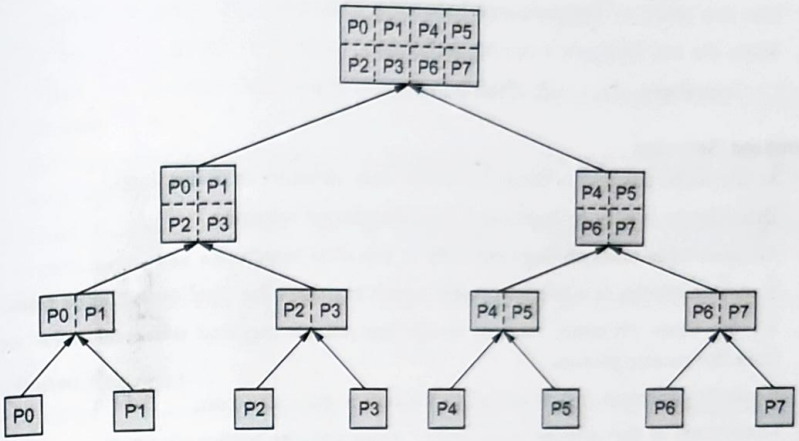


Fig. 2.4.17 An example of hierarchical mapping of a task-dependency graph

- At the fourth level, eight leaf tasks can be mapped to one process each.
- This scheme is also suitable for the parallel quick sort.
- Hierarchical mapping can also be applied for the problem of sparse matrix factorization.
- In sparse matrix factorization, high-level computations are represented by a task-dependency graph which is known as an elimination graph.
- The tasks, which are closer to the root, in the elimination graph can be further decomposed into subtasks by data decomposition.
- Further task partitioning is applied at the top layer and array partitioning is applied at the bottom layer according to hybrid decomposition scheme.
- To conclude, a hierarchical mapping can have many layers and different decomposition and mapping techniques may be suitable for different layers.

2.4.2 Schemes for Dynamic Mapping

- Load balancing is a important issue which is to be addressed when multiple tasks are doing work parallelly.
- As described in section 2.4.1 in static mapping the work is assigned to task before execution of the algorithm.

- In some cases this can lead to imbalance in distribution of the work to the processes.
- So to maintain workload balance dynamic mapping is used.
- It is also called as **dynamic load-balancing.**
- There are two Dynamic mapping techniques
  1. Centralized 2. Distributed.

Centralized Schemes

- As the name suggests a common central data structure is maintained.
- This scheme is easy to implement than distributed scheme.
- All executable tasks are kept centrally in this data structure.
- A master process is a special process which manages the pool of available tasks.
- All the other processes (slaves) which are not having any work take the work from the master process.
- A newly generated task is added to the central data structure.
- Limitations of this scheme is scalability issues and the bottleneck issue.
- In this, as number of slave processes increase the central data structure is accessed heavily leading to bottleneck at master process.

Self scheduling

- To understand the concept of self scheduling, consider the example of sorting the rows of n × n matrix A by quick sort algorithm

```
for (i = 1; i < n; i++)
Sort (A[i], n)
```

- As the number of elements to be sorted vary in each iteration, mapping of tasks lead to load imbalance.
- To address the problem of load imbalance, central pool of indices of the rows which are not sorted is maintained.
- When any process is idle, it takes available index, sort the row corresponding to it and delete that index.
- This will be done till indices are available in the work pool.
- In this case, in parallelly working processes, the iteration of a loop is independently scheduled.
- This is called as **self scheduling.**

Chunk scheduling

- To balance the computation a single task is assigned to a process at a time.

- But if the task is assigned less computation, for example, individual loop iteration in the above example, bottleneck can be there while accessing shared work pool.
- Consider average size of a task M is the time to assign work to a process.
- Accordingly only M/Δ processes can be assigned the task.
- To avoid this chunk scheduling is used.
- When any process needs work instead of a single task, group of tasks (chunk) is assigned to it.
- But if chunk size i.e. number of tasks assigned in each step is large it may lead to load imbalance.
- Load balancing problem can be addressed by decreasing chunk size while execution of the program.
- Initially chunk size can be kept large and it can be decreased as number of iterations, to be executed are decreasing.

Distributed Schemes

- Instead of maintaining central pool, in distributed scheme each process can send or receive work from any process.
- As the name suggests the tasks are distributed among the processes.
- Problem of bottleneck is avoided in this case.
- Some of the important parameters of this scheme are :
  1. Decision of forming the pairs of sending and receiving processes.
  2. Initiation of work transfer by sender or receiver.
  3. The amount of work transfer in each exchange.
  4. Whether the work can be assigned when process has finished its work or when it has very less work remaining and about to finish it.

Review Questions

1. *What are different partitioning techniques used in matrix vector multiplication ?* **SPPU : May-17, Marks 7**
2. *Explain graph partitioning with suitable example.*
3. *Discuss mapping techniques for load balancing.*
4. *What is static and dynamic mapping.*
5. *Discuss the block distribution.*
6. *Write a short note on -*
   a) Cyclic and block cyclic distribution b) Randomized block distribution
   c) Graph partitioning d) Mappings based on task partitioning e) Hierarchical mapping

7. *Explain the scheme for dynamic mapping.*
8. *Differentiate between static and dynamic mapping techniques for load balancing.* **SPPU : May-19, Marks 6**
9. *Define and explain the following terms - Granularity.* **SPPU : May-19, Marks 2**
10. *Explain mapping techniques for load balancing.* **SPPU : Dec.-19, Marks 6**

2.5 Methods for Containing Interaction Overheads **SPPU : Dec.-19**

- A parallel algorithm will become efficient if it has minimum interaction overhead.
- The overhead which is caused due to interaction depends on the factors like :
  1. Volume of data exchanged during interactions
  2. The frequency of interaction
  3. The spatial and temporal pattern of interactions, etc.
- In this section we will discuss some techniques to reduce interaction overheads in the parallel program.
- These techniques make use of some or all the three factors mentioned above while devising the decomposition and mapping schemes for the algorithms or while programming the algorithm in a given model.

2.5.1 Maximizing Data Locality

- In many parallel algorithms the access to some common data is needed for task execution by different processes.
- Consider the example of sparse matrix-vector multiplication y = Ab as shown in Fig. 2.1.6.
- In this example each task compute individual elements of vector y.
- For doing this, each task must access all the elements of input vector b.
- In addition to this interaction may also happen if processes require data generated by other processes
- In all such cases interaction overhead can be reduced by making use of the local data or recently fetched data.
- Data locality enhancing techniques include wide range of schemes that try to minimize the volume of nonlocal data that are accessed, maximize the reuse of recently accessed data, and minimize the frequency of accesses.
- This scheme is similar in nature to the concept of use of cache memory in modern processors.

2.5.1.1 Minimize Volume of Data-Exchange

- The interaction overhead can be reduced by minimizing overall volume of shared data, accessed by concurrent processes.
- This can be achieved by making maximum consecutive references to the same data(increasing temporal data locality).
- To maximize the access of local data, it has to brought in the local memory or cache.
- This can be achieved by using proper decomposition and mapping schemes.
- As discussed in section 2.4.1, in matrix multiplication problem, the use of two dimensional mapping reduces the amount of shared data (i.e., matrices A and B) that needs to be accessed by each task to $2n^2/\sqrt{p}$ as compared to $n^2/p + n^2$ in one dimensional mapping.
- It is observed that less volume of nonlocal data need to be accessed in higher dimensional distribution.
- One more way to decrease the amount of shared data is to locally store the intermediate results generated.
- For example, consider the computation of dot product of two vectors of length n in parallel.
- In this each of the p tasks multiplies n/p pairs of elements.
- To reduce the number of accesses to the shared location where the result is stored to p from n, a partial dot product by each task is kept locally.
- After getting all the partial products, the shared location can be accessed only once to add all of them.

2.5.1.2 Minimize Frequency of Interactions

- Generally a high startup cost is associated with each interaction in parallel programs.
- Thus minimized interaction frequency is very important to reduce interaction overhead.
- To achieve this the algorithm can be restructured, such that shared data can be used in large pieces (increasing spatial locality).
- Note that reconstruction of the algorithm does not reduce the overall volume of shared data.
- In case of shared address space model each time a word is accessed, if a program is restructured to have spatial locality, fewer cache lines are accessed instead of fetching an entire cache line containing many words.

- In a message-passing system, spatial locality enables fewer message-transfers over the network because each message can transfer larger amounts of useful data.
- The example of this technique is parallel sparse matrix-vector multiplication.
- In parallel formulation of sparse matrix-vector multiplication, a process interacts with other processes to access elements of the input vector that it may need for its local computation.
- To minimize frequency of interactions, a process can first collect all the nonlocal entries of the input vector that it requires, and then perform an interaction-free multiplication.

2.5.2 Minimizing Contention and Hot Spots

- If same resources are used by the multiple tasks at the same time then contention can be caused.
- The contention can be caused by :
  - Simultaneous transmitting of data over same interconnection lines.
  - Simultaneous access to save memory block.
- If multiple processes are sending message to some process at the same time, only one operation can be done at a time so other tasks have to wait.
- Consider the example of multiplication of two matrices A and B.
- C = AB
- Let p be number of tasks.
- If we consider two dimensional partitions each task computes unique C_{ij} by formula.
- $C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j}$ In this any one of $\sqrt{p}$ steps, $\sqrt{p}$ tasks access same block of A and B.
- The tasks working on same row of C access same block of A i.e. for computing $C_{0,0}, C_{0,1}, \cdots, C_{0,\sqrt{p}-1}$, $A_{0,0}$, will be read at once.
- Same with the case of the columns of C where same block of B is accessed.
- With this context the contention will be caused due to concurrent accessing of blocks A and B.
- This contention can be reduced by modifying the order in which block multiplications are performed by using formula

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}$$

- All the tasks working on the same row of C access block. $A_{*,(*+j+k)\%\sqrt{p}}$ which is different for each task.
- Same will be with the blocks of B.
- The processes computing first block row of the process which calculates $C_{0,j}$ access $A_{0,j}$ from first block row of A instead of $A_{0,0}$.

2.5.3 Overlapping Computations with Interactions

- Computation and interaction in a parallel execution should go hand in hand.
- We know that based on task dependency graph some processes may have to wait for shared data or to get the additional work.
- This waiting time can be reduced by doing some useful computation during this time.
- This is known as overlapping computation with communication.
- Note that overlap between computation and communication may increase with increase in the granularity of the task.
- This overlapping is possible if interaction is initiated early enough so that it is completed before it is needed for computation.
- One way to achieve this is to structure the parallel program such that independent computations are identified and performed before the interaction.
- This can be achieved if interaction pattern is predictable or if a process has multiple tasks which are ready for execution, thus if one task is waiting for interaction the process can take up another task for execution.
- Overlapping can be achieved in case of dynamic mapping scheme by anticipating the additional work needed by the process a priori, so that it does not have to wait till request of work is getting serviced.
- Overlapping computations with interaction should be supported by
  1. Programming paradigm
  2. The operating system
  3. Hardware.
- Concurrent processing of interactions and computations must be facilitated by programming model and these mechanisms must be supported by hardware.
- In separate address space model non-blocking message passing primitives provides this facility.
- In non blocking functions the control is returned to the program before completion of sending and receiving operations.
- Thus the interactions are initiated without interrupting the computations.

- In this case the interaction overhead can be reduced significantly, with the hardware support for concurrent execution of computation with message transfers.
- In case of shared-address-space this overlapping is facilitated with pre fetching hardware.

The memory access

- The prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed.
- In the absence of prefetching hardware, the same effect can be achieved by a compiler that detects the access pattern and places pseudo-references to certain key memory locations before these locations are actually utilized by the computation.
- The degree of success of this scheme is dependent upon the available structure in the program that can be inferred by the prefetch hardware and by the degree of independence with which the prefetch hardware can function while computation is in progress.

2.5.4 Replicating Data or Computations

- To reduce interaction overhead replication of data is a useful technique.
- While dealing with data centric applications, in a parallel algorithm the processes may need frequent access to shared data structures like hash table.
- If the copy of shared data structure is kept with each process interaction overhead can be eliminated.
- In the shared address space model, replication is affected by caches.
- In message passing model data replication is more beneficial as access to read only data is more expensive.
- But there are some limitations with data replication :
  1. Memory requirement increases.
  2. If number of processes running concurrently is more, the amount of memory to store the data increases in turn increasing the size of overall problem on a parallel computer.
- Data replication is beneficial if small amount of data is taken into consideration.
- Sometimes in some operations, processes need the intermediate results also but in some cases instead of getting the results generated from some other process, it will be a cost effective option for a process to generate the result itself.

- For example, while performing the Fast Fourier Transform on an N-point series, N distinct powers of w or "twiddle factors" are computed and used at various points in the computation.
- Different processes require overlapping subsets of these N twiddle factors in parallel implementation of FFT.
- The best way of implementing this in a message-passing model is, each process can locally compute all the twiddle factors it needs.
- Even if in this case number of twiddle factor computations is more it will be still be faster than serial formulation.

2.5.5 Using Optimized Collective Interaction Operations

- Generally it is observed that many times interaction patterns of concurrent activities performed by groups of tasks are static and regular.
- It is typically observed in collective operations like broadcasting some data to all processes or addition of the numbers.
- These collective operations are classified into three categories :
  1. Operations that are used by the tasks to access data.
  2. Operations are used to perform some communication-intensive computations.
  3. Operations used for synchronization.
- To minimize the overheads due to data transfer as well as contention highly optimized implementations of these collective operations have been developed.
- MPI (message passing interface) is one of such standards which provides the libraries and functions like MPI_Broadcast, MPI_Allgather, etc for optimized implementations of these operations.
- Due to this the algorithm designer can focus only on the functionality achieved by these operations and not on their implementation.

2.5.6 Overlapping Interactions with Other Interactions

- To understand overlapping interactions with other interactions let's consider the example of message broadcast among the processes.
- Fig. 2.5.1 shows communication operation one-to all broadcast between four processes P0,P1,P2 and P3 in a message-passing paradigm.
- Aim is to broadcast data from P0 to all other processes.
- According to algorithm and as shown in the Fig. 2.5.1 (a) in the first step, P0 sends the data to P2.

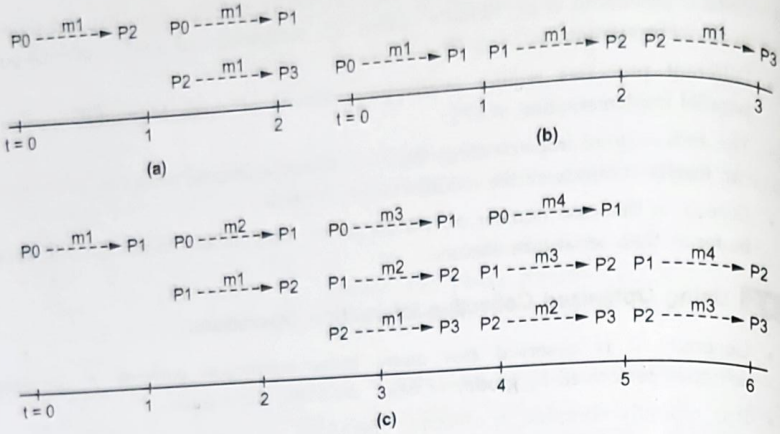


Fig. 2.5.1 : Illustration of overlapping interactions in broadcasting data from one to four processes

- In the second step, P0 sends the data to P1, in the same time step P2 sends the same data that it had received from P0 to P3.
- The operation is thus complete in two steps because two interactions of the second step can be completed in one time step only.
- This is called **overlapping interactions.**
- If underlying hardware supports efficient data transfer, the effective volume of communication can be reduced by overlapping interactions between pairs of processors.
- Consider the simple broadcast algorithm for the same set of processes as shown in Fig. 2.5.1 (b)
- This algorithm takes three steps to finish the same operation.
- It has been observed that in some cases the simple algorithm in 2.5.1 (b) will increase the amount of overlap than 2.5.1 (a).
- Consider the example of broadcast operation of four data structures one after the other.
- If we implement the first strategy of two steps then total eight steps are required draw the diagram.
- If second simple algorithm is implemented as shown in the Fig. 2.5.1 (c) the same operation can be finished in six steps in pipelined fashion.
- But this method is expensive for a single broadcast operation.

Review Questions

1. *What are the methods for reducing interaction overheads ?*
2. *Explain the methods for containing interaction overheads.* **SPPU : Dec.-19, Marks 8**

2.6 Parallel Algorithm Models

- A parallel computer system should be flexible and easy to use.
- It should exhibit good programmability in supporting various parallel algorithmic models.
- Parallel programming models provides different ways to structure algorithms to run on a parallel system.
- Parallel algorithm model is structured by selection of a appropriate decomposition and mapping technique and applying the appropriate strategy to minimize interactions.
- Following parallel algorithm models will be discussed in this section :

2.6.1 The Data-Parallel Model

- To understand data parallel model, consider the example of dense matrix multiplication explained in section 2.1.1
- All the tasks in this problem are identical but they are applied on unique and different data per task.
- This type of parallelism in which identical operations are applied concurrently on different data items is called **data parallelism.**
- Formally data parallel model can be described as, the model in which tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data.
- This is one of the simplest algorithm models.
- The computation can be carried out in phases and the data on which the computations are carried out may be different in different phases.
- To achieve synchronization among the tasks or to get the fresh data to the tasks, the computation phases are intermixed with interactions.
- If data is partitioned uniformly and then if static mapping is applied, load balancing can be achieved.
- So decomposition in this case should be based on data partitioning.
- Data-parallel algorithms can be implemented in both shared-address-space and message passing paradigms.

- There will be less programming efforts if applied on shared address space model, but separate address space allow better control of placement.
- Interaction overheads are reduced by overlapping computation and interaction.
- If size of a problem is bigger, degree of data parallelism increases. In this case more processes are used to solve bigger problems.

2.6.2 The Task Graph Model

Divide and conquer

- As discussed earlier the task dependency graph is used to represent the concurrent computations which can be performed in any parallel algorithm.
- In some problems (like database query example) it plays a major role and in some problems it is less important (dense matrix multiplication)

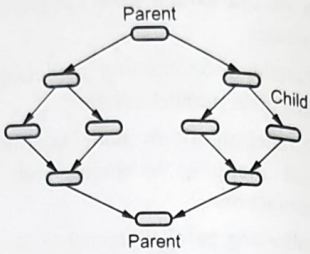


Fig. 2.6.1 The task graph model

- In some parallel algorithms which typically follows divide and conquer strategy, the task dependency graph is used in mapping.
- The type of parallelism that is expressed by independent tasks in a task-dependency graph is called **task parallelism.**
- In the task graph model, the interrelationships among the tasks are used to promote locality or to reduce interaction costs.
- The problems in which the amount of data associated with the tasks is more as compared with the computation, task graph model is used.
- To reduce the cost due to data movement within the tasks static mapping is used.
- In some cases dynamic mapping can also be used, but in any case interaction overhead is minimized by making use of task-dependency graph.
- Various interaction reducing techniques like reducing volume and frequency of interaction by reducing the access time by efficient mapping of the tasks can be applied for efficient implementation. Also interaction can be overlapped with computation for efficient implementation.
- The task graph model can be applied to parallel quicksort, sparse matrix factorization, and many parallel algorithms derived via divide-and conquer decomposition.

2.6.3 The Work Pool Model

Work pool model

- As shown in the diagram in the work pool model the work may be available at the beginning in the work pool or it can be generated dynamically by the processes over the time. In both the cases work will be available in the global work pool.

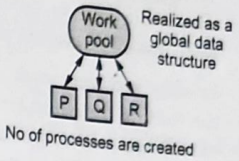


Fig. 2.6.2 The work pool model

- There will not be any pre mapping of the tasks onto processes.
- Mapping of tasks onto processes can be done dynamically to manage load balancing. So any process can execute any task.
- Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure.
- A termination detection algorithm is used to check the completion of all the tasks so that they can stop looking for more work.
- In the message-passing paradigm if the data associated with tasks is smaller than the computation, there will be less data interaction overhead. By this tasks can be readily moved around without causing too much data interaction.
- The granularity of the task is adjusted such that there will be balance between overhead of accessing the work pool for adding and extracting tasks and balancing the load.
- Examples are :
  1. Parallelization of loops by chunk scheduling with centralized mapping when the tasks are statically available.
  2. Parallel tree search where the work is represented by a centralized or distributed data structure where the tasks are generated dynamically.

2.6.4 The Master-Slave Model

Master slave model

- As shown in the Fig. 2.6.3 in the master-slave or the manager-worker model, one or more master processes generate work and allocate it to slave or worker processes.

Fig. 2.6.3 The master - slave model

- There are various ways in which master slave model can be implemented :
  1. If master estimates the size of tasks or if load balancing can be achieved by random mapping then tasks can be allocated apriori.
  2. To assign small pieces of work to the slaves at different times. In case if master takes more time to generate the work the slaves can be assigned pieces of work instead of keeping them idle.
  3. In some problems work should be carried out in phases. It is necessary that the work in the particular phase must be finished to start with the next phase. In such case master synchronises slaves after each phase.
- Generally pre mapping of task onto processes will not be there. Any slave can execute any job assigned to it by master.
- This model can be generalized to the hierarchical or multi-level manager-worker model. In this top level mastres at the highest level of hierarchy assign large tasks to second level masters. Masters can take up part of the work and subdivide and assign the tasks to their slaves.
- The model is suitable to shared address space as well as message-passing paradigms as in both the paradigms the working principle is same i.e. the master gives out work and workers take the work from master.
- Note that if the tasks are too small and slaves are fast then it can cause bottleneck at master.
- Choosing correct granularity is very important to ensure that cost of doing work will always be more than cost of transferring work and the cost of synchronization.
- To overlap communication with computation and to reduce waiting time by slaves asynchronous interaction can be carried out.

2.6.5 The Pipeline or Producer-Consumer Model

Pipeline

- A pipeline can be considered as a chain of producers and consumers.
- Each process in the pipeline act as a consumer. It accepts and consumes a stream of data from the preceding process and produce the data for the process following it in the pipeline.
- Pipelines can be constructed in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles by the processes.

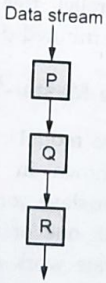


Fig. 2.6.4 The producer consumer model

- In the pipeline model, a stream of data is passed through a chain of processes, each of which perform some task on it.
- The simultaneous execution of different programs on a data stream is called **stream parallelism.**
- Execution of the new task by a process is triggered by new data arriving in the pipeline, except the process that initiates the pipeline.
- Generally static mapping of tasks onto processes is done.
- If coarse grain granularity is adapted the time to fill up the pipeline increases. In this case some of processes have to wait.
- Whereas in case of fine granularity interaction overhead increases because processes will need to interact to receive fresh data after smaller pieces of computation.
- Interactions can be reduced by overlapping interaction with computation.
- An example of a two-dimensional pipeline is the parallel LU factorization algorithm.

2.6.6 Hybrid Models

- If more than one algorithmic model is applied to solve the problem it is called as **hybrid model.**
- Different models can be used in hierarchy or in sequential way to different phases of a parallel algorithm
- For example in task dependency graph the data can trigger the tasks in pipelined manner.
- Hybrid model can be applied in parallel quicksort.

Review Question

1. *Discuss -*
   a) Data - parallel model *b) The task graph model*
   c) The work pool model *d) The master - slave model*
   e) The pipeline / producer - consumer.

2.7 The Age of Parallel Processing

SPPU : May-19

- In recent years, much has been made of the computing industry widespread shift to parallel computing.
- Nearly all consumer computers in the year 2010 are manufactured with multicore central processors.

- From the introduction of dual-core, low-end netbook machines to 8-and 16-core workstation computers, are not only related to supercomputers or mainframes.
- Command prompts are out and multithreaded graphical interfaces are in.
- Additionally electronic devices such as mobile phones and portable music players came up with parallel computing capabilities to enhance the performance.
- Cellular phones that only make calls are out; phones that can simultaneously play music, browse the Web, and provide GPS services are in.
- As a result, software developers now need to cope with a variety of parallel computing platforms and technologies in order to provide novel and rich experiences for an increasingly sophisticated base of users.

Evolution of the CPUs :

- For 30 years, one of the important methods for the improving the performance of consumer computing devices has been to increase the speed at which the processor's clock operated.
- Starting with the first personal computers of the early 1980s, consumer Central Processing Units (CPUs) ran with internal clocks operating around 1 MHz.
- 30 years later, most desktop processors have clock speeds between 1 GHz and 4 GHz, nearly 1,000 times faster than the clock on the original personal computer.
- Although increasing the CPU clock speed is certainly not the only method by which computing performance has been improved, it has always been a reliable source for improved performance.

Review Question

1. *Define and explain the following term - Task interaction graph.* **SPPU : May-19, Marks 2**

2.8 The Rise of GPU Computing

- A Graphics Processing Unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.
- GPUs are used in embedded systems, mobile phones, personal computers, workstations and game consoles.
- Modern GPUs are very efficient at manipulating computer graphics and image processing.
- Their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.

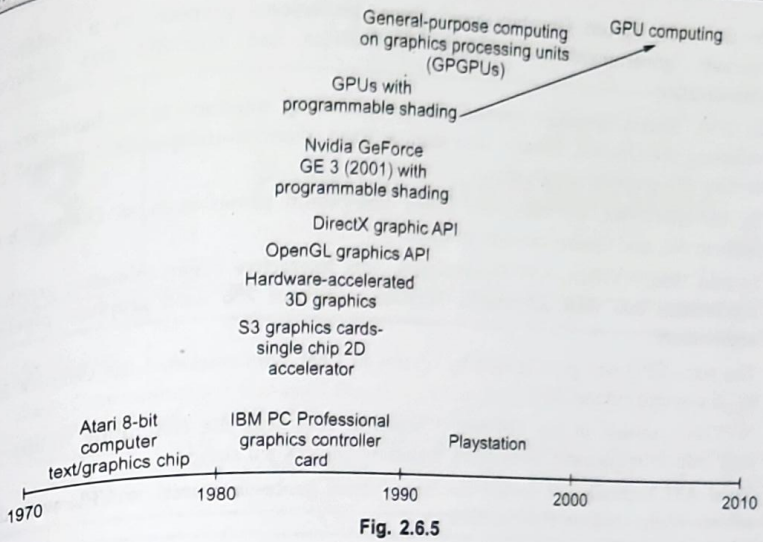


Fig. 2.6.5

- In a personal computer, a GPU can be present on a video card, or it can be embedded on the motherboard or-in certain CPUs-on the CPU die.
- In comparison to the central processor's traditional data processing pipeline, performing general-purpose computations on a Graphics Processing Unit (GPU) is a new concept. In fact, the GPU itself is relatively new compared to the computing field at large. However, the idea of computing on graphics processors is not new.

Review Questions

1. *Explain brief history of GPU's.*
2. *Explain the importance of GPU computing.*

2.9 A Brief History of GPUs, Early GPU

- We have already looked at how central processors evolved in both clock speeds and core count. In the meantime, the state of graphics processing underwent a dramatic revolution.
- In the late 1980s and early 1990s, the growth in popularity of graphically driven operating systems such as Microsoft Windows helped create a market for a new type of processor.
- In the early 1990s, users began purchasing 2D display accelerators for their personal computers, with hardware assisted bitmap operations for graphical operating systems.

- In the 1980, Silicon Graphics used three dimensional graphics in a variety of markets, government and defense applications and scientific and technical visualization
- In 1992, Silicon Graphics opened the programming interface to its hardware by releasing the OpenGL library, as a standardized, platform-independent method for writing 3D graphics applications.
- By the mid-1990s, the computer based first-person games such as Doom, Duke Nukem 3D, and Quake came to market.
- In mid 1990, NVIDIA, ATI Technologies, 3dfx Interactive began releasing graphics accelerators that were affordable. NVIDIA's GeForce 256 used graphics pipeline architecture.
- The term GPU was popularized by Nvidia in 1999, who marketed the GeForce 256 as "the world's first GPU".
- NVIDIA's release of the GeForce 3 series in 2001 was the computing industry's first chip to implement Microsoft's then-new DirectX 8.0 standard.
- Rival ATI Technologies coined the term "visual processing unit" or VPU with the release of the Radeon 9700 in 2002.

University Questions with Answers

Oct. - 2019

Q.1 *What are characteristics of task and interactions ?* **(Refer section 2.3)** **[4]**

Q.2 *Explain decomposition, task and dependancy graph.* **(Refer section 2.1.1)** **[6]**

Q.3 *Explain with suitable example - i) Recursive decomposition ii) Data decomposition iii) Exploratory decomposition* **(Refer section 2.2)** **[3]**

Q.4 *What are limitations of parallelization of any algorithm ?* **(Refer section 2.1.2)** **[4]**

May - 2019

Q.5 *Differentiate between static and dynamic mapping techniques for load balancing.* **(Refer section 2.4)** **[6]**

Q.6 *Define and explain the following terms : i) Granularity* **(Refer section 2.4)**
ii) Task interaction graph **(Refer section 2.7)**
iii) Degree of Concurrency. **(Refer section 2.1.2.1)** **[6]**

Dec. - 2019

Q.7 *Explain mapping techniques for load balancing.* **(Refer section 2.4)** **[6]**

Q.8 *Explain the methods for containing interaction overheads.* **(Refer section 2.5)** **[8]**

❑❑❑