Syllabus

Course Code	Course/Subject Name	Credits
DLO8011	High Performance Computing	4

Objectives :

1. To learn concepts of parallel processing as it pertains to high-performance computing.
2. To design, develop and analyze parallel programs on high performance computing resources using parallel programming paradigms.

Outcomes : Learner will be able to…

1. Memorize parallel processing approaches.
2. Describe different parallel processing platforms involved in achieving High Performance Computing.
3. Discuss different design issues in parallel programming.
4. Develop efficient and high performance parallel programming.
5. Learn parallel programming using message passing paradigm using open source APIs.

Module	Details	Hours
1.	**Introduction :** **Introduction to Parallel Computing :** Motivating Parallelism, Scope of Parallel Computing, Levels of parallelism (instruction, transaction, task, thread, memory, function) **Classification Models :** Architectural Schemes (Flynn's, Shore's, Feng's, Handler's) and Memory access (Shared Memory, Distributed Memory, Hybrid Distributed Shared Memory) **Parallel Architectures :** Pipeline Architecture, Array Processor, Multiprocessor Architecture, Systolic Architecture, Data Flow Architecture (**Refer Chapter 1**)	06
2.	**Pipeline Processing :** Introduction, Pipeline Performance, Arithmetic Pipelines, Pipeline instruction processing, Pipeline stage design, Hazards, Dynamic instruction scheduling (**Refer Chapter 2**)	08
3.	**Parallel Programming Platforms :** Parallel Programming Platforms: Implicit Parallelism : Trends in Microprocessor & Architectures, Limitations of Memory System Performance, Dichotomy of Parallel, Computing Platforms, Physical Organization of Parallel, Platforms, Communication Costs in Parallel Machines (**Refer Chapter 3**)	10

Module	Details	Hours
4.	**Parallel Algorithm Design :** **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 **(Refer Chapter 4)**	12
5.	**Performance Measures :** **Performance Measures :** Speedup, execution time, efficiency, cost, scalability, Effect of granularity on performance, Scalability of Parallel Systems, Amdahl's Law, Gustafson's Law, Performance Bottlenecks **(Refer Chapter 5)**	06
6.	**HPC Programming :** **Programming Using the Message-Passing Paradigm :** Principles of Message Passing Programming. **The Building Blocks :** Send and Receive Operations **MPI :** The Message Passing Interface, Topology and Embedding, Overlapping Communication with Computation, Collective Communication and Computation Operations, Introduction to OpenMP **(Refer Chapter 6)**	10

Module - I

Syllabus : Introduction to Parallel Computing : Motivating Parallelism, Scope of Parallel Computing, Levels of parallelism (instruction, transaction, task, thread, memory, function) **Classification Models :** Architectural Schemes (Flynn's, Shore's, Feng's, Handler's) and Memory access (Shared Memory, Distributed Memory, Hybrid Distributed Shared Memory) **Parallel Architectures :** Pipeline Architecture, Array Processor, Multiprocessor Architecture, Systolic Architecture, Data Flow Architecture.

Module - II

Syllabus : Introduction, Pipeline Performance, Arithmetic Pipelines, Pipeline Instruction processing, Pipeline stage design, Hazards, Dynamic Instruction scheduling.

Module - III

Syllabus : Parallel Programming Platforms : Implicit Parallelism : Trends in Microprocessor & Architectures, Limitations of Memory System Performance, Dichotomy of Parallel, Computing Platforms, Physical Organization of Parallel, Platforms, Communication Costs in Parallel Machines.

Module - IV

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.

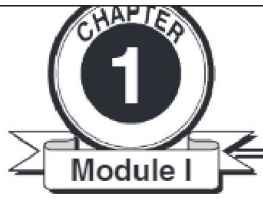
Module - V

Syllabus : Performance Measures : Speedup, execution time, efficiency, cost, scalability, Effect of granularity on performance, Scalability of Parallel Systems, Amdahl's Law, Gustafson's Law, Performance Bottlenecks.

Module - VI

Syllabus : Programming Using the Message-Passing Paradigm : Principles of Message Passing Programming.
The Building Blocks : Send and Receive Operations **MPI :** The Message Passing Interface, Topology and Embedding, Overlapping Communication with Computation, Collective Communication and Computation Operations, Introduction to OpenMP.

❑❑❑



Introduction

Syllabus

Introduction to Parallel Computing : Motivating Parallelism, Scope of Parallel Computing, Levels of parallelism (instruction, transaction, task, thread, memory, function)

Classification Models : Architectural Schemes (Flynn's, Shore's, Feng's, Handler's) and Memory access (Shared Memory, Distributed Memory, Hybrid Distributed Shared Memory)

Parallel Architectures : Pipeline Architecture, Array Processor, Multiprocessor Architecture, Systolic Architecture, Data Flow Architecture.

1.1 Necessity of High Performance

- There is always a need of increase in the speed for a computing system. The computations required from a system have always been increasing. All the complicated operations in research, medical, weather forecast, artificial intelligence, automation, defence, etc required powerful computation, leading to the necessity of high performance.
- Many things have been implemented to make the system faster. In the beginning, the processor developers came up with a prefetch system (in 8086 processor).
- Then it was found that although the instructions were prefetched, it was not performing as fast as expected. Researchers came with new concepts to increase the speed. They came up with different things like cache memory, internal cache, pipelined processors, superscalar processors, parallel computing etc. These things evolved one after the other to increase the speed of the system. All these are to be studied in this text.

1.2 Motivation for Parallelism

We will take a short glance into the history of parallel computing to better understanding its origin despite its recent rise in popularity. Parallel execution was present ever since the early days of computing when computers were still mechanical devices. In accelerating computing speeds, the role of parallelism has been recognized for several decades.

1.2.1 Increase Number of Transistors in IC Moore's Law

The computational power enhancement through increase number of transistors in IC :

Moore's Law (1965)

- "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000."
- This statement is by Moore in 1965, which predicted that over the next 10 years computers will become fast, very fast. Now that 10 years elapsed and it turned out that computers continued to become fast and also computer speed is going to be essentially doubling every 18 months and that happen by variety of techniques.

- In 1975, Moore predicted that we have essentially saturated out intelligence, we cannot make innovation into making things better and faster.
- But still just by advent of technology of being able to lay **thinner wires on silicon**, we will able to pack in more and **more transistors** which will in turn make computers faster. There are two parts, one was that you can drive the clocks much faster, drive the gates using a faster clock which have saturated in last few years.
- In modern processor, the power of CPU increases by adding additional transistors in the integrated circuit or chip.

1.2.2 Memory / Disk Speed Improvement

- The overall speed of the computation is not only dependent on the speed of the processor but other factors are also involved for the same.
- Like memory and disk speed also play important role in computation speed improvement.
- Parallel platforms provide increased bandwidth to the memory system.
- Using principle of locality of reference which is used to manage the mismatch between the memory and the processor speed and using bulk access, we can apply memory optimization techniques for parallel algorithm design.
- Parallel platforms also provide higher aggregate cache memory.
- In parallel computing, most of the application areas rely on the ability to pump data to memory and disk faster and not on the computational rates.

1.2.3 Data Communication Improvement

- As the networking infrastructure evolves, the vision of the Internet as one large computing platform has emerged.
- In many applications like databases and data mining, the volume of data is such that they cannot be moved.
- Any analyses on this data must be performed over the network using parallel techniques.

1.3 Scope of Parallel Computing

The use of parallel computing is in many fields such as Aerodynamics, Automotive, Biology, consulting, Database, Defence, Electronics, Energy, Environment modelling, Finance, Geophysics, Information service, Life Sciences, Medicine, Telecommunication, Transportation, Weather- climate research, Artificial Intelligence, Computer vision etc.

Let us see some of these applications in detail.

1.3.1 Application of Parallel Processing in Weather-Climate Research

Weather forecast is a very important requirement for agriculture, fishing, etc. The weather-climate forecast requires a lot of parameters. It requires various information like :

1. Humidity
2. Wind direction and speed.
3. Daily maximum and minimum air temperatures
4. Total solar radiation
5. Carbon dioxide concentration
6. Ocean sea ice
7. Cloud physics parameterization and many more things.

All these parameters are to be worked on simultaneously. Climate models are said to be computationally intensive because

1. There is a lot of stuff to calculate. The prediction is based on statistics of previous century on the time scale with precision of a minute. Hence the calculation is too much. For example, the huge earth (almost a sphere) has to be divided into discrete units, and thus forming a grid of latitudes and longitudes. The cells formed are of non-uniform size as shown in Fig. 1.3.1.

Fig. 1.3.1 : Non-uniform cells of the earth

These calculations require spherical co-ordinates (θ,ϕ) which are uniform. It requires spectral transforms or FFTs. Thus the computation of the parameters for such a huge grid will require huge time on non parallel system. Another option is dividing the surface into parts to form two dimensional plane an work on these small parts simultaneously. This will also require special type of topological arrangement of processors. The algorithm whatsoever will require lots of computations, must have very high accuracy, vector processing, communications, huge cache memory, petaflop computing, large database etc.

1.3.2 Application of Parallel Processing in Medical

- In medical, there is a requirement of very fast computation. This can be implemented by a huge scale of parallelism. The modern computing has made many things viable in the medical industry. Especially many medical applications that require imaging and processing on these images, require parallel computing.
- Magnetic Resonance Imaging (MRI) for example that is used see inside the human body requires very fast as well as huge computations. These computations that develop the 3D imaging of inside of the body, need to also have very accurate calculations. Magnetic resonance imaging (MRI) is done to find bleeding, tumors, injury, blood vessel diseases, infection etc. It uses magnetic field and radio frequency pulses to make the pictures of the organs inside the body. Thus while scanning the body, the image capturing, computing and reproducing must be fast enough so as to not allow the patient remain in the radio frequency waves for long.
- Various parallel mechanisms specially MPP (Massively Parallel Processing) are used. Array processors and multicomputer processing are normally under the category of MPP.

1.4 Levels of Parallelism

1.4.1 Levels of Parallel Processing / Software Parallelism

Parallel processing can be done in four different levels:

1. **Transaction (program) or Job level**

The program level parallelism requires the development of parallel processable algorithms. The implementation of these algorithms will depend on the efficiency of the allocation processor of the limited resources (hardware and software) to the multiple programs to solve the large computational problem. Examples of such systems are weather forecasting, medical consulting, oil exploration etc. This is an example of large grain parallelism i.e. the grains on which parallelism is implemented is very large.

2. **Task or Function level**

This can also be refered to as thread level parallelism. The different parts of the codes or threads are to be executed simultaneously on the different processing elements and hence achieving high performance. The details of this kind of parallelism will be discussed in section 1.7. This is also an example of medium grain of parallelism which is also sometimes said to be control level parallelism.

3. **Inter-Instruction level**

Different instructions of a program can be executed simultaneously in a single processor by the different units performing different stages of execution of these instructions. This also is a key type of feature implemented in the single processor to achieve the instruction level parallelism and will be also studied in detail in the section 1.7. This is also referred to as fine grain level of parallelism.

4. **Intra Instruction level**

There are different operations to be performed in a single instruction and similar operations to be performed on a huge number of data. This can be achieved by various units like carry look ahead adder and carry save adders instead of the traditional ripple adder. This is a case of very fine grain parallelism.

5. **Memory level parallelism**

- Memory Hierarchy makes the system faster. Memory access is an operation in the system that requires maximum time. If a faster memory is required, it has to be smaller to have low access time as well as the cost of such memory is very high. Hence we need to implement a special system with the principles of locality of reference. These principles ensure that if you have a small amount of fast memory and a huge amount of slow memory, you will get high performance by following the principles. The principles are stated below.
- The locality of reference principle is comprised of two components:

a. Temporal locality	b. Spatial locality

- Let us see these principles of locality of reference.
  - **a. Temporal locality :** The programs have loops. These loops are executed many times. Instead of bringing the instructions from slow memory every time, they must be brought and kept in the faster memory. Thus when running the same instructions in the second iteration onwards they would be available in faster memory and hence get the access faster.
  - **b. Spatial locality :** Normally the instructions and data are stored sequentially. Hence whenever a data is required by the processor, a huge block of data must be brought from the slower memory to the faster memory.
- Hence the processor is connected to the system in the following manner
  - i. Registers inside the CPU
  - ii. Internal (inside CPU) Cache
  - iii. External (outside CPU) Cache
  - iv. Internal or Main memory (RAM)
  - v. External memory (Hard disk, CD etc.)
- The advantage of memory hierarchy that results into parallel operation is that there are multiple buses. Each bus can be concurrently used to perform the different transfers in a system.

6. **Thread level parallelism**
   - This is another very important feature supported by many Operating Systems (OS) that allows multiple threads or processes to be executed by the processor simultaneously on a time sharing basis.
   - This also sometimes gives parallel processing and hence increasing the speed; especially because a task may be waiting for an operation to be completed and this time can be utilized by another task or thread.

1.4.2 Future Trends in Parallel Processing

- A lot of progress has been done in the last four decades in parallel computing. The progress of semiconductors has also supported these features.
- Parallel processing still needs some progress in vector and array processing. The programming of these systems is still quite complicated. Also based on the image processing and signal processing there are required to be function specific parallel processors. Medical field is also requiring new methods of calculations with fast processing where parallel computing still has scope to improve.

1.5 Parallelism in Uni-Processor

- Parallelism refers to performing multiple operations simultaneously in a system.
- There are many ways of achieving this parallelism in a single processor or uni-processor. This can be done by

1. Overlapping the CPU and Memory or I/O operations.
2. Pipelining
3. Multiple functional units
4. Memory Hierarchy
5. Multi-threading

1.5.1 Overlapping the CPU and Memory or I/O Operations

- This is a very basic parallelism implemented in the Intel's 8086, wherein we had instructions prefetched from the memory before they are to be executed. Also for I/O operations a special dedicated I/O processor can be connected.
- Hence all the operations i.e. accessing the data or instructions from memory, accessing I/O devices and internal ALU operations can be done simultaneously. In the 8086 processor, there are two separate units to perform the memory accesses and the ALU operations named as Bus Interface Unit (BIU) and the Execution Unit (EU).

1.5.2 Pipelining

- A processor has many resources like the ALU, buses, registers, etc. An attempt to utilize all these attributes to their fullest or continuously can be achieved by pipelining. In a pipelined system the instructions flow through the processor as if the processor was a pipe. The instructions move from one stage to another to accomplish the assigned operation. Hence at most of the times each unit of the processor is busy handling one or the other instruction, making the attribute of the processor being used continuously.
- In a non-pipelined system, the processor fetches an instruction from memory, decodes it to determine what the instruction was, read the instructions inputs from the register file, performs the computation required by the instruction and writes the result back into the register file. This approach is also called as unpipelined approach.

- The problem with this approach is that, the hardware needed to perform each of these steps (Instruction fetch, instruction decode, register read, instruction execution and register write-back) is different and most of the hardware is idle at any given moment. Waiting for the other parts of the processor to complete their part of executing an instruction.
- Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.
- Two stage pipelining includes two stages i.e. Fetch instruction and Execute instruction.
- These two operations are performed for one instruction and the next instruction overlapping i.e. when the first instruction is being executed the next is fetched and when this instruction is executed the next is fetched and so on.
- This method of execution the instructions in pipeline speeds up the processor operation.
- This also makes sure that all the units of the processor are busy operating and none of them is starving.
- Thus with the help of pipelining the operation speed of the processor increases i.e. more the number of pipeline stages, faster becomes the processor, but complex in design.
- A simple two stage instruction pipeline stage is as shown in Fig. 1.5.1.

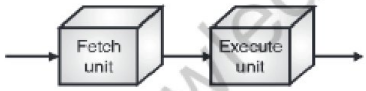


Fig. 1.5.1 : Two stage pipelining

- Pipelining is implemented in almost all the modern processors. We will discuss in more detail about pipelining in chapter 2.

1.5.3 Multiple Functional Units or Hardware Level Parallelism

A processor may have multiple functional units. There may be many units that perform the same operation. Whenever a particular operation is required the unit which is free will perform the operation. For example there may be multiple ALUs and whenever there are more ALU operations to be performed, they will be forwards to the ALU which is free. Examples of such processors are Pentium, Itanium etc. Pentium has two ALUs and hence two arithmetic / logic operations can be carried out simultaneously by them.

1.5.4 Memory Hierarchy

- Memory Hierarchy makes the system faster. Memory access is an operation in the system that requires maximum time. If a faster memory is required, it has to be smaller to have low access time as well as the cost of such memory is very high. Hence we need to implement a special system with the principles of locality of reference. These principles ensure that if you have a small amount of fast memory and a huge amount of slow memory, you will get high performance by following the principles. The principles are stated below.
- The locality of reference principle is comprised of two components :

1. Temporal locality	2. Spatial locality

- Let us see these principles of locality of reference.

1. **Temporal locality :** The programs have loops. These loops are executed many times. Instead of bringing the instructions from slow memory every time, they must be brought and kept in the faster memory.

Thus when running the same instructions in the second iteration onwards they would be available in faster memory and hence get the access faster.

2. **Spatial locality :** Normally the instructions and data are stored sequentially. Hence whenever a data is required by the processor, a huge block of data must be brought from the slower memory to the faster memory.

- Hence the processor is connected to the system in the following manner
  a. Registers inside the CPU
  b. Internal (inside CPU) Cache
  c. External (outside CPU) Cache
  d. Internal or Main memory (RAM)
  e. External memory (Hard disk, CD etc.)
- The advantage of memory hierarchy that results into parallel operation is that there are multiple buses. Each bus can be concurrently used to perform the different transfers in a system.

1.5.5 Multi-threading

- This is another very important feature supported by many Operating Systems (OS) that allows multiple threads or processes to be executed by the processor simultaneously on a time sharing basis.
- This also sometimes gives parallel processing and hence increasing the speed; especially because a task may be waiting for an operation to be completed and this time can be utilized by another task or thread.

1.6 Instruction Level Parallelism vs. Thread Level Parallelism

1.6.1 Pipelining

- The classification of pipelining according to the level of processing, namely:
  (a) Instruction level pipelining or Instruction pipeline
  (b) ALU level pipelining or Arithmetic pipeline
  (c) Processor level pipelining
- Processor level pipelining is a MISD system. It has multiple instructions operating on a single data. But each processor performs a different operation on the data. The floating point addition (discussed in chapter 2) is an example of this.
- An arithmetic pipeline may be a static pipeline or dynamic pipeline. The static arithmetic pipelines can perform only fixed functions while the dynamic pipelines can perform different operations.
- Instruction level parallelism refers to pipelining. Pipelining is overlapping of various instructions to be executed simultaneously at different stages in the processor as already discussed.
- A processor has many resources like the ALU, buses, registers, etc. An attempt to utilize all these attributes to their fullest or continuously can be achieved by pipelining. In a pipelined system the instructions flow through the processor as if the processor was a pipe. The instructions move from one stage to another to accomplish the assigned operation. Hence at most of the times each unit of the processor is busy handling one or the other instruction, making the attribute of the processor being used continuously.
- In a non-pipelined system, the processor fetches an instruction from memory, decodes it to determine what the instruction was, read the instructions inputs from the register file, performs the computation required by the instruction and writes the result back into the register file. This approach is also called as unpipelined approach.

- The problem with this approach is that, the hardware needed to perform each of these steps (Instruction fetch, instruction decode, register read, instruction execution and register write-back) is different and most of the hardware is idle at any given moment. Waiting for the other parts of the processor to complete their part of executing an instruction.
- Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.
- Two stage pipelining includes two stages i.e. Fetch instruction and Execute instruction.
- These two operations are performed for one instruction and the next instruction overlapping i.e. when the first instruction is being executed the next is fetched and when this instruction is executed the next is fetched and so on.
- This method of execution the instructions in pipeline speeds up the processor operation.
- This also makes sure that all the units of the processor are busy operating and none of them is starving.
- Thus with the help of pipelining the operation speed of the processor increases i.e. more the number of pipeline stages, faster becomes the processor, but complex in design.
- A simple two stage instruction pipeline stage is as shown in Fig. 1.6.1.

Fig. 1.6.1 : Two stage pipelining

- Pipelining is implemented in almost all the modern processors. We will discuss in more detail about pipelining in chapter 2.

1.6.2 Thread Level Parallelism

- Thread level parallelism makes use of multiple threads of execution that are independent of each other. A thread is nothing but a process or a program with its independent data.
- Multiple threads may be in execution in a system simultaneously or may be one at a time depending on the number of processing elements in the system.
- Thread level parallelism is more effective then Instruction level parallelism as the instructions of different threads are being executed which are not dependent.
- In real systems, thread level and instruction level parallelism are used together. This makes the system cost effective.

1.7 Explicitly Parallel Instruction Computing (EPIC)

- EPIC is a term designed by the joint venture of HP (Hewlett-Packard) and Intel, when designing the Itanium processor. EPIC uses multiple functional units in the processor to perform multiple operations simultaneously.
- We will see an actual example of EPIC processor i.e. Itanium to better understand the concept of EPIC
- EPIC is a new design philosophy going beyond the RISC and CISC processors that are available today.

- EPIC technology is evolved by combining VLIW (Very Large Instruction Word, wherein each instruction length is large enough to accommodate multiple operations), OOO (Out-of-order execution) and superscalar (multiple execution units). VLIW and OOO will be discussed in chapter 2 and Chapter 3.
- EPIC technology enables greater instruction level parallelism than previous processor architectures, supporting higher levels of performance. The Itanium architecture is based on EPIC technology.
- The EPIC architecture uses VLIW type instruction wording which contains multiple instructions in a single block called as bundle as shown in Fig. 1.7.1.

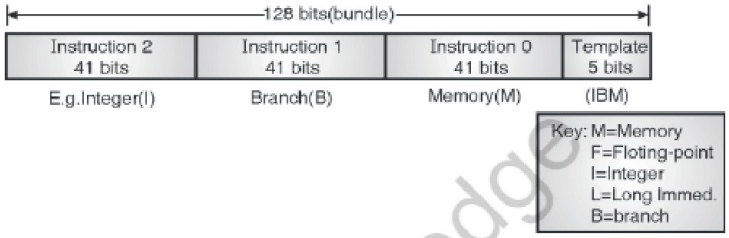


Fig. 1.7.1 : A 128-bits bundle

- Bundles instructions are sent to the CPU together other instructions. The "bundles" are put together in an "instruction group" with other instructions.
- The Itanium Architecture-64 (IA-64) utilizes 128 bit bundles, organized as a template plus three IA-64 instructions. The template contains information provided by the compiler to the processor.
- The template contains the information of which type of instructions are combined to be executed in parallel and hence to make a bundle, as shown in Fig. 1.7.1.
- Instructions in a bundle are such combined that they do not affect each other with the data they are working on, so they can run together without causing contention. Instruction groups are a series of bundles that are given together to the processor, of which 6 can be executed per clock.
- There is no limit to the size of an instruction group, and an instruction group can begin or end in the middle of a bundle. The instructions are bundled and grouped together when program is compiled i.e. the compiler itself decides the instructions that can be executed together.
- This simplifies the process of running multiple instructions at once on an Itanium CPU, allowing it to make greater use of multiple execution units without having to rely on complex on-die logic to determine what operations can run in parallel.
- The Itanium will still use on-die logic to improve upon instruction level parallelism, but EPIC instructions, at the minimum, provide a parallel execution for the Itanium processor. Hence, compiler technology and programming algorithms will have a massive impact on Itanium performance.
- The compiler adds branch hints, register stack and rotation, data and control speculation, and memory hints into EPIC instructions. Thus, EPIC makes processing much faster.

1.8 Architectural Schemes of Parallel Processing

Parallel processing is the ability to process multiple tasks simultaneously. There are various architectural classifications of parallel computing. These classifications are given by different people like Flynn, Feng, Handler and Shore's etc. We will see the classification of parallel processing laid down by these four people.

1.8.1 Flynn's Classification of Parallel Computing

A method introduced by Flynn, for classification of parallel processors is most common. This classification is based on the number of Instruction Streams (IS) and Data Streams (DS) in the system. There may be single or multiple streams of each of these. Hence accordingly, Flynn classified the parallel processing into four categories :

1. Single Instruction Single Data (SISD)
2. Single Instruction Multiple Data (SIMD)
3. Multiple Instruction Single Data (MISD)
4. Multiple Instruction Multiple Data (MIMD)

1. Single Instruction Single Data (SISD)

- In this case there is a single processor that executes one instruction at a time on single data stored in the memory.
- In fact, this type of processing can be said to be uniprocessing, hence uniprocessors fall into this category.
- Fig. 1.8.1 shows this type of system. You will notice there is a Control Unit (CU) that accepts the instruction from the processor and decodes it. The Processing Element (PE) accesses the data from the memory and performs the operation on this data as per the signal given by control unit. The Memory Module (MM) is connected to the PE and the CU for the data and the instruction streams respectively.

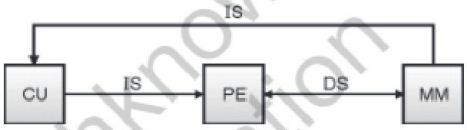


Fig. 1.8.1 : SISD computer

2. Single Instruction Multiple Data (SIMD) :

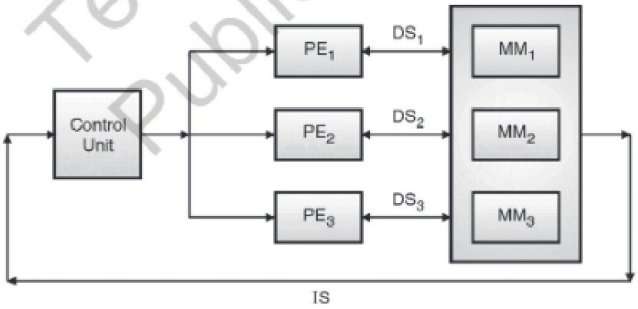


Fig. 1.8.2 : SIMD organisation

- In this case the same instruction is given to multiple processing elements, but different data. This kind of system is mainly used when many data (array of data) have to be operated with same operation. Vector processors and array processors fall into this category.
- Fig. 1.8.2 shows the structure of a SIMD system.

3. Multiple Instruction Single Data (MISD)

- In case of MISD, there are multiple instruction streams and hence multiple control units to decode these instructions. Each control unit takes a different instruction from the different memory module in the same memory.
- The data stream is single. In this case the data is taken by the first processing element. This processing element performs an operation on the data given to it and forwards the result to the next processing element for further operation. This processing element performs a similar operation and so on the final result reaches back to the same memory module.

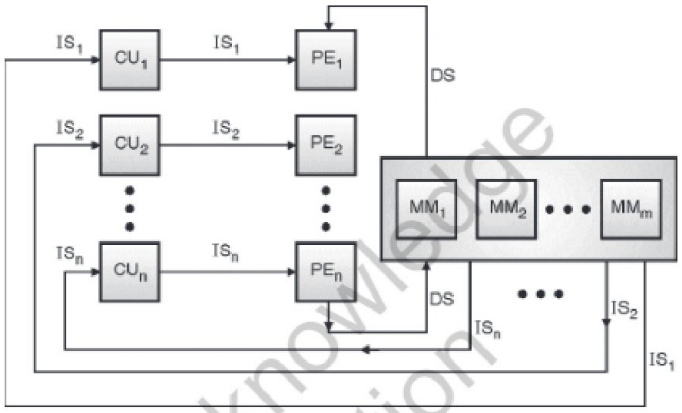


Fig. 1.8.3 : MISD computer

- MISD is not used much as its applications are very limited and are expensive. This system is not used much, but can be used in cases where in a data has to undergo many computations to get the result for e.g. to add two floating point numbers. Fig. 1.8.3 shows the implementation of such a system.

4. Multiple Instruction Multiple Data (MIMD)

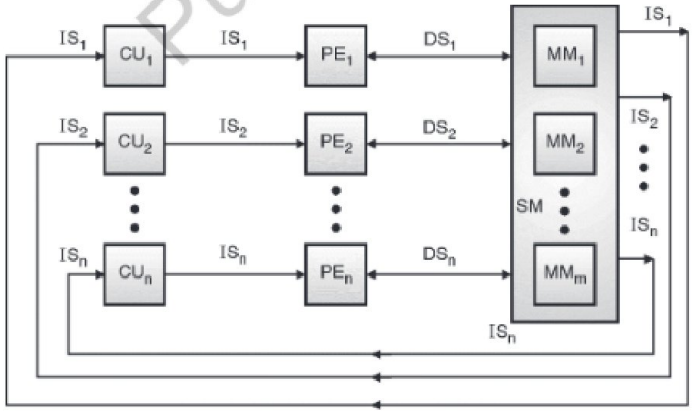


Fig. 1.8.4 : MIMD computer

- This is a complete parallel processing example. Here each processing element is having a different set of data and different instructions.
- Examples of this kind of systems are SMPs (Symmetric Multiprocessors), clusters and NUMA (Non-Uniform Memory Access). Fig. 1.8.4 shows the structure of such a system.

1.8.2 Feng's Classification of Parallel Computing

Feng also proposed a system for classifying the parallel processing systems. This classification is based on the number of bits in a word and the word length. The parallelism is based on the parallelism of bits and words. Hence we can have the bits of a words processed in parallel or serially. Similarly we can have words processed in parallel or serially. Thus resulting in the following four categories:

1. Word Serial Bit Serial (WSBS)
2. Word Serial Bit Parallel (WSBP)
3. Word Parallel Bit Serial (WPBS)
4. Word Parallel Bit Parallel (WPBP)

1. Word Serial Bit Serial (WSBS)

In this case one bit of a selected word is processed at a time. This corresponds to serial processing and hence requires maximum processing time.

2. Word Serial Bit Parallel (WSBP)

In this case all the bits of a selected word are processed simultaneously, but one word at a time. Here bit parallel indicates selection of all the bits of a word. Hence it is slightly parallel processing.

3. Word Parallel Bit Serial (WPBS)

In this case one selected bit from all the specified words are processed at a time. Here word parallel indicates selection of multiple words. This is again slight parallelism. WSBP can be said to be row parallelism while WPBS can be said as column parallelism.

4. Word Parallel Bit Parallel (WPBP)

- In this case all the bits of all the specified words are operated on simultaneously. Hence this gives the maximum parallelism thereby minimum time for execution.
- The processors are classified according to Feng's classification can be shown in the graphical representation with number of bits vs. number of words in parallel. Fig. 1.8.5 shows this graphical representation.

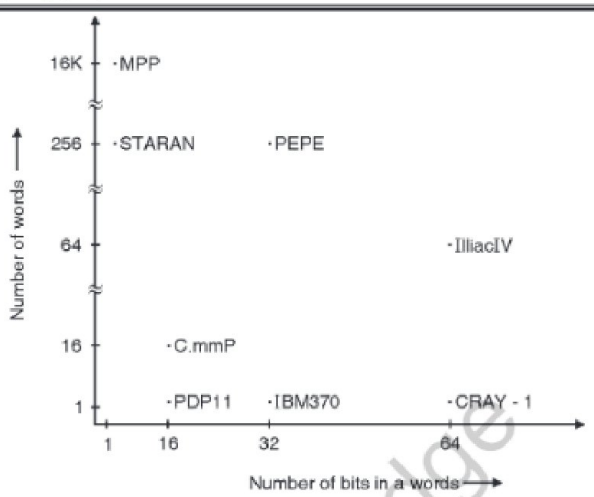


Fig. 1.8.5 : Processor on graph of number of words vs. word size executed in parallel

- You will notice in Fig. 1.8.5 that processors like IBM370, Cray-1, PDP11 have one word executed in parallel but the word size varies from 16 to 64 bits. Thus these processors can be said to be WSBP processors.
- STARAN and MPP processors have one bit of a word executed at a time but multiple words together. These processors can hence be categorised under WPBS.
- Processors like C.mmP, PEPE and IlliacIV are having multiple bits and multiple words being executed simultaneously and hence can be categorised under WPBP.

1.8.3 Handler's Classification of Parallel Processing

- Handler's method of classification takes a very important aspect into consideration i.e. Pipelining. Handler stated that a computing system (C) can be characterized as :

$$T(C) = (K \times K', D \times D', W \times W')$$

where,

- K is the number of Processor Control Units (PCUs) in the system
- K′ are the number of processors that are pipelined
- D is the number of Arithmetic and Logical Units (ALUs) in the system
- D′ is the number of pipelined ALUs
- W is the number of bits in a word or the word size of the Processing Elements (PEs) or ALUs
- W′ is the number of pipeline stages in the PEs or ALUs
- Let us define certain processors in this manner.

Note : If the second term (i.e. the terms K', D' or W') is '1', then it can be omitted.

- The Advanced Super Computer (ASC) of Texas Instruments (TI) is characterized as :

  TI-ASC (1, 4, 64 × 8)

  which means that has one PCU, 4 ALUs of 64-word and each word of 8-bit to be operated in parallel.
- Let us take an example of PEPE processor. It is characterized as:

PEPE (1 × 3, 288, 32)

- Which means that this processor has one processing control unit and three pipelined processors. It has 288 ALUs of 32-bit word size each.
- Another example of CRAY-1 processor. It is characterized as:

  CRAY-1 (1, 12 × 8, 64 × 1-14)

- Which means that this processor has one processing control unit. It has 12 ALUs pipelined as 8 ALUs in each pipeline. Also the word length of these ALUs is 64 bits and the number of words that can be operated on simultaneously can vary from 1 word to 14 words.

1.8.4 Shore's Classification

The shores classification is on the basis of organization of the components of the system. There are six machines recognized by him as complete classification

1.8.4(A) Machine 1: Von Neumann Architecture

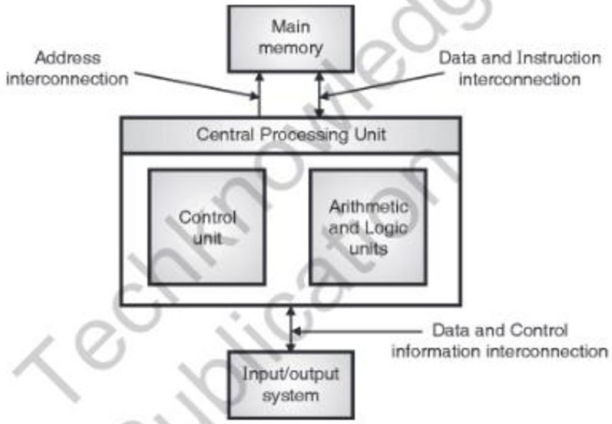


Fig. 1.8.6 : Von Neumann Architecture of a computer

- Here conventional architecture refers to the uni-processor system. Some of the parallelism features can definitely be implemented on a single processor to improve the speed, but there is a limitation to the same. Let us discuss certain basics of single processor.
- The architecture of processors began with the IAS's Von Neumann Computer. Fig. 1.8.6 shows the architecture of the Von Neumann architecture.
- This system has three units CPU, Memory and I/O devices. The CPU has two units Arithmetic Unit and Control unit. Let us discuss these units in detail.

1. Input Unit

- A computer accepts inputs from the user through these devices i.e. input devices. The commonly used input devices are keyboard and mouse.
- Besides that, there are devices like joystick, camera, scanner etc which are also input devices. The devices input the data accepted from the user in a proper coded form understood by the computer.

2. **Output Unit**
   - The result is given back by the computer to the user through an output device. Input devices and output devices are also called as human interface devices, because they are used to interface the human to the computer.
   - The mainly used output devices are monitor and printer. But there are many other output devices like plotter, speaker etc.
3. **Arithmetic Logic Unit (ALU)**

   Arithmetic or logic operations like multiplication, addition, division, AND, OR, EXOR etc are performed by ALU. Operands are brought into the ALU, where the necessary operation is performed.
4. **Control Unit**

   The control unit as we know is the main unit that controls all the operations of the system, inside and outside the processor. The memory or I/O devices have to be controlled by the computer to perform the operation according to the instruction given to it.
5. **Memory Unit**
   - Memory is used to store the programs and data for the computer. The instructions from the programs are taken by the processor, decoded and executed accordingly.
   - The data is also stored in the memory. The data is taken from memory and the operation is performed on that data, as well as the results are stored in the memory.
   - In some cases the input to an operation and the result may also be from input and output devices. Memory in the Von Neumann system has a special organization wherein the data and instructions are stored in the same memory.

1.8.4(B) Machine 2

This organization is similar to machine 1 except that Data Memory (DM) Fetches a bit slice from all the words in the memory and PU is organized to perform the operations in a bit serial manner on all the words. If the memory is regarded as a two dimensional array of bits with one word stored per row, then the machine 2 reads vertical slice of bits and processes the same, whereas the machine 1 reads and processes a horizontal slice of bits Fig. 1.8.7. Examples are ICLDAP and MPP

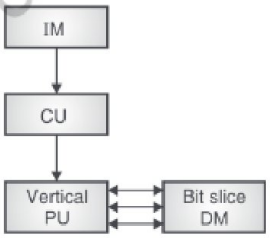


Fig. 1.8.7

1.8.4(C) Machine 3

In case of machine 3, it is combined version of machine 1 and 2. The memory is an array of bits with horizontal as well as vertical readability. Hence there is a possibility of having both horizontal as well as vertical processing units. An example of machine using this system is Omenn 60. Fig. 1.8.8 shows the machine 3 architecture.

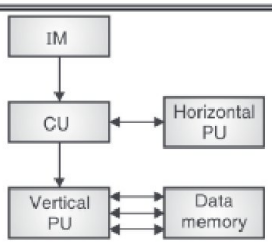


Fig. 1.8.8

1.8.4(D) Machine 4

In this case the processing unit (PU) and data memory (DM) are replicated and its combination is renamed as a processing element (PE). A single control unit is used to issue instructions to all PEs. There is no communication amongst PEs. A well known example of such machine is PEPE. Fig. 1.8.9 shows the architecture of machine 4

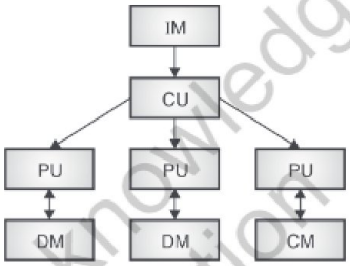


Fig.1.8.9

1.8.4(E) Machine 5

In this case, the machine 4 is added with a feature of inter processor element (PE) communication. LLIAC IV and many SIMD processors fall into this category of classification. Fig. 1.8.10 shows the architecture of machine 5.

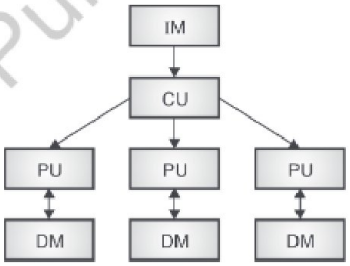


Fig. 1.8.10

1.8.4(F) Machine 6

Machine 6 uses a special system called as associate processor, wherein there is a association of the CPU and the DM. The data memory and CPU are combined. Machine based on such architectures span a range from simple associative memories to complex associative processors. Fig. 1.8.11 shows the architecture of Machine 6.

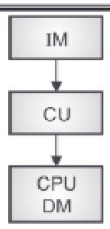


Fig. 1.8.11

1.9 Memory Access

1.9.1 Shared Memory

- In this case as the name says, there is a shared memory which is shared by all the processors in the parallel processor system so as to implement inter-processor communication (IPC). In such case we need two major things,
  1. Restricted access to the memory locations for the different processors i.e. those data that are to be accessed by only some particular processors must be restricted from being accessed by all other processors.
  2. Synchronization amongst these processor so as to implement mutual exclusion i.e. only when the processor that produces the data has updates the result in the memory, then the processor that has to access this data must access this memory location so as to read the same.
- The shared memory programming system looks as as shown in Fig. 1.9.1.

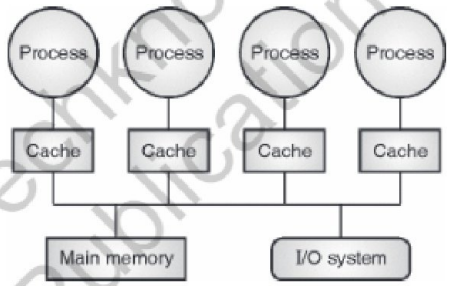


Fig. 1.9.1 : Shared memory architecture for inter process communication

- The main issues in shared memory programming are access of critical sections which has to be protected, maintaining memory consistency, synchronization fast data movements etc.
- Critical Section of a program is that part of the memory which accesses a shared variable from the memory, which must be synchronized such as to allow only one processor accessing the data at any given time. For implementing such a system, we need to use mutual exclusion. Mutual exclusion semaphore is a single bit, which is to be first read to check the availability of the resource (the shared variable), then if the resource is available the semaphore bit is toggled to indicate that it is not available any more. Thereafter any other authorised processor tries to access this resource it finds the semaphore indicating that the resource is not available and hence waits for it to be available. Once the previous processor that got the access of this shared resource, completes its operation, the semaphore is again toggled making it available for other processors.
- The **Lock()** method is used in parallel programming languages to lock the shared variable so that no other processor can access the same data simultaneously.

- Certain care have to be taken during the execution of this critical section of the code. There should not be a deadlock i.e. a circular wait by two processors trying to access the same resource. Also no interrupt is to be accepted or acknowledged during the execution of the critical section of the code. No other processor's request to access the resources should be accepted during the execution of the critical code section.
- Thus the **features** of the shared memory parallel programming can be listed as below :
  1. Shared memory parallel computers generally have the ability for all processors to access all memory as global address space.
  2. Multiple processors can operate simultaneously and independently and still share the same memory resources.
  3. The changes in a memory location that are done by one processor are visible to all other processors.

1.9.2 Distributed Memory

- In case of distributed memory, each processor has a separate memory. The data for each processor is stored in its memory. Each processor can access the memory as and when required without have any wait time as there is no other processor accessing the same memory.
- Special message passing algorithms are to be implemented to share data from one processor to another. This message passing sometime delays the task execution
- Fig 1.9.2 shows an example of distributed memory.

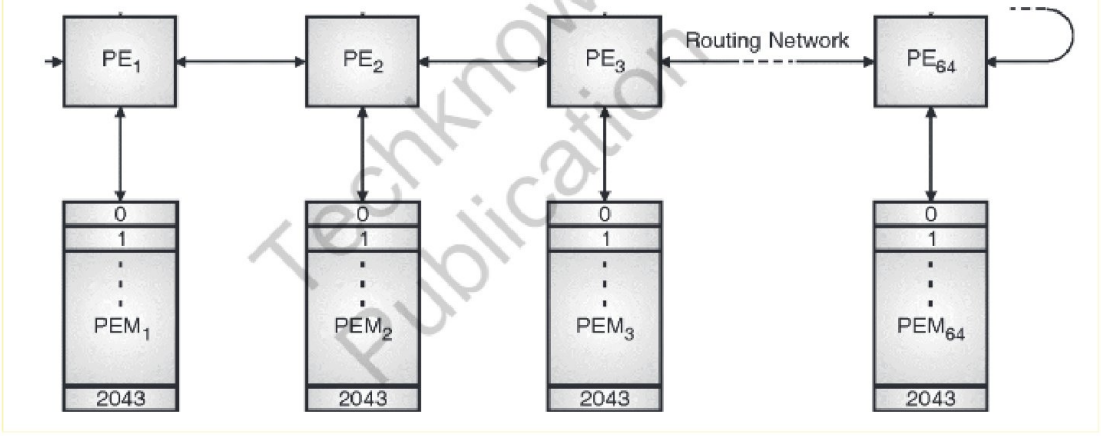


Fig. 1.9.2

1.9.3 Difference between Shared Memory System and Distributed Memory System

Sr. No.	Shared memory system	Distributed memory system
1.	In this case the multiple processors will have a shared memory	In this case each processor has its own memory.
2.	In this case the processors can communicate using these shared memory locations.	In this case it takes a longer time for the processor to take the data from another process.
3.	This system can use the shared memory subsystem.	This system requires message passing methods to communicate amongst themselves.

1.9.4 Hybrid Distributed Shared Memory

- As seen in the above sub-sections, there are pros and cons of shared memory as well as distributed memory. In case of shared memory no time is required in message passing, but the processors need to wait to access memory if another processor is accessing it. In case of distributed memory, the processors need not wait to access memory as each processor has a dedicated memory for itself, but there is special time requirement for message passing.
- A hybrid system comprises of distributed memory for each processor, but a shared memory to share the data amongst the processor. Hence the distributed memory is used by the processors to perform their tasks without waiting for any other processor. To pass the message, shared memory is used.
- Fig. 1.9.3 shows the Hybrid Distributed shared memory system

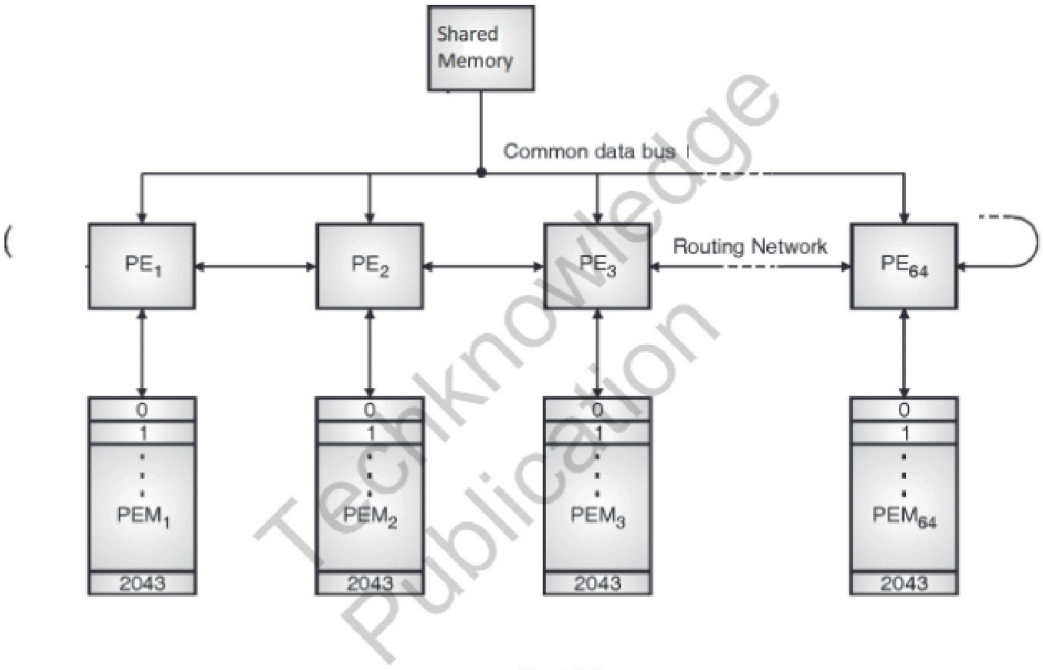


Fig. 1.9.3

1.10 Pipeline Architecture

- A processor has many resources like the ALU, buses, registers, etc. An attempt to utilize all these attributes to their fullest or continuously can be achieved by pipelining. In a pipelined system the instructions flow through the processor as if the processor was a pipe.
- The instructions move from one stage to another to accomplish the assigned operation. Hence at most of the times each unit of the processor is busy handling one or the other instruction, making the attribute of the processor being used continuously.
- This chapter deals with advanced pipelining and superscalar design in the processor development. We will go through the concepts and design issues of the linear and non-linear pipelining. We will also discuss collision-free scheduling techniques for performing dynamic functions. Techniques to design instruction pipelines, arithmetic pipelines are also discussed. We will study in detail about this pipelining in chapter 2.

1.11 Array Processor

- For a operation to be performed on a vector or array data, using multicomputer makes many redundant units like memory, instruction decoder, address decoder, control unit, ALU.
- This redundancy is avoided in case of a SIMD array. Here only one control unit and one program memory is connected and all the other related units like instruction register, instruction decoder and address decoder for instructions are also only one set and hence more resources can be used for the actual computational elements. This is shown in Fig. 1.11.1.

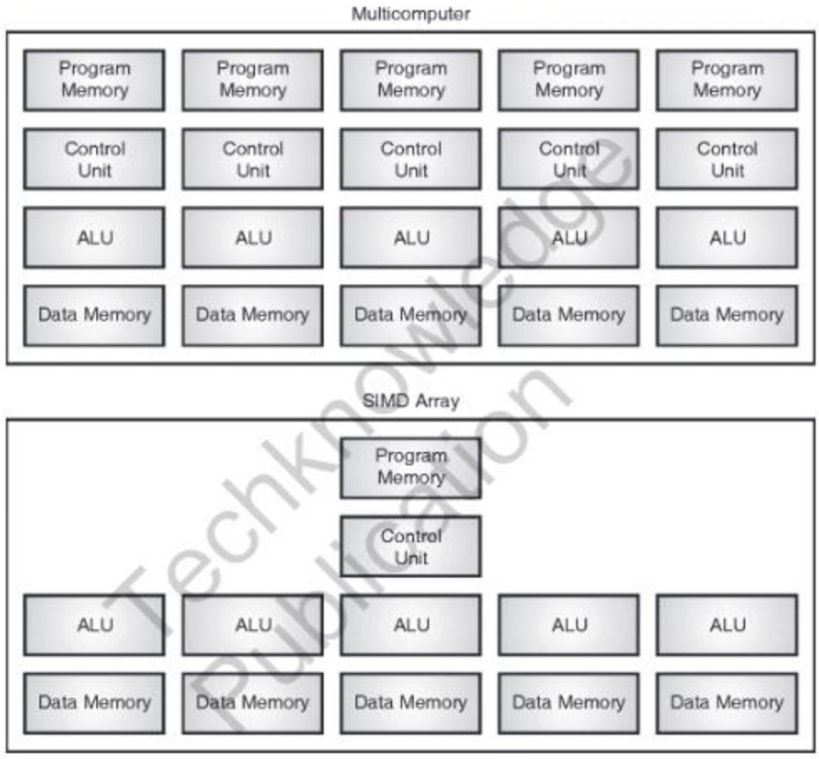


Fig. 1.11.1 : Multicomputer vs. SIMD architecture

- Thus the concept of SIMD design is to provide the maximum number of computational elements. This count has been up to a maximum of 256K in one machine, although 16K is more typical.
- This concept can make use of very simple processors, each connected with a small amount of data memory and in this way have huge number of elements.
- These can be machines with one-bit ALUs. Hence if an 8-bit addition is required, we will use one ALU and hence require 8 pulses. But since there are a huge number of such ALUs, for example say 256 ALUs, then 256, 8-bit data addition can go on simultaneously, and all the 256 sets of data can be added in 8 clock pulses. Thus giving an average of 32-byte operations in one clock pulse.
- In some SIMD processors, even for single data, parallelism is achieved by dividing the huge data into smaller parts. The speciality of SIMD processors is that if there are 256 computational units in a SIMD processor the performance will be 256 times that of a single processor. But this may not be true for a 256 processor system.

1.11.1 Programming Model

- SIMD has a very simple programming model amongst all parallel processors. This is because, it requires only one control unit. There is only one instruction to be executed on multiple data. Also there is no need of synchronization and no coherency as each ALU operates on a different data.
- The only complication is the programming of SIMD computers. A lot of programmer's interaction is required during the compilation of the code.
- SIMD programs normally operate on arrays and hence have parallel data types. Many operations are possible on arrays and parts of arrays.
- Fig. 1.11.2 shows a sample programmers model of the SIMD processor. There are multiple computational ALUs and a separate data memory for each of these computational units. The data operands and the results of each of the ALU is stored in this data memory.

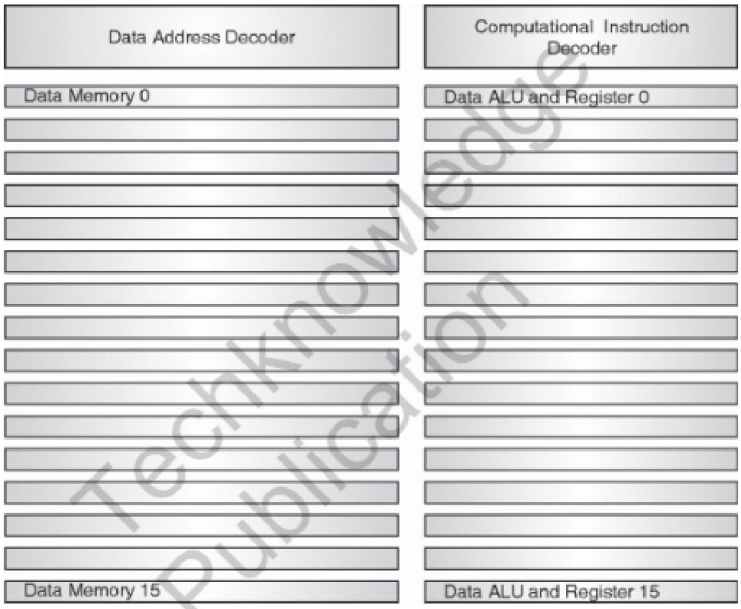


Fig. 1.11.2 : Sample programmers model of a SIMD processor

- We can perform operations like adding arrays, multiplying arrays etc. We can perform operations like sorting on these SIMD systems.
- For example if A, B and C are three arrays, then we can write: A = B + C, to add them.
- Another example, of sorting the elements of an array, we can write :
- WHERE A > A(Next) THEN Swap(A, A(Next))
- We will see some parallel algorithms to do these kind of operations in the further sections of this chapter
- Another example of the "WHERE" statement is for two way branch like a if-else statement of 'C' or Java programming language. For example if the two arrays 'A' and 'B' are to be added if a scalar value 'c' is positive and perform subtraction of the same two arrays if the condition is false; this can be written as :

  WHERE c >= 0 THEN A = A + B ELSEWHERE A = A – B
- Since the above statement is a single instruction, the processor performs one of the two tasks based on the condition. Hence there is no branching to be taken and hence no branch penalty.

1.11.2 Case Study of Illiac-IV SIMD Processor Architecture

– The Illiac-IV processor has a control unit with 64 processing elements (PE). Each processing element has a memory of 2K words. This is shown in the Fig 1.11.3 . Each processing element can access its local memory as well as communicate with its neighbour processors.

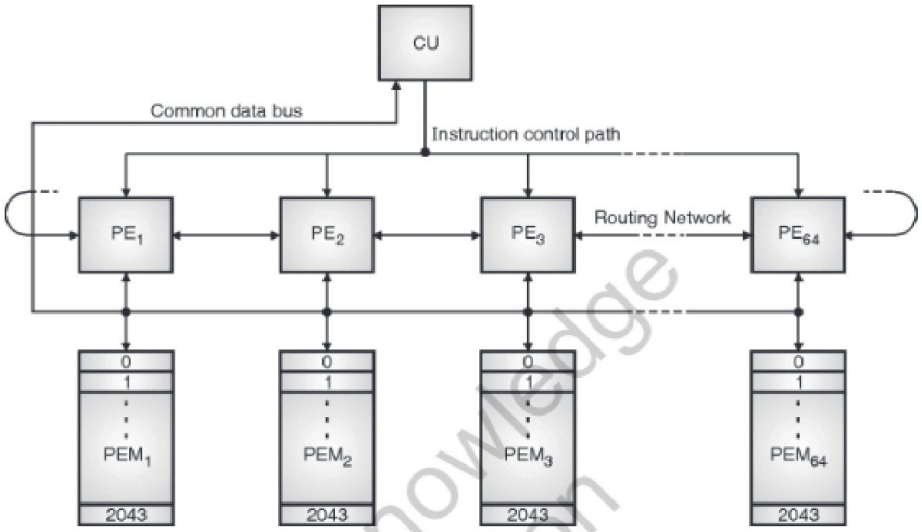


Fig. 1.11.3 : SIMD processor configuration

– Each processing element is an ALU, attached with the working registers and local memory. Masking schemes are available in the Illiac-IV SIMD processor to enable and disable the processing elements.

1.11.3 Masking and Data Network Mechanism

– Many a times all the processing elements of a SIMD processor need not operate simultaneously. There are various masking schemes used to enable or disable a set of processing elements. The control unit keeps a track of the enabled and disabled processing elements and it can also enable or disable the processing elements as required.

– In a network of computers, message passing algorithm is used as data routing mechanism. We know there are routers connected to route the data to the required computer. Similarly in a system of multiple processing element(PE), the inter-PE communication is obtained by various mechanisms like broadcast (one to all), multicast (many to many), shifting, rotating, shuffle, exchange etc.

– The programmer's model of a Illiac-IV SIMD processor, shown in Fig. 1.11.3 includes a set of multiple registers. Some of these registers are common for all the processing elements in the processor, while some are separately available for each processing element. These registers are :

1. A set of working registers and flags. It includes a set of operand registers for input as well as output, which are A_i, B_i, C_i. It also includes a status flag called as S_i, which indicates whether the processing element is enabled (active) or disabled (inactive). The control unit has a masking register 'M' which has the information of all active and inactive processing elements.
2. A local index register (I) : It is used for indexing a particular memory location.
3. An address register (D) : This register is used to store the address of the processing element.
4. Data routing register (R) : The data routing register is connected to other processing elements with which the communication is to be performed. The array processors may have multiple data routing registers.

1.11.4 Inter PE Communication

- Basically the communication in SIMD processor is synchronous. Every processor transfers data at the same instance, to its neighbour in the topology. The topologies used for SIMD arrays are meshes, grid, cube, hypercube etc. We will see these topologies in the next section.
- There are various issues that together define the inter processor element communication are operation mode, control strategy, switching methodology and network topology. As discussed earlier, network topology will be studied in the following section.
- The different operation modes for the inter processor elements are synchronous, asynchronous and combined communication (i.e. synchronous and asynchronous both). In case of synchronous operating mode, all the processing elements perform the communication simultaneously. All the processing elements forward the data synchronized with a common clock pulse. In case of asynchronous communication mode, the different processing elements forward the data in an asynchronous manner. The communication requests are dynamically executed. A combined system is designed such that it can switch between the synchronous and asynchronous operating modes.
- The different control strategies are centralized and distributed control. If there is a centralized unit to control the inter processor communication then it is called as centralized strategy, while if the control is distributed amongst the processing elements itself, then it is called as distributed control strategy. SIMD mostly use centralized control strategy.
- The different switching methodologies used are packet switching and circuit switching. In case of a circuit switching there is a dedicated path for communication between the two processing elements while in case of packet switching a packet of data is forwarded with the route and the data is transmitted via various processing elements to the right place as required. In case of SIMD systems, mostly the circuit switching methodology is used.
- Another important aspect to decide the inter processor communications is how the processing elements are connected. The different methods of connecting the processing elements is referred to as network topologies. The network topologies can be static or dynamic i.e. the connections can be passive and fixed or they can be changed during the operation. We will study these topologies, especially the ones used in SIMD processors in the following section.

1.11.5 Interconnection Network of SIMD

In this section we will see the various methods of connecting the SIMD multistage networks. We will see the different topologies of connecting the SIMD processors.

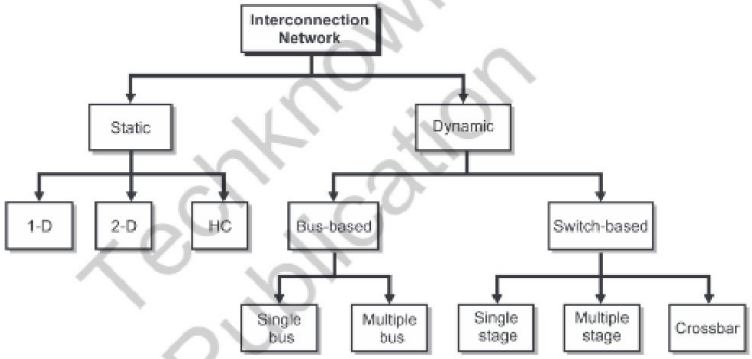
1.11.5(A) Need and Types of Routing in Array (SIMD) Processors

- As discussed earlier there are two types of networks namely static and dynamic.
- The static network have passive connections i.e. the connections among the processing elements are fixed and cannot be reconfigured to have a different connection path.
- For example if there are three processors connected such that processor 'a' is connected to processor 'b', and processor 'b' is connected to processor 'c'. Then, if data is to be transmitted from the processor 'a' to processor 'c', it cannot be directly transmitted; instead it is to be routed through the processor 'b'.
- In case of dynamic network connections, the interconnection can be reconfigured to establish new paths as and when required. Considering the previous example of transmitting the data from processor 'a' to processor 'c', in case of dynamic network, we can establish a path dynamically between the processors 'a' and 'c'.
- Table 1.11.1 shows the differences of the static and dynamic networks.

Table 1.11.1

Sr. No.	Static Networks	Dynamic Networks
1.	The connecting paths between the processing elements of the static networks is static or passive.	The connecting paths between the processing elements is dynamic or active.
2.	Establishing of links between two processor during the execution of program or dynamically is not possible.	Establishing of links between two processor during the execution of program or dynamically is possible.
3.	The static network is made up of fixed processor to processor or point to point connections	Dynamic networks are made up of channels that can be switched to be present and being removed or disconnected.
4.	These networks are used in a distributed system.	These networks are used in a shared memory system with multiple processors
5.	There are various types of static networks like cube, hypercube etc.	There are various types of dynamic networks like single stage dynamic network, multi stage dynamic network etc.

- The different network topologies are given in the following classification chart.



- The **number of links** is one of the important parameter to be considered for the cost aspect of the network. More the number of links more is the cost. But if the performance increases because of a slight increase in number of links, then it is affordable.
- The **degree** is yet another term that is important in measurement of cost involved in making the topology. This term corresponds to the maximum number of adjacent processors a processor is connected directly to. In this case we consider the worst case condition i.e. the processor that has the maximum number of processors connected to it directly.
- The **diameter (Permutation cycle)** is important parameter to measure the performance of a network. It is the maximum number of processors that a message has to route through to reach the farthest processor. In case of a network there will be multiple processors, hence the worst case condition is considered to measure this parameter.

1.12 Multiprocessor Architecture

Whenever we have, single or centralized approach that relies on a single CPU, it has some drawbacks, when compared with multiple processor system. Normally, employing multiple processors in medium to large systems offer several significant advantages over the centralized approach that relies on a single CPU and extremely fast memory.

Advantages of multiprocessor systems

- System tasks may be allocated to special purpose processors whose designs are optimized to perform certain types of tasks, simply and efficiently.
- Very high levels of performance can be attained when multiple processors can execute simultaneously i.e. parallel processing.
- Robustness can be improved by isolating system functions so that a failure or error in one part of the system has a limited effect on the rest of the system.
- The natural partitioning of the system promotes parallel development of sub systems, breaks the application into smaller more manageable tasks, and help isolate the effects of system modification.

8086 architecture in maximum mode, supports two types of processors :

- Co-processor [Math-co-processor/Numeric data processor - NDP] [NDP 8087]
- Independent processor [IOP 8089]
- An independent processor is one that executes its own instruction stream. The 8086, 8088 and 8089 are examples of independent processors.
- Coprocessor 8087 has been designed in such a manner that, it can obtain its instructions from another processor called host. The coprocessor monitors instructions fetched by the host and recognizes certain of these as its own and executes them. In short, coprocessor, in effect, extends the instruction set of its host processor.

1.13 Loosely Coupled and Tightly Coupled Systems

There are two basic multiprocessor configurations.

1. Closely coupled configuration (shared memory system)
2. Loosely coupled configuration (distributed and shared memory system)

1.13.1 Closely Coupled Configuration

- **Closely coupled configuration** is a system wherein all the processors in the system use the same resources viz. memory and I/O devices.
- Closely coupled configuration has three possibilities.

1. 8086 – 8087 configuration
2. 8086 – 8089 configuration
3. 8086 – 8087 and 8089 configuration

1.13.1(A) 8086-8087 Configuration

Before interfacing 8086 with 8087, we need to use the pin configuration of Numeric Data Processor 8087. 8087 pins can be grouped as follows.

(1) AD_{15} - AD_0 (Input/output lines) : (Address / Data)

These lines constitute the time multiplexed memory address and data bus. A_0 is analogus to the $\overline{BHE}$ for the lower byte of the data bus. These lines are active high. These line will be input lines for 8087 when CPU is having the control of the bus.

(2) A_{19}/S_6 - A_{16}/S_3 (Input/output lines) : (Address / Status)

These lines are multiplexed address/status lines. Initially, address information will be available, after that status information will be available.

For 8087 controlled bus cycle.

S_6, S_4 and S_3 = Reserved (currently high)

S_5 = Always low

These lines will be monitored by 8087, when CPU is having the control of the bus.

(3) $\overline{BHE}/S_7$ - (Input/output line) : (Bus HIGH Enable)

This pin is used to enable data onto the most significant half of the data bus. i.e. D_8 to D_{15}.

$\overline{BHE}$ = 0 during read/write cycle when a byte is to be transferred on the high portion of the bus. S_7 information available during T_2, T_3, T_w and T_4.

(4) $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$ - (Input/output lines) : (status lines)

These lines are encoded as follows :

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Operation
0	x	x	Unused
1	0	0	Unused
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive

Status is driven during T_2 cycle, and is returned to passive status (1, 1, 1) during T_3 or during T_w.

(5) $\overline{RQ0}$ / $\overline{GT0}$ (Input/output) : (Request/Grant)

This pin is used by 8087 to gain control of the local bus from the CPU for operand transfers or on behalf of another bus master. It must be connected to one of the two processor request/grant pins.

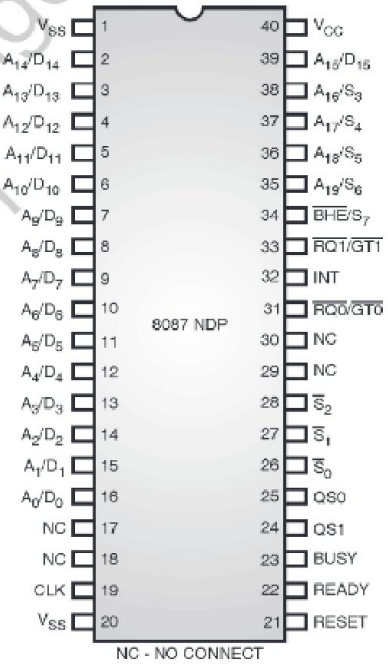


Fig. 1.13.1 : Pin configuration of NDP 8087

(6) $\overline{RQ1}$ / $\overline{GT1}$ (Input/output) : (Request/Grant)

This request/grant pin is used by another local bus master to force the 8087 to request the local bus. If the 8087 is not in control of the bus, then it will make request through this line.

(7) QS1, QS0 (Input) : (Queue status)

QS1, QS0 provide 8087 with status to allow tracking of the CPU instruction queue.

QS1	QS0	
0	0	No operation
0	1	First byte of opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from the queue

(8) INT (Output) : (Interrupt)

This line is used to indicate that, an unmasked exception has occurred during numeric instruction execution, when 8087 interrupts are enabled. Normally this signal is routed through 8259 (PIC).

(9) BUSY (Output) : (Busy)

This signal indicates that the 8087 NEU is executing numeric instruction. It is connected to the CPU's $\overline{TEST}$ pin to provide synchronization. Busy is active high.

(10) Ready (Input) : (Ready)

Used for slower peripheral devices.

(11) RESET (Input) : (Reset)

This causes processor to immediate terminate its present activity.

(12) CLK (Input) : (Clock)

Clock provides basic timing for processor and bus controller. It is asymmetrical with 33% duty cycle.

Communication Protocol between NDP and CPU

- CU [Control unit] keeps 8087 operating in synchronization with its host CPU. 8087 instructions are intermixed with CPU instructions in a single instruction stream. The CPU fetches all instructions from memory; by monitoring the status $\overline{S_0}$ – $\overline{S_2}$, S_6 emitted by CPU, the control unit determines when an instruction is being fetched. The 8087 monitors the data bus in parallel with the CPU to obtain instructions that pertain to 8087.
- CU maintains an instruction queue that is identical to queue in the host CPU.
- To keep track of queue, CPU automatically determines if CPU is 8086 or an 8088 immediately after reset and matches its queue length accordingly. By monitoring the CPU's queue status line (QS0, QS1), the CU obtains and decodes instructions from the queue in synchronization with the CPU.
- A numeric instruction appears as an ESCAPE instruction to CPU. Both, the CPU and 8087 decode and execute the ESCAPE instruction together. ESC will wake up 8087 NDP. The 8087 only recognizes the numeric instructions.
- The start of numeric instruction operation is accomplished when the CPU executes the ESCAPE instruction. The instruction may or may not identify a memory operand.
- CPU distinguishes between ESC instructions those reference to memory and those that do not.

- 8087 instruction can have one of three memory reference options.
  1. Not reference memory.
  2. Load an operand word from memory into 8087.
  3. Store an operand word from 8087 to memory.
- If no memory reference is required, 8087 simply executes its instruction. Same way CPU simply proceeds to next instruction.
- If memory reference is required, then CPU calculates the operand's address using any one of its available addressing modes, and then performs dummy read of the word at location. CU uses dummy read cycle initiated by the CPU to capture and save the address (20 bit physical address) that CPU places on the bus.
- The dummy read cycle is normal read cycle except that CPU ignores the data it receives.
- If the instruction is load, the CU additionally captures data word when it becomes available on the data bus.
- If the data required is longer than one word, CU will immediately capture the bus, using Request/Grant protocol; and reads rest of the information in consecutive bus cycles.
- If instruction is of store operation the CU captures and saves the store address as in a load, and ignores the data word that follows in the dummy read cycle. When 8087 is ready to perform the store, the CU obtains the bus from CPU and writes the operand starting at specified location.
- Now 8086 is executing its instruction and 8087 is performing some arithmetic operation, deactivating $\overline{TEST}$ pin.
- While executing, if 8086 require the output, from the task handed over to 8087, it has to WAIT. In wait state, it will check for $\overline{TEST}$ pin to activate. This indicate 8087 has completed the job.

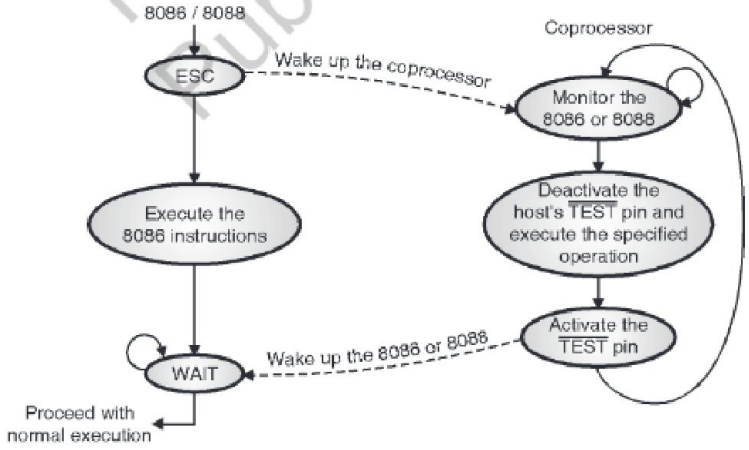


Fig. 1.13.2 : Synchronization between 8086 and its coprocessor

The Fig. 1.13.3 shows interfacing of 8086 with 8087.

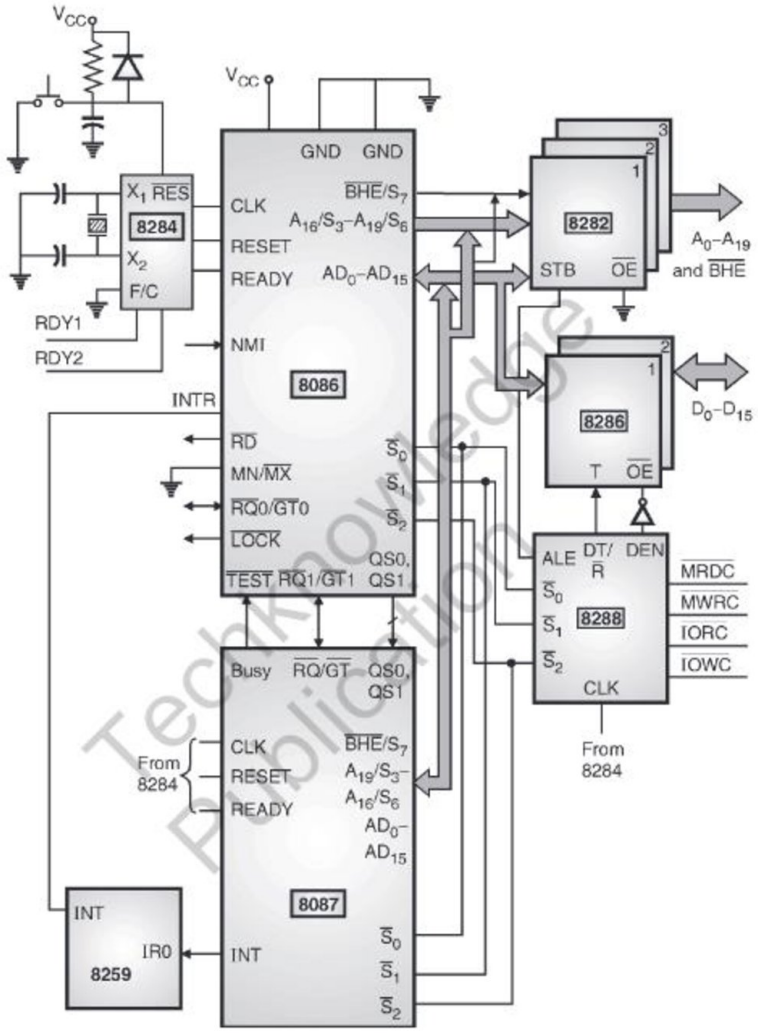


Fig. 1.13.3

1.13.1(B) 8086-8089 Configuration

Again, for interfacing 8086 with 8089, let us first see the pin diagram of 8089. The pin diagram of 8089 is shows in Fig. 1.13.4, which has almost the same pins to be interfaced to 8086 as that were in case of 8087.

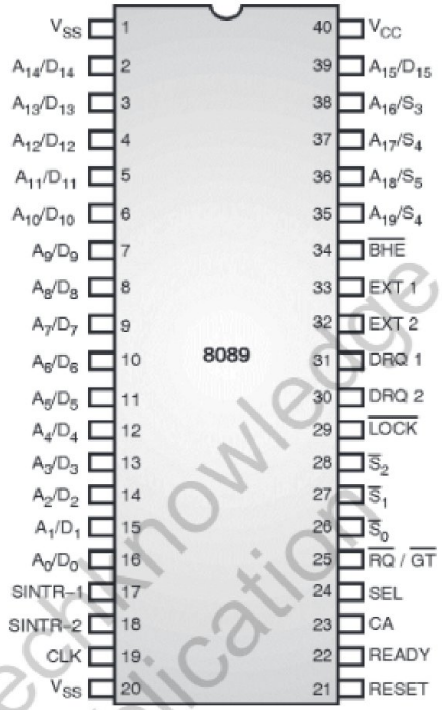


Fig. 1.13.4 : Pin Configuration of (IOP 8089)

Interfacing 8086 and 8089 : Local Configuration

Refer Fig. 1.13.5

- CLK, Reset and Ready given to 8086/8088 as well as IOP 8089.
- $\overline{S_0}$ – $\overline{S_2}$ from 8086 given to 8089 as well as bus controller 8288.
- AD_0 – AD_{15} and A16/S3 to A19/S7 lines of 8086 given to 8282 latch for demultiplexing address and data bus.
- A_1 to A_{15} lines of 8086 also given to 8286 (data bus buffer).
- Output from 8282 is A_0 to A_{19} with $\overline{BHE}$ signal.
- A_1 to A_{15} lines given to address decoder for generating chip select for IOP.
- INT pin of IOP is given to 8259, for generating interrupt for 8086, whenever required.
- $\overline{RQ}/\overline{GT}$ of IOP 8089 connected to $\overline{RQ}/\overline{GT}$ of microprocessor.
- When 8089 is directly connected to 8086/8088, the $\overline{RQ}/\overline{GT}$ lines built into all these processors are used to arbitrate use of a local bus. First we will see how $\overline{RQ}/\overline{GT}$ of CPU operates.

- An external processor sends a pulse to the CPU to request use of the bus.
- The CPU finishes its current bus cycle, if one is in progress, and sends a pulse to the processor to indicate that it has been granted the bus.
- When the external processor is finished with the bus, it sends a final pulse to the CPU, to indicate that it is releasing the bus.

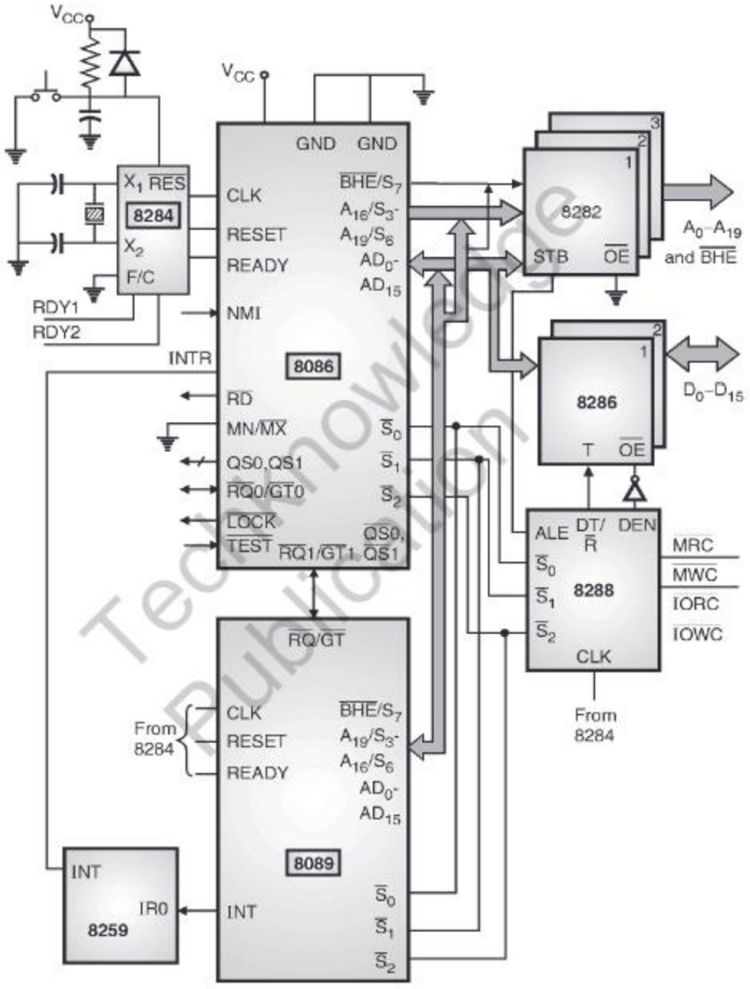


Fig. 1.13.5

(1) The host sets up message in memory and then wakes up the independent processor by sending a command to one of the independent processor's ports.

(2) The independent processor then accesses the shared memory to get the assigned task and executes the task in parallel with the host.

(3) After the task is completed, the external processor notifies its host of the completion by using either a status bit or an interrupt request.

(4) The message format, totally depends upon independent processor and the application. The message should specify which operation is to be performed, the input parameters and the addresses of the locations in which to store the result.

Similarly the third configuration of closely coupled i.e. 8086 - 8087 and 8089, can be implemented together as seen in sections 1.13.1 and 1.13.2.

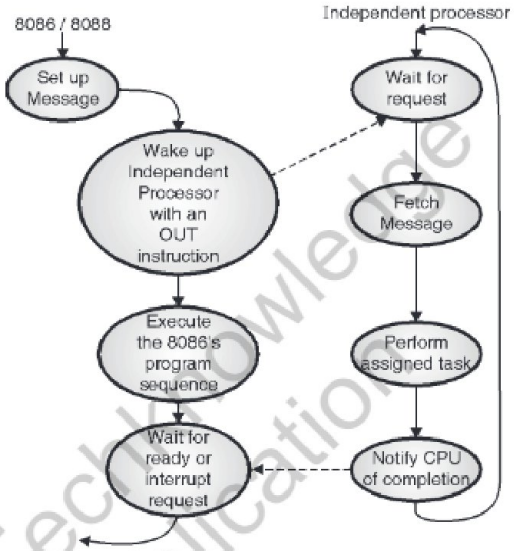


Fig. 1.13.6 : Interprocessor communication through shared memory

1.13.2 Loosely Coupled Configuration

- In loosely coupled configuration, user designs medium size or large system. The general idea is depicted in Fig. 1.13.7.
- Each module in the system may be the system bus master and consists of processor capable of being bus master. Processor module may contain closely coupled configuration on LOCAL basis. Several modules may share the system resources and system bus control logic.
- As shown each bus master runs independently and there are no direct connections between them. Interprocessor communication is made possible through shared resources. In addition to shared resources, each module may include its own memory and IO devices.
- The processor in separate modules can simultaneously access their private subsystems through their local buses and perform their local data references and instruction fetches independently, thus improving degree of concurrent processing.

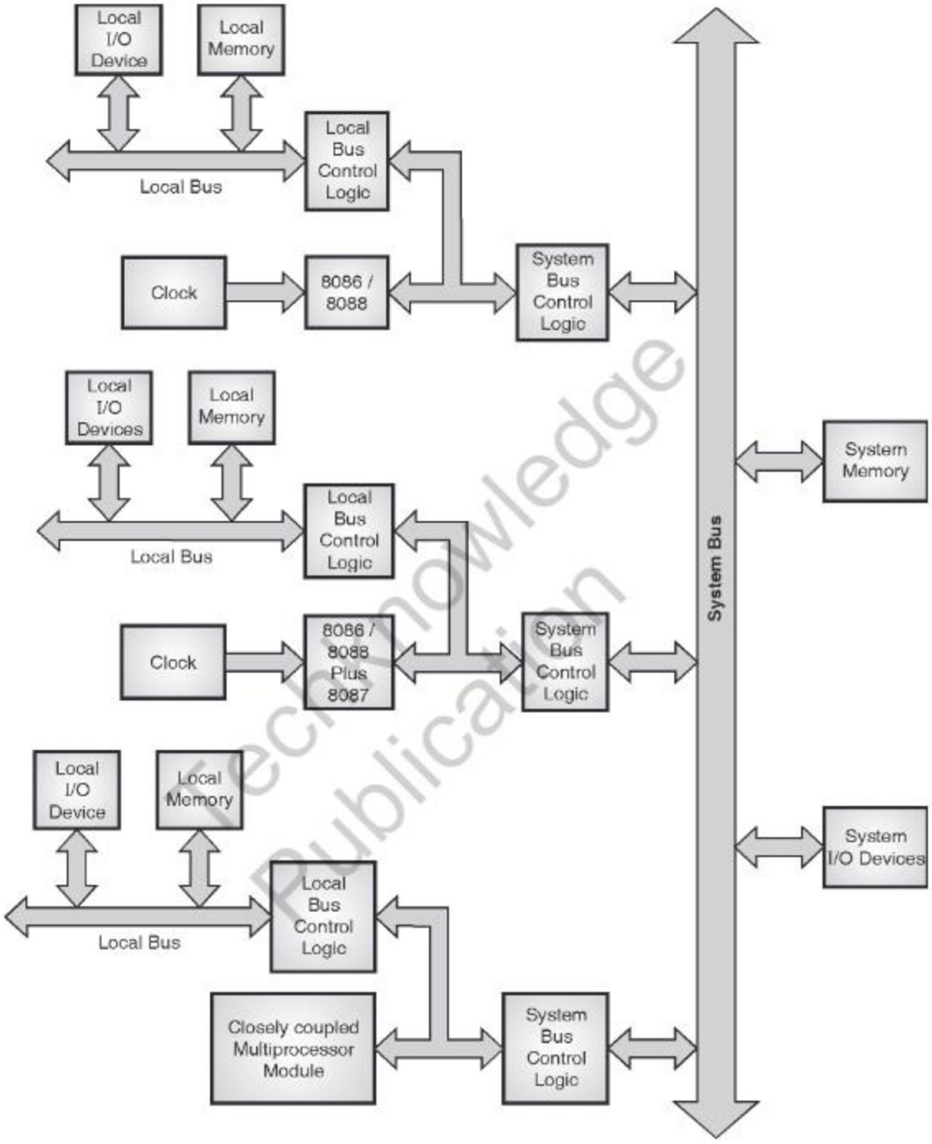


Fig. 1.13.7 : Loosely coupled configuration

Advantages of loosely coupled system

1. Better system throughput by having more than one processor.
2. The entire system does not breakdown or collapse if there is failure in one module. The module that is faulty can be detected and replaced.

3. The system structure is flexible. One or more modules can be added or removed, in order to change the system configuration, without disturbing the other modules in the system.
4. A greater degree of parallel processing is achieved as each processor can have a local bus to access memory or I/O devices.

1.13.3 Difference between Closely Coupled and Loosely Coupled

Sr. No.	Closely (Tightly) Coupled	Loosely Coupled
1.	Single Host CPU is used i.e. one CPU module with some processors.	Multiple CPU modules are used. CPU modules may have closely coupled configuration. Each module may contain multiple processors
2.	It has local bus only.	It has local as well shared system bus.
3.	No shared system memory or IO. Only local memory and IO.	It has shared system memory and IO, shared among CPU modules. It also has local memory and IO for each module.
4.	No bus arbitration logic required.	Bus arbitration logic required.

1.13.4 Processor Characteristics for Multiprocessing

There are various characteristics desirable from a multiprocessing system. They are

1. Efficient context switching
2. Processor recoverability
3. Large virtual and physical address space
4. Effective synchronization primitives
5. Inter processor communication techniques
6. Instruction set

1. Efficient Context Switching

- When the processor completes a particular task or leaves a task in between and goes to another task there must be a proper context switching taking place. Context switching refers to storing the data or context of current task in proper memory; then loading the data or context of the new task to be executed.
- The context must be stored in a memory that should be allocated for each task. So that the task context can be easily retrieved from the corresponding memory location by the processor when it returns to the same task or when another processor starts handling the corresponding task.

2. Processor Recoverability

- There will be many processors in a multiprocessor system. In case of the failure of a particular processor, another processor must take over the charge and handle the incomplete tasks by the failed processor.
- This is possible only if the processes and processors are considered to be two different entities. A process will remain in the pool of ready processes until it is completed. Once the process is completed it will not be found in the pool. If the processor handling this process fails due to some reason, the process will remain in the pool and some other processor will handle it.

- But for this care has to be taken that the registers storing the data for the process should be such that they are a part of memory and sharable; so that the other processor which handles this task should be able to access these data.

3. Large Virtual and Physical Memory Space

- Since the data to be stored in memory will be for each task, a huge memory will be required. The physical accessible memory should be huge so that the data can be accessed faster from the physical memory. The context of multiple tasks can be stored and the data retrieving as well as storing can be done faster from the physical memory.
- Also since there will be many processors and many processes the virtual memory to store the data and programs for all the processes will required to be very large. All these huge memory must be accessible by the processors.
- Thus the processors must have enough address line to access the huge physical memory. The page translation mechanism should also be capable to handle huge virtual memory.

4. Effective Synchronization Primitives

- There must be proper methods of synchronizing the entire system. The resources are to be shared as discussed in the earlier section in loosely coupled system.
- To allow proper usage, there must be special means for synchronization. The synchronization of usage of the shared bus is to be implemented using a special mechanism called as bus arbitration (discussed in a later section of this chapter). Similarly all the resources and processors themselves are to be synchronized.
- Hence to synchronize these things in a multiprocessor system, there must be proper mechanisms for the same.

5. Interprocessor Communication Techniques

- Inter processor communication is another very important aspect that must be supported by the processors in the multiprocessor system. In a multiprocessor system as discussed earlier, the tasks to be performed are distributed amongst multiple processors of the system.
- One processor will perform a part operation and forward the further work to the next processor. This requires some mechanism for communication amongst the processors.
- A simple method of communication among the processors is one processor storing the data in memory and the other accessing from the same. But there have to be proper synchronizations in the same. We will be seeing different methods to do this interprocessor communication in this chapter.

6. Instruction Set

- The instruction set is another aspect wherein we need to concentrate when selecting the processors for a multiprocessor based system. There must be instructions to perform various bus operations besides the complicated processing operations. The instructions must be provided to support concurrent operations in the system.

– In a multiprocessor system, the main advantage will be when the multiple processors are concurrently performing different operations. Thus, there must be instructions in the processor to support the concurrent operations in the system.

1.14 Inter Processor Communication Network

– Communication amongst processor is very important factor in a multiprocessor system. There are various mechanisms to do this. One such mechanism is making a network of the processors to allow communication amongst them.

– Also to make communication networks, there are various methods. We will see some of them in this section namely multi port model crossbar switch and time shared bus.

1.14.1 Multiport Memory

– As the name says, a multiport memory has multiple ports. A port of a memory can be considered to be a entry to access memory i.e. provide the address and access (read or write) the data. Hence if there are multiple ports to a memory means there are multiple addresses that can be given to the processor and corresponding data can be accessed simultaneously.

– We can have multiple processors connected on these multiple ports and hence accessing the multiple data.

– For example, there are three processors connected to a three-ported memory. One processor wants to read data from memory location 2000H, another wants to read from location 2673H and the third processor wants to write to memory location 35FFH. All these operations can be done simultaneously with the same memory, but each processor connected on different ports. Thus multiple processors will be able to access the same memory simultaneously for their required access.

– Fig 1.14.1 shows the structure of this multiport memory.

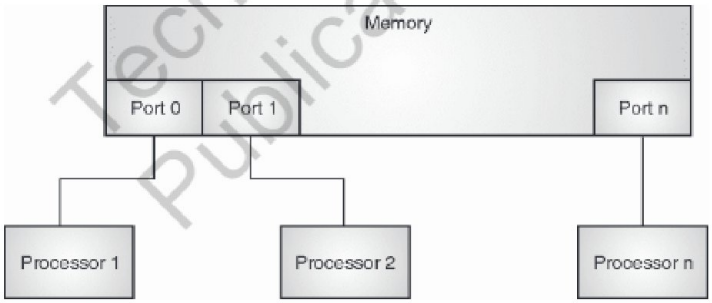


Fig. 1.14.1 : Multiport memory

– Each processor i.e. processor 1 to processor n, in Fig. 1.14.1, can access the multiport memory simultaneously.

– One very important thing to be noted here is that the number of processors should be equal to the number of ports in the memory. Hence the major disadvantage of this system is that the system is not scalable. If we want to increase the number of the processors then the memory also has to be changed i.e. to increase the number of ports we need to change the memory itself. Another disadvantage is that the memory with multiple ports will also cost higher.

– The major advantage of the multiport memory system is easy to implement. Another major advantage is high throughput as there are many processors accessing the memory simultaneously at their maximum speed.

– This will also allow communication amongst processor through the memory locations, as the same memory can be accessed by all the processors.

– There are some restrictions to be taken care for maintaining coherency (most updated data must be available to all the processors, and not the old stale data). The same memory location must not be read and written simultaneously. There may be special cases for this, but the operations for read and write must be in order of their occurrence to be executed.

1.14.2 Crossbar Switch

– This is a complicated system. In this case there is a switch of size M × N, where say 'M' is the number of processors and 'N' is the number of memory chips. Each port of the M × N switch can be connected to the required port and hence establish the connection between the two.

– The advantage is that since there is a direct connection between the required ports, the data transfer is quite fast. Another advantage is that the functional unit at each junction of memory and processor is very cheap as it is a simple switch.

– A major disadvantage is that the arbitration mechanism required here is quite complicated. The mechanism that will decide the connections is called as arbitration. Another disadvantage compared to multiport memory is that the throughput is lower as there is no separate connection for each processor to the memory at all given times. The arbitration mechanism used also takes some time to handover the bus to the required processor.

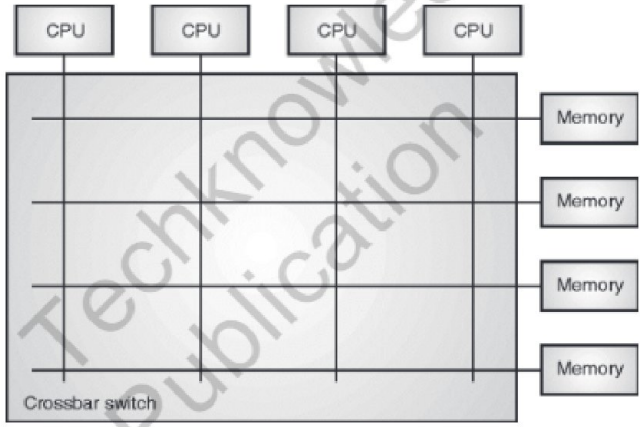


Fig 1.14.2 : Crossbar switch interconnection

– As shown in Fig. 1.14.2, there are processors on one side of the crossbar switch and memory on the other side. The crossbar switch connects the required CPU to the corresponding memory as per its demand and hence establishes the connection.

1.14.3 Time Shared Bus

– This is the most basic type of sharing the memory. In this case, there will be only one bus and all the processors connected to the same bus. Also the memory will be connected on the same bus.

– Whenever a processor wants to access the memory, it requests for the bus using its arbiter. Whenever the processor gets the bus, based on its priority, the processor accesses the shared memory.

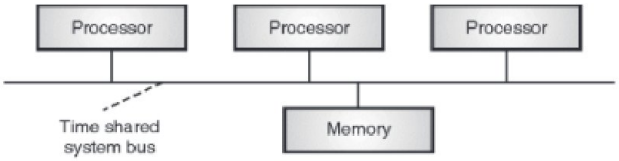


Fig 1.14.3 : Time shared bus

– As shown in the Fig. 1.14.3, the processors access the memory through the shared bus, shared on the basis of time.

1.15 Topologies of Connecting Multiple Processors

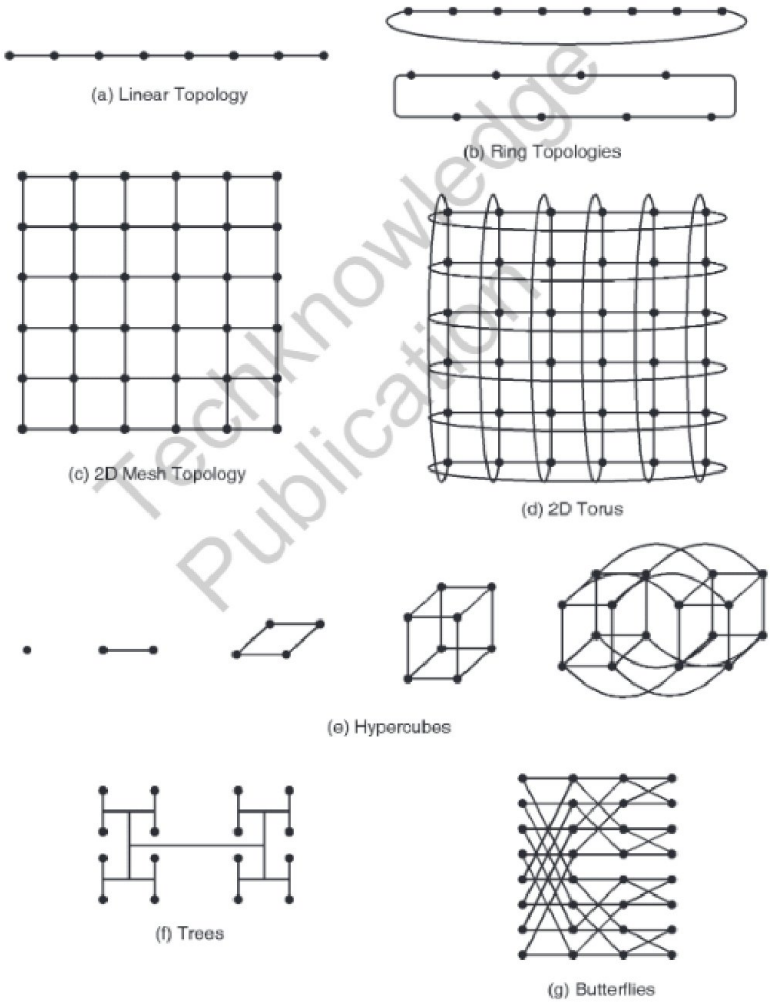


Fig. 1.15.1 : Processor connection topologies

- There are various ways of connecting multiple processors in a system. We will see these methods of connections in this section. Fig 1.15.1 shows the different topologies of connecting the processors.
- The basic topology shown in Fig. 1.15.1(a) i.e. linear topology is like a pipeline. Here the processors are connected one after the other linearly.
- The next one is the ring topology as shown in the Fig. 1.15.1(b). It is similar to the linear topology, but the last processor is again connected to the first processor, like a ring.
- The 2D mesh topology as shown in the Fig. 1.15.1(c), has a mesh like connection. Here each processor (or processing element) is connected to all its neighbours in the 2D plane. The 2D torus is a similar connection as 2D mesh, shown in Fig. 1.15.1(d). The only difference is that the mesh is made of ring topology instead or linear topology.
- The Fig. 1.15.1 (e), shows the different cases of Hyper cubes. They are with different dimensions viz. 0D, 1D, 2D, 3D and 4D. In case of 0D, there is only one processor, similarly as shown for other cases.
- The tree structured topology is shown in Fig. 1.15.1(f). Here, the structure is like the stem, branches, sub-branches and so on. Some processors directly connected by the stem, or the main connection. This connection, then has sub connections or branches.
- The butterfly connection is a special one used for FFT (Fast Fourier Transform) as shown in Fig. 1.15.1(g)
- All these topologies seen here are used for some or the other special application. We will be using them at some places according to the application. For example, the 2D mesh topology will be used for matrix multiplication, butterfly topology will be used for FFT etc.

1.16 Systolic Architecture

- Systolic architecture is a design to arrange data flow for high throughput with less memory access. It is different from pipelining. It consists of nonlinear array structure, multidirection data flow, with each Processing Element (PE) having small local instruction and data memory. It represents algorithms directly by chips connected in a regular pattern.

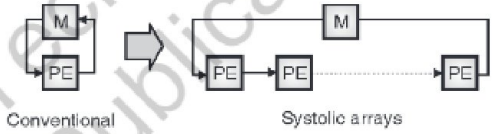


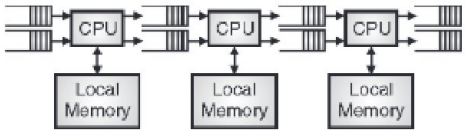
Fig. 1.16.1 : Comparison of conventional and systolic array architecture

- Hence it can be said that replacing a PE in conventional system with an array of PE's without increasing I/O bandwidth makes the systolic array architecture.

1.16.1 Communication Styles

There are two different methods used to communicate between the PEs of the systolic array. It could either be through a buffer queue connected between the PEs as shown in Fig. 1.16.2(a); or it could be through the memory connected to the buffer queue as shown in Fig. 1.16.2(b).

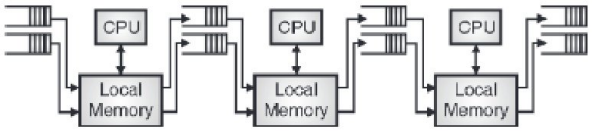
Systolic communication



(a) Buffer queue connected between the CPUs

Fig. 1.16.2 contd....

Memory communication



(b) Buffer queue connected between the PEs through the local memory

Fig. 1.16.2

1.16.2 Matrix Multiplication using Systolic Array Architecture

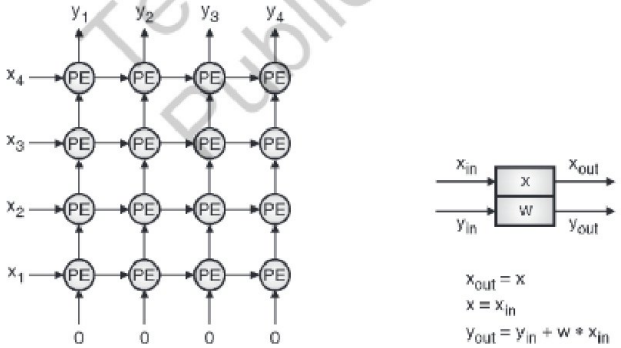
– The matrix multiplication can be implemented by the following algorithm :

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, n$$

↓ Recursive algorithm

```
for i = 1 to n
    y (i, 0) = 0
        for j =  0 to n
y (i, 0)=y (i, 0) + a (i, j) * x (j, 0)
```

– The Systolic Array Representation of Matrix Multiplication can be implemented as shown in Fig. 1.16.3(a). The processing elements required to perform this operation is shown in Fig. 1.16.3(b).



(a) Systolic array for matrix multiplication (b) Each processing element (PE) of the systolic array of multiplication

Fig. 1.16.3 : Systolic array to implement multiplication

1.17 Data Flow Computers

– A data flow computer has the instructions executed according to the availability of data. Any instruction should be ready for execution whenever the data is available. Instruction execution is independent of the physical location of that instruction in the program memory.

– Since the instructions need not be ordered in the sequence of execution, there is no need of the program counter (PC). The control-flow computers use shared memory to store the data and program, which may cause errors in other instruction or data while executing a particular instruction.

– In case of data flow computers, there is no need of shared memory as the instructions are stored in the instruction itself. Hence there are no problems related to the shared memory.

1.17.1 Data Flow Graphs

Data flow graphs are used to represent programs for data driven computations. An example of data flow graph to perform the operation z = (x – y) * 2, is shown in Fig. 1.17.1.

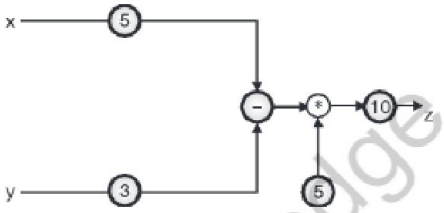


Fig. 1.17.1 : Data flow graph of z = (x – y) * 5

As shown in the Fig. 1.17.1, the inputs x and y are given to the subtract instruction. When the result operand is ready, the instruction for multiplication is being executed. The template implementation of the above data flow graph can be as shown in Fig. 1.17.2.

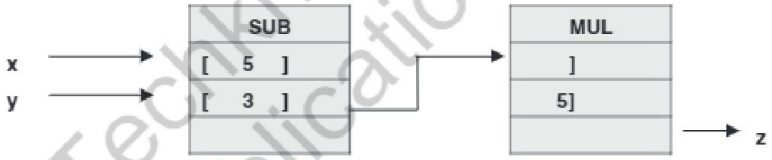
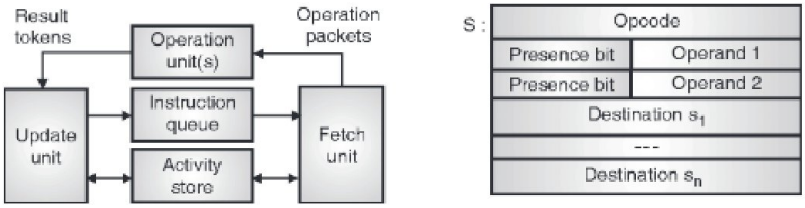


Fig. 1.17.2 : Template implementation

1.17.2 Static Dataflow

It combines control and data into a template like a reservation station, except that they are held in memory. They can inhibit parallelism among loop iterations and also re-use of template.



(a) Block diagram of static dataflow computer **(b) op-code structure**

Fig. 1.17.3

The block diagram of the static dataflow computer is shown in Fig. 1.17.3 (a). The different blocks of the same are explained below.

1. **Instruction Queue:** When the instruction becomes ready for execution, the address of the activity template is entered in the instruction queue. Each activity template has unique address.

2. **Fetch and Update Units:** Instruction fetching and data accessing operations are performed by fetch and update units.
3. **Operation Unit:** The specified operation is performed by the operation unit. The generated result is passed to each destination field in the template.
4. **Activity store:** This stores the activity templates.

– The Fig. 1.17.3(b) shows the structure of an opcode. The opcode has the operands and the corresponding presence bits to indicate if the operands are presented (ready) or not.

– As shown in Fig. 1.17.3 (b), an instruction consists of opcode, operands and their presence bits and the destinations of the result. Thus when an instruction is executed, all those instruction's operand fields are updated which are the destination for the executed instruction.

– Also, the corresponding presence bits are set, to indicate if the operands are available or not. The fetch unit fetches the instruction from the instruction queue. If the instruction has both the presence bit set, it is forwarded to operation unit. If the operands are not ready (i.e. presence bit is clear), the instruction is given back to the queue.

– The operation unit operates on the instructions whose operands are available and then forwards it to the update unit. The update unit, updates those instruction's operand fields which are the destination for the executed instruction.

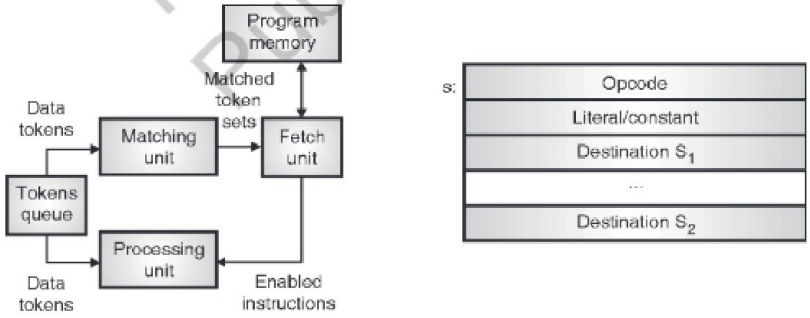
1.17.3 Dynamic Dataflow

– Dynamic dataflow computers have separate data tokens and control. It has a tagged token i.e. a labeled packet of information. The data token is tagged with the context descriptor and is then called as a tagged token. The tagging is achieved by attaching a label with each token.

– The label identifies the context of that token. This allows multiple iterations to be simultaneously active with shared control (instruction) and separate data tokens. A data token can carry a loop iteration number. The operation can be described as below

1. Match token's tags in matching store via associative search. If match not found, make entry and wait for partner. This requires large associative search to match tags.
2. When there is a match, fetch corresponding instruction from program memory and execute the instruction.

– The block diagram and the opcode structure of the dynamic dataflow computer is shown in Fig. 1.17.4.



(a) Block Diagram of Dynamic Data Flow Computer **(b) Opcode structure**

Fig. 1.17.4

Advantages of Dataflow computer

1. No program counter
2. Data-driven
3. Execution inhibited only by true data dependences

4. Stateless / side-effect free
5. Further enhances parallelism

Disadvantages of dataflow computer

1. No program counter leads to very long fetch/execute latency
2. Spatial locality in instruction-fetch is hard to exploit
3. Requires matching (e.g., via associative compares)
4. No shared data structures
5. No pointers into data structures (implies state)

Review Questions

Q.1 How to improve speed of communication operations?

Q. 2 What are the applications of Parallel Computing?

Q. 3 Difference between Shared Memory and Distributed system?

Q. 4 Explain Array Processor.

Q. 5 What is Multiprocessor Architecture?

Q. 6 Explain Systolic Architecture and Data Flow Computers.

Q. 7 Explain in brief general classification of parallel computer architectures based on following techniques.

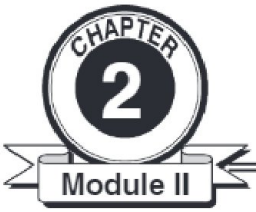
(i) Flynn's classification

(ii) Feng's classification.

Q. 8 What are the levels of parallel processing ? Explain in brief.

Q. 9 Explain Handlers classification of parallel computer Architecture.

❑❑❑



Pipeline Processing

Syllabus

Introduction, Pipeline Performance, Arithmetic Pipelines, Pipeline instruction processing, Pipeline stage design, Hazards, Dynamic instruction scheduling

2.1 Principles and Implementation of Pipelining

- A processor has many resources like the ALU, buses, registers, etc. An attempt to utilize all these attributes to their fullest or continuously can be achieved by pipelining.
- In a pipelined system the instructions flow through the processor as if the processor was a pipe. The instructions move from one stage to another to accomplish the assigned operation. Hence at most of the times each unit of the processor is busy handling one or the other instruction, making the attribute of the processor being used continuously.
- This chapter deals with advanced pipelining and superscalar design in the processor development. We will go through the concepts and design issues of the linear and non-linear pipelining. We will also discuss collision-free scheduling techniques for performing dynamic functions. Techniques to design instruction pipelines, arithmetic pipelines are also discussed.

2.2 Pipeline Performance: Speedup, Efficiency and Throughput

- A linear pipeline of k stages will take k + (n – 1) clock cycles to execute n instructions; the first instruction will take k clock cycles and the remaining n – 1 instructions will take one clock cycle each (assuming there are no dependency of the instructions). Hence with the clock cycle width being τ, the total time required to execute these n instructions will be

$$T_K = \tau [k + (n-1)] \quad \text{...(2.2.1)}$$

- An equivalent non-pipelined system the time required to execute n instructions will be

$$T_1 = nk\tau \quad \text{...(2.2.2)}$$

- Thus the speedup factor of a k-stage pipelined system can be given as :

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)} \quad \text{...(2.2.3)}$$

- Hence as the number of instructions n increases, S_k tends to k. Thus, **'k' stage pipeline processor can be at most 'k' times faster than the corresponding non-pipelined processor.** The speed up factor as a function of the number of instructions is shown in the Fig. 2.2.1.

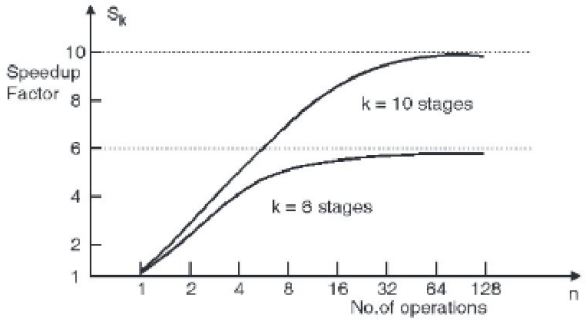


Fig. 2.2.1 : Relationship of speedup factor with the no. of operations

- Also there is a limit to the number of stages. As the number of stages increases the delay and skewing increases. Hence the finest pipelining or micro-pipelining that has the subdivision of the stages at almost gate level should consider this optimal number of stages. Most of the systems have the number of stages varying from 2 to 15. Very few systems are designed to have the number of stages beyond 10.
- This is because the increase in the number of stages should take care that the PCR (performance to cost ratio) has also increased. But beyond a particular number of stages, termed k_0 (optimal no. of stages), the PCR starts reducing as shown in the Fig. 2.2.2.

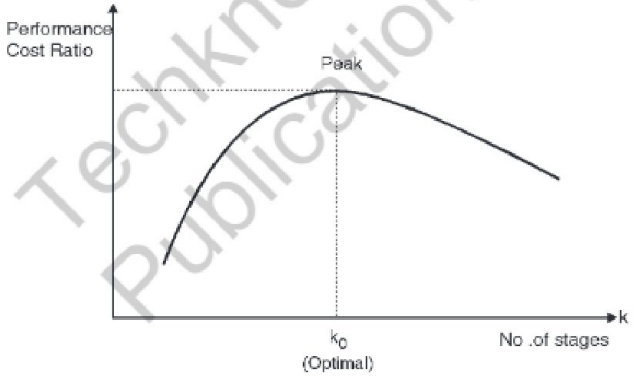


Fig. 2.2.2: Graph of PCR vs. k

- To execute a program on sequential non-pipelined system, say the time required is 't'. Thus to execute this program on a k-stage pipeline with equal flow-through the time required = $\frac{t}{k}$ + d, where d is latch delay.

$$\text{Hence,} \quad f = \frac{1}{\frac{t}{k} + d}$$

- The performance / cost ratio (PCR) can be defined as :

$$PCR = \frac{f}{(c + kh)} = \frac{1}{\left(\frac{t}{k} + d\right)(c + kh)} \qquad \ldots(2.2.4)$$

- where h is the cost of each latch, c is the cost of all logic gates of a stage, d is the latch delay, f is the pipeline frequency, k is the number of stages and t is the flow-through delay of each stage. The optimal number of stages can be given as

$$k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}} \quad \ldots(2.2.5)$$

- The efficiency of such a system is defined as (using Equation (2.2.3))

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)} \quad \ldots(2.2.6)$$

- The pipeline throughput H_k is defined as number of operations performed per unit time.

$$H_k = \frac{\text{Efficiency}}{\text{One unit time } (\tau)} = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)} \quad \ldots(2.2.7)$$

- Hence the maximum throughput as discussed earlier will be equal to f, when the number of instructions n tends to ∞.

2.3 Arithmetic Pipelines

An arithmetic pipeline may be a static pipeline or dynamic pipeline. The static arithmetic pipelines can perform only fixed functions while the dynamic pipelines can perform different operations.

2.3.1 Multiply Pipeline Design

Let's consider the multiplication of two 8-bit integers. The following process is used for the multiplication operation. The partial products P0 to P7 are obtained by multiplying each bit of the multiplier with the multiplicand.

								1	0	1	1	0	1	0	1	=	A
								×) 1	0	0	1	0	0	1	1	=	B
								1	0	1	1	0	1	0	1	=	P_0
							1	0	1	1	0	1	0	1	0	=	P_1
						0	0	0	0	0	0	0	0	0	0	=	P_2
					0	0	0	0	0	0	0	0	0	0	0	=	P_3
				1	0	1	1	0	1	0	1	0	0	0	0	=	P_4
			0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_5
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_6
+)	1	0	1	1	0	1	0	1	0	0	0	0	0	0	0	=	P_7
0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	1	=	P

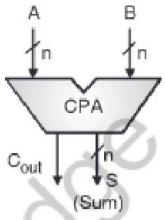
Fig. 2.3.1

- As seen in the above multiplication, the partial product P_j is obtained by multiplying the multiplicand with the j^{th} bit of the multiplier. According to the arithmetic pipeline design shown in Fig. 2.3.3, the first stage i.e. stage S_1, generates the partial products.

- The partial products based on the value of bit no. i.e. j, has its size, varying from 8 bits to 15 bits. The second stage i.e. stage $S_{2'}$ has two levels of four CSA (Carry Save Adder). The carry propagate adder (CPA) gives the sum at each bit level and the final carry.
- The CSA is a circuit that not only calculates the sum at each bit level, but also generates the carry generated at each bit level separately.
- Hence the same full adders are capable of adding three operands. The sum and the carry are then added. The final sum is the combination of the sum and one carry bit. Fig. 2.3.2 shows the difference between a CPA and a CSA.

e.g. n = 4

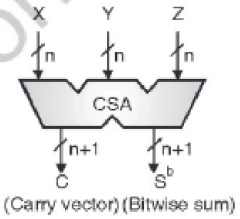
	A =	1 0 1 1
+)	B =	0 1 1 1
	S = 1	0 0 1 0 = A + B



(a) An n-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry look ahead technique

e.g. n = 4

	X =	0 0 1 0 1 1
	Y =	0 1 0 1 0 1
⊕	Z =	1 1 1 1 0 1
	S^b = 0	1 0 0 0 1 1
+)	C = 0	1 1 1 0 1 0
	S = 1	0 1 1 1 0 1 = S^b + C = X + Y + Z



(b) An n-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y and Z and C is a carry vector generated without carry propagation between digits

Fig. 2.3.2 : The concept of CPA and CSA

- The stage $S_{2'}$ merges the partial products by performing partial addition. Similarly stage S_2 which uses two CSA, generates the partial addition.
- As seen in the Fig. 2.3.3, each CSA is given three operands, which generates equal number of sum bits and carry bits. The three operands are given to next CSA for addition. Hence we finally get a 16-bit product of the entered 8-bit numbers at stage S4.

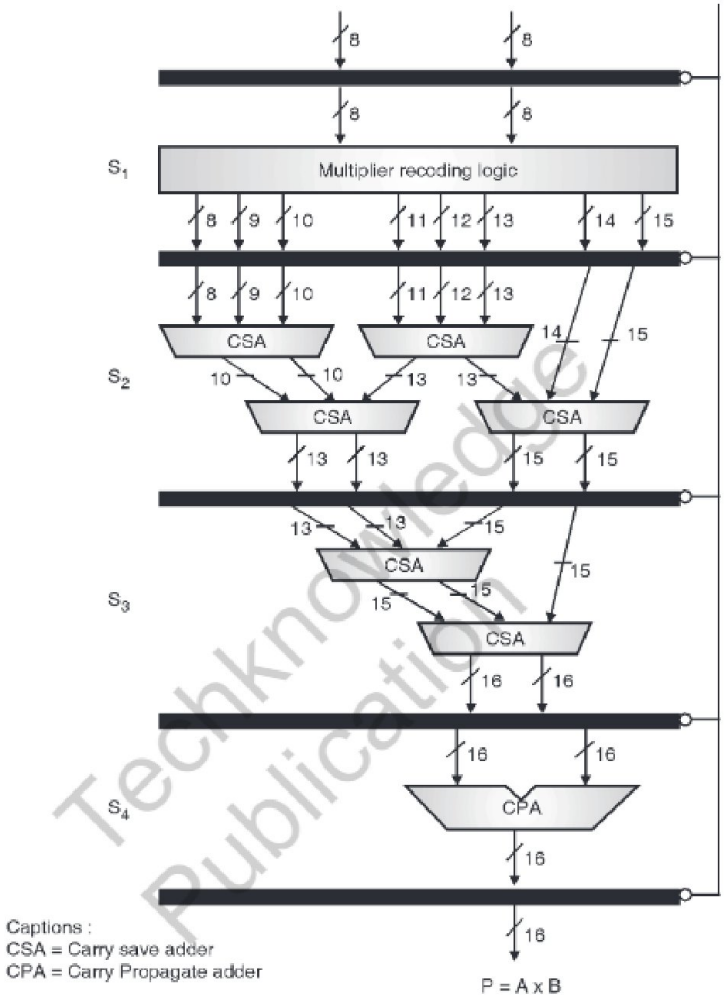


Fig. 2.3.3 : An example pipeline unit for 8-bit multiplication

2.3.2 Floating Point Addition Pipeline

- Some functions of the ALU can be pipelined. Complex functions like floating point addition can be decomposed. The floating point addition of two normalized FP numbers, x and y can be decomposed as following operations :
- Compare the exponents
- Align the mantissas
- Add the mantissas
- Normalize the result
- Hence this four-stage FP pipelined adder can be implemented as shown in Fig. 2.3.4.

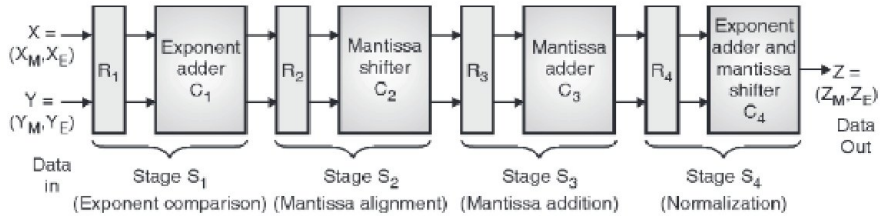


Fig. 2.3.4 : Four stage pipeline for floating point addition

The exponent of the two numbers to be added is compared. The mantissa of the number with smaller exponent (X_M) is shifted right which is called as mantissa alignment. The new mantissa is $X_M \rightarrow (X_M, Y_E) = (X_M, X_E)$ i.e. the exponent of Y, becomes the exponent of X; and then mantissa of X is such adjusted that the same exponent as that of Y can be given to X.

2.3.2(A) Steps involved in Designing of an Arithmetic Pipeline

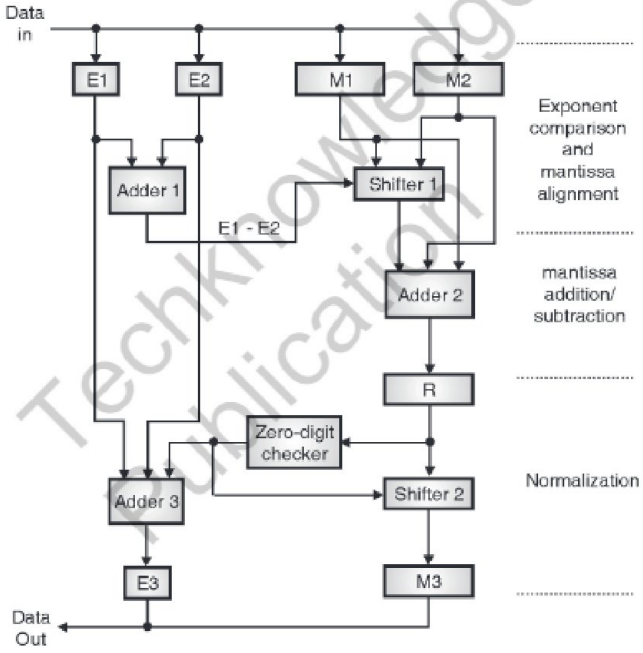


Fig. 2.3.5 : Non-pipelined version of floating point addition

- To design an arithmetic pipeline the following steps are to be followed:
  1. Finding a multistage sequential algorithm to compute the required arithmetic function. Care should be taken that the algorithm's steps are balanced.
  2. Placing buffer registers between the stages
- Let us first consider the non-pipelined floating point addition as it is implemented in the IBM System/360 Model 91 computer. The non-pipelined system to perform this floating point addition operation is shown in Fig. 2.3.5. As shown in the Fig. 2.3.5, there are four registers to store the mantissas and exponents of each of the operands, namely X and Y. First the exponents are subtracted by the adder 1.

- The number with smaller exponent gets its mantissa shifted towards right for the difference of exponents number of times. Thereafter the mantissas are added by the adder 2 and the final result is given to adder 3. The adder 3, performs the required normalization, combines the result exponent and the mantissa to give the final result.
- The pipelined version of the system shown in Fig. 2.3.5. The FP add unit of the IBM System/360 computer.
- In this case the exponents are subtracted in stage S1 and stored in the register E5. In stage S2. In stage S2, the shifting operation is performed and given to register M6 after shifting, the other mantissa being placed in M7. The larger mantissa is passed to register E6 of stage S3, from register E4 in stage S2. The stage S3 adds the mantissas and stores in register R of stage S4.
- The larger exponent is passed to register E7. This is adjusted in stage S4 by shifting the mantissa so as to normalize the result. The zero-digit checkers checks for the bit before fraction point is '0' or '1', which is required for normalization. The final result is obtained as mantissa in register M3 and exponent in register E3.

2.4 Linear Pipelining

In a linear pipelined processor there are n stages connected linearly to perform different operations. These may perform different operations to execute an instruction, perform arithmetic operations or memory access operations. In a linear pipelined processor with suppose n stages, the partially processed instructions are passed from stage i to stage i + 1, where i varies from 1 to k – 1. The linear pipeline can either be a synchronous system or asynchronous system.

2.4.1 Asynchronous and Synchronous Linear Pipelining

- In case of an asynchronous linear pipeline system, there is a set of handshaking signals between the two stages. Whenever a stage (say stage i) completes its operation, it places the result on the input lines of next stage (i.e. stage i + 1) and enables the ready (or strobe) signal.
- The next stage (i.e. stage i + 1) on completing its operation, accepts the data from its input lines and indicates this to the previous stage (i.e. stage i) by giving an acknowledgement signal.
- On this, the stage which had placed the data (i.e. stage i), also checks its input if it has previous stage (i.e. stage i – 1) completed its operation and is ready with the result. It also repeats the same process as explained with stage i and i + 1. This can be explained as shown in the Fig. 2.4.1.

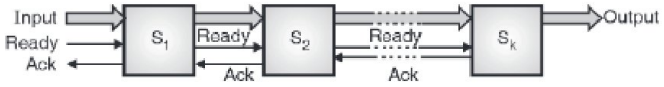


Fig. 2.4.1 : Asynchronous linear pipelining system

- Hence the asynchronous linear pipelined system will have variable throughput rate and will experience different amount of delay at each stage.
- In case of a synchronous linear pipelined system the stages are separated by latches. Whenever a stage completes its part of operation it stores the result in the latch. The clock signal is synchronously given to all the latches, such that on reception of the clock signal each stage takes the output of the latch connected to its input. This system is shown in Fig. 2.4.2.

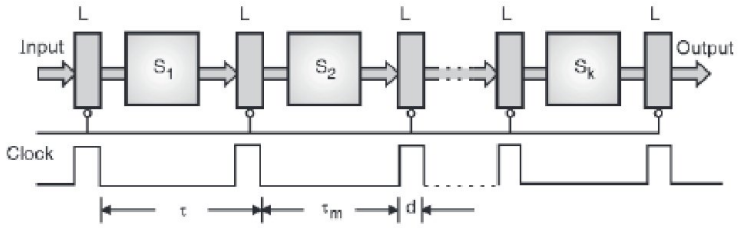


Fig. 2.4.2 : Synchronous linear pipelining system

- The latches are infact master-slave flipflops. The time required by each stage is expected to be equal; and it is this time that determines the clock period as well as the speed of the pipelined system.
- The utilization of the stages or the utilization pattern of stages in a synchronous pipeline can be represented by the reservation table. The reservation table follows a diagonal line for synchronous linear pipeline as shown in Fig. 2.4.3.

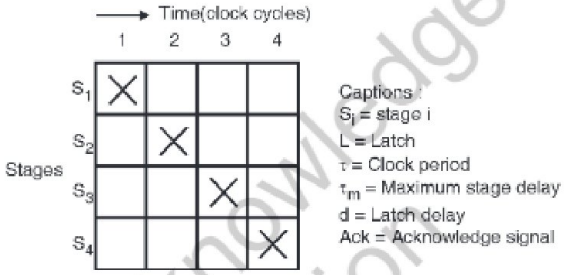


Fig. 2.4.3 : Reservation table of a synchronous linear pipeline

- Reservation table is a space-time diagram showing streamline pattern. Hence as seen in Fig. 2.4.3, for an n-stage pipeline, n clock pulses are required to execute the instruction. Once the pipeline is filled up completely, the processor completes one instruction execution every clock pulse.

2.4.2 Clocking and Timing Control

- The clock cycle, τ as shown in Fig. 2.4.2, can be calculated as discussed below. Let τ_i, be the delay time for stage S_i. Hence the **clock cycle** time can be given as :

$$\tau = \max [\tau_i]_1^K + d = \tau_m + d$$

- where τ_m the maximum stage delay and d is, as shown in the Fig. 2.4.2, the 'on' period of the clock pulse.
- The data from each stage is latched in the master flipflop of latch register during the rising edge and given to the slave flipflop during the falling edge. In fact, $\tau_m >> d$; hence we can say that, $\tau \cong \tau_m$. Hence the pipeline frequency can be given as = 1 / τ. This frequency f, is also termed as the throughput of the system as it gives the time required for one instruction to come out of the pipeline.
- The actual **throughput** of the pipeline may be less than the maximum throughput given by f, which may be because of more than one clock pulses, may be required for the initiation of successive instructions. The initiation of successive instructions may take more clock pulses because of their data or control dependency. The clock pulse at each stage is expected to arrive simultaneously.
- But, because of the time delay of the path, different stages get the pulse at different time offset *s*; this problem is referred to as **clock skewing**.

– Assuming the shortest logic path to get the clock at a delay of t_{min} and the longest logic path to get the clock pulse at delay of t_{max}; to avoid this problem the $\tau_m \geq t_{max} + s$ and $d \leq t_{min} - s$. Thus the clock period with skew is :

$$d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s$$

Hence in ideal case, $s = 0$, $t_{max} = \tau_m$ and $t_{min} = d$. Hence, even with the clock skewing $\tau = \tau_m + d$

Ex. 2.4.1 : Consider the execution of a program of 15,000 instructions by a linear pipeline processor with a clock rate of 25 MHz. Assume that the instruction pipeline has five stages and that one instruction is issued per clock cycle. The penalties due to branch instructions and out-of-sequence executions are ignored.

(a) Calculate the speedup factor in using this pipeline to execute the program as compared with the use of an equivalent non-pipelined processor with an equal amount of flow-through delay.

(b) What are the efficiency and throughput of this pipelined processor ?

Soln. :

Given : n = 15000, f = 25 MHz, k = 5

$$\text{Speed up factor } (S_k) = \frac{nk}{k + (n-1)} = \frac{15000 \times 5}{5 + (15000 - 1)} = 4.9986$$

$$\text{Efficiency } (E_k) = \frac{S_k}{k} = 0.99973$$

$$\text{Throughput } (H_k) = \frac{E_k}{k} = f\,E_k = 25 \text{ MHz} \times 0.99973 = 24.9933 \text{ MIPS}$$

Ex. 2.4.2 : A non-pipeline system takes 50 ns to process a task. The same task can be processed in a six stage pipeline with a clock cycle of 10 ns. Determine the speedup and the efficiency of the pipeline for 100 tasks. What is the maximum speedup and efficiency that can be achieved ?

Soln. :

Given : For the non-pipeline system : $t_n = 50$ ns

For the pipeline system : $k = 6$, $t_p = 10$ ns

Number of tasks n = 100

$$\text{Speed up } (S_k) = \frac{T_1}{T_k} = \frac{nk\tau_1}{k\tau + (n-1)\tau} = \frac{100 \times 50}{6 \times 10 + (100 - 1) \times 10} = 4.7619$$

Maximum speedup will occur when the number of tasks (n) is very large (n >> k). Hence neglecting the term k – 1, we have Max Speedup $= \frac{n\tau_1}{n\tau} = \frac{50}{10} = 5$

$$\text{Efficiency } (E_k) = \frac{S_k}{k} = \frac{4.7619}{6} = 0.7936$$

Considering the max speedup the max efficiency $= \frac{5}{6} = 0.8333$

2.5 Non Linear Pipeline Processors

– A dynamic or multi-function pipeline is called as non-linear pipeline. In a linear pipeline the operations that are being performed are fixed; each stage as a fixed operation.

– But in a non-linear pipeline allows feed forward and feedback connections in addition to the streamline connection. It may also have more than one output i.e. the output need not be from the last stage. An example of three stage non-linear pipeline system is shown in Fig. 2.5.1.

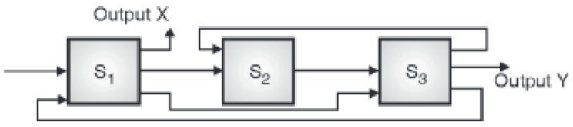


Fig. 2.5.1 : A 3-stage non linear pipeline

- In the Fig. 2.5.1 besides the three stages connected in streamline, there are also some feedback and feed forward connections.
- The feed forward connection is from S_1 to S_2 and feedback connections from S_2 to S_2 and S_3 to S_1. The reservation table for such connections may be different for different operations. There are two examples of different operations say, X and Y, for which the reservation table is shown in Fig. 2.5.2.

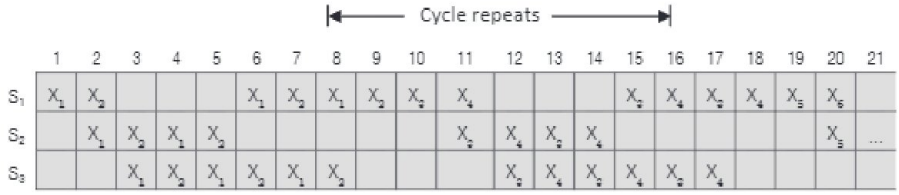
	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Fig. 2.5.2(a) : Reservation table for function X

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

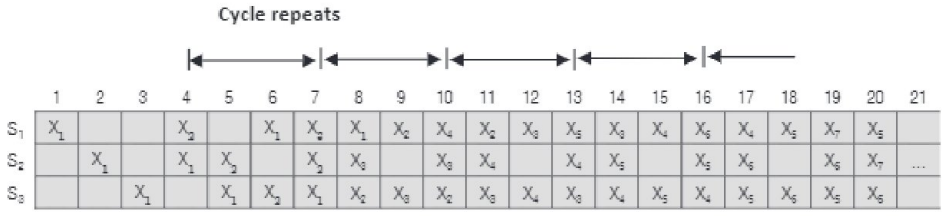
Fig. 2.5.2(b) : Reservation table for function Y

- The Fig. 2.5.2 shows the requirement of different stages at different times for doing the corresponding operation. For e.g. the function X, has first to be given to S_1, then S_2, then S_2, then S_2, then S_2, then S_1, then S_2 and finally to S_1 which will give the output. Similarly the function Y is first given to S_1, then S_2, then S_2, then S_2, then S_1 and finally to S_2 which will give the output. The number of columns in a reservation table corresponds to the **evaluation time** of that function. Hence from Fig. 2.5.2, function X has an evaluation time of 8 clock cycles while function Y has an evaluation time of 6 clock cycles. A pipeline initiation table consists of different time when the next time the same function can be initiated.
- The number of time units (clock cycle) required between the two initiations of a function is called as the **latency period** between them. Some valid latencies or latency sequences that do not cause any collision are shown in Fig. 2.5.3. The latencies that do not cause collision are called as **latency sequence** while latencies that cause collision are called as **forbidden latencies**. Examples of forbidden latencies are shown in Fig. 2.5.4.

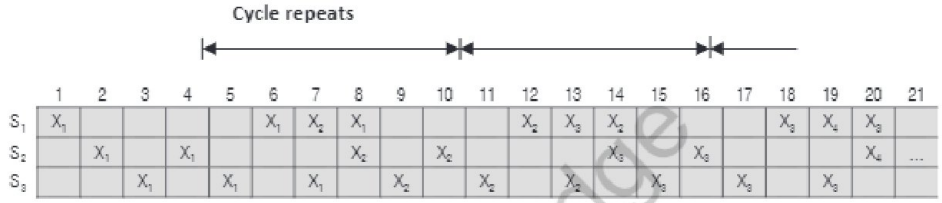


(a) Latency cycle (1, 8) = 1, 8, 1, 8, 1, 8, ..., (With an average latency of 4.5)

Fig. 2.5.3 contd,,,

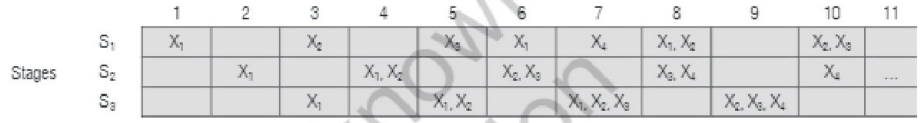


(b) Latency cycle (3) = 3, 3, 3, 3, ... (With an average latency of 3)

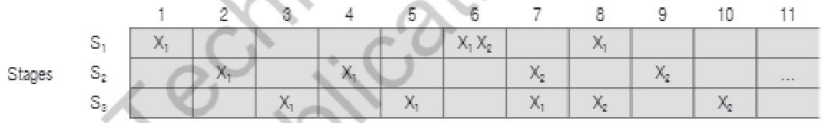


(c) Latency cycle (6) = 6, 6, 6, 6, ..., (With an average latency of 6)

Fig. 2.5.3 : Example latencies of function X that do not cause collision



(a) Collision with scheduling latency 2



(b) Collision with scheduling latency 5

Fig. 2.5.4 : Example forbidden latencies i.e. latencies that cause collision of function X

- As shown in Fig. 2.5.3, forbidden latencies are 2 and 5. Besides, 4 and 7 are also forbidden latencies. To detect the forbidden latency, you need to check the distance between the two marks on the reservation table in a row. For e.g. in case of function X, as shown in Fig. 2.5.2(a), the distance between two marks in S_1 is 5 or 2.
- Average latency of a latency cycle is defined as the ratio of sum of all latencies to the number of latencies along the cycle. For e.g. (1,8) latency as shown in Fig. 2.5.3(a), has an average latency of (1+8)/2 = 4.5. The average latency of a constant cycle, i.e. latency cycle that contains only one latency value, is same as the constant value. For e.g. the average latency of the cycle 3 and 6, as shown in Figs. 2.5.3 (b) and (c), are 3 and 6 respectively.

2.5.1 Collision Free Scheduling or Job Sequencing

As seen in non-linear pipelining, we cannot sequence the instructions (job) as in linear pipelining; else there would be collision. Hence we need to have optimal job sequencing or scheduling technique. In a non-linear pipelining while scheduling, the main aim is to obtain smallest average latency without any collision. This pipeline design theory requires the concepts of collision vectors, state diagrams, single cycles, greedy cycles and minimal average latency (MAL).

1. Collision vectors

- As seen in the previous section, we can separate the permissible latencies and the forbidden latencies using the reservation table. For a reservation table with n columns, the maximum forbidden latency (m) should be ≤ n – 1. The permissible latency (p), must be as small as possible. An ideal case would be p = 1. This smallest latency i.e. p = 1 is possible in linear pipelining, but in non-linear pipelining, it is difficult to achieve.
- A **collision vector** is an 'm' bit binary vector (C = C_m C_{m-1}.....C_2 C_1), that shows the set of permissible and forbidden latency. In a collision vector, a bit C_i is '1' if the latency i causes a collision, else it is '0'. For e.g., the reservation tables seen in Fig. 2.5.2, will have the collision vectors as C_x = (1011010) and C_y = (1010). Thus for C_x, there is a permissible latency 1, 3 and 6 while forbidden latencies of 7, 5, 4 and 2.

2. State Diagrams

- From the collision vector, we can make the state diagram for the pipeline. The collision vector C_x, achieved above is called as the initial collision vector. When loaded in a register and shifted right, each bit at the output corresponds to an increase in latency. A '1' at the output indicates collision, while a '0' indicates no collision. A '0' is inserted from the left for every clock cycle. This can be implemented as said by a right shift register and OR gates, as shown in Fig. 2.5.5.

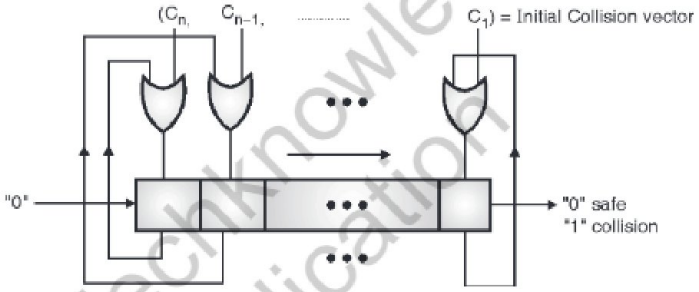
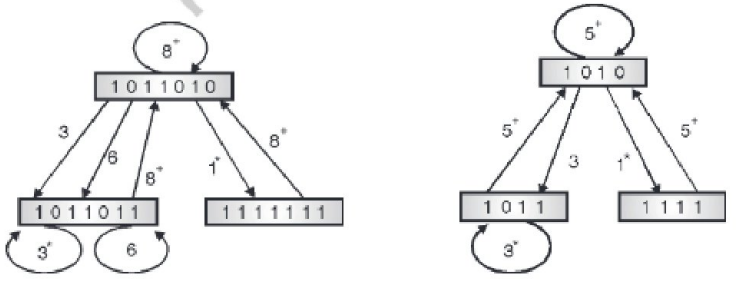


Fig. 2.5.5 : n-bit right shift register for state transition

- The state transition diagram can be constructed using this state register. The next state at time t + p, where p is the permissible latency and t is some no. of clock pulses, obtained by shifting the register for p times and ORing it with the initial collision vector. The state diagrams for the collision vectors C_x and C_y are as shown in Fig. 2.5.6.



(a) State diagram for collision vector C_y (b) State diagram for collision vector C_y

Fig. 2.5.6 : State diagrams for collision

- A transition example can be explained as below. For e.g. the three bit shifts with the initial vector of function X, will result in 0001011; this when ORed with the initial collision vector results in 1011011.

– The state diagram (Refer Fig. 2.5.6 (a)) shows this transition for the function X. If the number of shifts is greater than m, the next state is same as the initial collision vector. For e.g. if the number of shifts is 8 or more in Fig. 2.5.6 (a), it comes back to the initial collision state. This transition is denoted by 8^+.

3. Single cycle and Greedy cycle

A single cycle is one in which any state appears not more than once. For example for the X function state diagram shown in Fig. 2.5.6 (a), the different single cycles are (3), (6), (8), (1,8), (3,8) and (6,8). A cycle that travels for more than one time through the same state is a greedy cycle. some greedy cycles in the Fig. 2.5.6(a) are (1,8,3,8), (1,8,6,8), (3,6,3,8,6) etc.

4. Minimal Average Latency (MAL)

We have already studied the minimal average latency in the previous sections. There are some bounds on this value of MAL. These bounds are listed below :

(i) The lower bound of MAL is the maximum number of checkmarks in any row of the reservation table

(ii) The MAL should be lower than or equal to the latency of any greedy cycle in a reservation table.

(iii) The upper bound of MAL is equal to the number of 1s in the initial collision vector plus 1.

2.5.2 Delay Insertion

– The reason of inserting delay states is to modify the reservation table and hence generating a new collision vector. The new collision vector should generate a state diagram and in turn generate greedy cycles so as to meet the lower bound of MAL.

– Let us take an example of a system wherein inserting delays generates lower bound MAL. The system, its reservation table and the state diagram are shown in Fig. 2.5.7.

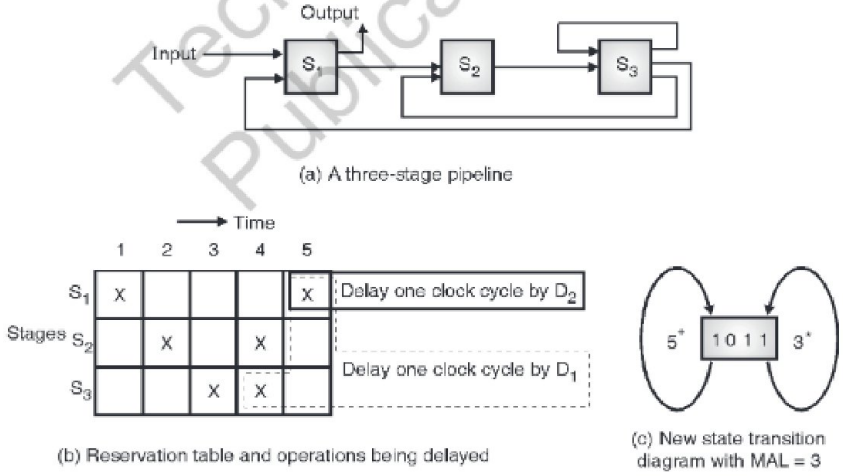
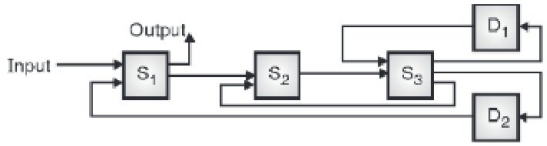
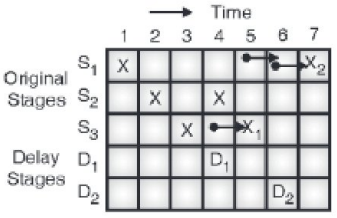


Fig. 2.5.7

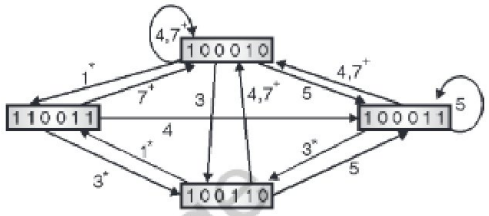
– The collision vector of this is C_x = (1011). Hence we have a permissible latency of (3) and (6). This is shown in the state diagram in Fig. 2.5.7(c). After inserting a delay the clock cycles, we get a lower bound MAL, as shown in Fig. 2.5.8.



(a) Insertion of two noncompute delay stages



(b) Modified reservation table



(c) Modified state diagram with a reduced MAL = (1+3) /2 = 2

Fig. 2.5.8

– As seen in Fig. 2.5.8, the operation X_1 is delayed by 1 clock and X_2 by 2 clock pulse. The new collision vector is hence, C_x = (100010). Hence the state diagram can be designed as shown in Fig. 2.5.8(c). The permissible latencies are (4), (5,4), (1,3), (1,3,4), etc. Hence the MAL = (1+3)/2 = 2.

Ex. 2.5.1 : Consider following pipeline reservation table.

	1	2	3	4
S_1	X			X
S_2		X		
S_3			X	

(a) What are the forbidden latencies ?

(b) Draw the state transition diagram.

(c) List all simple cycles and greedy cycles.

(d) Determine the optimal constant latency cycle and the minimal average latency.

(e) Let the pipeline clock period be τ = 20 ns. Determine the throughput of this pipeline.

Soln. :

(a) Collision Vector = (100)

X_1	X_2	X_3	X_1 X_4	X_2 X_5	X_3	X_4	X_5
	X_1	X_2	X_3	X_4	X_5	...	...
		X_1	X_2	X_3	X_4	X_5	...

Collision with latency 1

X_1			X_1 X_2			X_2 X_3			X_3
	X_1			X_2			X_3		...
		X_1			X_2			X_3	...

Collision with latency 3

Hence latencies (1) and (3) are forbidden latencies.

(b) State transition shift register

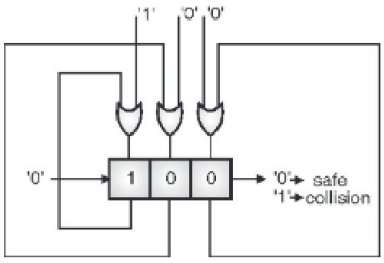


Fig. P. 2.5.1

Using the above shift register, we can generate the following state transition Fig. P. 2.5.1(a).

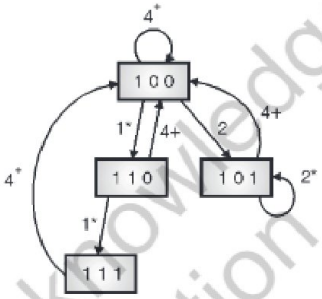


Fig. P. 2.5.1(a)

(c) As seen in the state transition diagram,

Simple cycles : (2), (4), (1,4), (1,1,4) (2,4) etc

Greedy cycles : (1,4,2,4) , (1,1,4,2,4) etc.

(d) Optimal constant latency (2)

∴ Minimum average latency (MAL) = 2

(e) Throughput

Cycle repeats

S_1	x_1		x_2	x_1	x_3	x_2	x_4	x_3	x_5	...
S_2		x_1		x_2		x_3		x_4		...
S_3			x_1		x_2		x_3		x_4	...

As seen in the above table, one instruction is executed every two cycles.

$$\therefore \text{Throughput} = \frac{1}{2} \times f = \frac{1}{2} \times \frac{1}{20 \text{ nsec}}$$

$$= 25 \text{ MIPS}$$

Ex. 2.5.2 : A non-pipelined processor X has a clock rate of 25 MHz and an average CPI (cycles per instruction) of 4. Processor Y, an improved sucessor of X, is designed with a five-stage linear instruction pipeline. However, due to latch delay and clock skew effects, the clock rate of Y is only 20 MHz.

(a) It is a program containing 100 instructions is executed on both processors, what is the speedup of processor Y compared with that of processor X?

(b) Calculate the MIPS rate of each processor.

Soln. :

(a) Program has 100 instruction.

i.e. n = 100

Time taken by a non-pipelined (X) processor to execute this program (T1) = nkτ.

where n = number of instruction = 100

k = number of cycles per instruction = 4

τ = clock width = $\frac{1}{25\ MHz}$

$l_1 = nk\tau = 100 * 4 * \frac{1}{25\ MHz}$

= 16 μsec.

For a 5-stage pipelined (Y) processor, the time required to execute n-instruction (T_k) = [k + (n – 1)] τ.

where k = 5 stages

n = 100 instructions

$\tau = \frac{1}{f} = \frac{1}{20MHz} = 0.05\ \mu sec$

$T_k = [5 + (100 - 1)] * 0.05\ \mu sec$

= 5.2 μsec

$\therefore$ Speedup = $\frac{T_1}{T_k} = \frac{16\ \mu sec}{5.2\ \mu sec} = 3.07$

(b) Non-pipelined processor (X)

Time taken	Instructions
16 μsec	100
1 sec	x

$\therefore$ MIPS rate, (x) = $\frac{100\ Instruction}{16\ \mu sec}$ = 6.25 MIPS (Million instructions per second)

Pipelined processor (Y)

Time taken	Instructions
5.2 μsec	100
1 sec	x

$\therefore$ MIPS rate, (x) = $\frac{100\ Instruction}{5.2\ \mu sec}$ = 19.23 MIPS

Ex. 2.5.3 : Consider the following pipeline reservation table

	0	1	2	3	4	5	6	7	8
1	×								×
2		×	×					×	
3				×					
4					×	×			
5							×	×	

(i) Determine latencies in the forbidden list F and collision vector C

(ii) Draw state Transition diagram

(iii) List all simple cycles and greedy cycles

(iv) Determine MAL

Soln. :

(i) Forbidden latencies F = (1, 5, 6, 8)

∴ Collision vector C = (10110001)

(Since the total number of tasks are 9, there are 8 bits in collision vector i.e. C_8 to C_1)

(ii) State transition diagram

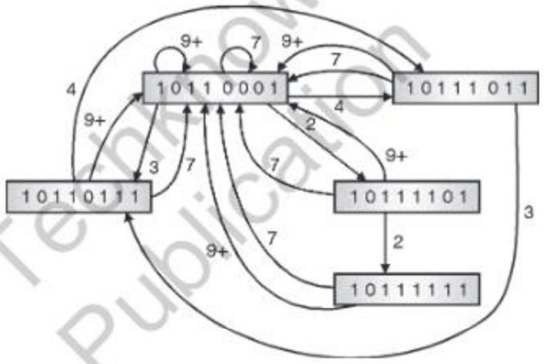


Fig. P. 2.5.3

(iii) Latency cycles : (7), (2, 7), (2, 2, 7), (4, 7), (4, 3), (3, 7), (4, 3, 4, 7), (9), (2, 9), (4, 9) (3, 9)

Simple cycles : (7), (2, 7), (4, 7), (4, 3), (3, 7), (9), (2, 9), (4, 9), (3, 9)

Greedy cycles : (2, 2, 7), (4, 3, 4, 7)

(iv) Optimal control latency : (4, 3)

∴ Minimum average latency (MAL) = $\frac{4+3}{2} = 3.5$

Ex. 2.5.4 : Consider the following pipeline reservation table

Stage \ Clock cycle	0	1	2	3	4	5	6
S_1	×		×				×
S_2				×		×	
S_3			×		×		

(i) Determine latencies in Forbidden list F and collision vector C

(ii) Draw the state transistor diagram

(iii) List all simple cycles and greedy cycles

(iv) Determine minimum average latency (MAL)

(v) For a pipeline clock period τ = 20 ns. Determine maximum throughput of the pipeline

Soln. :

(i) Forbidden latencies = (2, 4, 6)

Collision vector C = (101010)

(Since the total number of tasks are 7, there are 6 bits in the collision vector i.e. C_6 to C_1)

(ii) State transition diagram

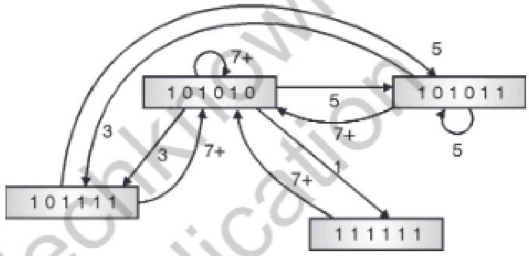


Fig. P. 2.5.4

(iii) Latency cycles : (7), (1, 7), (3, 7), (3, 5), (5), (3, 7, 5, 3, 7), (5, 3, 7)

Simple cycles : (7), (1, 7), (3, 7), (3, 5), (5), (5, 3, 7)

Greedy cycles : (3, 7, 5, 3, 7)

(iv) Optimal constant latency : (1, 7) or (3, 5)

$\therefore$ Minimal average latency (MAL) $= \frac{1+7}{2} = 4$

(v) Since MAL = 4, 1 instruction will be executed every 4 cycles.

$$\therefore \text{ Throughput } = \frac{1}{4} * f = \frac{1}{4} * \frac{1}{20 \text{ nsecs}} \qquad \left(\because \text{ frequency} = \frac{1}{\tau} = \frac{1}{20 \text{ nsecs}}\right)$$

$$= \frac{1}{70 \text{ nsecs}} = 25 \text{ MIPS}$$

Ex. 2.5.5 : For a unifunction pipeline, the forbidden set of latencies is as given below.

F = {1, 3, 6} with the largest forbidden latency = 6

(i) Obtain collision vector (ii) Draw the state diagram

(iii) State all simple and greedy cycles (iv) Obtain MAL

Soln. :

(i) Collision vector (C) = (100101)

(ii)

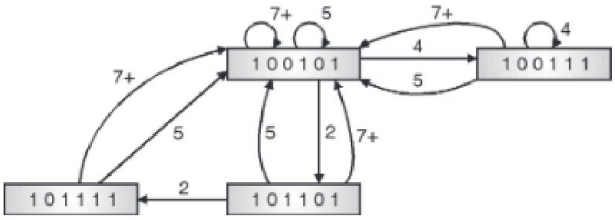


Fig. P. 2.5.5

(iii) Latencies : (5), (4), (4, 5), (2, 5), (2, 2, 5), (7), (2, 7), (4, 7)

Simple latencies : (5), (4), (4, 5), (2, 5), (7), (2, 7)

Greedy latency : (2, 2, 5)

(iv) Optimal latency : (2, 2, 5)

$\therefore$ Minimal average latency (MAL) = $\frac{2+2+5}{3} = 3$

Note : Optimal latency is the latency that gives minimal average i.e. the average as minimum.

Ex. 2.5.6 : For the following reservation table, determine collision vector, state transition diagram and MAL.

Stage \ Clock cycle	1	2	3	4	5	6	7	8
S_1	×					×		
S_2		×				×	×	
S_3			×		×			×
S_4				×				

Also find the throughput for τ = 10 nsec

Soln. :

(i) Collision vector (C) = (0011111)

(ii) State transition diagram : (Refer Fig. P. 2.5.6)

Fig. P. 2.5.6

(iii) Latencies : (6), (7)

Simple latencies : (6), (7)

Greedy latencies : NIL

(iv) Optimal latency : (6)

Minimal average latency (MAL) = 6

(v) Since MAL = 6, 1 instruction takes 6 clock pulses

$\therefore$ Throughput $= \frac{1}{6} * f = \frac{1}{6} * \frac{1}{10 \text{ nsec}}$

$= 16.67$ MIPS

Ex. 2.5.7 : For the following reservation table, determine collision vector state transition diagram and MAL.

	1	2	3	4	5
S_1	×			×	
S_2			×		×
S_3		×			

Also find the throughput for τ = 25 nsecs

Soln. :

(i) Collision vector (C) = (0110)

(ii) State transition diagram : (Refer Fig. P. 2.3.7)

(iii) Latencies : (4), (1,4)

Simple latencies : (4), (1,4)

Greedy latencies : NIL

(iv) Optimal latency : (1,4)

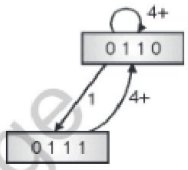


Fig. P. 2.5.7

Minimal average latency (MAL) $= \frac{1+4}{2} = 2.5$

(v) Since MAL = 2.5, 1 instruction takes 2.5 clock pulses

$$\therefore \text{ Throughput } = \frac{1}{2.5} * f = \frac{1}{2.5} * \frac{1}{25 \text{ nsec}}$$

$$= 16 \text{ MIPS}$$

Ex. 2.5.8 : A certain pipeline with the four stages S1, S2, S3 and S4 is characterized by the following Table P. 2.5.8.

Table P. 2.5.8

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
S_1	X					X	
S_2			X				X
S_3		X		X			
S_4			X		X		

(i) Determine the latencies in the forbidden list F and the collision vector C.

(ii) Determine the minimum constant latency L by checking the forbidden list

(iii) Draw the state diagram for this pipeline and determine MAL.

Soln. :

(i) Forbidden latencies = (2, 4, 5)

Collision vector C = (011010)

(ii) State transition diagram

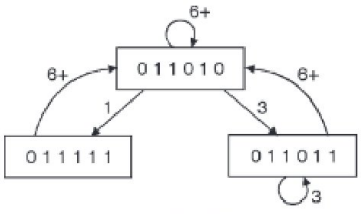


Fig. P. 2.5.8

(iii) Latency cycle : (6), (1, 6), (3), (3, 6), (3, 3, 6), (6, 1, 6), (3, 6, 6)

Simple cycles : (6), (1, 6), (3), (3, 6)

Greedy cycles : (3, 3, 6), (6, 1, 6), (3, 6, 6)

(iv) Optimal constant latency : (3)

Minimum average latency (MAL) = 3

2.6 Pipeline Instruction Processing

- Instruction pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.
- Generally, the processor fetches an instruction from memory, decodes it to determine what the instruction was, read the instructions inputs from the register file, performs the computation required by the instruction and writes the result back into the register file. This approach is called unpipelined approach.
- The problem with this approach is that, the hardware needed to perform each of these steps (Instruction fetch, instruction decode, register read, instruction execution and register write-back) is different and most of the hardware is idle at any given moment. Waiting for the other parts of the processor to complete their part of executing an instruction.
- Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.
- Each instruction takes the same amount of time to execute in a pipelined processor as it would in a non-pipelined processor, but the rate at which instructions can be executed is increased by **Overlapping Instruction Execution**.
- **Latency** is the amount of time that a single operation takes to execute.
- **Throughput** is the rate at which operations get executed.
- In a non-pipelined processor,

  $$\text{Throughput} = \frac{1}{\text{latency}}$$

  [expressed as operations/second or operations/cycles]

  In a pipelined processor,

  $$\text{Throughput} > \frac{1}{\text{latency}}$$

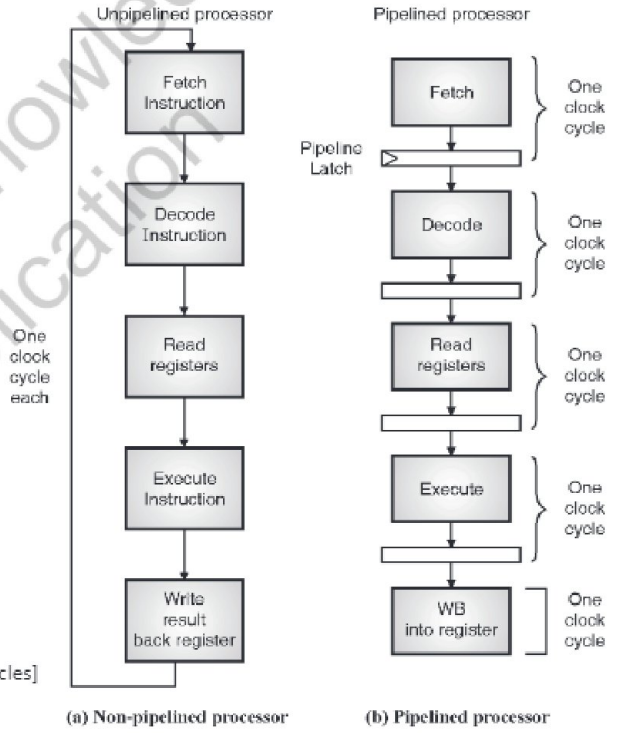


Fig. 2.6.1

- **Pipelining :** To implement pipelining, designers divide a processor's data path into sections called stages and place pipeline latches between each section.

  As shown in Fig. 2.6.1, at start of each cycle, the pipeline latches read their inputs and copy them to their outputs.

  The amount of data path that a signal travels through in one cycle is called a stage of the pipeline.

A **five-stage pipeline** is shown in Fig. 2.6.1.

Stage 1 : Fetch block

Stage 2 : Decode block

Stage 3, 4, 5 are subsequent blocks in execution process.

Fig. 2.6.2 shows the Instruction flow in a pipelined processor.

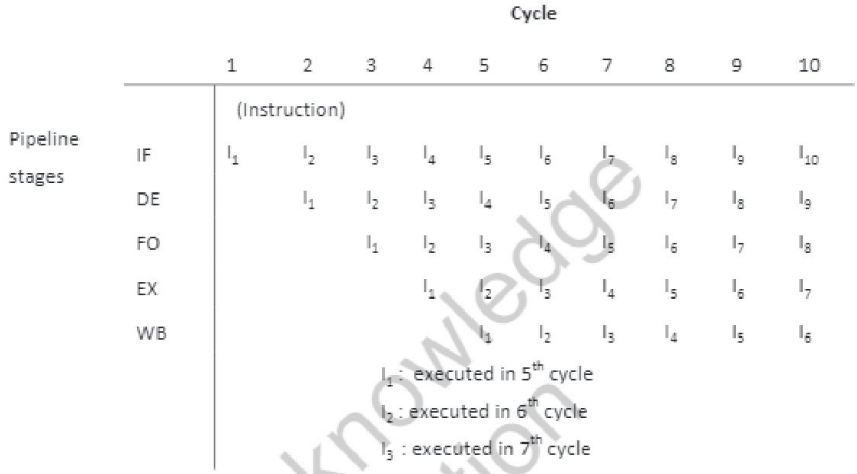


Fig. 2.6.2

- Cycle time of a pipelined processor is calculated as

$$\text{Cycle time (pipelined)} = \frac{\text{Cycle time (unpipelined)}}{\text{Number of pipeline stages}} + \text{Pipeline latch latency}$$

- As, the number of pipeline stages increases, the pipeline latch latency increases which in turn limits the benefit of dividing a processor into a very large number of pipelining stages.

Ex. 2.6.1 : An unpipelined processor has a cycle time of 25 ns. What is the cycle time of a pipelined version of the processor with 5 evenly divided pipeline stages, if each pipeline latch has a latency of 1 ns ?

Soln. :

$$\text{Cycle time pipelined} = \frac{\text{Cycle time unpipelined}}{\text{Number of stages of pipeline}} + \text{Pipeline latch latency}$$

$$= \frac{25 \text{ ns}}{5} + 1 \text{ ns} = 6 \text{ ns.}$$

To find the speedup of the execution process in a pipelined processor,

$$\text{Execution time pipelined} = (K + n - 1)\,\tau$$

$$\text{Execution time unpipelined} = (K\,\tau) \times n$$

Where, n = number of instructions

τ = time taken for each stage

K = number of stages in pipeline

Ex. 2.6.2 : If a processor executes 100 instructions in a pipelined (5 stage) processor and unpipelined processor. What is the speedup achieved by pipelining technique if the time taken for each stage is 20 ns.

Soln. :

n = 100 instruction, K = 5, τ = 20 ns

$$\text{Execution time pipelined} = (5 + 100 - 1) \times \tau$$

$$= (5 + 99) \times 20 \text{ ns} = 2080 \text{ ns.}$$

$$\text{Execution time unpipelined} = (K\tau)\, n = 5 \times 20 \text{ ns} \times 100 = 10000 \text{ ns.}$$

$$\text{Speedup ratio is} = \frac{10000}{2080} = 4.80 \text{ times.}$$

2.7 Pipeline Stage Design

2.7.1 Non-Pipelined System vs. Two Stage Pipelining

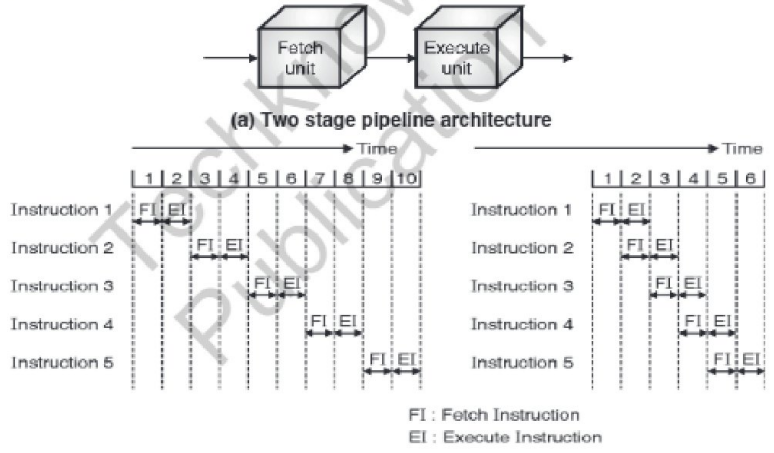


Fig. 2.7.1

- In a non-pipelined system, the processor fetches an instruction from memory, decodes it to determine what the instruction was, read the instructions inputs from the register file, performs the computation required by the instruction and writes the result back into the register file. This approach is also called as unpipelined approach.

- The problem with this approach is that, the hardware needed to perform each of these steps (Instruction fetch, instruction decode, register read, instruction execution and register write-back) is different and most of the hardware is idle at any given moment. Waiting for the other parts of the processor to complete their part of executing an instruction.
- Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.
- Two stage pipelining includes two stages i.e. Fetch instruction and Execute instruction.
- These two operations are performed for one instruction and the next instruction overlapping i.e. when the first instruction is being executed the next is fetched and when this instruction is executed the next is fetched and so on.
- This method of execution the instructions in pipeline speeds up the processor operation.
- This also makes sure that all the units of the processor are busy operating and none of them is starving.
- Thus with the help of pipelining the operation speed of the processor increases i.e. more the number of pipeline stages, faster becomes the processor, but complex in design.
- Two stage instruction pipeline stage is as shown in Fig. 2.7.1(a).
- In case of a system without pipelining the time required for executing a set of instructions is much more than the time required to execute the same set of instructions in a pipelined system.
- The comparison of the execution of five instructions in a system with and without pipeline is shown in Figs. 2.7.1 (b) and 2.7.1(c).
- You will notice that the time required for executing five instructions on a non-pipelined system is 10 clock pulses while that on two stage pipelined processor is 6 clock pulses. Thus the number off clock pulses required in two stage processor will always be x/2 + 1, where 'x' is the number of clock pulses in non-pipelined instructions and '2' is the number of stages. If we increase the number of instructions, we can make the expression as x/2 (since '1' is negligible for huge number of instructions) clock pulses for a two stage pipelined processor, wherein 'x' clock pulses in case of non-pipelined processor.
- If we try to generalize this expression, we can write it as x/n, where x is number of clock pulses required for non-pipelined instructions and 'n' is the number of stages of a pipelined processor.
- Thus we can say that the speed-up achieved by a pipelined processor can be maximum 'n' times of the non-pipelined processor.

2.7.2 Six Stage CPU Instruction Pipeline

The flowchart given in Fig. 2.7.2 gives an instruction flow through the six stage pipelined processor.

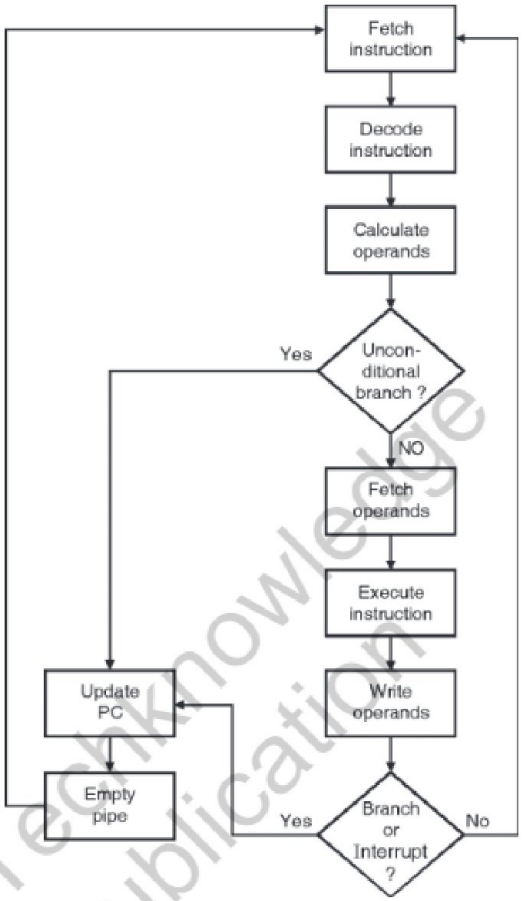


Fig. 2.7.2 : Six stage pipeline flowchart

This can be shown on a time scale as in Fig. 2.7.3.

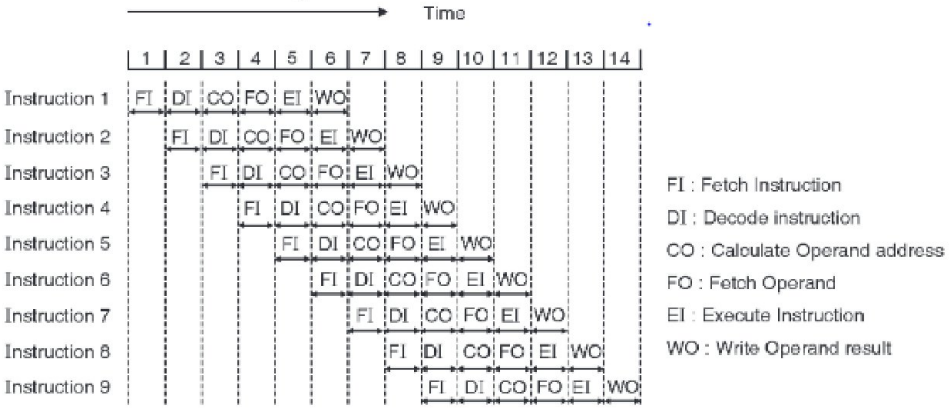


Fig. 2.7.3 : Six stage pipelined processor timing diagram

- Pipelining is termed as overlapped parallelism as in case of pipelining the instructions are overlapped. The execution of the instructions is overlapped in such a manner that there are several instructions under different process in the pipelined processor as shown in Fig. 2.7.3.
- As shown in the Fig. 2.7.3, for e.g. during the 6^{th} clock pulse, there are six instructions in the processor. Instruction 1 has its result being written back, instruction 2 is being executed, instruction 3 has its operands being fetched, instruction 4 has the address of operands being calculated, instruction 5 is being decoded and instruction 6 is being fetched. This shows how pipelining is a overlap parallelism of instructions. This six stage pipeline system can be implemented with six units as shown in Fig. 2.7.4

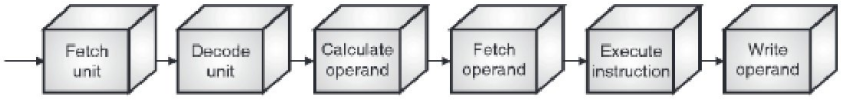


Fig. 2.7.4 : Six stage pipelined architecture

- In pipelining, when a branch instruction is executed the system causes a huge waste of time i.e. processor units starving. The instructions in the pipeline are the sequential instructions. If a branching instruction is given the next instruction to be executed is not the sequential one, instead it is the instruction at target instruction.
- Hence the sequential instructions in the pipeline are to be cleared and instructions from target are to be fetched. Clearing the sequential instructions from the pipeline is called as flushing of the pipeline.
- These problems are discussed in detail in section 2.5. Also the solutions to the same are discussed in that section. This is as shown in the timing diagram in Fig. 2.7.5.

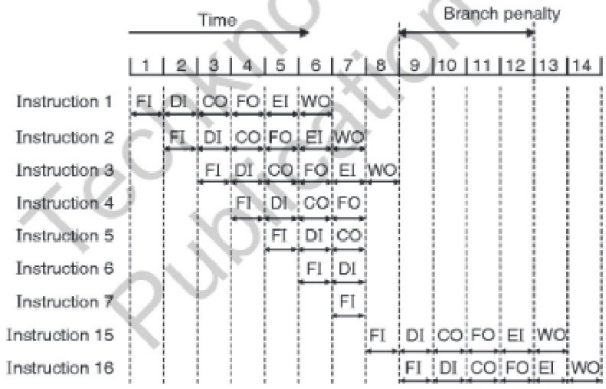


Fig. 2.7.5 : Branch in a six-stage pipeline

Ex. 2.7.1 : Draw the space-time diagram for a six-segment pipeline showing the time it takes to process eight tasks.

Soln. :

Clock cycles		1	2	3	4	5	6	7	8	9	10	11	12	13
Segment :	1	T1	T2	T3	T4	T5	T6	T7	T8	–	–	–	–	–
	2	–	T1	T2	T3	T4	T5	T6	T7	T8	–	–	–	–
	3	–	–	T1	T2	T3	T4	T5	T6	T7	T8	–	–	–
	4	–	–	–	T1	T2	T3	T4	T5	T6	T7	T8	–	–
	5	–	–	–	–	T1	T2	T3	T4	T5	T6	T7	T8	–
	6	–	–	–	–	–	T1	T2	T3	T4	T5	T6	T7	T8

It takes 13 clock cycles to process 8 tasks.

2.8 Instruction Dependencies (Pipeline/Instruction Hazards)

- Pipelining increases processor performance by increasing instruction throughput, because several instructions are overlapped in the pipeline, cycle time can be reduced, increasing the rate at which instructions execute.
- Pipeline Hazards occur when instructions read or write registers that are used by other instructions. The type of conflicts are divided into three categories :

(1) Structural Hazards (Resource Conflicts)
(2) Data Hazards (Data Dependency Conflicts)
(3) Branch difficulties (Control Hazards)

(1) Structural Hazards (Resource Conflicts)

- These hazards are caused by access to memory by two instructions at the same time. These conflicts can be slightly resolved by using separate instruction and data memories.
- Structural hazards occur when the processor's hardware is not capable of executing all the instructions in the pipeline simultaneously.
- Structural hazards within a single pipeline are rare on modern processors because the Instruction Set architecture is designed to support pipelining.

(2) Data Hazards (Data Dependency)

- This hazards arises when an instruction depends on the result of a previous instruction, but this result is not yet available.
- These are divided into four categories :

(a) RAR Hazard

- RAR Hazard occurs when two instructions both read from the same register. This hazard do not cause a problem for the processor because reading a register does not change the register's value. Therefore, two instructions that have RAR Hazard can execute on successive cycles.

Ex. 1 : Instructions having RAR Hazard

ADD r_1, r_2, r_3
SUB r_4, r_5, r_3 — Both Instructions read r_3, creating RAR

(b) RAW Hazard

This hazard occurs when an instruction reads a register that was written by a previous instruction. These are also called as **data dependencies** (or) **true dependencies**.

Ex. 2 : Instructions having RAW - Hazard

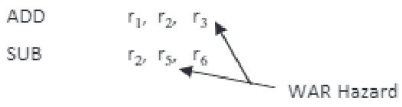
ADD r_1, r_2, r_3
SUB r_4, r_5, r_1 — Subtract reads the output of the addition creating RAW hazard

(c) WAR and WAW

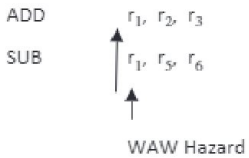
- They are also called as name dependencies.
- These hazards occurs when the output register of an instruction has been either read or written by a previous instruction.

– If the processor executes instructions in the order that they appear in the program and uses the same pipeline for all instructions, WAR and WAW hazards do not cause any problem in execution process.

Ex. 3 : Instruction having WAR Hazard



Ex. 4 : Instructions having WAW Hazard



(3) Branch Hazards

Branch instructions, particularly conditional branch instructions, create data dependencies between the branch instruction and the previous instruction, fetch stage of the pipeline. Since the branch instruction computes the address of the next instruction that the instruction fetch stage should fetch from, it consumes some time and also some time is required to flush the pipeline and fetch instructions from target location. This time wasted is called as branch penalty.

2.8.1 Methods to Resolve the Data Hazards and Advances in Pipelining

The methods used to resolve the **data** hazards are discussed in the following sub sections.

2.8.1(A) Pipeline Stalls

– The hardware inserts a special instruction called (NOP) i.e. no operation instruction known as a bubble into the flow of execution stage of pipeline to resolve the RAW hazard between two instructions.

– This method is also called as hardware interlocks.

– This approach detects the hazard and maintains the program sequence by introducing delays to resolve the data hazards (RAW).

2.8.1(B) Operand (Internal) Forwarding (or) Bypassing

– This technique uses a special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline stages.

– Example of a RAW dependency exists between two instructions, instead of transferring an ALU result into a destination register, the hardware checks the destination operand, and if it is needed as a source in the next instructions, it passes the result directly into the ALU input; bypassing the register file.

– This method requires additional hardware paths through MUX (Multiplexers).

– In this case there is a multiplexer between the two stages, wherein the data required by the next instruction is forwarded.

– Let us see this with a program example.

– If we take the following sequence of instructions,

ADD r_1, r_2, r_3
SUB r_4, r_1, r_3

– We will notice that the destination of instruction 1 is the source for instruction 2. In this case the ALU stage or the execution stage of the pipeline will forward the data to the next instruction as shown in the Fig. 2.8.1. The Fig. 2.8.1 assumes that the system is 6 stage pipelined system.

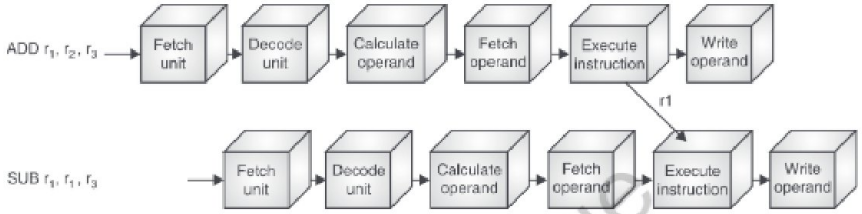


Fig. 2.8.1 : Operand forwarding mechanism in pipelining for resolving a data hazard

– As shown in Fig. 2.8.1 the data i.e. the value of register r1 is passed from the first instruction to the second instruction. Actually the value of the register r1 is updated by the write operand stage of the first instruction. But, before that the same is required by the execute stage of second instruction. Hence the value of this register is passed to the second instruction.

2.8.1(C) Dynamic Instruction Scheduling (or) Out-Of-Order (OOO) Execution

– This is another interesting and very widely used technique because of the speed up given by it. It is used in Pentium IV processor.

– Here the execution of the instructions of a program is done out-of-order i.e. not in the sequence as the instructions were written by the programmer. As and when the resources of an instruction is available, the execution of that instruction is done. If, for an instruction the resources are not available, it is kept in waiting state and the further instructions whose resources are available will be executed.

– But, you would think that this approach will have a problem. The logic implemented by the programmer will not be followed properly i.e. wrong sequence of instructions will be executed. The answer to this is that, although the instructions are executed out-of-order, but the write-back is done in order, and hence the final result of the program is in sequence.

– The compiler is designed in such a way that, while translating from high-level language to machine language program, it detects the data dependencies and re-orders the instructions.

– If necessary to delay the loading of the conflicting data it inserts no-operation instruction (NOP).

2.8.2 Handling of Branch Instructions to Resolve Control Hazards

The methods used to resolve the control hazards are discussed in the following sub sections.

2.8.2(A) Pre-Fetch Target Instruction

– One way of handling a conditional branch is to prefetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed.

– If the branch condition is successful, the pipeline continues from the branch target instructions else sequential instructions are executed.

2.8.2(B) Branch Target Buffer (BTB)

- The BTB is an associative memory included in the fetch segment of the pipeline.
- Each entry in BTB consists of the address of a previously executed branch instructions and the target instruction for that branch. It also stores the next few instructions after the branch target instructions. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction.
- If it is in the BTB, the instruction is available directly and prefetch continues from the new path.
- If the instruction is not in BTB, pipeline shifts to a new instruction streams and stores the target instruction.

2.8.2(C) Loop Buffer

- This is like a BTB, but is a high speed register file maintained by the instruction fetch segment of the pipeline. When a program loop is detected in the program, it is stored in the loop buffer.
- The program loop can be executed directly without having to access memory until loop mode is removed by final branching out.

2.8.2(D) Branch Prediction

- A pipeline with branch prediction uses additional logic to speculate the outcome of a conditional branch instruction before it is executed.
- The pipeline then begins pre-fetching the instructions stream from the predicted path.
- A correct prediction eliminates the wasted time caused by branch penalties.
- A detailed operation of branch prediction logic will be discussed in the later sections of this chapter.

2.8.2(E) Pipeline Stall (Delayed Branch)

Compiler detects branch instruction and rearranges the machine language code sequence by inserting useful instructions and rearranges the code sequence to reduce the delays incurred by Branch Instruction.

2.8.2(F) Loop Unrolling Technique

- This is a very superb solution to handle the stalls due to branching in loops.
- In this case a code which has a loop that has to be executed multiple times, will be actually stored multiple times (or unrolled) so as to remove the need of branching.
- Let us see how this can be implemented with an example.
- If there is a code for adding an array of 5 numbers, the loop can be written as shown in the code below (using processor 8086) :

Label	Instructions
	MOV AX,0000H
	MOV CX,0005H
AGAIN :	ADD AX,[SI]
	INC SI
	DEC CX
	JNZ AGAIN

This loop can be unrolled to avoid stalling as shown below :

Label	Instructions
	MOV AX,0000H
	ADD AX,[SI]
	INC SI
	ADD AX,[SI]
	INC SI
	ADD AX,[SI]
	INC SI
	ADD AX,[SI]
	INC SI
	ADD AX,[SI]

- Thus you will notice that we have unrolled the loop, and written the loop for the number of times it was to be repeated. This totally removes the pipeline stalls due to the loops.
- The **advantage** clearly seen of this method is that there is no scope for pipeline stall and hence the performance will increase.
- The major **disadvantage** of this method is that the memory required will be more as the loop has to be unrolled and stored in memory. In our example the loop was to be repeated for only 5 times, but if the loop was larger and had to be repeated for say 100 or 1000 times, the memory consumed would be very huge.

2.8.2(G) Software Scheduling or Software Pipelining

- In case of software pipelining, the iterations of a loop of the source program are continuously initiated at regular intervals, before the earlier iterations complete. Thus taking advantage of the parallelism in data path.
- It can be said that software scheduling, schedules the operations within a loop, such that an iteration of the loop can be pipelined to yield optimal performance.
- The sequence of the instructions before steady state are called as **PROLOG,** while the ones after the steady state are called as **EPILOG.**
- Let us see this with an example. Suppose the source code is

```
for(i=0;i<=n-1;i++)
a[i]=a[i]+10;
```

- When this loop is executed by a processor, the processor will do the following:

```
for(i=0;i<=n-1;i++)
{
    Load a[i];
    Add a[i]+10;
    Store a[i];
}
```

- Here you will notice that the three instructions inside the loop (in each iteration) are the same i.e. each of the three instructions have to operate on the data a[i].
- When this is converted to pipeline, it will look as shown in the Fig. 2.8.2. But the three instructions, one below the other are dependent and hence cannot be pipelined. But the instructions that are circled can be pipelined.

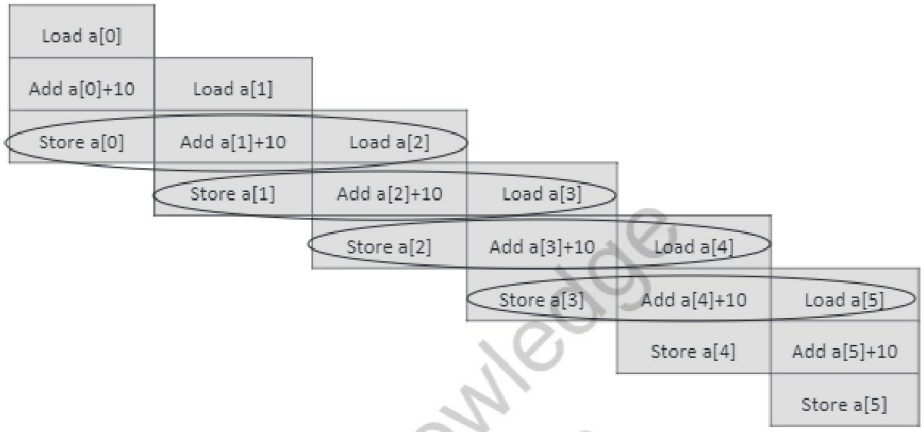


Fig. 2.8.2 : Software pipelining

- You will notice in the Fig. 2.8.2, that the three instructions that are circled are store, add and load. These instructions are always independent i.e. they have different data to operate on.
- For example in the first circle: Store a[0], Add a[1]+10 and Load a[2] is performed. Each of these instructions are using different data.
- Thus the code can be changed to the following

```
Load a[0]
Add a[0]+10
Load a[1]
for(i=2;i<=n-3;i++)
{
    Store a[i-2];
    Add a[i-1]+10;
    Load a[i]
}
Store a[n-2];
Add a[n-1]+10;
Store a[n-1]
```

- Thus, you will notice inside the for loop i.e. for each iteration, each of the three instructions are working on different data and hence are not dependent on each other and hence allowing pipelining of the three instructions without any hazards.

2.8.2(H) Trace Scheduling

- In a general pipelining, the instructions are scheduled in sequence. This results in a problem or hazard on a branching instruction as discussed in the previous section. Trace scheduling is a good solution to avoid hazard due to branching. Let us see how this can be implemented.
- In this case the probability of branch to be taken or not taken is found. Based on this the code is written with all instructions in sequence, such that no branching will be required for most of the times according to the probability calculated earlier. This code is called as the trace.
- The other blocks of code are made for less probable cases i.e. if branching is taken. Hence this trace code and the other blocks of code are written, with minimizing branches. Let us see this with a program example. Suppose the source code is:

```
if(a[i]==0)
a[i]=a[i]+10;
else a[i]=a[i]+1;
x[i]=x[i]*x[i];
```

- The number is to be squared in the above code. If the number is zero then 10 is to be squared and stored there, while if it is any other number its incremented value is to be squared and stored in the same place. When this is converted to assembly program, it will look as shown below :

Label	Instruction
	Load a[i] into say AL
	Compare AL with 0
	If not equal to zero then branch to label over
	Add AL with 10
	Branch to label next
over:	Increment AL
next:	Multiply AL with itself
	Store the result in a[i]

– This can be divided into four blocks as shown in the Fig. 2.8.3.

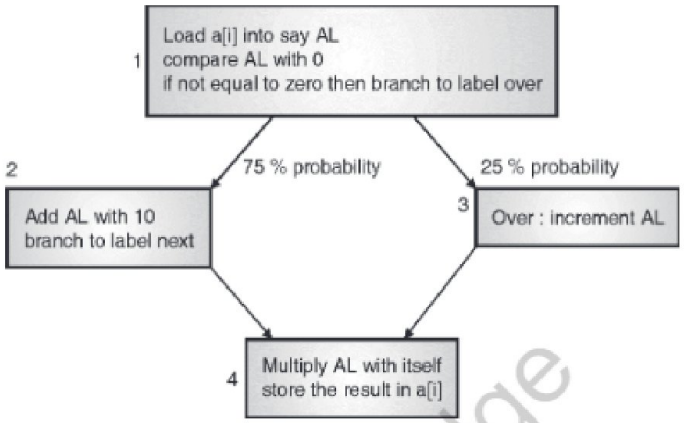


Fig. 2.8.3 : Division of Blocks of the code

– Since the path 1-2-4 is the most probable path, this will make the trace. The path 1-3-4 will be a separate block. This is shown in the Fig. 2.8.4.

Trace

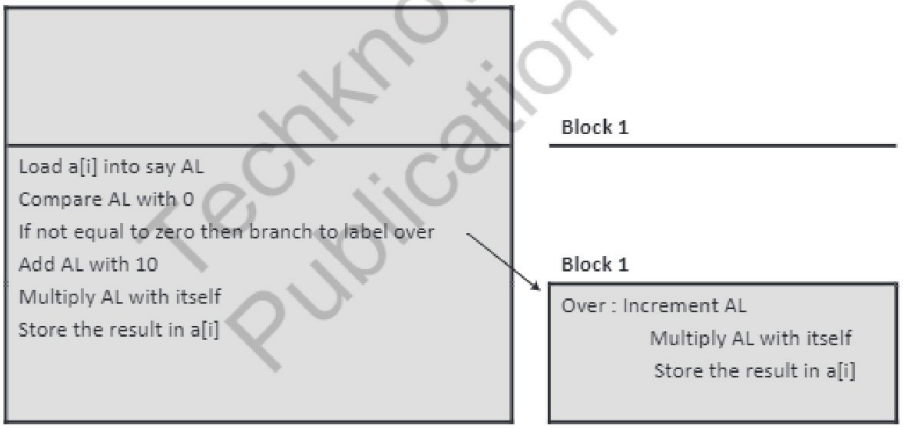


Fig. 2.8.4

– Hence in most of the cases i.e. 75% cases the trace will be executed and hence no branching will be required. Although in 25% cases we will need to branch to Block 1, but there would be only one branching and not multiple branching as required in the previous case.

2.8.2(I) Predicated Execution

– This is also a method that removes the branches. Here each instruction has a predicate that decides whether the instruction is to be executed or not. If the predicate is true then the instruction is executed, else it is not executed. The predicate is a condition bit. If the bit is '1' then the instruction is to be executed else it is not to be executed.

– Each instruction has the operands and a predicate. This removes the branching instructions and hence the stall of pipeline.

- An example of predicate instruction is given below

  CMOVZ AX,BX,CX
- This instruction copies the contents of register BX into register AX, if the predicate register CX is zero. Else the contents of BX are not copied into AX
- Predication mainly implements the if-else statement and hence the branching required for if-else is removed. It can remove the branching required for all the instructions.
- Hence we can say that predication totally removes the need of handling the branches in a pipelined system. The only disadvantage of predication is that the instruction size increases.
- Predication is used in IA-64 processors of Intel, ARM processor

2.8.2(J) Speculative Loading

- This is a process implemented in EPIC processors discussed in chapter 1. In this case the data is brought from the memory, well before it is needed.
- The compiler indicates the data that will be required in the later parts of the program and the corresponding data is brought and kept in the processor.
- This removes the latency of memory accesses required for the data to be brought from the memory.
- As the data required later is speculated and brought in advance it is called as speculative loading of data.

2.8.2(K) Register Tagging

- Register tagging is normally done by a unit called as Reservation Station (RS) in a processor.
- This reservation station is used in order to resolve the data or resource conflicts amongst the multiple instructions entering the processor.
- The operands are made to wait in the reservation station until their data dependencies are resolved.
- A tag is used to identify each reservation station, and the tag unit keeps on monitoring these reservation stations.
- This tag unit also monitors all the registers used currently or the reservation stations. This technique is called as register tagging.
- This mechanism allows to resolve the register conflicts and hence the resultant data hazards.
- The reservation stations can also be used as buffers between the various stages of pipeline in the processor. These stages can work simultaneously once the conflict is resolved.

2.8.3 Branch Prediction

Branch prediction foretells the outcome of conditional branch instructions. Excellent branch handling techniques are essential for today's and for future microprocessors. Requirements of high performance branch handling :

- An early determination of the branch outcome (the so-called branch resolution),
- Buffering of the branch target address in a BTAC (Branch Target Address Cache),
- An excellent branch predictor (i.e. branch prediction technique) and speculative execution mechanism,
- Often another branch is predicted while a previous branch is still unresolved, so the processor must be able to pursue two or more speculation levels, and
- An efficient rerolling mechanism when a branch is mispredicted (minimizing the branch misprediction penalty).

2.8.3(A) Misprediction Penalty

The performance of branch prediction depends on the prediction accuracy and the cost of misprediction. Misprediction penalty depends on many organizational features:

- The pipeline length (favouring shorter pipelines over longer pipelines),
- The overall organization of the pipeline,
- The fact if misspeculated, instructions can be removed from internal buffers, or have to be executed and can only be removed in the retire stage,
- The number of speculative instructions in the instruction window or the reorder buffer. Typically only a limited number of instructions can be removed each cycle.
- Misprediction is expensive (11 or more cycles in the Pentium II).

2.8.3(B) Static Branch Prediction

- Static Branch Prediction predicts always the same direction for the same branch during the whole program execution.
- It comprises hardware-fixed prediction and compiler-directed prediction.
- Simple hardware-fixed direction mechanisms can be :
  (a) Predict always not taken
  (b) Predict always taken
  (c) Backward branch predict to be taken, forward branch predict not to be taken
- Sometimes a bit in the branch opcode allows the compiler to decide the prediction direction.

2.8.3(C) Branch-Target Buffer or Branch-Target Address Cache

The Branch Target Buffer (BTB) or Branch-Target Address Cache (BTAC) stores branch and jump addresses, their target addresses, and optionally prediction information. The BTB is accessed during the IF stage.

Branch address	Target address	Prediction bits
...	...	...

Fig. 2.8.5

2.8.3(D) Dynamic Branch Prediction

The hardware influences the prediction while execution proceeds. Prediction is decided on the computation history of the program. During the start-up phase of the program execution, where a static branch prediction might be less effective, the history information is gathered and dynamic branch prediction gets more effective. In general, dynamic branch prediction gives better results than static branch prediction, but at the cost of increased hardware complexity.

2.8.3(E) One-bit Dynamic Branch Predictor

- A one-bit predictor correctly predicts a branch at the end of loop iteration, as long as the loop does not exit.

  In nested loops, a one-bit prediction scheme will cause two mispredictions for the inner loop :

- One at the end of the loop, when the iteration exits the loop instead of looping again, and one when executing the first loop iteration, when it predicts exit instead of looping.

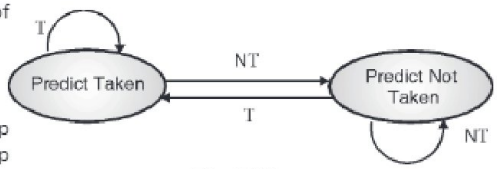


Fig. 2.8.6

- Such a double misprediction in nested loops is avoided by a two-bit predictor scheme.

2.8.3(F) Two-bit Prediction

A prediction must miss twice before it is changed when a two-bit prediction scheme is applied.

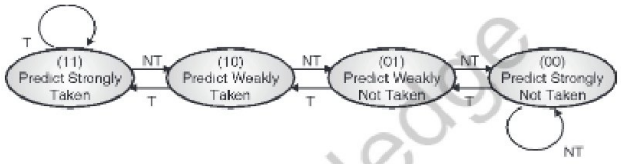


Fig. 2.8.7

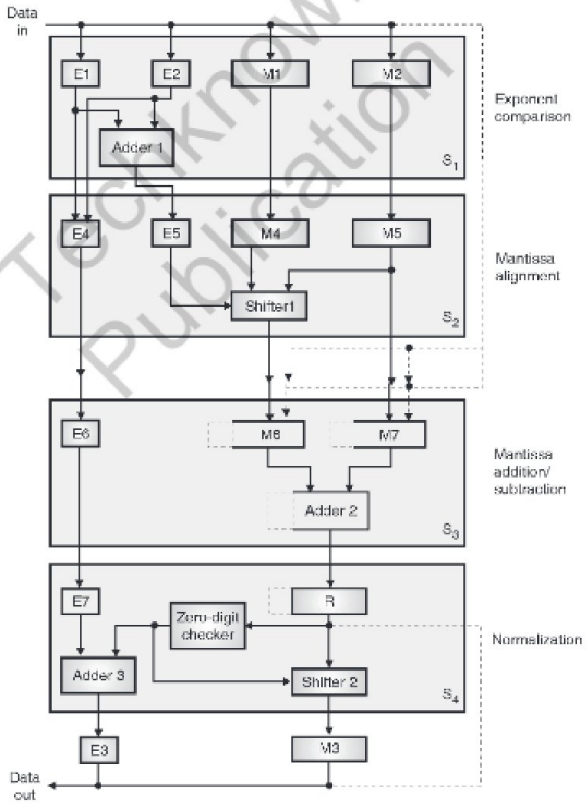


Fig. 2.8.8 : Pipelined version of the FP add unit of the IBM System/360 computer

2.9 Classification of Pipelining

There are various ways of classifying the pipelining mechanisms. They are

1. Linear and Non-linear pipelining
2. Classification of pipelining according to the level of processing, namely:
   a) Instruction level pipelining or Instruction pipeline
   b) ALU level pipelining or Arithmetic pipeline
   c) Processor level pipelining
3. Uni-function pipeline and Multi-function pipeline
4. Static and Dynamic Pipeline
5. Scalar and Vector Pipeline

- The first method of classification of pipelining is just seen in the previous sections i.e. sections 2.3 and 2.4. We have seen the operation of linear and non-linear pipelining in these sections.
- The second method is also seen in the sections i.e. 2.5 and 2.7. We have seen the instruction pipelining and arithmetic pipelining. We will now see the processor pipelining. In case of processor level pipelining, we have multiple processors each doing a part of the operation of a huge task. The next processor in the pipeline continue the further operation and so on. We will be seeing these types of pipelining in the further chapters in this book
- The remaining three methods of classification of pipelining are discussed in the following sub sections.

2.9.1 Uni-function and Multi-function Pipeline

- A uni-function pipeline is a dedicated pipeline i.e. the pipeline will always do the same operation. The operations of a pipeline cannot be changed
- A multi-function pipeline is one, that can perform multiple or different operations simultaneously or at different times. For example a multi function arithmetic pipeline may perform floating point addition at one time, multiplication at another time and so on.

2.9.2 Static and Dynamic Pipeline

In case of static pipeline, the functions of the pipeline stages are not configurable during run time. They can be configured only during the design time. While, in case of dynamic pipelining, the pipeline stages can dynamically change their operation i.e. they can be configured to change their function as and when required according to the operation to be performed

2.9.3 Scalar and Vector Pipeline

In case of static pipeline, the functions of the pipeline stages are not configurable during run time. They can be configured only during the design time. While, in case of dynamic pipelining, the pipeline stages can dynamically change their operation i.e. they can be configured to change their function as and when required according to the operation to be performed.

Review Questions

Q. 1 Explain various types of data Hazards observed in pipeline processor. How these Hazards could be detected and resolved ?

Q. 2 Compare between :

(i) Static and Dynamic pipeline

(ii) Unifunctional and Multifunctional pipeline.

Q. 3 Discuss With suitable examples, the following techniques to resolve hazards with respect to pipelining processors.

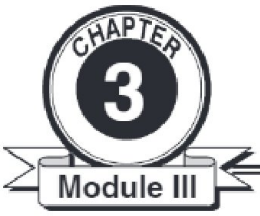
(i) Trace scheduling .

(ii) Software scheduling.

Q. 4 Write a detail note on register tagging technique

Q. 5 Discuss various pipeline hazards. Give Hazard detection and resolution techniques

❑❑❑



Parallel Programming Platforms

Syllabus

Parallel Programming Platforms: Implicit Parallelism : Trends in Microprocessor & Architectures, Limitations of Memory System Performance, Dichotomy of Parallel, Computing Platforms, Physical Organization of Parallel, Platforms, Communication Costs in Parallel Machines.

3.1 Parallel Programming Platforms : Implicit Parallelism

3.1.1 Pipelining Execution

- A processor has many resources like the ALU, buses, registers, etc. An attempt to utilize all these attributes to their fullest or continuously can be achieved by pipelining. In a pipelined system the instructions flow through the processor as if the processor was a pipe. The instructions move from one stage to another to accomplish the assigned operation. Hence at most of the times each unit of the processor is busy handling one or the other instruction, making the attribute of the processor being used continuously.
- In a non-pipelined system, the processor fetches an instruction from memory, decodes it to determine what the instruction was, read the instructions inputs from the register file, performs the computation required by the instruction and writes the result back into the register file. This approach is also called as unpipelined approach.
- The problem with this approach is that, the hardware needed to perform each of these steps (Instruction fetch, instruction decode, register read, instruction execution and register write-back) is different and most of the hardware is idle at any given moment. Waiting for the other parts of the processor to complete their part of executing an instruction.
- Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions.

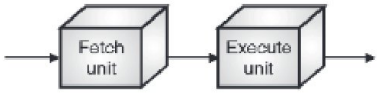


Fig. 3.1.1 : Two stage pipelining

- Two stage pipelining includes two stages i.e. Fetch instruction and Execute instruction.
- These two operations are performed for one instruction and the next instruction overlapping i.e. when the first instruction is being executed the next is fetched and when this instruction is executed the next is fetched and so on.
- This method of execution the instructions in pipeline speeds up the processor operation.
- This also makes sure that all the units of the processor are busy operating and none of them is starving.
- Thus with the help of pipelining the operation speed of the processor increases i.e. more the number of pipeline stages, faster becomes the processor, but complex in design.

- A simple two stage instruction pipeline stage is as shown in Fig. 3.1.1.
- Pipelining is implemented in almost all the modern processors.

3.1.2 Superscalar Execution

Superscalar processors are those processors that have multiple execution units. Hence these processors can execute the independent instructions simultaneously and hence with the help of this parallelism it increases the speed of the processor. It has been seen that the number of independent consecutive instructions is around 2 to 5. Hence the instruction issue degree in a superscalar processor is restricted from 2 to 5.

3.1.2(A) Pipelining in Superscalar Processors

- The pipelining is the most important representation of demonstrating the speed increase by the superscalar feature of the processors. Fig. 3.1.2 shows the timing diagram of a two issue superscalar and 4-stage pipeline processor.

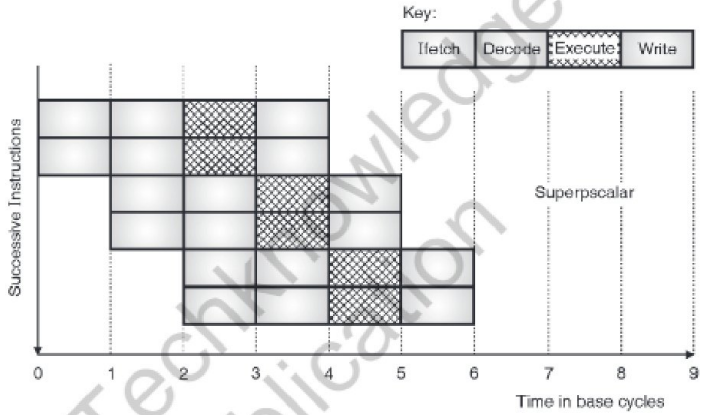


Fig. 3.1.2 : Timing diagram of the superscalar processor with degree m=2

- Hence to implement multiple operations simultaneously, we need to have multiple execution units to execute each instruction independently. Fig. 3.1.3 shows a block diagram of a typical superscalar processor.
- As shown in the Fig. 3.1.2, the instruction fetch unit fetches m instructions (where m is the degree of the processor superscalar). The ID (instruction decode) and rename unit, decodes the instruction and then by the use of register renaming avoids instruction dependency. The instruction window then takes the decoded instructions and based on some pairability rules, issues them to the respective execution units.
- The instructions once executed move to the Retire and write back unit, wherein the instructions retire and the result is written back to the corresponding destination.

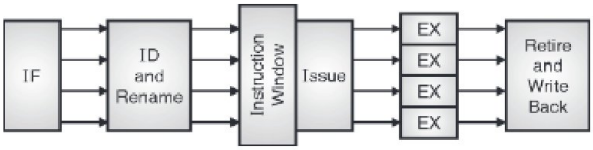


Fig. 3.1.3 : Block diagram of a typical superscalar processor

– Most of the processors to be studied in this subject are superscalar processors. Hence we will see the detailed concept of superscalar with those processors. In chapter 3, we will see the two issue superscalar processor i.e. Pentium processor. Further chapters will also explain the other superscalar processors like Pentium II, Pentium III, Pentium IV, Itanium I, Itanium II, Sun SPARC etc.

3.1.3 Very Long Instruction Word Processors (VLIW)

VLIW (Very Long Instruction Word) has around 256 to 1024 bits per instruction. It is a combination of horizontal micro-coding and superscalar. It also has a large register file.

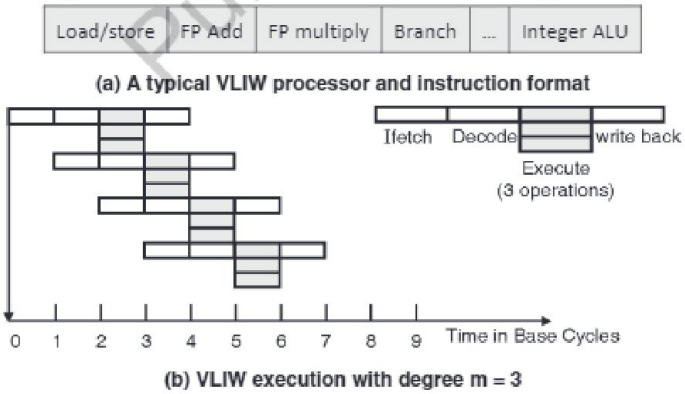
3.1.3(A) Horizontal vs. Vertical Micro-coding

VLIW processor uses horizontal micro-coding. Let us understand how is the horizontal micro-coding different from vertical micro-coding.

Sr. No.	Vertical Programming	Horizontal Programming
1.	Decodes the instructions one by one as they are available.	All the instructions are given simultaneously.
2.	This is comparatively slower.	This is comparatively faster.
3.	Less no. of bits for each instruction are used.	More no. of bits for each instruction.
4.	Low degree of parallelism.	High degree of parallelism .

3.1.3(B) VLIW Instruction and Pipelining

The instruction format and the pipelining of VLIW processors is as shown in Fig. 3.1.4. As seen in Fig. 3.1.4 (a), the VLIW instruction consists of many operations in a single instruction. Some operations like FP Add (Floating Point Addition), FP Multiply (Floating Point Multiplication), Branching instruction, Integer ALU operation etc. Hence on issuing one instruction itself many operations are done simultaneously during the execute cycle of the pipelining. This multiple operations in the execution stage of the VLIW processors is shown in the Fig. 3.1.4 (b).



(a) A typical VLIW processor and instruction format

(b) VLIW execution with degree m = 3

Fig. 3.1.4 : Instruction structure and pipelining of VLIW processors

3.1.4 VLIW Processor Structure

– To perform multiple operations in a single execution stage, we need to have separate units to perform each of these operations. To perform the different operations like floating point add, multiply, branching, integer ALU etc, we need separate units for each of these operations this is shown in the Fig. 3.1.5. Fig. 3.1.5 shows the architecture of a typical VLIW processor with all the above mentioned units that can follow the instruction format and pipelining given in the Fig. 3.1.4.

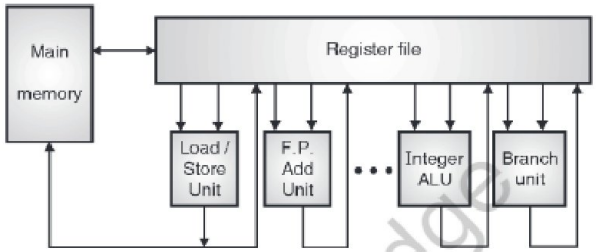


Fig. 3.1.5 : Typical VLIW architecture

– The special characteristic of a traditional VLIW Processor is that the instruction has multiple operations. The multiple operations given in the instruction are independent operations and have no flow dependences between these operations.

3.1.4(A) Advantages, Disadvantages and Applications

1. Advantages

(a) No runtime dependence checks against previously or simultaneously issued operations.

(b) No runtime scheduling decisions.

(c) No need for register renaming.

2. Disadvantages

(a) No tolerance for any difference in the types of functional units.

(b) No object code compatibility.

(c) In Superscalar and VLIW processors, more than a single instruction can be issued to the execution units per cycle.

(d) Superscalar machines are able to dynamically issue multiple instructions each clock cycle from a conventional linear instruction stream.

(e) VLIW processors use a long instruction word that contains a usually fixed number of instructions that are fetched, decoded, issued, and executed synchronously. Hence, Superscalar has dynamic issue, while VLIW has static issue.

3. Application

It is suitable for Digital signal processing.

3.2 Trends in Microprocessor and Architectures

Before the conventional processors, we had processors made using vacuum tubes, transistors and so on.

3.2.1 Mechanical Era (1600s-1940s)

- **Wilhelm Schickhard :** In 1623, Wilhelm Schickhard made an automatic mechanical system to add, subtract, multiply and divide
- **Blaise Pascal:** In 1642, Blaise Pascal mass produced first working machine (50 copies) but that could only add and subtract.
- **Gottfried Leibniz :** In 1673, Gottfried Leibniz improved on Pascal's machine to add, subtract, multiply and divide.
- **Charles Babbage** is called as the "Father of modern computer". He designed a concept of computer that had the Modern structure i.e. I/O, storage (Memory) and ALU. He also developed a machine that could add in 1 second, multiply in 1 minute
- **Herman Hollerith** in 1889 made a Tabulating Machine Company, to tabulate the census for his country.
- **Konrad Zuse** in 1938 built the first working mechanical computer, named as Z1. It was a Binary machine

3.2.2 Electronic Era

This era had quite a few generations based on the components used to make the CPU.

3.2.2(A) Generation 1 Vacuum Tubes (1945 - 1958)

Vacuum tubes were used as basic electronic component in this generation. This generation had two remarkable milestones in the field of computing. They were ENIAC and Von Neumann's system.

- **ENIAC (Electronic Numerical Integrator And Computer) :** It is regarded as the first electronic computer. It used decimal number system.
- **IAS (Institute for Advanced Studies) :** This was another computer made using vacuum tubes. This architecture was also known as the "Von Neumann" architecture.

3.2.2(B) Generation 2 Transistors (1958 - 1964)

- With the advent of semiconductors in electronic industry, transistors replaced vacuum tubes.
- The major advantage of transistors over the vacuum tubes were that the transistors are small, cheaper and have less heat dissipation

3.2.2(C) Generation 3 IC (1964 onwards)

- This was the generation of Integrated Circuits.
- Many transistors can be integrated on a single IC, and hence reduce the size of the CPU.
- Earlier in this generation there were ICs available to perform separate operations required in a CPU. Then all these functions came on a single IC to give a single chip CPU.
- This generation saw many processors with special features to increase the speed as the number density of components on an IC went on increasing.
- A lot of progress has been done in the last four decades in parallel computing. The progress of semiconductors has also supported these features.

- Parallel processing still needs some progress in vector and array processing. The programming of these systems is still quite complicated. Also based on the image processing and signal processing there is required to be function specific parallel processors. Medical field is also requiring new methods of calculations with fast processing where parallel computing still has scope to improve.

3.3 Limitations of Memory System Performance

- Effective performance of a computer program on a computer relies not just on the speed of processor but also on the ability of the memory system to feed the data to the processor.
- Memory system performance is largely captured by two parameters, **latency and bandwidth.**
- Latency is the time from the issue of a memory request to the time the data is available at the processor. Latency does not provide complete information about performance of the memory system.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

To improve the memory latency, cache can be used,

- Caches are small and fast memory elements between the CPU and main memory.
- Cache memory is used to enhance the access speed of any storage devices.
- The properties of cache are low-latency and high bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache. As if any application is already fetched by the processor then it is saved in cache and next time it can be fetched by the cache.
- Cache hit - the fraction of data references satisfied by the cache is called the cache hit ratio and otherwise it is called cache miss ratio.
- Now we will discuss bandwidth as one of the parameter that affects memory performance.

3.3.1 Impact of Memory Bandwidth

- Memory Bandwidth is the rate at which data can be read from or stored into memory by a processor.
- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.

3.3.2 Hiding Memory Latency Techniques

Latency means extra time or the time delay. The extra time is required to access the memory because the memories are comparatively slower than the processors is one major cause of latency. Also in a multiprocessor system many a times we have shared memory which has more latency. Hence it is necessary to either reduce this latency or atleast hide it from the processor. There are various latency hiding mechanisms we will be studying in this section. They are as listed below :

1. Pre-fetching technique i.e. bringing the instructions and the data before they are actually needed. Hence reducing the memory access time, or hiding the latency from the processors.
2. Multiple coherent caches that will reduce the cache misses.
3. Relaxed memory consistency models that allow buffering as well as pipelined access for memory accesses.
4. Multiple context support processors that allow switching from one context to another whenever the first one has a long latency. This one is nothing but multithreading.

3.3.2(A) Pre-fetching Techniques

This method of data hiding brings the data and instructions before they are needed by the execution unit of the processor. But for this technique it is necessary to have a knowledge of the data and instructions that will be expected by the processor. Pre-fetching can be classified in two manners. The first classification is bounded and non-bounded pre-fetching. Another classification is on the control of this pre-fetching i.e. whether the control is hardware or software. We will see all these methods in the following sub sections.

1. Bounded and Non-bounded Pre-fetching

- In this case as the name says there is some binding between the pre-fetching. For example, if an instruction accesses a data from a memory location whose address is given by a register pointer. And this register pointer is initialized in one of the instruction. This address or pointer is also to be pre-fetched.
- Here, if another processor modifies the memory location that initializes the reference or pointer during the delay between the pre-fetch of this address and the actual initialization of the reference (i.e. register pointer), then the pointer reference will be initialized with wrong value and hence wrong data will be accessed.
- This will also be true if the data that may not be a reference or pointer, but is pre-fetched before the actual execution of that instruction. Similar to the above case if that data is modified by another processor before the actual execution of this data then there will be a wrong operation i.e. a stale data (old data) will be considered instead of the most updated version of the data.
- A major disadvantage in such cases i.e. bounded pre-fetching, is getting a stale data.
- In this case also the data is brought into the processor before the actual execution of the instruction i.e. the data is pre-fetched. But another major care taken here to avoid the disadvantage of the bounded pre-fetching is that the cache coherency protocol keeps an eye on this data.
- Thus if another processor alters the value of this memory location which is pre-fetched, the cache coherency protocol monitors this and also updates the pre-fetched data by the processor. This totally removes the problem of stale data being pre-fetched.

2. Hardware and Software Controlled Pre-fetching

- Hardware controlled pre-fetch is automatic hardware system that fetches the instructions and data automatically into the cache based on the principles of locality of reference.
- But this type of pre-fetch has the limitation i.e. the data and instructions fetched are wrong in case of a branching.
- In case of software controlled pre-fetching there are instructions that are given to the processor to fetch the data in advance, so as to keep the buses occupied during the execution of ALU instructions.
- We know that most of the processors have pipelining implemented in them. In such cases, many a times the buses of the processor are free while the internal operations are carried out by the ALU and the internal cache. As we know principles of locality of reference assure 90% of the accesses from the cache memory. This makes the system bus free for most of the time, hence if the instructions are given for pre-fetching the data that will be required later, the operations will be faster.
- An important thing for software pre-fetching is that the processor must have instructions to support the instructions for pre-fetching. There must be special instructions in the instruction set of such processors, that allow pre-fetching. This can be said to be a disadvantage of software controlled pre-fetching i.e. the extra instructions and the circuit implementation for the same inside the processor or the overhead included in adding these instructions.

- The most important advantage of any type of pre-fetching is that the code is always available in the pipeline and hence the stages of the pipeline never starves. The experiments have been carried out that show that there is a drastic improvement in the performance when the pre-fetching is done compared with respect to when no pre-fetching is done.

3.3.2(B) Multiple Coherent Caches

- In case when we have a single cache for each processor, then the maintenance of coherency is very complicated. Hence many multiprocessor supporting processors do not have any cache. But there is another better solution that allows the use of cache and still give a higher performance.
- The multiple coherent system says cache the private and shared data. Since there are multiple processors operating on a task divided into multiple parts, they may access the same data. If the data is once cached for one processor, since this shared data is cached in all the caches, the processor need not face the miss. Hence no latency for the access of the shared data.
- We can have the shared and private data cacheable. This is seen to increase the performance substantially. The main reason being that this reduces the number of read and write misses in case of the shared data being cached. The statistical data shows the performance increase.
- One such implementation is called as Scalable Coherent Interface (SCI) specified as a standard by IEEE. This cache coherency model protocol maintains distributed directories in the individual processor cache module.
- Fig. 3.3.1 shows the implementation of SCI protocol.

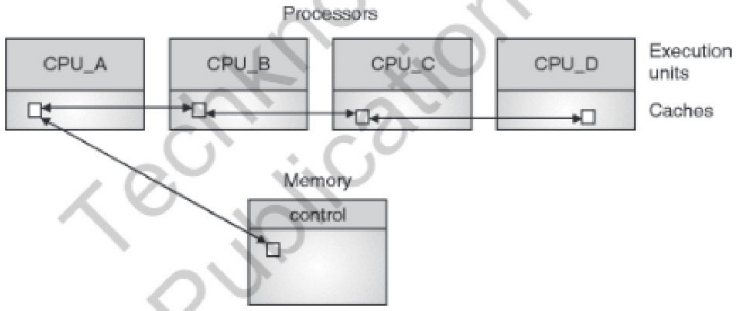


Fig. 3.3.1 : SCI cache coherence protocol

- In each processor cache there is a directory for the local or private data and a separate directory for the shared data. There are additional bits in the tag for each address to indicate the CPU. Hence, one can say that there is a doubly linked list maintained.
- Whenever a data is required by suppose CPU_A in the Fig. 3.3.1, it checks for the availability of this data in the caches of the other three processors besides its own cache. If it doesn't gets this data then it accesses it from the main memory. All other processors maintain the presence of this data in their coherent cache directories. Now if another processor requires the same data, it need not access the memory, instead it will be available from the cache as indicated in the coherent cache directory.

3.3.2(C) Relaxed Memory Consistency

There are some memory consistency models that define the order in which the memory operations from multiple processors to be carried out. We will see some of these consistency models in this section.

1. Sequential Memory Consistency

- **Sequential Memory consistency** guarantees that the store, FLUSH and atomic load-store instructions of all processors appear to be executed by memory serially in a single order called the memory order.
- Also, the sequence of store, FLUSH and atomic load-store instructions in the memory order for a given processor is identical to the sequence in which they were issued by the processor.
- Fig. 3.3.2 shows the ordering constraints for this model.

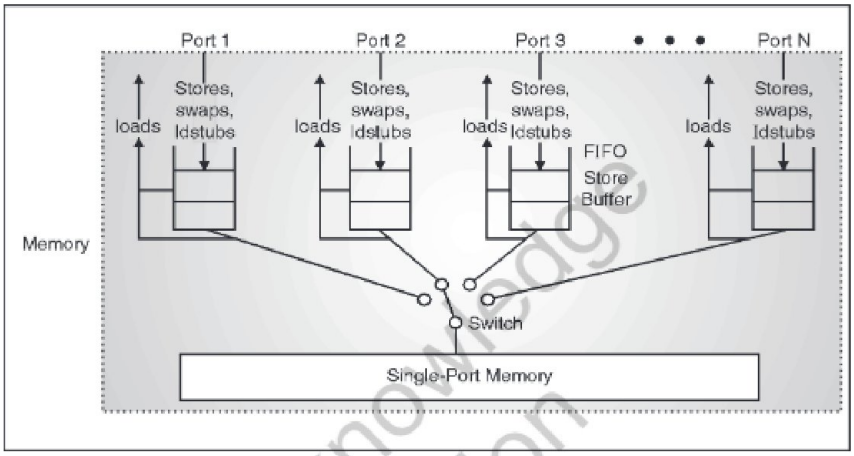


Fig. 3.3.2 : Sequential model of memory

- Stores, FLUSHes, and atomic load-stores issued by a processor are placed in its dedicated FIFO Store Buffer. The order in which memory executes the operations for a given processor is the same as the order in which the processor issued them. The order of these operations corresponds to the order in which the switch is thrown from one port to another.
- To maintain consistency, a load by a processor first checks its Store Buffer to see if it contains a store to the same location (atomic load-stores do not need to be checked because they block the processor). If it does, then the load returns the value of the most recent such store from the buffer; else the load goes directly to memory.
- Since all loads do not go to memory, loads in general do not appear in the memory order.
- A processor is blocked from issuing further memory operations until the last load returns a value.
- An atomic load-store behaves like both a load and a store. They are placed in the Store Buffer like a store, and it blocks the processor like a load. Hence, load does not need to check for atomic load-stores in the Store Buffer.

2. Processor Consistency

- In 1989, Goodman proposed a new memory consistency model called as processor consistency. This model proposes that all the memory write operations issued by individual processors must be in the order of the program.
- We know, in multiprocessor system, the different parts of a program are executed by different parts of the processor. The part of the program that is executed by an individual processor has its memory stores in order of the program but for the different processors the memory stores may not be in order of the program. Also the order of reads from each processor is not restricted unless the data is shared by another processor.

- There are two conditions with respect to other processors that ensure processor consistency :
  1. Before a read is allowed for other processor, all earlier read operations must be performed.
  2. Before a write is allowed for other processor, all earlier read and write operations must be performed.
- This guarantees that a write that appears previously in the program will be executed first i.e. in order.
- Let us see the comparison of these consistency memory modes

Sr. No.	Sequential Consistency	Processor Consistency
1.	A multiprocess is said to be sequentially consistent if the result of the execution is as if all the operations are sequentially executed.	The write operations appeared from different processes must appear in the same sequence as they occur.
2.	It is a weaker model of correctness because the real time of ordering of events need not be maintained	This system is said to be stronger than the coherence system but weaker than the sequential consistency model
3.	It is possible to list all the actions by all processes in the system in one linear order	Interleaving of write issued by two different processes is viewed differently by each process, as long as coherency is maintained.

3. Release Consistency

- This is the most relaxed memory consistency model. It was introduced in 1990 by Gharachorloo and his team. This consistency model uses two terms called as acquire and release.
- An acquire operation is a read operation while a release operation is a write operation.
- There are three conditions to be satisfied for the processors to ensure the release consistency :
  1. All previous acquire operations must be completed before any acquire or release operation.
  2. All previous acquire and release operations must be performed before any release operation.
  3. In some special cases wherein ordering restrictions are there, processor consistency is to be used.
- The **advantage** of relaxed consistency model is that the performance increases by hiding the write latency while the disadvantage is hardware complexity and complex programming. Infact the combination of various latency hiding techniques result in much better performance.

3.4 Dichotomy of Parallel Computing Platforms

Parallel program specify **concurrency** and **interaction** between concurrent subtasks. Sometimes Concurrency is also referred to as the **control structure** and interaction referred as **communication model**.

3.4.1 Control Structure of Parallel Platform

- Parallelism can be expressed at various levels of granularity, from instruction level to processes.
- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- In this, system is divided into multiple smaller parts and is granular in structure.
- Granularity is used to describe about the division of a task into numbers of a smaller subtasks.
- Granularity in parallel program is considered as different level. Example program level and instruction level.

- There are two types :
  1. **Fine grained :** When system is divided into large networks of small part is called fine grained.
  2. **Course grained :** System is divided into smaller numbers of large parts is called as course grained.
- Processing units in parallel computers either operate under the **centralized control of a single control unit** or **work independently**.
- Two types of models :
  1. SIMD model (Single Instruction Stream, Multiple Data Stream)
  2. MIMD model (Multiple Instruction Stream, Multiple Data Stream)

3.4.1(A) SIMD Architecture

- For a operation to be performed on a vector or array data, using multicomputer makes many redundant units like memory, instruction decoder, address decoder, control unit, ALU.
- This redundancy is avoided in case of a SIMD array. Here only one control unit and one program memory is connected and all the other related units like instruction register, instruction decoder and address decoder for instructions are also only one set and hence more resources can be used for the actual computational elements
- This is shown in Fig. 3.4.1.

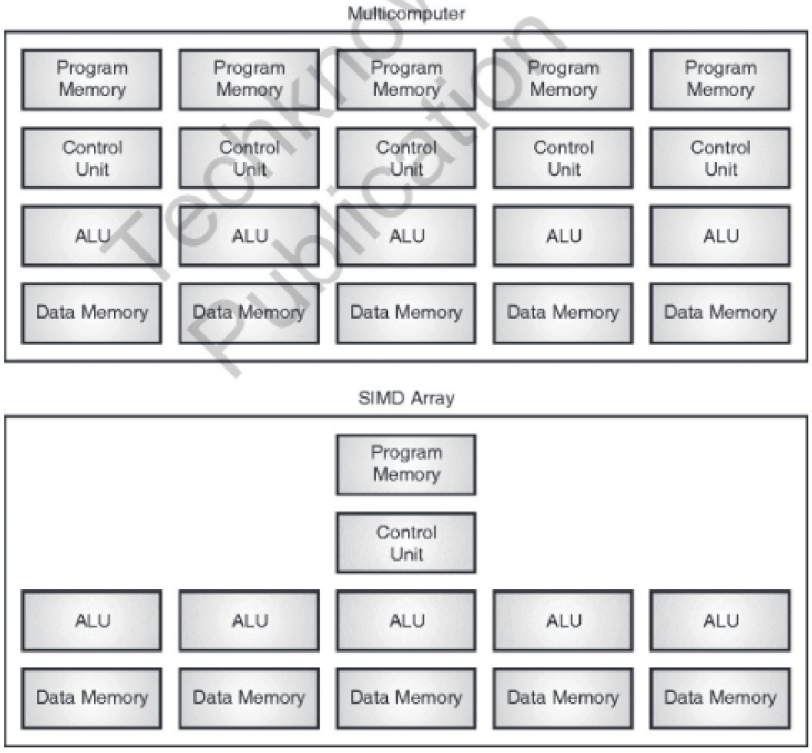


Fig. 3.4.1 : Multicomputer vs. SIMD architecture

- Thus the concept of SIMD design is to provide the maximum number of computational elements. This count has been up to a maximum of 256K in one machine, although 16K is more typical.
- This concept can make use of very simple processors, each connected with a small amount of data memory and in this way have huge number of elements.
- These can be machines with one-bit ALUs. Hence if an 8-bit addition is required, we will use one ALU and hence require 8 pulses. But since there are a huge number of such ALUs, for example say 256 ALUs, then 256, 8-bit data addition can go on simultaneously, and all the 256 sets of data can be added in 8 clock pulses. Thus giving an average of 32-byte operations in one clock pulse.
- In some SIMD processors, even for single data, parallelism is achieved by dividing the huge data into smaller parts. The speciality of SIMD processors is that if there are 256 computational units in a SIMD processor the performance will be 256 times that of a single processor. But this may not be true for a 256 processor system.

3.4.1(B) MIMD Architecture

- This is a complete parallel processing example. Here each processor has its own control unit; each processing element can execute different instructions on set of different data.
- Examples of this kind of systems are SMPs (Symmetric Multiprocessors), clusters and NUMA (Non-Uniform Memory Access). Fig. 3.4.2 shows the structure of such a system.

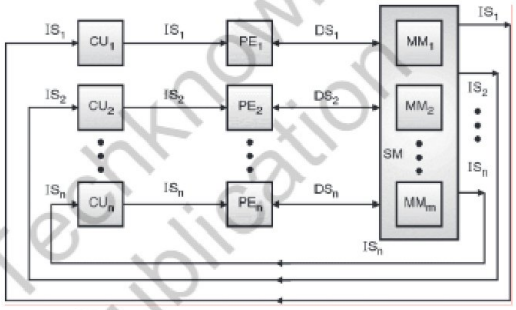


Fig. 3.4.2 : MIMD architecture

Table 3.4.1 : Comparison between SIMD and MIMD

SIMD	MIMD
It is also called as Array processor.	It is also called as multiprocessor.
Here, single stream of instruction is fetched.	Here, multiple streams of instructions are fetched.
The instruction stream is fetched by shared memory.	The instruction streams are fetched by control unit.
Here, instruction is broadcasted to multiple processing elements.	Here, instructions streams are decoded to get multiple decoded instruction streams.
SIMD computers require less hardware than MIMD.	Requires more hardware.

3.4.2 Communication Model of Parallel Platform

- There are two primary forms of data exchange between parallel task.
- The basic forms are :
  1. Shared address space approach
  2. Message passing approach

3.4.2(A) Shared-Address Space Platform

- Platforms that provides shared data space are called shared-data machines or multiprocessors.
- This platform provides common data space accessible by all processors included in the system. Part or all of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared address space.

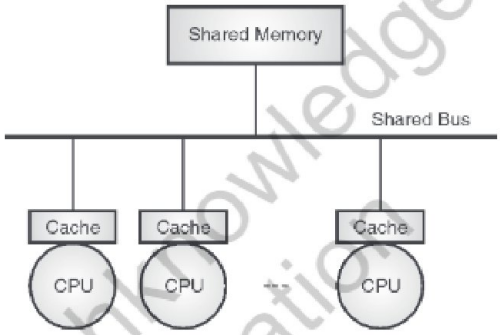


Fig. 3.4.3

Classification of shared address platform :

- The shared address platform is classified as,
  1. NUMA (Non-Uniform Memory Access)
  2. UMA (Uniform Memory Access)
- If the time taken by a processor to access any memory word in the system is identical, the platform is classified as a Uniform Memory Access (UMA) else, a Non-Uniform Memory Access (NUMA) machine.
- The main difference between the NUMA and UMA is the location of the memory.

1. NUMA (Non-Uniform Memory Access)

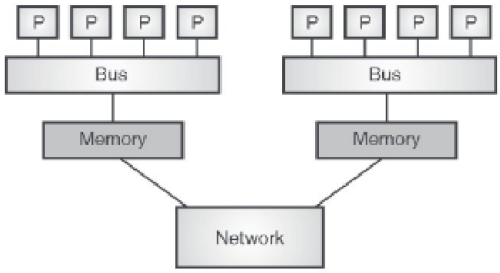


Fig. 3.4.4 : NUMA architecture

- In the NUMA shared memory architecture, each processor has its own local memory module that it can access directly. At the same time, it can also access any memory module belonging to another processor using shared bus or any type of interconnect.
- These systems have a shared logical address space, but physical memory is distributed among CPUs, so that access time to data depends on data position.
- A processor has direct path to the block of its local memory attached to it.
- All processors can see all memory.
- Access to the memory of other processors is slower.

2. UMA (Uniform Memory Access)

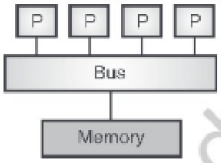


Fig. 3.4.5 : UMA Architecture

- All processors share a unique centralized primary memory, so each CPU has the same memory access time.
- Each processor gets equal priority to access the main memory of the machine.
- These systems are also called as Symmetric Shared - Memory Multiprocessors (SMP).

3.4.2(B) Message – Passing Platform

- Platforms that support messaging are also called message passing platforms or multicomputer.
- These platforms exchange messaging for sharing data.
- This model allows multiple processes to read and write data to the message queue without being connected to each other.
- Messages are stored on the queue until their recipient retrieves them.

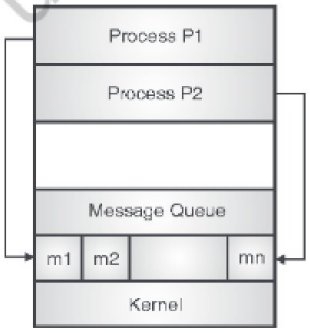


Fig. 3.4.6 : Message Passing Model

- In Fig. 3.4.6 both the processes P1 and P2 can access the message queue and store and retrieve data.
- These platforms comprise of a set of processors and their own exclusive memory.

3.5 Physical Organization of Parallel Platforms

- Here we are going to discuss the physical architecture of parallel computers.
- Here conventional architecture refers to the uni-processor system. Some of the parallelism features can definitely be implemented on a single processor to improve the speed, but there is a limitation to the same.

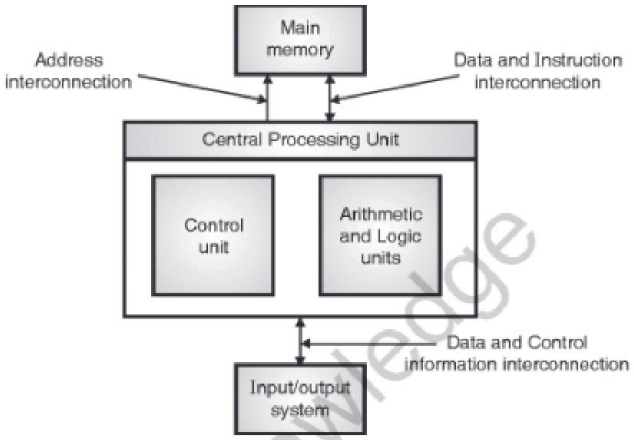


Fig. 3.5.1 : Von Neumann Architecture of a computer

- Here conventional architecture refers to the uni-processor system. Some of the parallelism features can definitely be implemented on a single processor to improve the speed, but there is a limitation to the same. Let us discuss certain basics of single processor. The architecture of processors began with the IAS's Von Neumann Computer. Fig. 3.5.1 shows the architecture of the Von Neumann architecture.
- This system has three units CPU, Memory and I/O devices. The CPU has two units Arithmetic Unit and Control unit.

3.5.1 Evolution of Parallelism

- The features in processors were slowly developed to give parallelism and hence fast processing. Fig. 3.5.2 shows this architectural evolution tree.
- You will notice in the Fig. 3.5.2, that the first step towards parallelism was Look ahead i.e. overlap of fetch and execute and the concept of parallelism in functions. The parallelism in functions was implemented by two mechanisms as seen in the Fig. 3.5.2, pipelining and multiple functional units. In the second mechanism, multiple functional units were implemented that would operate simultaneously.
- Vector instructions are a kind of huge array of data that has a common operation to be performed on them. For vector instructions, initially pipeline processors were used controlled by software looping. Later explicit processors were made for them. There were again two variations in the vector processing, memory to memory and register to register. The first one makes use of memory to store the operands i.e. the operands are loaded and stored in memory; while the second one uses registers to store the operands.
- The register to register architecture further evolved into two types of processors namely Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD).

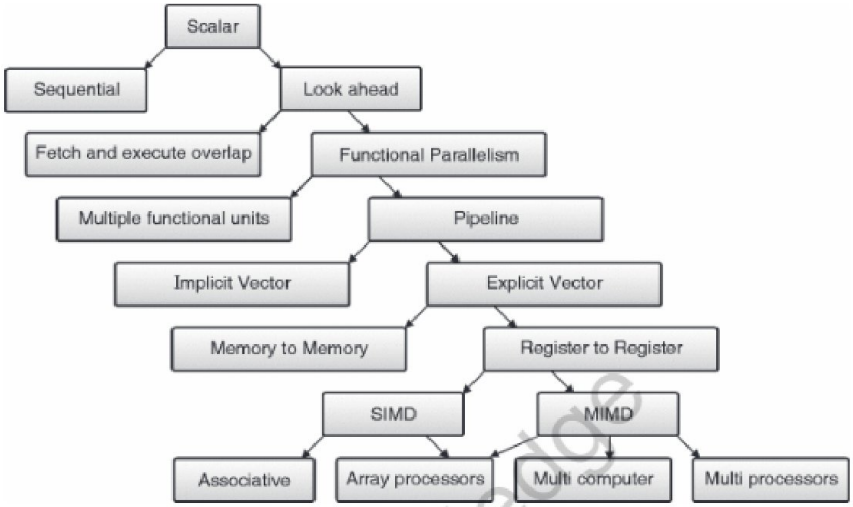


Fig. 3.5.2 : Architectural evolution tree

3.5.2 Ideal Model for Parallel Computing

- Without considering the physical constraints and implementation details, the ideal model should give a suitable framework for developing algorithms.
- PRAM provides an ideal model of a parallel computer for analyzing the efficiency of parallel algorithms. It helps to write parallel algorithm without any architecture constraints.

PRAM (Parallel Random Access Machine)

- The PRAM is used by parallel algorithm designers to model parallel algorithmic performance.
- It is a shared memory multiprocessor
- It has unlimited number of processors which is able to access the shared memory in constant time and has unlimited local memory.
- It can be suitable for modern day architectures , example GPU
- PRAM architecture model consists of control unit, global memory and an unbounded set of processors each with own private memory.

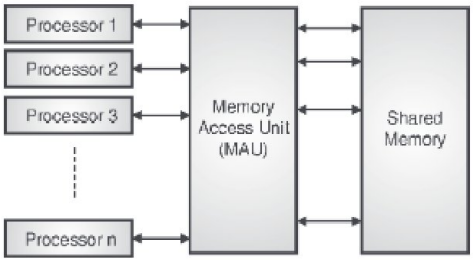


Fig. 3.5.3 : PRAM Architecture

- An active processor reads from global memory, performs computation and writes to global memory back.

- It executes in SIMD model.
- Various PRAM models differ in how they handle read or write conflicts.
  1. **EREW :** Exclusive Read Exclusive Write - p processors can simultaneously read and write the content of p distinct memory locations.
  2. **CREW :** Concurrent Read Exclusive Write - p processors can simultaneously read the content of p' memory locations , where p' < p and simultaneously write the content of p distinct memory locations.
  3. **CRCW :** Concurrent Read Concurrent Write
     - **(i) COMMON :** All processors writing to same global memory must write the same value.
     - **(ii) ARBITRARY :** One of the competing processor's value is arbitrarily chosen.
     - **(iii) PRIORITY :** Processor with the lowest index writes its value.
- Any PRAM model or algorithm can execute any other PRAM model or algorithm.
- For example, possible to convert PRIORITY PRAM to EREW PRAM.

3.6 Communication Costs in Parallel Machines

- The parallel computing platform provides facility for execution of parallel programs. Parallel programs composed of processes or tasks which communicate to achieve the goal of parallel computing.
- Two models for communication :
  1. Message passing (MP)
  2. Shared memory (SM)

  Communication is a major overhead in parallel programs.
- The efficiency of parallel algorithms is closely related to the cost of communicating. In order to improve the parallel efficiency of large problems on numerous processors, one has to control and minimize the communication cost.
- The communication cost depends on several factors like nodes used for parallel programming, protocol used in communication, the network topology, routing etc.

3.6.1 Issues Affect the Overall Communication

1. **Message passing cost in parallel computers :**

   To transfer a message over a network, total time required is calculated based on the following factors :

   - **(i) Startup time (t_s) :** Time spent at sending and receiving nodes.
   - **(ii) Per-hop time (t_h) :** It is a function of number of hops and includes factors such as network delays, switch latency etc.
   - **(iii) Per-word transfer time (t_w) :** It includes all overheads that are determined by the length of the message. This includes error checking and correction, bandwidth etc.

2. **Store-and-forward routing :**
   - A message traversing multiple hops is completely received at an intermediate hop before being forwarded to the next hop.
   - The total communication cost for a message of size m words to traverse l communication link is,

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

- In most platforms, t_h is small and the above expression can be represented by,

$$t_{comm} = t_s + mt_w$$

3. **Packet routing :**
   - Communication resources are poorly used by store-and-forward routing.
   - Packet routing breaks message into packets and pipelines them through the network.
   - Since packets may take different paths, each packet must carry sequencing, routing information, error checking and other header related information.
   - The total communication time for packet routing is,

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

   Where; t_w accounts for overheads in packet headers.

4. **Cost-through routing :**
   - It takes the concept of packet routing to an extreme by further dividing messages into basic fixed size units called **flow control digits or flits.**
   - Since flits do not contain overheads of the packets, they are typically small and thus header information must be minimized.
   - This is done by forcing all flits to take the same path, in sequence.
   - A tracer message first programs all intermediate routers. All flits then take the same route.
   - No sequence numbers are needed.
   - The total communication time for cut-through routing is :

$$T_{comm} = t_s + t_h l + t_w m$$

   - This is identical to packet routing, however, t_w is much smaller.

5. **Cost models for shared address space machines :**
   - While the basic messaging cost applies to this shared address space machines as well, a number of other factors make accurate cost modeling more difficult.
   - The reason for this is as follows :
     - Memory layout is typically determined by the system.
     - Finite cache sizes can result in cache thrashing.
     - Overheads associated with invalidate and update operations are difficult to quantify.
     - Spatial locality is difficult to model.
     - Prefetching can play a role in reducing the overhead associated with data access.
     - False sharing is often an important overhead in many programs
     - Contention in shared accesses is often a major contributing overhead in shared address space machines.
   - Any cost model for shared-address-space machines must account for all of these overheads.
   - we can use the same expression $t_s + t_w m$ to account for the cost of sharing a single chunk of m words between a pair of processors in both shared-memory and message-passing paradigms with the difference that the value of the constant t_s relative to t_w is likely to be much smaller on a shared-memory machine than on a distributed memory machine (t_w is likely to be near zero for a UMA machine).

Review Questions

Q. 1 Explain basic working principle of Superscalar Processor.

Q. 2 What are the limitations of Memory System Performance?

Q. 3 Write a short note on Communication Cost in Parallel machine.

Q. 4 What are the Advantages of VLIW processor ?

Q. 5 What are the Disadvantages of VLIW processor ?

Q. 6 Write short note on NUMA Multicomputer

Q. 7 Write short note on UMA multicomputer.

Q. 8 Write short note on Communication Costs in Parallel Machines.

Q. 9 Explain SIMD, MIMD architecture.

Q. 10 Explain store-and-forward and packet routing with its communication cost.

❑❑❑

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

4.1 Principles of Parallel Algorithm Design - Preliminaries

- Parallel algorithms are mainly meant for executing the concurrent part of the algorithm simultaneously by different processors. The final output of the different processors are finally collected together and the result is obtained.
- It is easy for some algorithms to be divided into parts that can be concurrently executed, and hence achieve a high level of parallelism. For example, if the numbers from 1 to 100 are to be checked, to find which are prime and which are not. If one processor is doing this operation it will take a long time, as it can check only one number at a time. But if we have 100 processors, we can divide the task amongst 100 processors, each checking one number to be prime or not. Hence this operation can be done concurrently.
- But there are quite a few operations that cannot be done concurrently. In this case, we need to find the concurrent operations, or else the algorithm has to be executed on one of the processors of the so many processors in the multiprocessor system. Most of the operations in the numerical methods are iterative, with each iteration dependent on the previous iteration. Hence they cannot be executed concurrently.

4.1.1 Preliminaries

For designing parallel algorithms the following steps are to be followed :

1. We need to find the pieces of work or tasks that can be done concurrently.
2. Then these tasks are to be mapped onto multiple processors i.e. we need to map the processes vs processors
3. The distribution of input/output & intermediate data across the different processors is also to be considered as this requires a lot of extra time.
4. Thus we also require the management of the accesses of shared data.
5. Finally the synchronization of the processors at various points of the parallel execution is also to be done. The most important things to be considered while making parallel algorithms is to maximize the concurrency, reduce the overheads due to parallelization and hence maximize the speedup.

4.1.2 Decomposition, Tasks and Dependency Graphs

4.1.2(A) Decomposition

- When we think about how to parallelize a program we use the concepts of decomposition.
- Decomposition is the process of dividing a computation into smaller parts. Some or all parts of this computation may be executed in parallel.
- The number and size of tasks into which a problem is decomposed determines the **granularity** of the decomposition.
- When a computation is divided into a large number of small tasks, it is referred as **fine-grained** decomposition.
- Whereas the **course-grained** decomposition consists of a small number of large tasks.
- To achieve a high degree of concurrency, we should have good decomposition techniques.

4.1.2(B) Tasks

- The first step in developing a parallel algorithm is to decompose the problem into tasks that are candidates for parallel execution.
- Task is an indivisible sequential unit of computation. It is a programmer defined units of computation into which the main computation is subdivided by means of decomposition.

4.1.2(C) Task-Dependency Graphs

- Tasks may have dependencies on other tasks. If the input of task B is dependent on the output of task A, then task B is dependent on task A.
- Task dependency graphs are used to describe the relationship between tasks.
- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next.

For Example :

(i) There is an edge between two tasks T_1 and T_2 if T_2 must be executed after T_1 task. A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other (Refer Fig. 4.1.1).

(a) Task T_2 is dependent on T_1 **(b) Task T_3 is dependent on T_1 and T_2**

Fig. 4.1.1

(ii) Here in graph 1 (Refer Fig. 4.1.2), Task 7 is dependent on Task 4 and Task 7 , 6, 5,3,2,1.

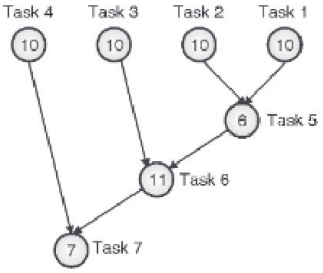


Fig. 4.1.2 : Graph 1

In dependency graph we have start and finish node.

- Start node- the nodes with no incoming edges
- Finish node- the nodes with no outgoing edges

Basic features of task dependency graph

1. **Critical path**
   - Critical path is the longest directed path between any pair of start and finish nodes.
   - The sum of the weights of nodes along this path is the critical path length.
   - For graph 1, critical path length is 10 + 6 + 11 + 7 = 34, Node 10 → 6 → 11 → 7
2. **Total amount of work**
   - If assumed that each tasks takes one unit of time then it is referred as total amount of work.
   - Here in graph 1, total amount of work is 10 * 4 + 6 + 11 + 7 = 64. As we have total 7 tasks and we are assuming each one takes 1 unit time.
3. **Maximum degree of concurrency**
   - Maximum number of tasks executed simultaneously in a parallel program is termed as the maximum degree of concurrency.
   - In the above graph 1, maximum degree of concurrency is 4.
4. **Average degree of concurrency**
   - The average number of tasks that can run concurrently over the entire duration of execution of the program is termed as average degree of concurrency.

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

   - For graph 1,

     $$\text{Average degree of concurrency} = \frac{64}{34} = 1.88$$

5. **Degree of concurrency**

   The number of tasks that can be executed in parallel. The degree of concurrency varies with the granularity of the decomposition.

4.2 Decomposition Techniques

De-composition Techniques are as follows :

1. Data decomposition
2. Recursive decomposition
3. Exploratory decomposition
4. Speculative decomposition

4.2.1 Data Decomposition

- If we want to execute a problem which is having large amount of data parallely then data decomposition is the technique.
- In data decomposition, the data is partitioned across various tasks.

  Two steps are required for decomposition of computation :

  1. The data on which the computations are performed is divided into sub parts.
  2. These sub parts is then executed by different processors concurrently.

For example : Matrix multiplication

- Consider matrix multiplication problem, multiply matrix A and B of size 2*2 which will store results in matrix C.
- The output of matrix C is divided into four tasks as,

  T_1 = AE + BG

  T_2 = AF + BH

  T_3 = CE + DG

  T_4 = CF + DH

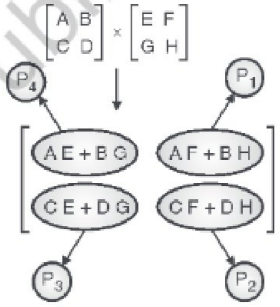


Fig. 4.2.1

- Now each task will be assigned to different four processors P_1, P_2, P_3 and P_4. All these processors will execute these 4 tasks concurrently so that execution will be faster.

Note :	In data decomposition, **data is different** with every processor but **computation is same**. As in above example all matrix elements in tasks are different but computation is same for all tasks i.e multiplication and addition.

4.2.2 Recursive Decomposition

- The recursive decomposition is based on providing concurrency in problem that can be solved in divide and conquer strategy.
- In divide and conquer, we follow the approach that to solve the problem directly if it is small and if it is not able to solve then decompose the problem into sub problems and solve the sub problems.
- Each sub problem is solved recursively and then the solution to original problem is obtained by combining the results of these sub problems.

For example : Merge sort

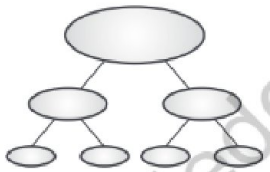


Fig. 4.2.2

4.2.3 Exploratory Decomposition

- Sometimes the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration i.e. search of a state space of solutions.
- In other words, your problem is in running state and in the running state it is also decomposed which is referred to as exploratory decomposition.

For example : Puzzle problem

- The puzzle problem is we have to transform from initial state to a desired final state.
- In this problem, there are 4 tiles numbered through 1 to 4 are arranged in 2×2 grid shown in Fig. 4.2.3

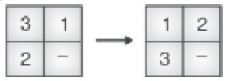


Fig. 4.2.3

- One tile is left blank, so that moves can be made. The 2 possible moves here are up and left with blank tile. So we have divided initial problem into 2 sub problems.
- So here we are searching whether we have reached till final goal and also concurrently applying moves. So we are executing problems at the same time we are decomposing problems into sub tasks.
- The same puzzle problem is possible with 4×4, 8×8, 15×15 grids. Only moves will be more.
- So in exploratory decomposition, computation is split into tasks, each task searching a different portion of the search space.

15 puzzle problem :

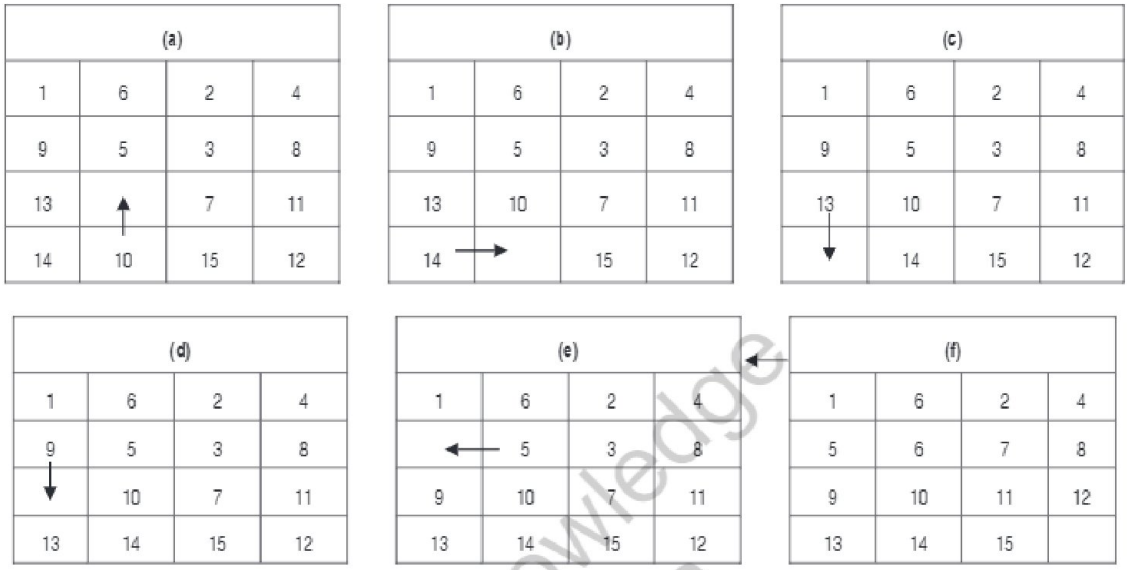


Fig. 4.2.4

- Fig. 4.2.4(a) shows the initial configuration of puzzle. Different sequence of movements right, up, down and left from initial to final steps are shown in Fig. 4.2.4(b) to Fig. 4.2.4(e). Fig. 4.2.4 (f) shows the final expected configuration.

4.2.4 Speculative Decomposition

- It is impossible to identify independent tasks whenever dependencies between tasks are not known in advance.
- In this decomposition, the action to be taken is based on output of the preceding part.
- This decomposition is used when a program may take one of many possible computationally significant branches depending on the output of the other computations that precede it.

For example : Switch case

- In switch case, among multiple available cases which case is to be considered is based on the input (expression) which has come from its preceding part.

```
Compute expr;
switch(expr)
{
    case 1:  compute t1;
             break;
  case 2 :   compute t2;
             break;
     -------
     -------
}
```

- Here lots of options are available to choose, but which one to select is depend on previous preceding code part.
- One more example of speculative decomposition is **topological sorting**.
- In topological sorting we will check incoming edges of every node and if the node doesn't have any incoming edge then only we can add that node in the sorting list.

  So here also we are looking for incoming edges (looking the previous state).

4.3 Characteristics of Tasks and Interactions

Once the problem has been decomposed into independent tasks, the characteristics of these tasks can have impact on performance of parallel algorithms.

We know that task is the basic unit of computations.

Characteristics of tasks :

1. Generation of tasks

The tasks that in used to implement parallel algorithms are generated as static or dynamic.

(i) Static task generation

- Before computation if all the tasks are known then it is static task generation.
- Data or recursive decomposition often leads to static task generation.
- **For example :** matrix and algorithms, image processing

(ii) Dynamic task generation

- Here tasks are not available a priori. In some cases, the tasks to be executed are created dynamically based on the decomposition of data. Tasks are generated as we perform computation.
- Exploratory and speculative decomposition can generate tasks dynamically.
- **For example :** quick and merge sort, puzzle game.

2. Size of task

- Every task takes some amount of time for its completion.
- Time duration required by the task to complete is termed as a size of task.
- Size will be either uniform or non-uniform.
- Uniform size - all tasks are of the same size. Matrix multiplication decomposition is of uniform size.
- Decomposition of merge sort is example of non-uniform size.

3. Knowledge of task size

We should know about the size of task i.e how much time the particular task will take for completion. Because for mapping task to different processors we should know in advance the size of the task. We will learn about mapping in section 4.4.

4. Data size

Data associated with the task must be available to the process performing the task while mapping. This will avoid excessive data movement overheads.

4.4 Mapping Techniques for Load Balancing

- We know that in parallel programming, processes execute a particular task. So **mapping is the technique used to assign tasks to processes.**
- The objective of mapping is that all tasks should complete in the shortest amount of time.
- Mapping techniques are used to solve the major sources of overheads that is
  - Load imbalance(some processes may spend being idle)
  - Inter-process communication
- So assigning a balanced load to each process is necessary. The main objective of load balancing is the amount of computation assigned to each processor is balanced so that some processors do not idle while others are executing tasks.
- There are two techniques of mapping for load balancing

1. Static mapping

- Before executing algorithm, tasks are distributed among processes.
- If tasks sizes are unknown, a static mapping can lead to serious load-imbalances.
- Static mapping is applicable for tasks that are generated statically and advance uniform computational requirements are known in advance.

2. Dynamic mapping

- Dynamic mapping is also referred to as dynamic load balancing
- Tasks are distributed among processes at runtime.
- It is applicable for tasks that are,
  - Generated dynamically and
  - If we have non-uniform computational requirements.
- There are two different classes of dynamic mapping

(i) Centralized dynamic mapping
(ii) Distributed dynamic mapping

(i) Centralized dynamic mapping

- In centralized dynamic mapping, all executable tasks are maintained in a common central data structure or they are maintained by a special process or a subset of processes.
  - **Master :** Process mange a group of available tasks.
  - **Slave :** Depend on master to obtain work.
- When a slave process has no work, it takes a portion of available work from master and when a new task is generated, it is added to the pool of tasks in the master process.
- The main problem with centralized dynamic mapping is,
  - When many processes are used, master process may become bottleneck.
- Solution to this problem is,

- Chunk scheduling : every time a process runs out of work it gets a group of tasks from master. In other words, a process will get chunks of tasks in a single step and not a single task.

(ii) Distributed dynamic mapping

- In distributed dynamic mapping, tasks are distributed among processes which exchange tasks at run time to balance work.
- Each process can send or receive work from other processes.
- The main problem with distributed dynamic mapping is that how the sending and receiving processes paired together at run time and who will initiate the work transfer (sending or receiving process).

4.5 Methods for Containing Interaction Overheads

- For an efficient algorithm, reducing the interaction overhead among concurrent tasks is important.
- The overhead that a parallel program incurs due to interaction among processes depends on many factors, such as, the volume of data exchanged during interactions, the frequency of interaction, the spatial and temporal pattern of interaction etc.
- General techniques used to reduce the interaction overheads are,
  1. Maximizing data locality
  2. Minimizing Contention and Hot-Spots
  3. Overlapping Computation with interaction
  4. Replicating data or computation
  5. Using Optimized collective interaction operations

4.5.1 Maximizing Data Locality

- For some parallel programs, the different processes require access to common input data or may be processes require data generated by other processes. In this case it will increase interaction overheads.
- The interaction overhead can be reduced by using techniques that promote the use of local data or data that have been recently fetched.
- Various schemes of data locality enhancing that try to,
  - Minimize the volume of non-local data that are accessed,
  - Maximize the reuse of recently accessed data and,
  - Minimize the frequency of accesses.

4.5.2 Minimizing Contention and Hot-Spots

- In previous sections we have learned that interaction overheads can be reduced by reducing frequency and volume of data transfer.
- Though, the data access and inter-task interaction patterns can often lead to contention that can increase the overall interaction overhead.
- A frequent source of contention for shared data structures or communication channel is because of centralized schemes for dynamic mapping.

- Contention occurs when,
  - Multiple tasks try to access the same resources concurrently.
  - Multiple simultaneous transmissions of data over the same interconnection link.
  - Multiple simultaneous accesses to the same memory block.
  - Multiple processes sending messages to the same process at the same time.
- The contention may be reduced by choosing a distributed mapping scheme.

4.5.3 Overlapping Computation with Interaction

- After an interaction has been initiated, the amount of time that processes spend waiting for shared data to arrive or to receive additional work can be reduced by doing some useful computations during this waiting time.
- There are number of techniques that can be used for overlapping computations with interaction.
- The simplest technique is initiating an interaction early enough so that it is completed before it is needed for computation.
  - For this, need to identify computations that can be performed before the interaction and do not depend on it.
  - Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original algorithm.
- In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process.
- Instead of this if the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance.
- On a shared-address-space architecture, the overlapping of computations and interaction is assisted by prefetching hardware.
- So if 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.

4.5.4 Replicating Data or Computation

Techniques that are used to reduce interaction overheads

1. Replication of data :

- Multiple processes may require frequent read-only access to shared data structure such as hash-table.
- Replicate a copy of the shared data structure on each process.
- After the initial interaction during replication, all subsequent accesses to this data structure are free of any interaction overhead.

2. Replicating computation :

- The processes in a parallel program often share intermediate results in some situations it may be more cost effective for a process to compute these intermediate results than to get them from another process that generates them.
- However, data replication increases the memory requirements of a parallel program as we need to store replicated data on each process.
- So data replication must be used selectively to replicate relatively small amount of data.

4.5.5 Using Optimized Collective Interaction Operations

- The interaction pattern among concurrent activities is often static and regular.
- A class of such static and regular interaction patterns are those that are performed by groups of tasks and they are used to perform certain type of computations on distributed data.
- A need to identify such collective interaction operations that appear frequently in many parallel algorithms to reduce the interaction overhead.
- For example, broadcasting some data to all processes

4.6 Parallel Algorithm Models

An algorithm model is typically a way of structuring a parallel algorithm by selecting decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

There are basically six different types of model :

1. Data parallel model
2. The task graph model
3. The task pool model
4. Master/Slave model
5. Pipeline/producer-consumer problem
6. Hybrid model

4.6.1 Data Parallel Model

- It is simple algorithm model.
- In this model, the tasks are statically mapped onto processes and each task performs similar operations.
- It is a result of identical operations being applied concurrently on different data items, is called data parallelism.
- The work may be done in phases. Data parallel computation phases are interspersed with interactions to synchronize tasks.
- It can be implemented in both shared-address-space and message passing paradigm.
- Interaction overhead can be minimized.
- The degree of data parallelism increases with size of problem.
- It solves large problems effectively.
- For example : Dense matrix multiplication.

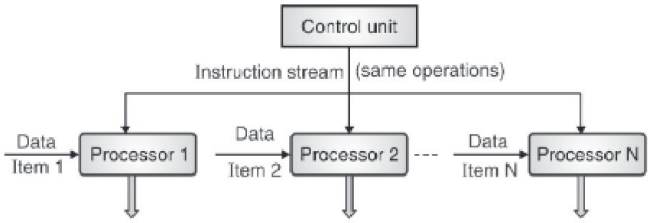


Fig. 4.6.1

4.6.2 Task Graph Model

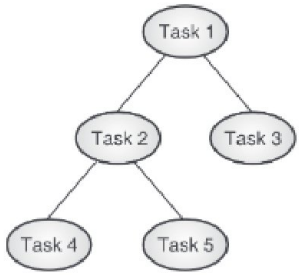


Fig. 4.6.2

- Starting from task dependency graph, the interrelationship among tasks are utilized to promote locality or to reduce interaction cost.
- It is used to solve problem in which amount of data associated with tasks is large.
- Tasks are mapped statically to optimize the cost of data.
- For example : Parallel quick sort, sparse matrix factorization
- This type of parallelism that is naturally expressed by independent tasks in task dependency graph called **Task Parallelism.**

4.6.3 The Task Pool Model

- It is also called as work pool model.
- It is characterized by dynamic mapping of tasks onto process for local balancing in which task may be performed by any process.
- No desired premapping.
- Mapping may be centralized or decentralized
- Pointer to task may be stored in physically shared list, queue, hash table or tree.
- Work may be statically available in the beginning or could be dynamically generated.
- Used when amount of data associated with task is relatively small.
- For example : Parallelization of loops by chunk scheduling

4.6.4 Master-Slave Model

- It is also called as Manager-Worker model.
- One or more master processes generate work and allocate to worker processes.
- Tasks may be allocated prior if manager can estimate size of task.
- In other scenario, workers are assigned small pieces of work at different times.
- No desired premapping is done, manager will allot work.
- The manager –Worker model can be generalized to hierarchical or multi-level model in which top level manager sends chunks of tasks to low level managers who further sub divides tasks.
- It is suitable to shared-address space or message passing paradigm.
- Master should not become bottleneck, which may happen if task are too small.
- It reduced waiting time.

4.6.5 Pipeline / Producer-Consumer Problem

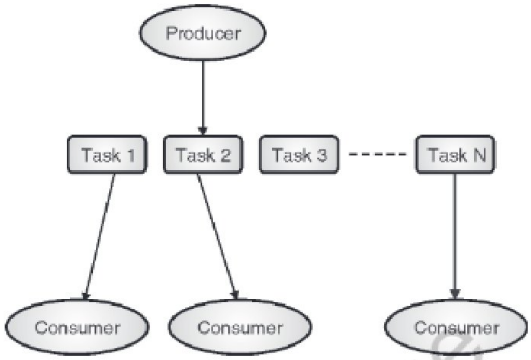


Fig. 4.6.3

- In this model, a stream of data is passed on through succession of processes, each of which perform same task.
- This simultaneous execution of different programs on a data stream is called Stream Parallelism.
- Processor could form pipeline in the form of arrays, trees, graphs etc.
- A pipeline is a chain of producer and consumer.
- Each process in pipeline can be viewed as consumer of sequence of data items for process preceding it and producer of data for process following it.
- It involves static mapping.
- Load balancing is function of task granularity.
- For example : Parallel factorization algorithm

4.6.6 Hybrid Model

- More than one model can be applicable to problem.
- It is a composition of multiple models.
- Task dependency graph is used.
- For example : Parallel Quicksort

As seen the array processors can operate on an array of data simultaneously and hence provide parallel processing. Let us see some algorithms implemented on array processor.

4.6.6(A) Scan Algorithms

These algorithms are as the name says that scan an array and perform the required operation. We will see the various applications of these scanning algorithms in the following sections.

1. Adding a Set of Elements of an Array

- If we want to add a set of elements in an array on a single processor system, then it will take us O(n) time. The same thing can be ideally done on an array processor in O(n/p) time. Since there are 'p' processors, each processor working on one element of the 'n' sized array simultaneously, the time required will be 1/p of what it is required on a single processor. A code to perform this operation of a array processors can be as given below:

```
total = segment[0];                    //each processor takes the first element in its segment as the initial total
for(i = 1; i < segment.length; i++)
{
    total += segment[i];               //all the remaining elements of the segment given to a processor are
                                       // added and stored in the variable total
}
if(pid > 0)                            // for all the processors except for the processor '0', send their
                                       //individual total to the processor '0'
{
  send(0, total);
}
else
{
    for(int k = 1; k < procs; k++)     //for processor '0' receive the total from each processor and add them.
    {
          total += receive(k);
    }
}
```

- In this program the variable "pid" is used to indicate the processor number or the identity of the processor number. The variable "procs" is the total number of processors in the parallel processing system. The method or the function send() is used to send a value to another processor.
- The parameters passed with this function are the "pid" of the processor to whom the value is to be passed and the value to be passed. Each processor is given a section of array to operate on called as "segment".
- The numbers written outside the circles are the "pid" in Fig. 4.6.4. Also in this Fig. 4.6.4, the elements of the arrays are taken as a, b, c and so on. And each array is given two elements of the array i.e. the segment size is two elements.
- Initially the variable "total" is initialized to first element of the segment in each processor. Then each processing element adds the elements in its segment (a part of the total array) using the first "for" loop seen in the program.
- The time taken for this is O(n / p) time, as expected by us. But then each processor has to forward its total to the processor "zero". This will require extra time for p - 1 transfers and their additions in the processor '0'. his is done in the second "for" loop and it will require O(p) time. Thus, the total time taken is O(n / p + p) i.e. more than what was expected. A diagrammatic representation of this method is shown in the Fig. 4.6.4.

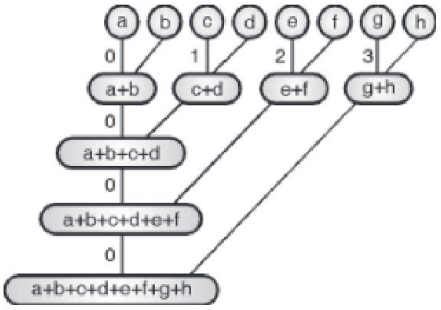


Fig. 4.6.4 : A method for parallel scan

- Hence you will notice that the time for communication is much more than the time for actual computation. In this case the first "for" loop is for computation and the second "for" loop is for communication. The first part i.e. the computation will take very less time as 'p' processors are working together, but the next "for" loop requires a lot of time.
- The second loop i.e. for communication, although for each processor is done simultaneously, but the processor '0' cannot accept or receive messages simultaneously. This process of message receiving is done as one by one. For a small array, where the value of 'p' is small this time required for communication is not an issue. But for long arrays with huge value of 'p', the extra time is definitely a big concern.
- Let us see another implementation to avoid the second loop. Fig. 4.6.5 shows how we can reduce the time required for communication. Here each alternate processor takes the element from the next processor, and adds them. Then in the next time unit, four alternate processors add, then eight alternate processors and so on.

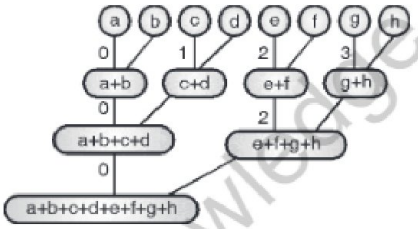


Fig. 4.6.5

- Thus in the example shown in Fig. 4.6.5 which has the array of 4 processors, after adding the elements in the individual segments, the processors '0' and '2' perform the partial additions of the total from processors '0' with '1' and '2' with '3' respectively. In the next round the processor '0' performs the addition of the total from the processor '0' with '2'.
- This algorithm will take O(n / p + log p) time, which is much better compared to the previous algorithm. Here the communication time is O(log p), where p is the number of processors. In our case, since there are 4 processors the time for communication is just 2. In case of huge arrays, this algorithm can give high performance.
- We have seen in these algorithms how parallel addition is done in array processor and hence increase the speed of operation. All operations that follow the law of associativity can use this algorithm, for example, multiplication, finding minimum or maximum value from an array etc.

2. Counting Initial 1's

- Another application of parallel scan can be to count the number of 1s in a given binary data.
- The parallel scan algorithm uses a mechanism wherein the following procedure is followed :

Step 1 : Each bit is written twice i.e. a bit a_i is replaced with (a_i, a_i)

Step 2 : The two bits are replaced with the two numbers, where the first digit gives the initial 1s in the segment given to the processor and the second digit is either a '0' or '1'. The second digit is '1' if all the bits in the said group are 1s else it is 0. The steps 1 and 2 are performed by the processor given the segment.

Step 3 : The processor takes this information from its neighbouring processor and performs the same operation as done is step 2

Step 4 : The process in step 3 is repeated until the final result is obtained by the first processor.

– For example if there is an array of 1s as (1,1,1,1,1,1,1,1,0,0,1,0,1,1,0,1,1,1), the implementation of the above algorithm is shown in Fig. 4.6.6.

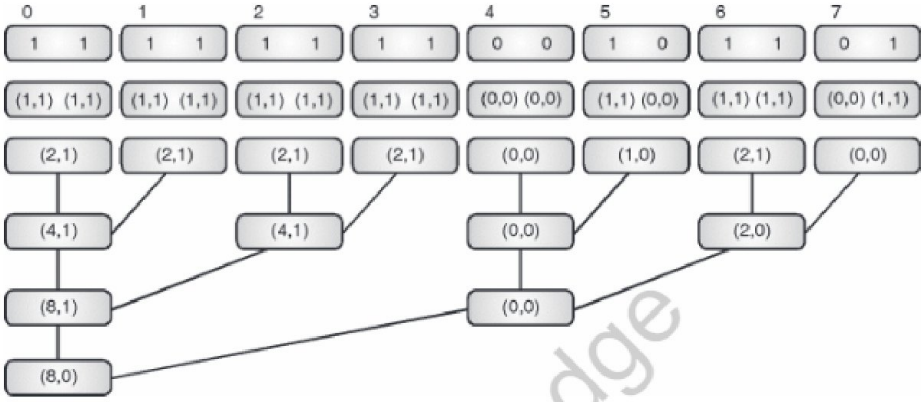


Fig. 4.6.6 : Counting number of initial 1s using array processor

– Thus in the example shown in Fig. 4.6.6, we perform the operation as discussed in the above algorithm and finally get the result as 8 initial 1s. To perform this operation of a uni-processor system, we would have taken 9 clock cycles, to check one by one each bit and reach upto the ninth bit where we get a '0'.

– But in this case of parallel algorithm we required only 4 clock cycles. The difference in the number of clock cycles further keeps on increasing as the number of initial 1s is larger.

3. Evaluating a Polynomial

– A polynomial of the following type is considered in this section

$$a_0 \cdot x^{n-1} + a_1 \cdot x^{n-2} + \ldots + a_{n-2} \cdot x + a_{n-1}$$

– Suppose we are given an array of coefficients of a polynomial and also given the value of the variable i.e. x.

– The algorithm to perform this operation is as given below :

Step 1 : Form the pairs of the coefficient and the variable value i.e. the values of a_i and x, forming (a_i, x).

Step 2 : Each processor takes a segment of two such pairs as formed in step 1, and gets a new pair with the value $(a_i \times x + a_{i+1}, x \times x)$.

Step 3 : The same process of step 2 is repeated from the values obtained from two consecutive processors until we reach to a single pair. The first value of this pair is the final result.

– Let us understand this algorithm with an example. The example is shown in Fig. 4.6.7. This example considers a polynomial $x^3 + x^2 + 1$ where x is 2.

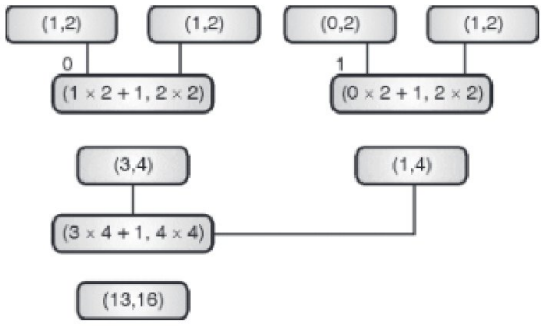


Fig. 4.6.7 : Evaluating a polynomial

- In the Fig. 4.6.7, we have taken the polynomial as discussed earlier. The same steps are followed at each processor level. For this small polynomial we require just 2 processing elements as shown in the Fig. 4.6.7.
- The result of the second processor is forwarded to the first processor which repeats the same operation and gives the result. The first term of the final result i.e. 13 is the value of the polynomial. The second term i.e. 16 would have been required in case of a larger polynomial to proceed further.

4. Parallel Prefix Scan

- Let us consider another application of scanning. Suppose we want to find the prefix scan of an array i.e. we need cumulative sum of all the previous elements of an element in the array. And this is to be done for all the elements. Then the following code will have to be used for doing this operation on a uniprocessor system.
- Assuming the array is named as 'a', with the elements indexed from '0' to 'n – 1'. We want the sum of the prefix elements as shown below :

$$(a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + a_1 + a_2 + \ldots + a_{n-1})$$

```
for(i = 1; i < array.length; i++)
{
  array[i] += array[i-1];
}
```

- Now if we have a multiple processor system, then we can perform these operations simultaneously and we will expect the time taken for the operation should be O(n / p + log p) time, since we required this much time for the sum of array elements in the previous section. But that is not the case, since the partial sums are required by all the processors. Let us see how this can be implemented in a multiprocessor based system.
- There are two phases of this algorithm namely the up-sweep phase and the down-sweep phase. The reason for the names can be understood from the example shown in the Fig. 4.6.8.

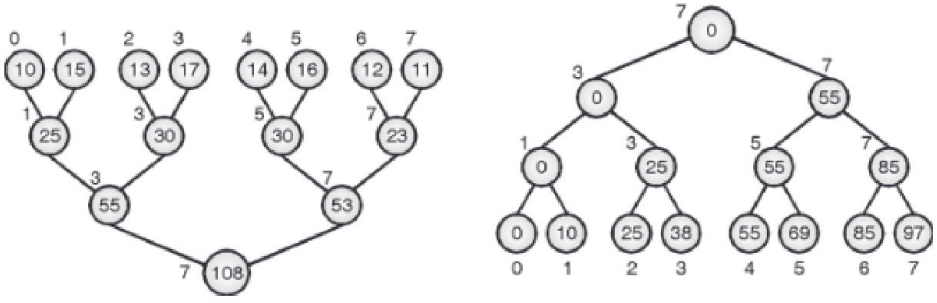


Fig. 4.6.8 : Parallel prefix scan

- The processors are numbered as shown in the Fig. 4.6.8. The left part in figure is up-sweep, wherein each alternate processor performs the addition as shown, and forwards the result to the next stage processor. This is similar to the parallel add algorithm done earlier in this section. In the Fig. 4.6.8, the up-sweep phase is shown in the upper four lines.
- The second part of the algorithm i.e. the down-phase can be better understood by the Fig. 4.6.9, where the lower four lines are for the down-phase. In the down phase shown in the right part of the fig., the processor 7, initially takes a value 0. This value is given to the left hand side processor, and the earlier value of the left hand side processor is added with this processor.
- For example, in this case the left hand side processor is 3. The processor 3 gets the value of processor 7 i.e. 0 and processor 7 gets the sum of its earlier value i.e. 0 and the value of processor 3 i.e. 55; hence processor 7 gets 55. The sequence of processors being used will be reverse as it was in the up-phase.

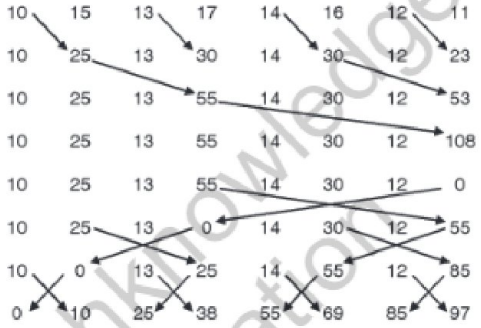


Fig. 4.6.9: Parallel prefix scan

- Thus for a set of eight elements in the array, we can notice in the Fig. 4.6.9, that we require only 6 clock pulses, 3 for the up-phase and 3 for the down-phase.

5. Sieve of Eratosthenes

- It is a very old algorithm to find the prime numbers upto a given integer number.
- This algorithm says to check and cross the multiples of the prime numbers already found. Thus those numbers which are not crossed when you reach to that element are said to be prime numbers.
- For example if we want to find the prime numbers upto 30, it is shown in the Fig. 4.6.10 with steps

Step 1 : 2 is a prime number and hence cross all multiples of 2, as they cannot be prime numbers. This is shown in Fig. 4.6.10 (a).

	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~
11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~
21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~

(a) Implementation of step 1

Fig. 4.6.10 contd....

Step 2 : Now we reach to 3, which is not crossed, hence it is a prime number. We need to cross all the multiples of 3, as shown in Fig. 4.6.10(b).

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~

(b) Implementation of step 2

Step 3 : Now we reach to 4, which is crossed hence it is not prime number. Next we reach to 5, which is not crossed, hence it is a prime number. We need to cross all the multiples of 5, as shown in Fig. 4.6.10 (c).

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~

(c) Implementation of step 3

Fig. 4.6.10

Step 4 : Now we reach to 6, which is crossed hence it is not prime number. Next we reach to 7, which is not crossed, hence it is a prime number. We need to cross all the multiples of 7. But the multiples of 7 are already crossed. Next is 8, 9, 10, which are already crossed and hence they are not prime. Now we reach to 11, which is not crossed and hence it is a prime number. Now we need to cross all multiples of 11, which are again all crossed. Next is 12, which is crossed, hence not prime number. 13 is again not crossed, hence it is prime. Similarly 17, 19, 23 and 29 are prime numbers while others are not prime numbers.

6. Parallel Addition Using Hypercube, Shuffle Exchange and 2D Mesh

- We have already seen some methods to perform parallel addition for multiple elements of an array. Let us see some more methods using an hypercube.
- We allocate n/p elements to each processor, where n is the total number of elements to be added and p is the number of processors.
- Hence each processor initially adds the elements given to it, resulting in p partial sums.
- These partial sums can be then added using the parallel algorithm.
- The different parallel algorithms can be used for this discussed below.

– The first method we will see is using the 4D hypercube SIMD model as shown in Fig. 4.6.11.

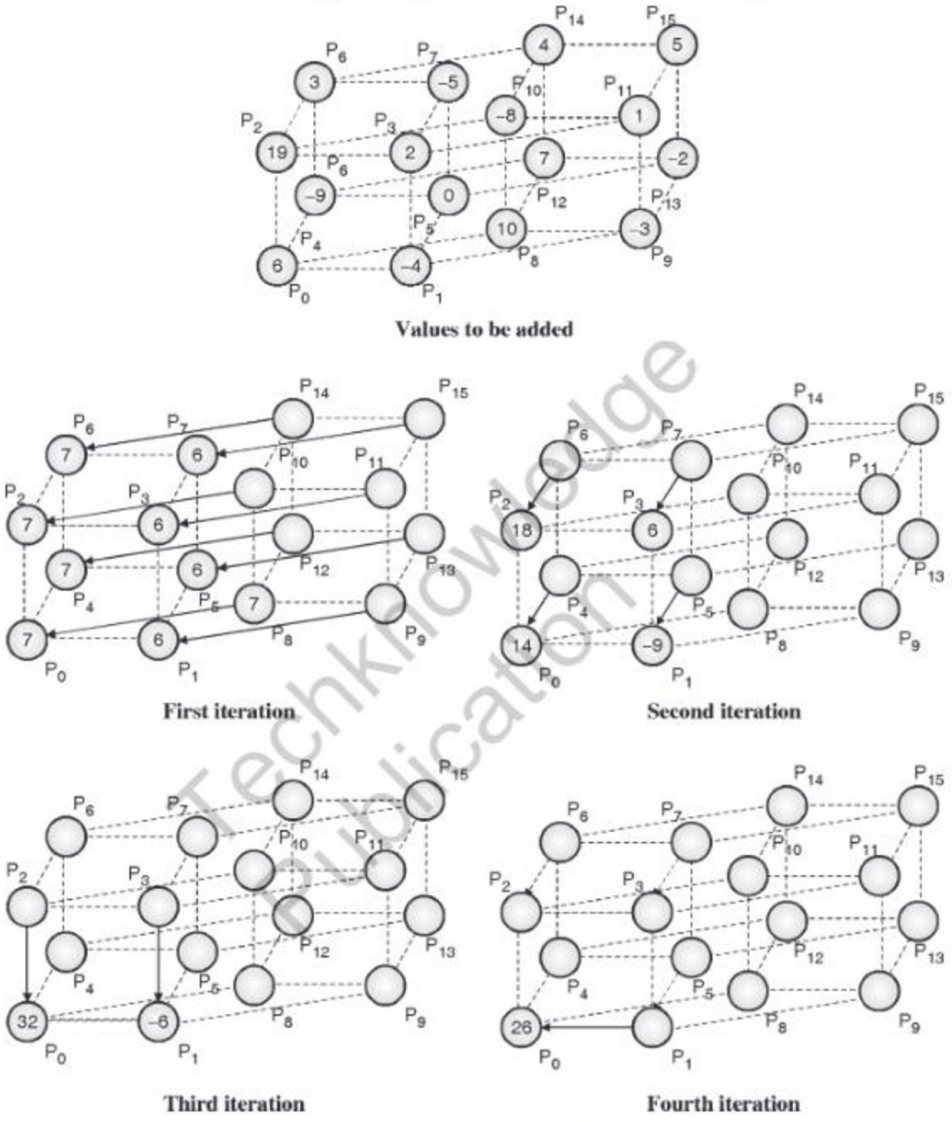


Fig. 4.6.11 : Hypercube SIMD implementation of parallel add

– In this case, first iteration as shown in the figure, all the elements on the back side are given to the ones on the front side, who add and store the new partial sum. Then in the second iteration, each processor on the upper cube gives the values to the lower cube and new sum is calculated. In the third and fourth iteration similarly the elements on the upper front processors are forwarded to obtain the final result.

– The second method is using the shuffle-exchange SIMD model as shown in Fig. 4.6.12.

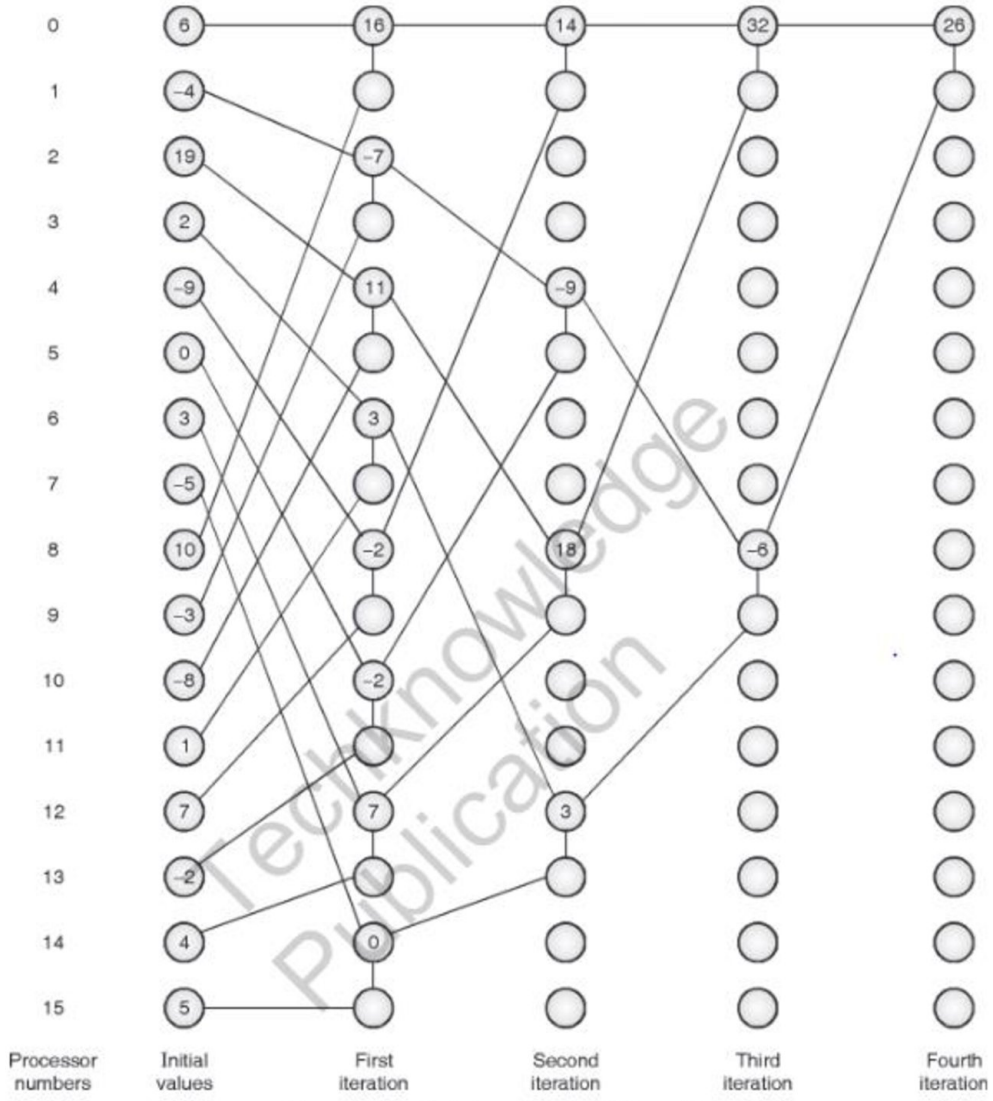


Fig. 4.6.12 : Shuffle exchange SIMD model

- The Fig. 4.6.12 is self explanatory, as it shows the passing of the values from one processor to another and hence performing the addition of all the elements and giving the final result.
- Another method of solving this array addition problem is using the 2D mesh as shown in Fig. 4.6.13.

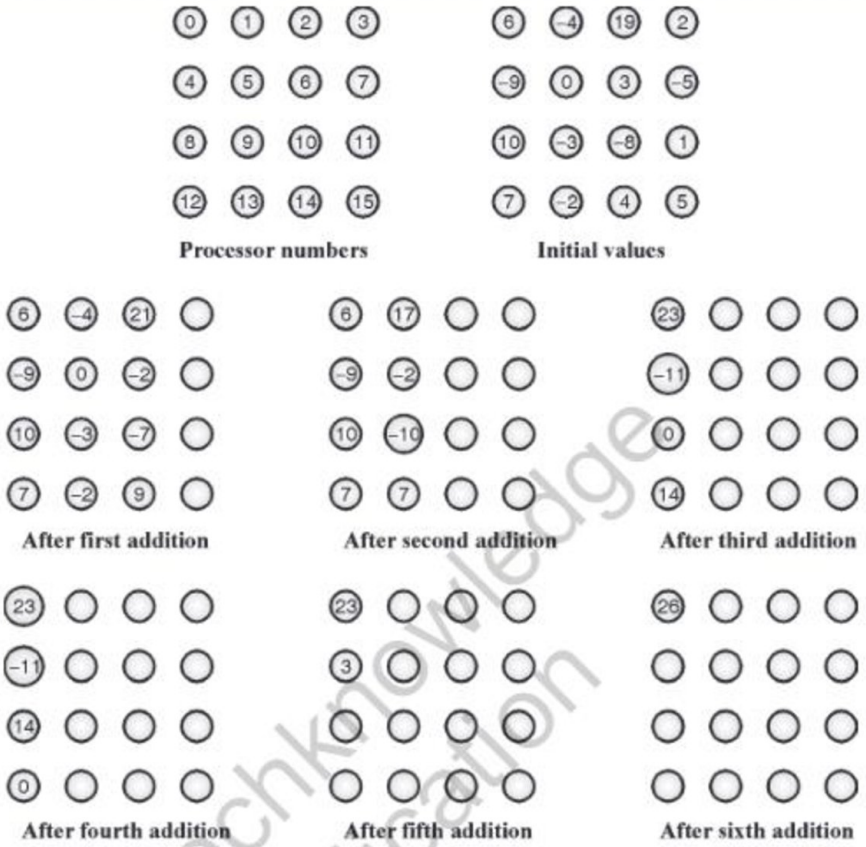


Fig. 4.6.13 : 2-D mesh for finding the sum

- In this case also the figure is self explanatory. Initially, each processor performs the addition of n/p processors and finds the partial sums. These partial sums are then added by them in the sixth addition or iteration as shown in Fig. 4.6.13.
- The same set of data is taken an the answer is found by first passing the values from the last column to the third, then from third to second and finally from second to first. The same three steps are repeated for the three rows, but now only for the first column, hence six iterations to get the final result.

7. Prefix Sum for an Array of N Greater than p

- We have seen in the previous subsection, for a parallel algorithm to find the prefix sum of an array elements.
- Here is another method for doing the same but in a different manner.
- If the number of processors are say 4 and the number of elements whose prefix sum is to be found is say 16, then the Fig. 4.6.14 shows an algorithm to achieve this.

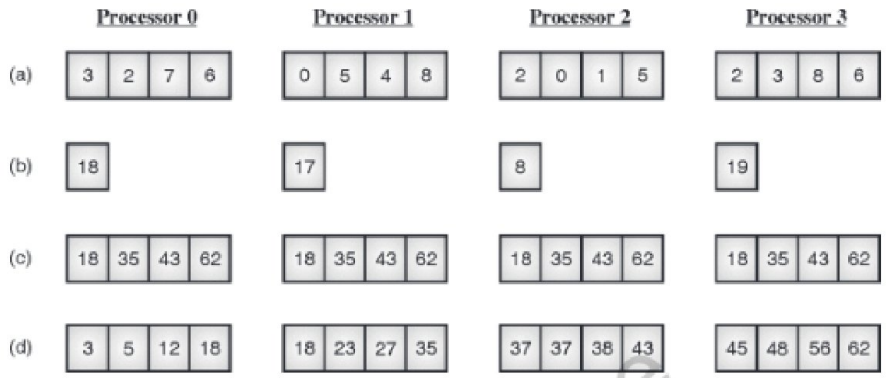


Fig. 4.6.14: Prefix sum calculation for n greater than p

- In step 1, each processor is given with its set of values equal to n/p.
- In step 2, all the processors perform the addition of their independent elements or segment given to them.
- In step 3, prefix sums of the local sums is calculated and distributed over all the processors.
- In the final step, each processor calculates the prefix sum of all its elements.

4.6.6(B) Matrix Multiplication

- A very lengthy process when carried out on a uni-processor system is the matrix multiplication.

$$\begin{matrix} a_{11}\ a_{12}\ a_{13} \\ a_{21}\ a_{22}\ a_{23} \\ a_{31}\ a_{32}\ a_{33} \end{matrix} * \begin{matrix} b_{11}\ b_{12}\ b_{13} \\ b_{21}\ b_{22}\ b_{23} \\ b_{31}\ b_{32}\ b_{33} \end{matrix} = \begin{matrix} c_{11}\ c_{12}\ c_{13} \\ c_{21}\ c_{22}\ c_{23} \\ c_{31}\ c_{32}\ c_{33} \end{matrix}$$

```
For I = 1 to N
    For J = 1 to N
        For K = 1 to N
            C[I, J] = C[I, J] + A[J, K] * B[K, J];
```

- For example if the matrices 3×3 are to be multiplied in a uni-processor system, we require 27 operations, as shown above in the 3-level nested for loop. Hence the time required is n^3, where n is the degree for the square matrix to be multiplied.
- There are two solutions to this in parallel processing. One is the use of Hypercube, and the other is use of a mesh.
- Fig. 4.6.15 shows the multiplication of a 2×2 matrix on a hypercube model.

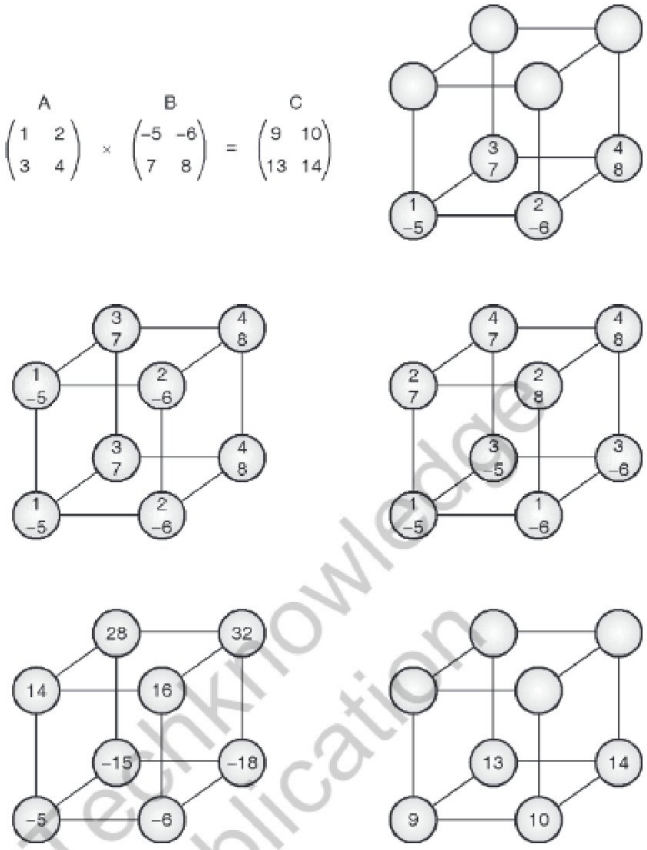


Fig. 4.6.15: Hypercube model for matrix multiplication

- Initially, the elements of the two arrays to be multiplied are given to the base processors of the hypercube, correspondingly with their row and column number.
- These values are then passed to the processors on the upper side of the hypercube.
- The upper processors copy the element of first matrix from right to left while the elements of second matrix from back to front processors respectively. At the same time the lower processors swap the elements of the first matrix from left to right and the elements of the second matrix from front to the back processors.
- The processors perform the multiplication of the elements with them of the two matrices. These values are then added by the lower processors, by receiving the products of the upper processors as shown in the last step in the Fig. 4.6.15.
- This algorithm involves communication steps in three loops, each with q/3 iterations (where 2^q is number of processors). Thus

$$T_{mul}(m, m^3) = O(q) = O(\log m)$$

- For matrix multiplication using a hypercube network, we need m^3 processors, where m is the number of rows and columns. For example, we just saw 8 processors are required for the multiplication of 2×2 matrix. Similarly if we want multiplication of 4×4 matrix, then we need 64 processors.

- Analysis in the case of block matrix multiplication of $m \times m$ matrices, can be calculated as below :
  1. The matrices are divided small matrices of $p^{1/3} \times p^{1/3}$ blocks each of size $(m / p^{1/3}) \times (m / p^{1/3})$
  2. Now here the number of block elements dealt by each communication step is $m^2 / p^{2/3}$ block elements
  3. The each computation or multiplication involved $2m^3/p$ arithmetic operations
- Hence the total time complexity for communication and computation can be given as below,

$$T_{mul}(m, p) = m^2 / p^{2/3} \times O(\log p) + 2m^3 / p$$

(Communication) (Computation)

- The second method using a 2-D mesh SIMD, is a very interesting method. It is also called as a systolic array architecture. This architecture of systolic array is shown in the Fig. 4.6.16.

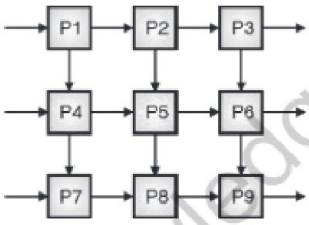


Fig. 4.6.16: Systolic array (or 2-D Mesh SIMD) for multiplication of a 3×3 matrix

- We need to modify the data before being given to these arrays of processors. The modification is that the columns 1 and 3 are to be swapped of the matrix 'a' and the rows 1 and 3 are to be swapped of matrix 'b'.

$$\begin{matrix} a_{13} & a_{12} & a_{11} \\ a_{23} & a_{22} & a_{21} \\ a_{33} & a_{32} & a_{31} \end{matrix} \qquad \begin{matrix} b_{31} & b_{32} & b_{33} \\ b_{21} & b_{22} & b_{23} \\ b_{11} & b_{12} & b_{13} \end{matrix}$$

(a) Matrix a after swapping columns 1 and 3 **(b) Matrix b after swapping rows 1 and 3**

Fig. 4.6.17: Matrices a and b after doing the needful change in the inputs

- Hence the input becomes as shown in Fig. 4.6.18.

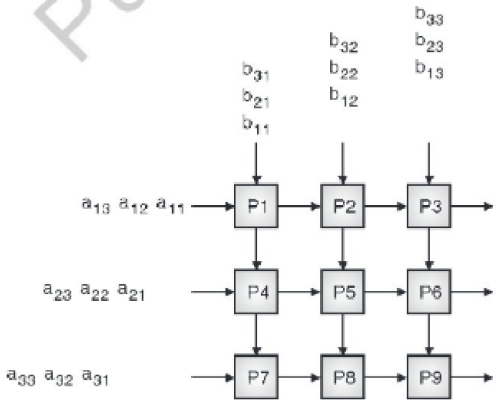


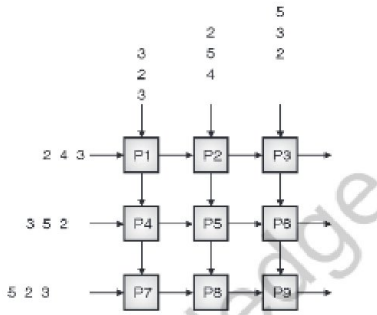
Fig. 4.6.18: Input sequence for matrix multiplication using systolic array architecture

- Finally the data is to be staggered as shown in Fig. 4.6.19.

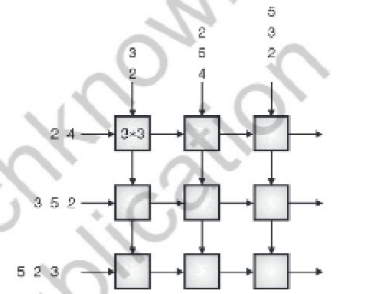
– Let us take an example for multiplication of two matrices.

$$\begin{matrix} 3 & 4 & 2 \\ 2 & 5 & 3 \\ 3 & 2 & 5 \end{matrix} \quad * \quad \begin{matrix} 3 & 4 & 2 \\ 2 & 5 & 3 \\ 3 & 2 & 5 \end{matrix} \quad = \quad \begin{matrix} 23 & 36 & 28 \\ 25 & 39 & 34 \\ 28 & 32 & 37 \end{matrix}$$

– Fig. 4.6.19 shows the sequence of giving the data and performing the operations at different stages with every clock tick and the corresponding output with every processor after that clock tick.

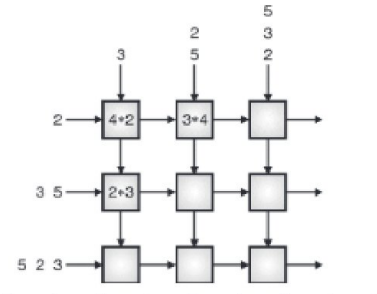


(a) Inputs swapped and staggered before being given



P1	P2	P3	P4	P5	P6	P7	P8	P9
9	0	0	0	0	0	0	0	0

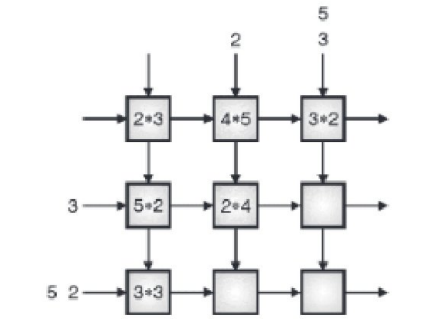
(b) After the first clock tick



P1	P2	P3	P4	P5	P6	P7	P8	P9
17	12	0	6	0	0	0	0	0

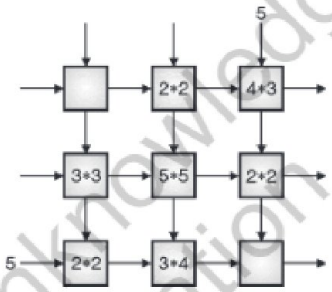
(c) After the second clock tick

Fig. 4.6.19 contd....



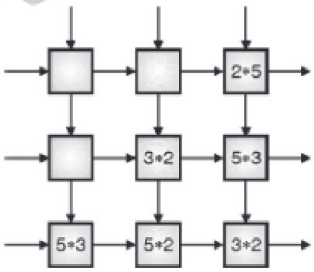
P1	P2	P3	P4	P5	P6	P7	P8	P9
23	32	6	16	8	0	9	0	0

(d) After the third clock tick



P1	P2	P3	P4	P5	P6	P7	P8	P9
23	36	18	25	33	4	13	12	0

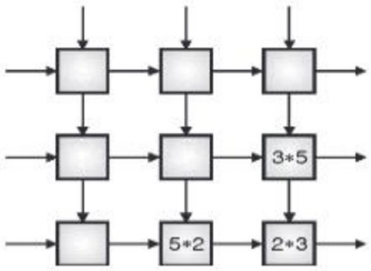
(e) After the fourth clock tick



P1	P2	P3	P4	P5	P6	P7	P8	P9
23	36	28	25	39	19	28	22	6

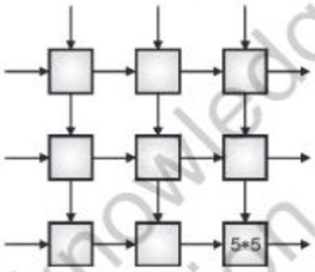
(f) After the fifth clock tick

Fig. 4.6.19 contd....



P1	P2	P3	P4	P5	P6	P7	P8	P9
23	36	28	25	39	34	28	32	12

(g) After the sixth clock tick



P1	P2	P3	P4	P5	P6	P7	P8	P9
23	36	28	25	39	34	28	32	37

(h) After the seventh clock tick

Fig . 4.6.19 : Matrix multiplication using Systolic array or 2D Mesh

- The Fig. 4.6.19 is self explanatory with the steps.
- **Time complexity :** Thus in the seventh clock tick, we get the final answer of a 3×3 matrix multiplication which take 27 clock ticks in a uni-processor system.

4.6.6(C) Parallel Sorting Algorithms

Till now we have been seeing some basic operations do be carried on array processors. Let us see something which actually requires a lot of time on uni-processor system and how can it be fastened in multiprocessor system i.e. sorting. We will see the different sorting algorithms followed to this.

1. Divide and Conquer Algorithm
2. Enumeration Sort
3 Odd-even Transportation Sort
4. Bitonic Merge Sort
5. Hyper Quick sort

1. Divide and Conquer Algorithm

- This algorithm as the name says divides the array or partitions the array at the pivot.
- Once the pivot is decided and the elements are partitioned, each processing element can concurrently perform operations of sorting.
- This is shown in Fig. 4.6.20. The first case pivot taken is 5, hence all numbers below 5 are given to one processor while the ones equal to and greater than 5 are given to the next processor. Now we get two arrays. These two arrays can be processed on concurrently or simultaneously by two different processors. Again the pivot taken is 3 for one array and the other array has the pivot taken is 9. The elements less than 3, are given to one processor and the ones equal to and above 3 are given to another processor and so on as shown in the Fig. 4.6.20.

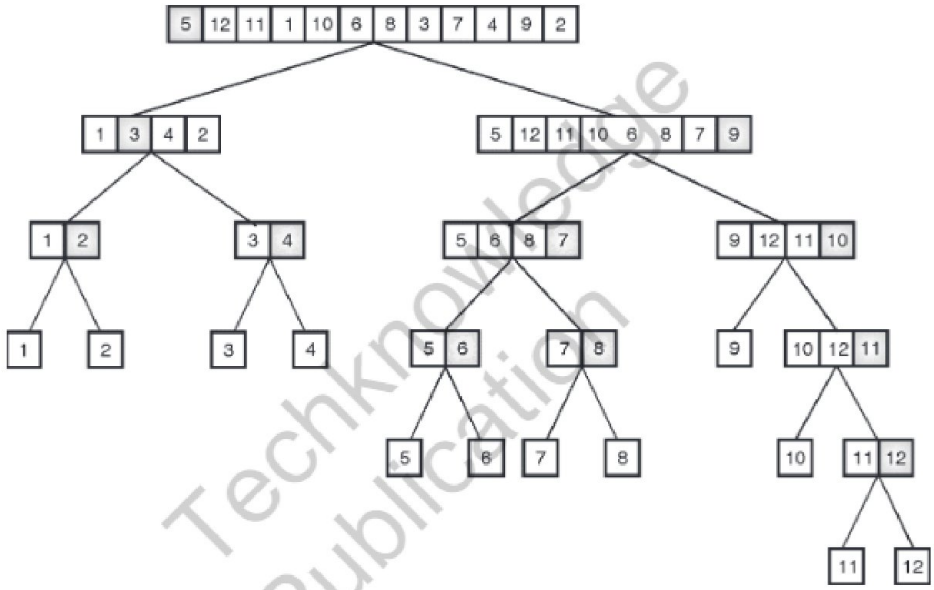


Fig. 4.6.20 : Divide and conquer sorting

2. Enumeration Sort

- This method finds the exact position of each element in the sorted list. This is done by comparing it with the other elements and finding a count of the number of elements with smaller value than this element. For example, if x elements are smaller than a_i, then the position of this element is (x + 1).
- To have parallelism, we can have each processor in a parallel processor system finding the position of one element and placing it in the shared memory.
- If one or more elements have same value, they will have the same position according to the above said algorithm. In this case the algorithm is to be amended.

3. Odd-even Transportation Sort

- This is a special method of sorting wherein, the odd-even and even-odd exchanges are performed alternately. It is also called as parallel bubble sort and is implemented using 1-D meshes.
- This algorithm takes exactly N steps to sort N numbers on M(N), which is optimal due to the diameter argument. Thus the cost of the algorithm is $O(N^2)$.

- In the first case all odd elements are compared with the consecutive next even element, and the same are swapped if not in sorted order. Then the even elements are compared with the consecutive odd next elements and again sorted if not in proper order.
- Hence if there are 8 elements in an array, we need 4 processors to compare the elements simultaneously.
- An example of this 8 elements sorting is shown in the Fig. 4.6.21.

Indices :	0	1	2	3	4	5	6	7
Initial values :	G	H	F	D	E	C	B	A
After odd-even exchange :	G	F	< H	D	< E	B	< C	A
After even-odd exchange :	F	< G	D	< H	B	< E	A	< C
After odd-even exchange :	F	D	< G	B	< H	A	< E	C
After even-odd exchange :	D	< F	B	< G	A	< H	C	< E
After odd-even exchange :	D	B	< F	A	< G	C	< H	E
After even-odd exchange :	B	< D	A	< F	C	< G	E	< H
After odd-even exchange :	B	A	< D	C	< F	E	< G	H
After even-odd exchange :	A	< B	C	< D	E	< F	G	< H

Fig. 4.6.21: ODD EVEN transportation sort

- Initially in the first iteration, 3 comparisons are done i.e. 1 with 2, 3 with 4 and 5 with 6. In next iteration 4 comparisons are done i.e. 0 with 1, 2 with 3, 4 with 5 and 6 with 7. This is to be repeated for 8 times for 8 elements. In case of a uniprocessor system this would have required 64 iterations in a nested for loop.

4. Bitonic Merge Sort

- A bitonic sequence is a special sequence which has values from $a_0,\ldots\ldots,a_{n-1}$, with the following properties :
  1. There exists an index i, where 0 <= i <= n – 1, such that a0 through ai is monotonically increasing and ai through an–1 is monotonically decreasing or
  2. There exists a case wherein with some cyclic shifts of indices, the first condition is satisfied.

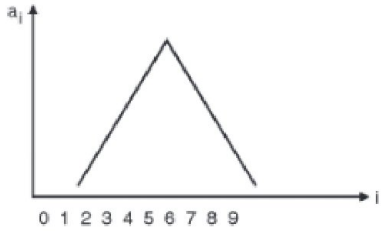


Fig. 4.6.22 : Bitonic sequence distribution

- Let us take an example to sort a set of characters, which is a bitonic sequence. The example is shown in Fig. 4.6.23.
- The sequence at the input if seen carefully is a bitonic sequence i.e. A, C, D, E, G, H, J, K, L, P, O, N, M, I, F, B. The sequence keeps on increasing up to 'P' and then starts decreasing.

- In this case, each processing element is given one element in sequence from the upper half of the array and one element in sequence from the lower half.
- The processor sorts the two elements given to it, and the same thing repeats as in the previous step. The upper element is retained by the upper processors while the lower element is passed on to the lower processors. In the lower half its reverse i.e. the lower element is retained while the upper element is passed to the upper processor.
- The processors that swap in the first two steps now, distribute in four parts and repeat the same step i.e. distribute and swap if required. Finally, they further reduce in eight parts and repeat the same step i.e. distribute and swap if required.

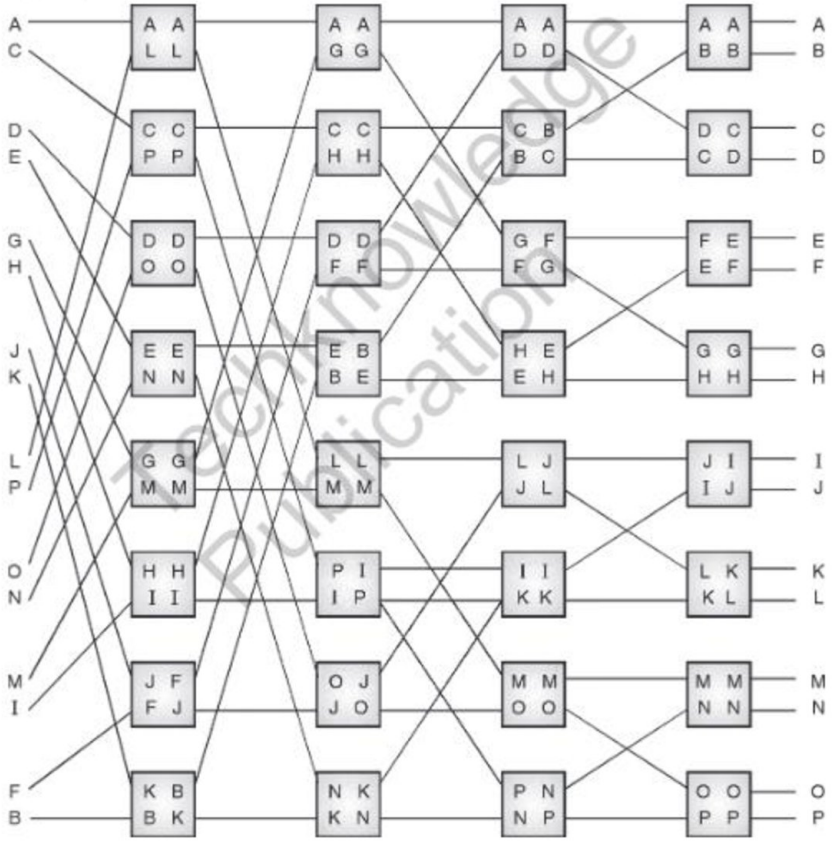


Fig. 4.6.23: Bitonic merge

- Finally the sequence is sorted as seen at the end in the Fig. 4.6.23.

Note : This sorting mechanism will work only for a special type of sequence i.e. bitonic sequence. Hence it cannot be used as a generalised sorting algorithm.

5. Hyper Quick sort

– In this method each processor is given n/p elements, where 'n' is the size of array and p is the number of processors. Each processor sorts these elements given to it.

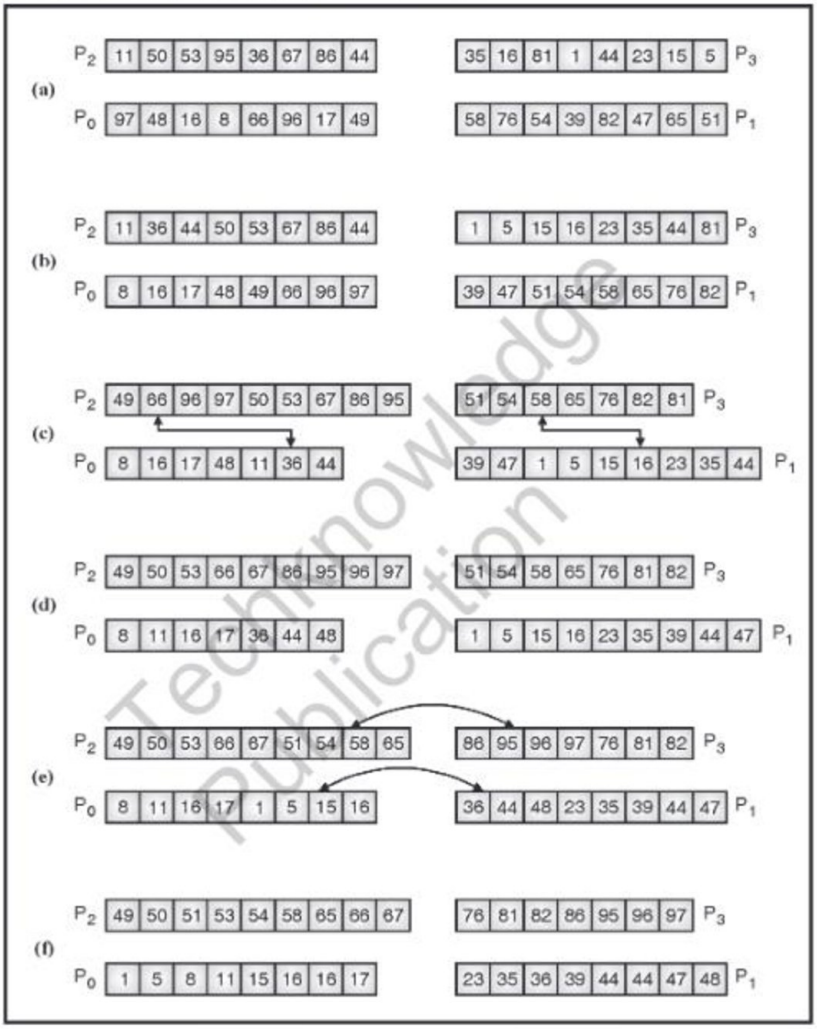


Fig. 4.6.24: Hyper quick sort

– Then the array is divided into two parts based on a centre point decided. The elements below that are given by the higher processor to the lower processor while the higher ones are given by the lower processor to the higher processor; for example as shown in the Fig. 4.6.24 there is a swapping done between processors 1 with 3 and between processors 2 with 4 in step (c) in Fig. 4.6.24.

– The same thing is repeated i.e. each processor sorts the elements given to it and then the centre point is taken and now the horizontally arranged processors swap the lower and upper elements as discussed in the previous point. But in this case the horizontally placed processors swap i.e. processors 1 with 2 and processors 2 with 4 as shown in step (e) in Fig. 4.6.24.

4.6.6(D) Fast Fourier Transform

– The Fast Fourier transform is a very important calculation for parallel processing as the time required for this is very large. And if it can be reduced with the parallel processing then it will be very advantageous.

– The FFT has the standard format as shown below :

$$\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^0 & \omega_4^2 \\ \omega_4^0 & \omega_4^3 & \omega_4^2 & \omega_4^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

– For example

$$\begin{pmatrix} \omega_4^0 & \omega_4^0 & \omega_4^0 & \omega_4^0 \\ \omega_4^0 & \omega_4^1 & \omega_4^2 & \omega_4^3 \\ \omega_4^0 & \omega_4^2 & \omega_4^0 & \omega_4^2 \\ \omega_4^0 & \omega_4^3 & \omega_4^2 & \omega_4^1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} 10 \\ -3-i \\ 0 \\ -3+i \end{pmatrix}$$

– The formula for 8 point FFT is done using the following method :

$$\begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & -j & -\frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & -1 & -\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} & j & \frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -\frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & j & \frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & -1 & \frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} & -j & -\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} & -j & \frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} & -1 & \frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & j & -\frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & \frac{1}{\sqrt{2}}+j\frac{1}{\sqrt{2}} & j & \frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & -1 & -\frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} & -j & \frac{1}{\sqrt{2}}-j\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 1 \\ \frac{1}{\sqrt{2}} \\ 0 \\ -\frac{1}{\sqrt{2}} \\ -1 \\ -\frac{1}{\sqrt{2}} \\ 0 \end{bmatrix}$$

– Let us see an SIMD mesh connected network for performing this operation as seen in the topologies earlier.

– The Fig. 4.6.25, shows along with an example values of the implementation of the 8-point FFT.

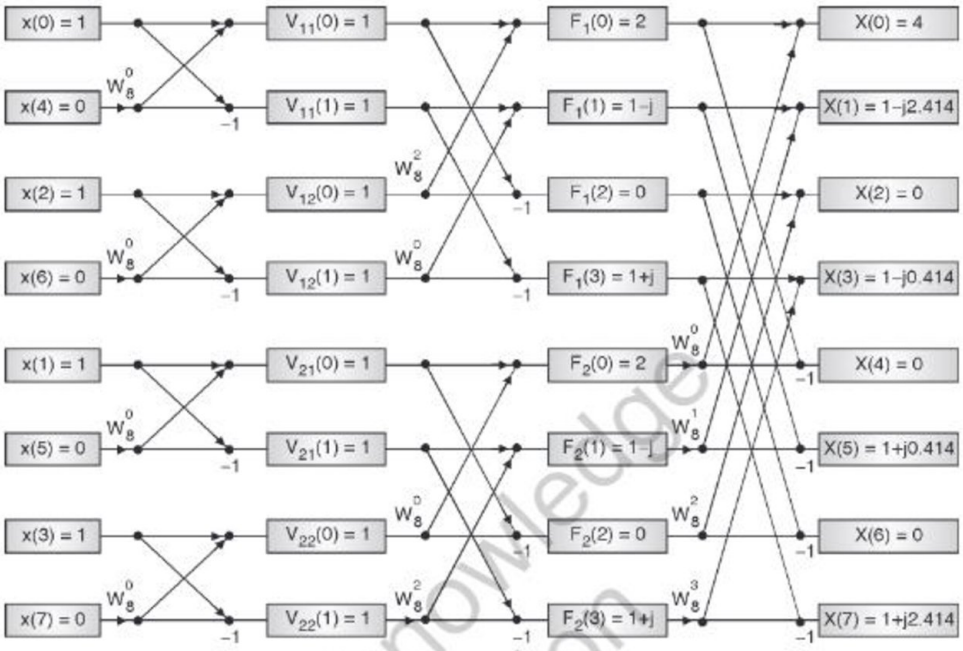


Fig. 4.6.25: 8-point FFT

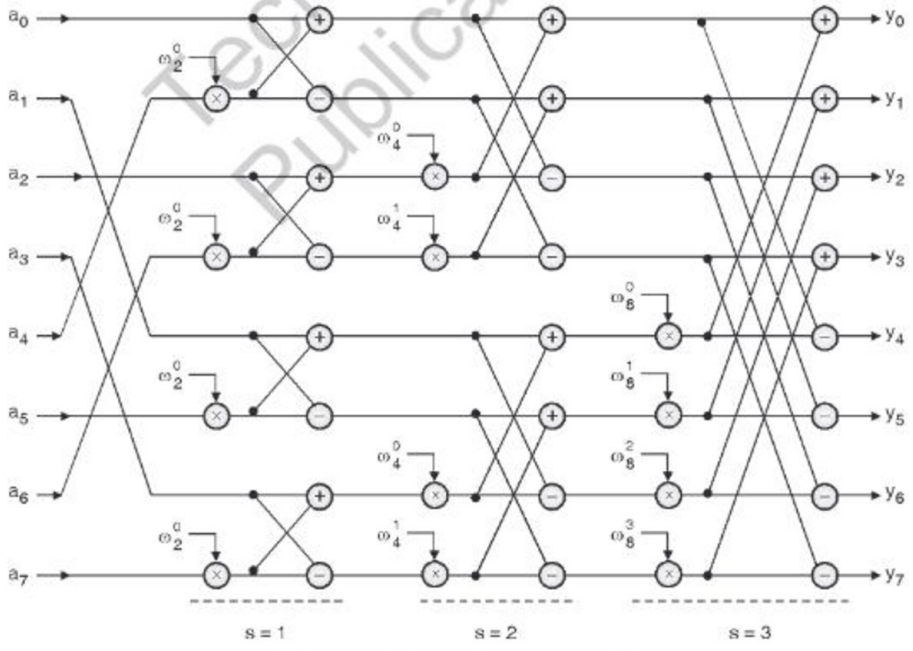


Fig. 4.6.26: FFT on a 2D mesh network

- This is a special topology of interconnection of processors, specially designed to be used for the calculation of FFT. This topology can be implemented using Hypercube or 2D mesh as shown in Fig. 4.6.26.
- The Fig. 4.6.26 shows the implementation of the FFT on a 2D mesh network, with eight processors in each stage for the 8-point FFT.
- Each processor in this case performs a computation for the three stages as shown in the Fig. 4.6.26. The multiplication and summation operations are performed by the independent processors and then the result is forwarded to the corresponding processor as connected in the network.

Review Questions

Q. 1 Explain Principles of Parallel Algorithm Design.

Q. 2 Explain the different methods for Containing Interaction Overheads.

Q. 3 Explain Parallel Algorithm Models.

Q. 4 What are the characteristics of Tasks and Interactions ?

Q. 5 Explain Decomposition, Tasks and Dependency Graphs.

Q. 6 Describe in brief Data-decomposition and Recursive decomposition.

Q. 7 What are the Characteristics of Tasks?

Q. 8 What is Mapping Techniques for Load Balancing?

Q. 9 Describe in detail for Methods for Containing Interaction Overheads.

Q. 10 Explain in detail Parallel Algorithm Models.

❑❑❑

CHAPTER 5

Module V

Performance Measures

Syllabus

Performance Measures : Speedup, execution time, efficiency, cost, scalability, Effect of granularity on performance, Scalability of Parallel Systems, Amdahl's Law, Gustafson's Law, Performance Bottlenecks

5.1 Performance Measures and Metrics of Parallel Processors

There are various parameters that are used to measure the performance of a parallel system. We will see these parameters in this section.

1. Sequential execution time
2. Parallel execution time
3. Speed-up
4. Efficiency
5. Clocks Per Instruction (CPI)
6. Million Instruction Per Second (MIPS)
7. Million Floating point Instructions per second (MFLOPS)
8. Throughput
9. Scalability
10. Cost

1. Sequential execution time

The time required for a program to be executed on a sequential (uni-processor) system is called as the sequential execution time with respect to parallel processors. It is represented by T(1).

2. Parallel execution time

The time required for a program to be executed on a n-parallel processor system is called as parallel execution time for 'n' processors. It is represented as T(n), where 'n' is the number of processors.

3. Speed-up

The speed increase because of the parallel system compared to the uni-processor system is called as the speed up. It is the ratio of the speed of parallel system to that of the sequential system. It can also be given as the ratio of time required to execute a program on sequential system to that of the parallel system (since time is inverse of frequency or speed). It is a very important metric to measure the performance of a parallel system. It is represented as S(n) and given as below :

$$S(n) = \frac{T(1)}{T(n)} \qquad \ldots(5.1.1)$$

4. Efficiency

- Efficiency of a parallel system is the ratio of the actual speed-up obtained by a system to the ideal speed-up that should be achieved according to the number of processors in the parallel system. The ideal time required to execute a program using 'n' processors should be T(1)/n i.e. the time required should be 1/n of the time required on a sequential or single processor system. Thus the efficiency (η) can be given as below :

$$\eta \text{ or } E(n) = \frac{\text{Actual speed – up}}{\text{Ideal speed – up}} = \frac{T(1)}{nT(n)} \quad \ldots(5.1.2)$$

- Since actual speed-up will be given as 1/T(n) and the Ideal speed-up will be given as n/T(1).

5. Clocks Per Instruction (CPI)

- This is as the name says, a measure of the clock pulses require per instruction. It is the ratio of the clock cycles required for a program to the number of instructions in the program. The time for one clock pulse is given as 'τ' and is the inverse of frequency (f). Let the number of instructions in the program be 'Ic', thus the time required to execute a program (T) can be given as :

$$T = I_c \times CPI \times \tau \quad \ldots(5.1.3)$$

- This is because there are I_c instructions and CPI is the clock cycles for one instruction. Thus $I_c \times CPI$ is the total number of clock pulses required to execute the program. The time for one clock pulse is τ, thus the total time required to execute the program will be as given in Equation (5.1.3).

6. Million Instruction Per Second (MIPS)

- This is a very widely used performance measure. As the name says it is the count of instructions executed per second in millions. For example, if a system has 5 MIPS, it means it can execute 5 million instructions in a second.
- Let us get an equation to find the value of MIPS. The time required to execute one instruction is $CPI \times \tau$. Thus the number of instructions executed in one second is :

$$\text{Instructions per second} = \frac{1}{CPI \times \tau} \quad \ldots(5.1.4)$$

- Thus, the MIPS count of instruction can be given as:

$$MIPS = \frac{1}{CPI \times \tau \times 10^6} \quad \ldots(5.1.5)$$

- We have simply divided the Instructions per second in Equation (5.1.4) by 106 i.e. 1 Million to get Million Instructions per second (MIPS). This equation can be written by replacing τ by 1/f. Also if 'C' is equal to total clock pulses to execute a program i.e. $C = CPI \times I_c$; then CPI can be given as $CPI = C/I_c$. With these replacements, the new expression for MIPS will be:

$$MIPS = \frac{f \times I_c}{C \times 10^6} \quad \ldots(5.1.6)$$

7. Million Floating point Instructions per second (MFLOPS)

This is similar to the MIPS, only the difference being here floating point instructions are taken into account. The same equations will work for MFLOPS, if the instruction count and CPI are replaced according to floating point instructions

8. Throughput

The throughput of a system is defined as the number programs executed per unit time. This is represented as W_s, and is given as below:

$$Ws = \frac{\text{Number of programs}}{\text{Time in seconds}} \quad ...(5.1.7)$$

9. Scalability and its significance

- A parallel system is said to be scalable if the same efficiency is obtained by increasing the number of processor. Since the efficiency is dependent on the number of processors, and it normally keeps on decreasing with the increase in the number of processors.
- In Equation (5.1.2), there is a term 'n' i.e. the number of processors in the denominator. Increase in this value of 'n' should not reduce the efficiency. To do this the value T(n) should reduce in the same proportion as that of the increase in 'n'. This will make the system to be scalable.
- In the following section we will see the scalability performance measure using various laws.

10. Cost

- The cost of a parallel algorithm or program is,

  Cost = Parallel running time × number of processors ($T_p \times p$)
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of $\Theta(1)$.
- Since $E = T_s / pT_p$, for cost optimal systems, $E = \Theta(1)$.
- Cost is sometimes referred to as work or processor-time product.
- **For Example :** Consider the problem of adding n numbers on n processors.
  - We have, $T_p = \log n$ (for $p = n$).
  - The cost of this system is given by $pT_p = n \log n$.
  - Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.

5.2 Effect of Granularity on Performance

- Granularity is the amount of work between cross-processor dependences.
- Generally, larger grain is better and fewer interactions, more local work this can lead to load imbalance.
- If the granularity is too fine, the performance can suffer from the increased communication overhead.
- On the other hand, if the granularity is too coarse, the performance can suffer from load imbalance.
- Often, using fewer processors improves performance of parallel systems.
- If we want to discuss granularity then we have to discuss scaling down concept.

- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called **scaling down a parallel system (downsizing)**.

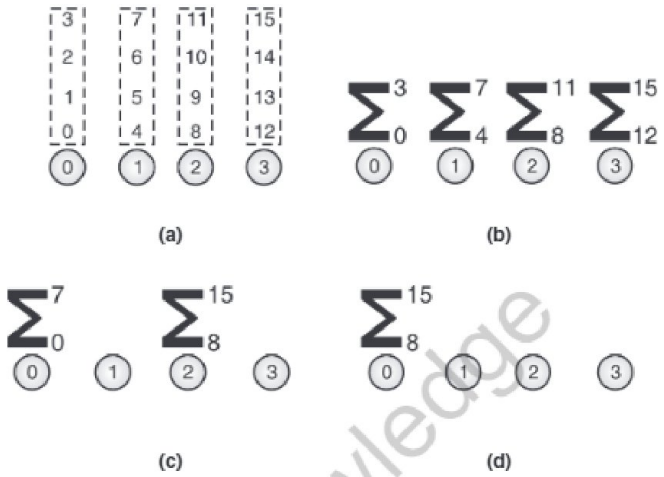


Fig. 5.2.1

- A simple technique of scaling down is to think of each processor in the original case as a virtual processor and to assign these virtual processors equally to scale down the available processors.
- Since the number of processing elements decreases by a factor of n/p, the computation at each processing element increases by a factor of n/p.
- The communication cost should not increase by this factor since some of the virtual processors assigned to physical processors might talk to each other. This is the basic reason for the improvement from building granularity.
- Consider example of building granularity

  Consider the problem of adding n numbers on p processing elements such that p < n and both n and p are powers of 2.
- Each of the p processors is now assigned **n / p** virtual processors because we assume the use of virtual processors here. The first **log p** of the **log n** steps of the original algorithm are simulated in **(n / p) log p** steps on **p** processing elements. Subsequent **log n – log p** steps do not require any communication.
- The overall parallel execution time of this parallel system is : $\Theta((n / p) \log p)$.
- The cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding **n** numbers sequentially. Therefore, the parallel system is not cost-optimal.
- Can we build granularity in the example in a cost-optimal fashion?
- Each processing element locally adds its **n / p** numbers in time $\Theta(n/p)$. The p partial sums on p processing elements can be can be added in time $\Theta(n/p)$. Fig. 5.2.1 describes a cost-optimal way of computing the sum of 16 numbers using four processing elements.

5.3 Scalability of Parallel Systems

- Scalability is another important measure of performance for a parallel algorithm.
- Algorithms are said to be scalable if the level of parallelism increases at least linearly with size of the problem.
- The architecture to implement an algorithm is scalable if it continues to yield the same performance per processor, even if the problem size increases and also the number of processors increase.
- It is seen that data parallel algorithms are more scalable as compared to the architecture scalable algorithms.
- **A parallel computer system is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size.**
- Scalability of a parallel system is the measure of its capacity to increase speedup in proportion to the machine size.
- Increasing the number of processors decreases the efficiency. And increasing the amount of computation per processor, increase the efficiency.
- To keep the efficiency fixed, both the size of problem and the number of processors must be increased simultaneously.

5.3.1 Isoefficiency Metric of Scalability

1. Isoefficiency function

- The isoefficiency function can be used to measure scalability of the parallel computing systems.
- It shows how the size of problem must grow as a function of the number of processors and in order to maintain some constant efficiency.
- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.
- The isoefficiency function does not exist for unscalable parallel computing systems. This is because the efficiency in such systems cannot be kept at any constant value as machine size increases, no matter how fast the problem size is increased.
- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?
- This rate determines the scalability of the system. The slower this rate, the better.
- Before we formalize this rate, we define the problem size W as the asymptotic number of operations associated with the best serial algorithm to solve the problem.
- We can write parallel runtime as,

$$T_p = W + T_o(W, p) / p$$

- The resulting expression for speedup is,

$$S = W / T_p = \frac{W p}{W + T_o(W, p)}$$

- A large isoefficiency function indicates a poorly scalable system.

5.4 Principles of Scalable Performance

There are some laws that govern the performance of the parallel processing system. These laws will be studied in this section.

5.4.1 Amdahl's Law

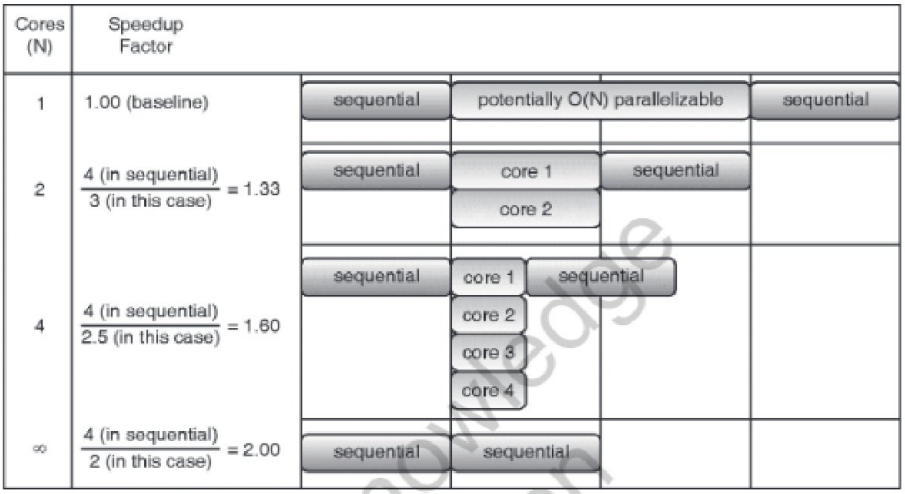


Fig. 5.4.1 : Amdahl's law

- Let the time requires for executing a task on a sequential system is t_s. If the 'f' fraction of the task is serial and (1 – f) is non-serial or parallel, then we can make the second part work in parallel on multiple processors but the first part has to work serially.
- For example if a part of code is serial, it has to always work serially on whatever number of processors be there. The part that is parallelizable will require lesser and lesser time when the number of processor increases. Thus the Speed-up also keeps on increasing.
- This can be explained with the Fig. 5.4.1. In this figure, four time units as shown are required to execute the program in case of a sequential system. When there are two processors, the sequential time remains the same, while the parallelizable time requires half the time i.e. one time unit, as there are two processors. When the number of processors are 4, the time required further halves to just a half time unit.
- Thus the speed-up for n-processors according to the Amdahl's law can be given as below :

$$S(n) = \frac{t_s}{ft_s + (1-f)\,t_s / n} \qquad \ldots(5.4.1)$$

where, t_s is the total time required on sequential system

- f is that fraction of the code which is sequential, hence 1-f is the parallelizable part of the task and 'n' is the number of processors.

- Equation (5.4.1) is called as Amdahl's law. The limitation of Amdahl's law is shown in Fig.5.4.1, the last case. If we keep on increasing the number of processors to infinity, the speed-up cannot go beyond 1/f (2 in this case), as 'f' is the fraction of the code which is serial. The graph for the speed-up factor vs. number of processors 'n' for Amdahl's law is as shown in Fig. 5.4.2.

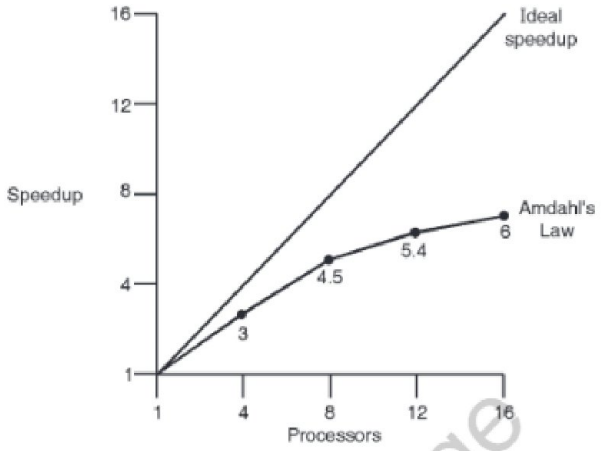


Fig. 5.4.2 : Speed-up vs number of processors for Amdahl's law

5.4.2 Gustafson's Law

- Gustafson law overcame the drawback of the Amdahl's law. Gustafson relaxed the problem size from being fixed to be of any size. Gustafson said that instead of having a fixed problem size or fixed workload we must assume that we have a fixed execution time.
- This is because in case of huge problem size, we will need to increase our system size i.e. number of processors instead of the execution time.
- The time for execution for whatever huge the workload is, the execution time must be fixed. According to this the speed-up factor will be numerically different as compared to the Amdahl's speed-up factor, and hence this is termed as scaled speed-up factor (S'(n)).
- Thus in case of Gustafson's law the execution time is fixed. Thus let the serial execution time be 's' and the parallel execution time be 'p' for 'n' processor system. Thus total execution time is s + p. If the execution is to be done on a sequential system, the execution time will be hence s + np. Thus the scaled speedup factor can be given as below :

$$S'(n) = \frac{s + np}{s + p} = \frac{s + np}{1} \quad \ldots(5.4.2)$$

assuming the time required on parallel system is 1 i.e. s + p =1.

Thus, $S'(n) = \frac{s + n(1 - s)}{s + (1 - s)}$ (since, s + p = 1, therefore p = 1 – s)

$$= \frac{n + s(1 - n)}{1}$$

$$S'(n) = n + s(1 - n) \quad \ldots(5.4.3)$$

- This equation is called as Scaled speed-up factor of Gustafson's law. The graph of speedup factor vs. sequential part of the task can be shown as in Fig. 5.4.3. Fig. 5.4.3 shows that there is no effect of the sequential part on the speedup factor of a system.

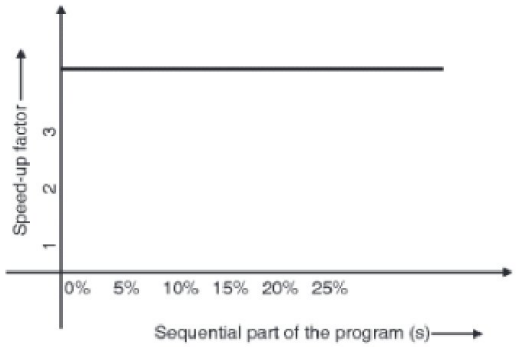


Fig. 5.4.3 : Speedup vs. Sequential part in the task

5.5 Performance Measure and Bottleneck in Parallel Algorithms

- The analysis of an algorithm mainly involves two things, namely
- **Space complexity :** This parameter corresponds to the space requirement in the memory for the particular program
- **Time complexity :** This parameter corresponds to the time required to perform the given algorithm. The time of the execution increases as the size of problem (array size) increases. Hence comparing two algorithms based on time or the number of instructions doesn't give an accurate performance measure of the algorithm.
- The only solution to this is to measure the time as a function of input size 'n'. Such an analysis is independent of machine time, programming style, etc.
- Let us take an example to measure the performance in this manner. We need to find the cost associated with each step in the algorithm and then calculate the total cost.

Algorithm 1	Cost	Algorithm 2	Cost
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...		...	
arr[N – 1] = 0;	c_1		
	---		---
$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$	

Algorithm 3	Cost
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$	

- Another very important method of measuring the performance of an algorithm is called as **Asymptotic Analysis.** In this case the measure of performance is done on the basis of how fast the function grows with respect to the problem size or the array size.

- There are various notations used for the asymptotic analysis, as listed below :
- O notation: asymptotic "less than":

  $f(n) = O(g(n))$ implies: $f(n)$ " < " $g(n)$
- Ω notation: asymptotic "greater than":

  $f(n) = \Omega(g(n))$ implies: $f(n)$ " > " $g(n)$
- Θ notation: asymptotic "equality":

  $f(n) = \Theta(g(n))$ implies: $f(n)$ " = " $g(n)$
- **Big-O Notation:** It is a very widely used notation. It is derived from the relationship of the time taken to execute the algorithm with respect to the size of the array on which the algorithm is to be executed. It indicates the degree of growth of the algorithm with increase of array size.
- We will see some examples of this notation below :

  $f(n) = 30n + 8$ is order n, i.e. O (n) means proportional to n.

  $f_B(n) = n^2 + 1$ is order n^2, i.e. $O(n^2)$ means proportional to n^2.
- This means $O(n^2)$ function is growing faster than any O(n) function.

  $3n^4 + 10n^2 + 50$ is $O(n^4)$

  $n^3 + 230n^2$ is $O(n^3)$

  10 is O(1)
- Let us take the same example again and find the performance according to the asymptotic analysis for which we have done time cost analysis.

Algorithm 1	Cost	Algorithm 2	Cost
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...			
arr[N – 1] = 0;	c_1		
	---		---
$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$	

- Both algorithms are of the same order: O(N)

Algorithm 3	Cost
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$	

- Whenever we have to design a parallel algorithm, we need to begin with decomposing the problem into small concurrent tasks.

- This can be done in different manner. The degree on concurrency is the number of tasks that can be executed in parallel. The degree of concurrency increases as the decomposition becomes finer in granularity i.e. the finer the concurrent tasks, more is the concurrency possible.

Scalability

- This is another important measure of performance for a parallel algorithm.
- Algorithms are said to be scalable if the level of parallelism increases at least linearly with size of the problem.
- The architecture to implement an algorithm is scalable if it continues to yield the same performance per processor, even if the problem size increases and also the number of processors increase.
- It is seen that data parallel algorithms are more scalable as compared to the architecture scalable algorithms.
- The performance of a parallel algorithm is also measured with various parameters listed below :

1. $$\text{Speedup} = \frac{\text{Worst-case running time of faster known sequential algorithm}}{\text{Worst-case running time of parallel algorithm}}$$

2. $$\text{Cost} = (\text{Parallel running time}) \times (\text{Number of processors used})$$

3. $$\text{Efficiency} = \frac{\text{Worst-case running time of fastest known sequential algorithm}}{\text{Cost of parallel algorithm}}$$

$$= (\text{Speedup}) / (\text{Number of used processors})$$

$$\leq 1$$

Review Questions

Q. 1 Discuss the various performance measures and metrics used for performance evaluation.

Q. 2 State and prove Amdahl's law. List different scalability metrics

Q. 3 Discuss term scalability. Explain Amdahl's Law for speedup performance.

Q. 4 Explain following terms with respect to parallel processing.

(i) Sequential execution time

(ii) Parallel execution time

(iii) Efficiency

(iv) Speedup

(v) Scalability

Q. 5 Interpret the Effect of Granularity on Performance parallel execution.

❑❑❑

CHAPTER 6

Module VI

HPC Programming

Syllabus

Programming Using the Message-Passing Paradigm : Principles of Message Passing Programming. **The Building Blocks :** Send and Receive Operations

MPI : The Message Passing Interface, Topology and Embedding, Overlapping Communication with Computation, Collective Communication and Computation Operations, Introduction to OpenMP

6.1 Parallel Programming Techniques

We have processors, memory and I/O devices in a system. The processors are considered as active devices while memory and I/O devices are considered as passive devices. Process is the basic computational unit in a multi processor system, where each processor operates on a processor. The programming model for a parallel system is required especially for inter-processor communication. There are various programming models to help this inter-processor communication; some of them are seen in the subsections below.

6.1.1 Shared Memory Parallel Programming

1. Restricted access to the memory locations for the different processors i.e. those data that are to be accessed by only some particular processors must be restricted from being accessed by all other processors.
2. Synchronization amongst these processor so as to implement mutual exclusion i.e. only when the processor that produces the data has updates the result in the memory, then the processor that has to access this data must access this memory location so as to read the same.

– The shared memory programming system looks as shown in Fig. 6.1.1.

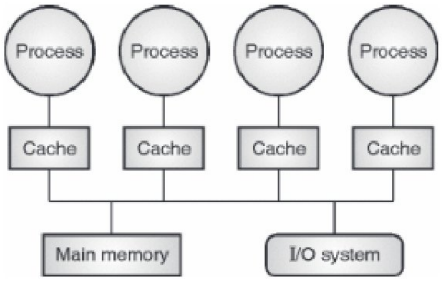


Fig. 6.1.1 : Shared memory architecture for inter process communication

– The main issues in shared memory programming are access of critical sections which has to be protected, maintaining memory consistency, synchronization fast data movements etc.

– Critical Section of a program is that part of the memory which accesses a shared variable from the memory, which must be synchronized such as to allow only one processor accessing the data at any given time.

- For implementing such a system, we need to use mutual exclusion. Mutual exclusion semaphore is a single bit, which is to be first read to check the availability of the resource (the shared variable), then if the resource is available the semaphore bit is toggled to indicate that it is not available any more.
- Thereafter any other authorised processor tries to access this resource it finds the semaphore indicating that the resource is not available and hence waits for it to be available. Once the previous processor that got the access of this shared resource, completes its operation, the semaphore is again toggled making it available for other processors.
- The **Lock()** method is used in parallel programming languages to lock the shared variable so that no other processor can access the same data simultaneously.
- Certain care have to be taken during the execution of this critical section of the code. There should not be a deadlock i.e. a circular wait by two processors trying to access the same resource. Also no interrupt is to be accepted or acknowledged during the execution of the critical section of the code. No other processor's request to access the resources should be accepted during the execution of the critical code section.
- Thus the **features** of the shared memory parallel programming can be listed as below :
  1. Shared memory parallel computers generally have the ability for all processors to access all memory as global address space.
  2. Multiple processors can operate simultaneously and independently and still share the same memory resources.
  3. The changes in a memory location that are done by one processor are visible to all other processors.

6.1.1(A) Fork / Join Parallelism

- Fork / Join is a mechanism to divide the huge task into smaller tasks and distribute these tasks amongst different processors.
- If the huge tasks are divided into smaller independent tasks they will run concurrently or in parallel until they are independent.
- Let us take an example to understand this. We will find the largest of 'n' numbers in a given huge array. This array must be stored in the shared memory, so as to allow all the processors to access the array. We will divide the task into two smaller tasks.
- Let us assume a method compute is used to do this operation.

Example 6.1.1 :

```
int compute
{
    int mid = start + (end - start) / 2;    // find the centre element
                                            // create the subtasks passing the array for them and the starting and ending indices
    left = new Task(array, start, mid);
    right = new Task(array, mid, end);
                                            // start both the parallel tasks
    left.fork();
    right.fork();
                                            // wait for parallel tasks to complete, and get their results
```

```
    int leftMax = left.join();
    int rightMax = right.join();
                                        // find and return overall max from the max of sub-ranges
    return maximum of leftMax, rightMax;
}
```

- In this example we have created a method called as compute(), which is supposed to find the largest number in the given array.
- The mid-point of the array is first found and the two parts of the array i.e. the left and right are created by passing the array and the start and end elements for the two tasks. The left task is passed with the start and end indices as start and mid-point indices of the array; while the right task is passed with the start and end indices as the mid-point an end indices of the array.
- The fork() method is used to start or spawn a new task. Hence using the fork() method we have started the left and right tasks.
- Then before going to the last step i.e. finding the maximum from the maximum of the two parts of the array, we need to wait for the two tasks "left" and "right" to be completed. The join() method ensures this. The join() method is used for both the tasks i.e. "left" and "right", ensuring both the tasks to be completed before proceeding to the next step.
- Finally, once the two tasks are completed the largest of the two results obtained from the two sub-task is found and returned.

6.1.2 Principles of Message Passing Programming

- This message passing may be a synchronized operation or may be asynchronized operation.
- There may be special methods to send and receive the messages with synchronizing signals in case of synchronous message passing, while there will be buffers required without any synchronizing signals.
- Let us see the two methods in the following sub sections.

6.1.2(A) Synchronous Message Passing Building Blocks: Send Receive Operations

- Unlike the shared memory, there is no requirement of any common memory for the two processors that require to share the message.
- Also, since there is no shared memory, there is no need of mutual exclusion. Here there has to be a special connection path between the two processors that want to communicate.
- Only one message can be sent at any given time. This communication can have only one sender at any given time but one or more receivers.
- There must be a dedicated channel which is used to pass the message from one processor to another or there may be a common channel for transmitting messages from one processor to another. The later one is used quite widely because of its cost effectiveness. Hence there may be a case wherein the channel requestor has to wait because the channel is busy. Hence it is a blocking method of communication.
- A proper protocol must be used so as to allocate the channel to the right sender-receiver pair so as to ensure no loss of time for the processors waiting for a message. If a processor cannot proceed without a message from another processor then it will have to wait for that processor and also for the channel to be free.

- The sender and receiver processes must be synchronized in space and time. Space communication refers to the availability of the communication channel while the time synchronization refers to the receiver being ready when the sender sends the message.
- The parallel programming languages "Ada" and "Occam" use Synchronous message passing communication model.
- In case of "Ada" a name addressing method is used, which has the node and processor ID to identify the node. In case of "Occam" a one way communication is established between the sender and receiver. "Occam" will be discussed in the section 6.5.3.
- The Send() and Receive() methods are used to perform these operations of sending from the sender and receiving by the receiver respectively. They are used to properly synchronize the message passing protocol.
- Fig. 6.1.2 shows an example of synchronous and asynchronous example in real life.

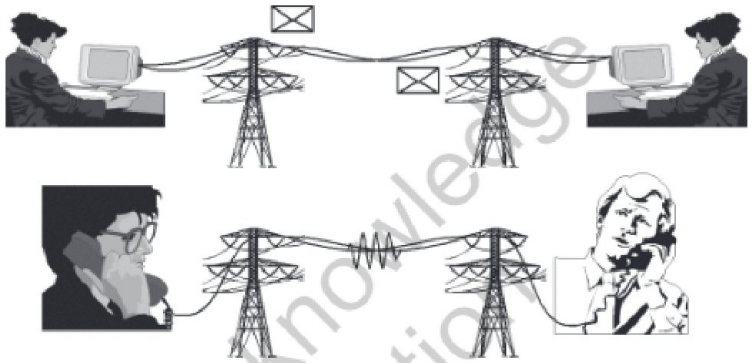


Fig. 6.1.2 : Synchronous and asynchronous message passing

- In case of synchronous message passing the channel is continuously available between the sender and the receiver. Also that both the sender and receiver must be available at the time of message passing. The telephone call is an example of such a case i.e. both the caller and the receiver must be present at the time of message passing for the communication to happen. Also, if the receiver of or sender is busy, the other person has to wait i.e. the communication can be blocked.
- In case of say SPMD (Single Program Multiple Data) system, there is communication to be done among the various processors performing the same operation on multiple sets of data. The result of each processor has to be forwarded to a particular processor to perform the further operation on that result. There are various algorithms that require to perform the message passing of such type, studied in chapter 4.

6.1.2(B) Asynchronous Message Passing Building Blocks: Send Receive Operations

- In this case there is a buffer between the sender and the receiver and hence the sending is never blocked.
- Whenever the sender wants to send a message, it puts the message into the buffer and the receiver takes the message whenever it is free.
- This message sending protocol is not synchronized with time and space. Thus the sender and receiver need not be available at the same time for the communication to happen. Also the sender need not wait for the channel to be free. Whenever it has to send a message the message is put into the buffer and the receiver will get it whenever the communication channel is free.

- The best example of this kind of communication in real life is again shown in Fig. 6.1.2 i.e. sending a mail or post. The E-mail when sent is put into the buffer, and the receiver need not be online at that time. Whenever the channel is free the mail is sent from the buffer, and when the receiver comes online, he gets the message.
- Similarly, in case of asynchronous message passing protocol, there is a buffer in the channel in which all the messages are stored by the sender processors and the receiver processors are given these messages one by one. There is no synchronization required as the sender need not bother about the message being received or not. It will automatically be handled by the communication channel buffers.
- Although we say that it is a non-blocking system, but it may be blocked in case if the buffer is full. Hence we need to increase the buffer size, so as to have a non-blocking type of communication.
- Asynchronous message passing protocols have advantage of having shorter delay time and better resource utilization. Hence many advanced systems use asynchronous type of message passing protocols.
- The "Linda" programming system used asynchronous type of message passing. It has a global mail-box, and whenever a process has to send a message the message is put into this global mailbox. The receivers are given the messages meant for them as and when the message comes in a FIFO manner. "Linda" will be studied in the 6.5.2.
- Here also we have the send() and receive() methods, to send the data and to receive the data respectively. Along with the data, the channel and the processor id are also to be sent. The data is stored in the buffers of the channel as discussed earlier and the data is hence said to be transferred asynchronously
- Let us see the comparison of the synchronous and asynchronous methods of message passing.

Sr. No.	Synchronous message passing	Asynchronous message passing
1.	There is a bound on the size of communications channels. (buffer space)	There is no bound on the size of communications channels. (buffer space)
2.	A process can have at most one message a time queued up on any channel.	A process can have any number of messages at a time queued up on any channel.
3.	Not until the message is received, can the sending process continue and send another message. Hence coherency is maintained.	Another message can be sent or received even if previous message is not sent. Special care is to be taken for coherency.
4.	Concurrency is reduced. When two processes communicate, at least one of them will have to block.	Concurrency is more as there are multiple channels for simultaneous transmission.
5.	Programs are more prone to deadlock. The programmer has to be careful that all send and receive statements match up.	There is no scope of deadlock.

6.1.3 Data Parallel Programming

- Data parallel algorithms are mainly of importance in SIMD computers. SIMD (Single Instruction Multiple Data) processors have single instruction that have to operate on multiple data. In our case, we will have multiple processors to whom the data will be distributed for the same operation to be performed. Since the processors are doing the same operation but on different data, the processors can normally operate independently.
- Data parallel algorithm is quite often also referred to as loop based parallelism, as in this case the multiple iterations of a loop are executed simultaneously.

- Data parallel programs mainly work by distributing the huge data into smaller sets. Hence a loop operating on an array can be divided into different independent parts and the different processors operating on their part of the data.
- Let us take a simple example of multiplying the corresponding elements of two matrices of sizes m × n.

```
do i=1,nm
    do j=1,n
        a(i,j) = b(i,j) * c(i,j)
    end do
end do
```

- Here the compiler can easily find out the independence of each of the iteration. Here by independence of each iteration it means that no iteration reads or writes the variables of another iteration.
- A program may also require communication amongst the processor depending on the operations required. In such case the independent part is executed concurrently or in parallel by all the processors and then for the sequential parts the communication will be done.
- Let us take an example of this.

```
do i=1,nm
    do j=1,n
        a(i,j) = b(i,j) * c(i,j)
    end do
end do
result = Sum(a)
```

- Thus in this case, the first part is same as discussed above i.e. finding the product of the corresponding elements of a matrix. Next, all the elements of the product matrix are to be added and the sum is the final result. In this case a communication will be required with all the processors to calculate the final result or the sum.
- Locality of the data to be operated on is very important for data parallel algorithms. If the data for the above matrices to be multiplied is in one processor then communication will be required for all operations. Hence it is required to distribute the data blocks to the different processors. The Distribute directive performs this operation. For example if we have 16 processors and we want to distribute to these 16 processors, we can do it in the following manner.

  Processors p (16)

  Distribute b onto p
- Here, 'p' is a variable that holds the number of processors and the array 'b' is distributed on these 'p' processors. These directives discussed above are for the Fortran 90 parallel programming language. These programming languages will be seen in the section 6.5.1.
- Fortran 90 or F90 allows a lot of operations or constructs that are to be performed on each element of an array. Some of the examples are given below (assuming arrays a and b) :

  1. a = b + c ; will add a scalar value 'c' to each element of the array 'b' and put the result in the corresponding element of array 'a'.
  2. a = a + 1.0 ; will add the value one to each element of the array 'a'.
  3. a = sqrt (a); will find and replace each element of 'a' with the square root of the corresponding element.

The features of the data parallel programming model can be as given below :

1. In this case, most of the parallel work is focused on performing operations on a data set.

2. The data set on which the operation is to be done is typically organized into a common structure, like an array or cube.
3. Multiple tasks work collectively on the same data structure, but each task works on a different part of the same data array.
4. Multiple tasks perform the same operation on their part of data, for example, "add 1 to every array element".

6.2 Introduction to Parallel Programming

- We have already discussed quite a few parallel algorithms in chapter 4. In this section we will discuss some concepts of parallel algorithms.
- Parallel algorithms are mainly meant for executing the concurrent part of the algorithm simultaneously by different processors. The final output of the different processors are finally collected together and the result is obtained.
- It is easy for some algorithms to be divided into parts that can be concurrently executed, and hence achieve a high level of parallelism. For example if the numbers from 1 to 100 are to be checked, to find which are prime and which are not. If one processor is doing this operation it will take a long time, as it can check only one number at a time. But if we have 100 processors, we can divide the task amongst 100 processors, each checking one number to be prime or not. Hence this operation can be done concurrently.
- But there are quite a few operations that cannot be done concurrently. In this case, we need to find the concurrent operations, or else the algorithm has to be executed on one of the processors of the so many processors in the multiprocessor system. Most of the operations in the numerical methods are iterative, with each iteration dependent on the previous iteration. Hence they cannot be executed concurrently.

6.3 Parallel Algorithms for Multiprocessors

- An *algorithm* is a finite set of steps to perform an operation or for solving a problem.
- The algorithms can be described in English, pseudo-code or diagrams.
- An algorithm can have zero to many quantities as input and also generate one or many outputs.
- A very important job while writing algorithms is to write the algorithm such that it is independent of the programming language.
- Pseudo-code is mostly used to do this, wherein the program is better structured compared to the simple English statements but not exactly structured a programming language statements
- We will consider an example to find the maximum element of an array. This algorithm is a pseudo code, however it is for a single processor and not parallel system.

```
Algorithm arrayMax(A, n):
        Input:  An array A storing n integers.
        Output: The maximum element in A.
        currentMax <-- A[0]
        for i <-- 1 to n -1 do
        if currentMax < A[i] then currentMax <-- A[i]
        return currentMax
```

- In pseudo Code the standard expressions used are "=-" for assignment as well as to check equality. The programming constructs if.....then.....else, while.....do, repeat....until, for.....do, etc

- For designing parallel algorithms the following steps are to be followed

  Step 1 : We need to find the pieces of work or tasks that can be done concurrently.

  Step 2 : Then these tasks are to be mapped onto multiple processors i.e. we need to map the processes vs processors

  Step 3 : The distribution of input/output & intermediate data across the different processors is also to be considered as this requires a lot of extra time.

  Step 4 : Thus we also require the management of the accesses of shared data.

  Step 5 : Finally the synchronization of the processors at various points of the parallel execution is also to be done

- The most important things to be considered while making parallel algorithms is to maximize the concurrency, reduce the overheads due to parallelization and hence maximize the speedup.

6.3.1 Characteristics of Parallel Algorithms

The following are the characteristics to be considered for a parallel algorithm

1. **Computational granularity :** If the algorithm can be divided into smaller parts to be executed concurrently, they can be executed faster. Hence a very important characteristic of parallel algorithms is granularity of computations
2. **Communication :** Inter processor (processor element) communication is required in most of the parallel algorithms. Besides communication is also required with memory. The communication method could be static or dynamic. Static communication methods are normally used for 'communication amongst SIMD processors while dynamic in=s required in MIMD processing systems.
3. **Uniformity of operations :** It is another important characteristic of parallel algorithm. There must be some operations that are uniform and hence can be distributed over a huge number of processors and all doing the uniform operation. Hence SIMD (i.e. single instruction and multiple data) can be used for uniformity of operations. If the operations are not uniform enough, then the MIMD systems can be used.
4. **Synchronization :** When communicating, the processors and memory need to synchronize. Only once the processor has completed its part can it forward the result to the next processor. Hence the synchronization is required for parallel algorithm.
5. **Memory requirement :** Normally a huge memory is required for the parallel algorithms. The array of data are to be stored. The input operands are array as well as the result is an array. To store all these arrays a huge memory is required. Hence memory and the data structures used can affect the efficiency of the parallel algorithms.
6. **Parallelism profile :** The distribution of the parallel tasks among the processors decides the degree of parallelism. The more the parallelism possible, more is the effectiveness of the parallel algorithms.

   These are some of the characteristics exhibited by the parallel algorithms.

6.3.2 Classification of Parallel Algorithms

- The parallel algorithms can be classified on various basis. Let us see these methods of classifying the parallel algorithms.

  1. Synchronised Parallel Algorithms
  2. Asynchronous Parallel Algorithm

- The most important thing in a parallel algorithm is the concurrent processes. More the **number of concurrent processes**, more is the parallelism possible. If the number of concurrent processes is just one, then it becomes a sequential algorithm.

- Hence one method of classifying the parallel algorithms is the number of concurrent processes in the algorithm.
- Another method of classifying the parallel algorithms is the method of decomposition used i.e. static or dynamic. The operation to be performed is to be decomposed to be performed on the different processors. This decomposition can be passive i.e. static or it can be dynamic i.e. changed during the time of execution. Based on this we have static and **dynamic decomposition** methods. In case of dynamic decomposition, the decomposition of the task i.e. distribution of the work to the various processors is done during the execution of the algorithm. In case of **static decomposition** method the division of the work for each of the processor is done before the beginning of the execution of the algorithm.
- Another method of classifying the parallel algorithm is **synchronous and asynchronous algorithms**. Those algorithms wherein the processor (or processing element) needs to wait for another processor to complete its operation, i.e. synchronization is required, are said to be synchronized algorithms. While those algorithms wherein, the processors use a shared memory, and after doing their part of the work, they have to simply write the result in the shared memory without waiting for another processors, is called as asynchronous parallel algorithm. In case of the synchronous parallel algorithms the time required for the execution of the processor increases as the waiting time for other processing elements is more.

6.3.2(A) Synchronised Parallel Algorithms

- For a synchronized parallel algorithm, we need to make use of semaphores to get synchronization. Let us see an example of a operation that can be done in parallel, but requires synchronization.
- Suppose the following operation is to be done on matrices A, B, C, D, I and J

$$Z = A \cdot B + (C + D) \cdot (I + G)$$

- We cannot perform the addition of the result of first product and sum of the next two matrices until their product and sum are calculated. This can be better understood by the diagrammatic representation shown in Fig. 6.3.1.

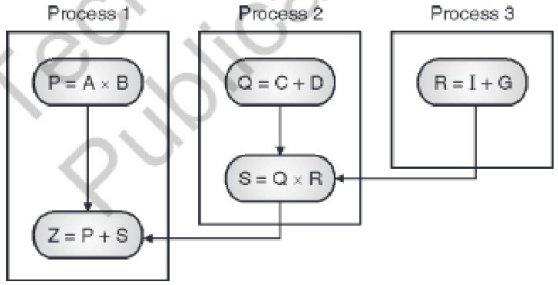


Fig. 6.3.1 : Synchronized parallel algorithm to perform a set of operations

- As shown in the Fig. 6.3.1, there are three processors required to perform the required operation in parallel. Since the multiplication operation will require more time, we have made the other two processors to perform the addition and get the final result to be added to the product of the first two matrices.
- The parallel algorithm for this can be written as shown below.

```
Var R,S:shared real;
Var Sp, Sr, Ss: semaphore;
Cobegin
    process P1          : begin
            P <-- A × B;
```

```
            P ( Sp);
            Z <-- P + S;
            end
      end P1
      process P2          :begin
            Q <-- C + D;
            P(Sr);
            S <-- Q × R;
            P(Ss)
            end;
      end P2
      process P3          :begin
            R = I + G;
            P(Sr);
            end
      end P3
coend
```

- The process 1 performs the multiplication and then waits for the semaphore Sp. At the same time the process 2 performs the addition of matrices C and D, while the process 3 performs addition of matrices I and G. Once the addition is completed by the processes 2 and 3, the process 2 performs multiplication of the two sums calculated earlier. Hence there is a semaphore used Sr, so that the multiplication is done after the completion of the previous addition. Hence the same semaphore is taken by both processes P2 and P3. Finally the process P1 waits for the completion of this multiplication semaphore Ss, and then the final addition is carried out by process P1.

6.3.2(B) Asynchronous Parallel Algorithm

- In case of asynchronous parallel algorithms, the processes must be independent of each other's result and hence achieve parallelism without any wait time.
- For example if there is an operation to be carried out by parallel processing system as shown below :

$$Z = X + Y$$

$$A = B + C$$

To perform this execution we can use the following parallel algorithm

```
Cobegin
      process P1:         begin
            Z = X + Y;
            end
      end P1
      process P2:         begin
            A = B + C;
            end
      end P2
coend
```

– Here the two processes are independent of each other and hence no semaphore is required. Since no sharing of variables required and no result is to be passed no communication is required.

6.3.3 Performance of Parallel Algorithms

– The analysis of an algorithm mainly involves two things, namely

1. **Space complexity :** This parameter corresponds to the space requirement in the memory for the particular program
2. **Time complexity :** This parameter corresponds to the time required to perform the given algorithm. The time of the execution increases as the size of problem (array size) increases. Hence comparing two algorithms based on time or the number of instructions doesn't give an accurate performance measure of the algorithm.

– The only solution to this is to measure the time as a function of input size 'n'. Such an analysis is independent of machine time, programming style, etc.

– Let us take an example to measure the performance in this manner. We need to find the cost associated with each step in the algorithm and then calculate the total cost.

Algorithm 1	Cost	Algorithm 2	Cost
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...		...	
arr[N – 1] = 0;	c_1		
	----------		----------
	$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

Algorithm 3	Cost
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

	$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2$

– Another very important method of measuring the performance of an algorithm is called as **Asymptotic Analysis.** In this case the measure of performance is done on the basis of how fast the function grows with respect to the problem size or the array size.

– There are various notations used for the asymptotic analysis, as listed below :

(i) O notation: asymptotic "less than":

f(n) = O(g(n)) implies: f(n) " < " g(n)

(ii) Ω notation: asymptotic "greater than":

f(n) = Ω (g(n)) implies: f(n) " > " g(n)

(iii) Θ notation: asymptotic "equality":

f(n) = Θ (g(n)) implies: f(n) " = " g(n)

– **Big-O Notation:** It is a very widely used notation. It is derived from the relationship of the time taken to execute the algorithm with respect to the size of the array on which the algorithm is to be executed. It indicates the degree of growth of the algorithm with increase of array size.

- We will see some examples of this notation below :

  f(n) = 30n + 8 is order n, i.e. O (n) means proportional to n.

  $f_B(n) = n^2 + 1$ is order n^2, i.e. $O(n^2)$ means proportional to n^2.

- This means $O(n^2)$ function is growing faster than any O(n) function.

  $3n^4 + 10n^2 + 50$ is $O(n^4)$

  $n^3 + 230n^2$ is $O(n^3)$

  10 is O(1)

- Let us take the same example again and find the performance according to the asymptotic analysis for which we have done time cost analysis.

Algorithm 1	Cost	Algorithm 2	Cost
arr[0] = 0;	c_1	for(i=0; i<N; i++)	c_2
arr[1] = 0;	c_1	arr[i] = 0;	c_1
arr[2] = 0;	c_1		
...			
arr[N – 1] = 0;	c_1		
	---		---
	$c_1+c_1+...+c_1 = c_1 \times N$		$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

- Both algorithms are of the same order: O(N)

Algorithm 3	Cost
sum = 0;	c_1
for(i=0; i<N; i++)	c_2
for(j=0; j<N; j++)	c_2
sum += arr[i][j];	c_3

	$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N^2 = O(N^2)$

- Whenever we have to design a parallel algorithm, we need to begin with decomposing the problem into small concurrent tasks.
- This can be done in different manner. The degree on concurrency is the number of tasks that can be executed in parallel. The degree of concurrency increases as the decomposition becomes finer in granularity i.e. the finer the concurrent tasks, more is the concurrency possible.

Scalability

- This is another important measure of performance for a parallel algorithm.
- Algorithms are said to be scalable if the level of parallelism increases at least linearly with size of the problem.
- The architecture to implement an algorithm is scalable if it continues to yield the same performance per processor, even if the problem size increases and also the number of processors increase.
- It is seen that data parallel algorithms are more scalable as compared to the architecture scalable algorithms.

The performance of a parallel algorithm is also measured with various parameters listed below :

1. Speedup = $\frac{\text{Worst-case running time of faster known sequential algorithm}}{\text{Worst-case running time of parallel algorithm}}$

2. Cost = (Parallel running time) × (Number of processors used)
3. $$\text{Efficiency} = \frac{\text{Worst-case running time of fastest known sequential algorithm}}{\text{Cost of parallel algorithm}}$$

$$= \text{(Speedup)} / \text{(Number of used processors)}$$

$$\leq 1$$

6.4 Message Passing Libraries for Parallel Programming Interface

- Parallel processing is the method of having many small tasks being distributed over various processing elements so as to solve one large problem.
- The growth of parallel systems is because of the major developments in distributed computing and MPP (Massively Parallel Processing). These are systems with few hundreds to few thousand processors. In such a system with so many processors, it is required by a to have communication among these processors with very high speeds.
- The message passing model is a choice of the various possible ones. There are various message passing libraries for parallel programming interface. Some of them are listed below :

1. Bulk Synchronous Parallel model (BSP)	2. Parallel Virtual Machine (PVM)
3. Message Passing Interface (MPI)	4. PThreads in shared memory

- Let us see these libraries in details.

6.4.1 BSP (Bulk Synchronous Parallel Model)

- This model divides the computations to be performed into supersteps. In each superstep a processor can perform a work on the local data and then send messages.
- At the end of the superstep, a barrier synchronization takes place and all processors receive the messages which were sent to them in the previous superstep
- It has small number of subroutines to implement.
  1. Process creation,
  2. Remote data access, and
  3. Bulk synchronization.

6.4.2 PVM (Parallel Virtual Machine)

- The Private Virtual Machine (PVM) uses a message passing model that allows the programmers to use the distributed computing over a various types of multiprocessing systems including MPP.
- This model derives its name from one major thing in PVM, that is it sees the collection of huge number of computers as one large virtual machine. It can aslo view a huge number of heterogeneous processors as a huge virtual system. A heterogeneous network is one which has multiple computers of different types i.e. different processors having different architecture, different data format, different computational speed, different machine and network loads.

6.4.2(A) Features of PVM

- The computing model of PVM is simple and general.
- The PVM software provides a framework wherein the parallel programs can be developed in a an efficient manner with the existing hardware.

- It transparently handles all message routing, task scheduling and data conversion across a network of heterogeneous computers.
- It accommodates a wide variety of application program structures.
- The user has to write the application as a collection of co-operating tasks. These tasks access the PVM resources through a library of standard interface routines.
- These routines help in initiation and termination of the tasks in the network and also in the communication and synchronization among the tasks.
- The PM message passing mechanism is especially made to handle the heterogeneous network. They also involve constructs for buffering and transmission. Communication constructs have sending and receiving data structures and also special mechanisms for broadcast, barrier synchronization and global sum.
- At any moment any task may start or stop other tasks or add or delete computers from the virtual system or Any process may communicate with any other processor
- PVM has gained widespread acceptance in high performance scientific computing because of its omnipresent nature. virtual machine concept, simple and complete programming interface.

6.4.2(B) PVM System

- This software design is done by a research group working on a research project on heterogeneous network computing.
- The objective of PVM is to enable the collection of heterogeneous computers to be used co-operatively for concurrent computations.
- The main principles on which PVM is implemented are given below :

1. **User Configured Host Protocol :** The computation tasks are to be executed on a set of machines which can be selected by the user. It could be either a single processor or a multiprocessor system with shared memory or distributed memory participating in the pool of host processors to perform the given task. This pool of host processors can be altered during the execution of the program.
2. **Translucent access to hardware :** The application programs can view the entire system as a collection of virtual processing elements without any attributes. It can also use the capabilities of a specific processor in the pool of host processors to allocate most appropriate processor for a given task.
3. **Process based computation :** The task, which is an independent thread is the unit of parallelism in PVM. Independent threads alternate between the tasks of computations and communications. There is no hard and fast process to processor mapping enforced by PVM except for some special tasks that is indicated by the used.
4. **Explicit message passing model :** All the processors that are performing various computational tasks with the help of data, functional or hybrid decomposition of the workload can co-operate amongst themselves by sending or receiving messages. The size of the message is limited by only one thing and that is the memory size, i.e. the message size can be as large as required.
5. **Heterogeneity support :** As already discussed the specialty of PVM is that it supports heterogeneous systems i.e. each computer in the system can be having different configurations in terms of architecture, data type etc. Hence the PVM message passing model permits the passing of message of more than one type amongst the processors connected in the system.
6. **Multiprocessor support :** The PVM uses a native method for passing messages. This takes the advantage of the hardware. Although the vendors provide their own PVM message passing models with the systems optimised for their systems but these can still communicate with the public PVM version.

- The PVM system consists of two parts.
  1. The first is a **daemon (called pvmd3)** that resides on all the computers making up the virtual machine. Another example of a daemon program is the mail program that runs in the background and handles all the incoming and outgoing e-mails on a computer. Pvmd3 is designed so any user with a valid login can install this daemon on a machine. When a user wants to run a PVM application, he has to first create a virtual machine by starting up PVM. The PVM application can then be started from a Unix prompt on any of the hosts. Multiple users can configure overlapping virtual machines, and each user can execute several PVM applications simultaneously.
  2. The second is a **library of PVM interface routines**. It contains a complete functional range of primitives that are needed for communication between tasks of an application. This library contains user callable routines for message passing, spawning processes, coordinating tasks and modifying the virtual machine.
- The PVM computing model is based on the notion that an application consists of several tasks. Each task is responsible for a part of the application's computational workload.
- Sometimes an application is parallelized along its functions, that is each task performs a different function. For example: input, problem setup, solution, output and display. This process is often called functional parallelism.
- A more common method of parallelizing an application is called data parallelism. In this method all the tasks are the same, but each one only knows and solves a small part of the data. This is also referred to as the SPMD (single program multiple data) model of computing. PVM supports either or a mixture of these methods. Depending on their functions, tasks may execute in parallel and may need to synchronize or exchange data, although this is not always the case.
- An example diagram of the PVM computing model is shown in Fig. 6.4.1, that demonstrates the heterogeneity of the PVM.

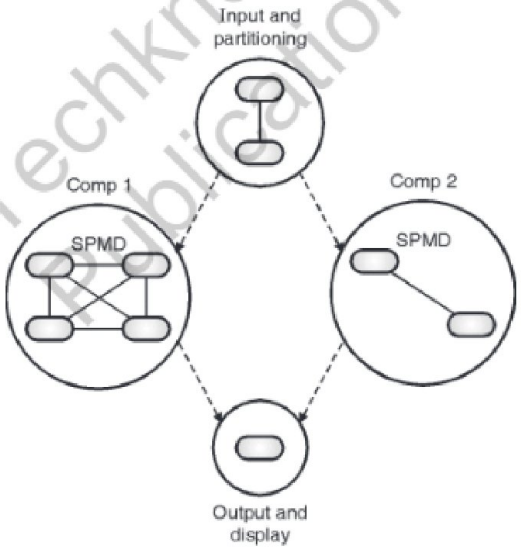


Fig. 6.4.1 : PVM Computing model

- Fig. 6.4.2 shows the architectural view of the PVM with multiple clusters and MPP.

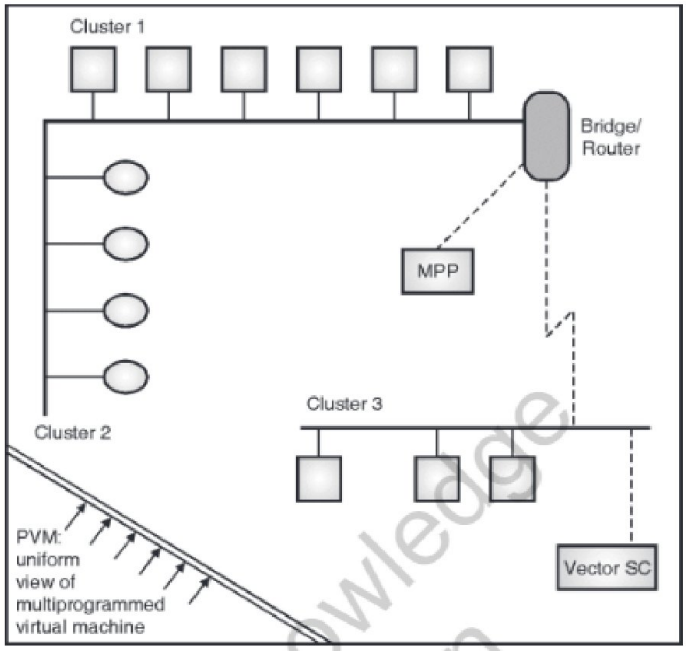


Fig. 6.4.2 : Architectural overview of PVM

- All PVM tasks are identified by an integer task identifier i.e. TID. Messages are sent to and received from these TIDs. Since TIDs must be unique across the entire virtual machine, they are supplied by the local "pvmd" and are not user chosen.
- Although PVM encodes information into each TID, the user is expected to treat the TIDs as opaque integer identifiers. PVM contains several routines that return TID values so that the user application can identify other tasks in the system.
- There are applications where it is natural to think of a group of tasks. And there are cases where a user would like to identify his tasks by the numbers 0 to p-1, where p is the number of tasks.
- PVM includes the concept of user named groups. When a task joins a group, it is assigned a unique number in that group.

Advantages of using PVM

1. Ability to establish a parallel machine using ordinary (low cost) workstations.
2. Ability to connect heterogeneous machines.
3. Use of simple C/Fortran functions.
4. Takes care of data conversion and low level communication issues.
5. Easily installable and configurable.
6. Portable since various message passing libraries available
7. It has a scalable parallelism
8. Fault tolerance by dynamically adding and removing a machine from the host pool.
9. Flexible as the definition and modification of the parallel virtual machine is easily possible

Disadvantages of PVM

1. Programmer has to explicitly implement all parallelization details.
2. Difficult debugging.
3. Performance of the PVM depends on the architecture and may be slightly slower than the native message passing methods
4. It is deficient in some of the message passing functions.

6.4.3 Message Passing Interface (MPI)

- The following is a list of various features of MPI :
  1. It provides communication among multiple concurrent processes
  2. It includes several point-to-point and collective communication methods among groups of processes
  3. It is made up of a library of routines callable from conventional programming languages like Fortran, C, and C++
  4. It has been universally adopted by developers and users of parallel systems that rely on MPI for message passing
  5. MPI closely matches computational model underlying the design methodology for developing parallel algorithms and hence provides natural framework for implementing the parallel algorithms
  6. It is mainly made to work on distributed-memory systems, but also works effectively on any type of parallel system
  7. The performance of MPI is better in most of the cases as it uses data locality
  8. It has more than 125 functions, with various options and protocols. A small subset of this is sufficient for most practical purposes.
- MPI is a library of functions that can be used in various standard languages like Fortran, C, and C++.
- In the this method to parallel programming, many processes execute programs written in a standard sequential language with calls to a library of functions for sending and receiving messages.
- In the MPI programming model, a operation comprises of one or more processes that communicate by calling library functions to send and receive messages to other processes. In most MPI implementations one process is created per processor. However, these processes may execute different programs. Hence, the MPI programming model is sometimes referred to as ***Multiple Program Multiple Data*** (MPMD) to distinguish it from the **Single Processor Multiple Data** (SPMD) model in which every processor executes the same program.
- Processes can use ***point-to-point or broadcast*** communication operations. For point to point communications they can send a message from one named process to another. A group of processes can call *collective* communication operations to perform commonly used global operations such as summation and broadcast.
- There are six functions in MPI system which are very important to send and receive messages. They are :
  1. MPI_INIT() : Initiate an MPI computation.
  2. MPI_FINALIZE() : Terminate a computation.
  3. MPI_COMM_SIZE() : Determine number of processes.
  4. MPI_COMM_RANK() : Determine my process identifier.
  5. MPI_SEND() : Send a message.
  6. MPI_RECV() : Receive a message.

- These six functions are sufficient to write a wide range of parallel programs.
- All these functions except the first two, take a communicator handle as an argument. A communicator identifies the process group and context with respect to which the operation is to be performed.
- A simple program example is given below to understand the library functions.

```
program main
begin
        MPI_INIT()      ;                                   // Initiate computation
        MPI_COMM_SIZE(MPI_COMM_WORLD, count) ;              // Find number of processes
        MPI_COMM_RANK(MPI_COMM_WORLD, myid);                // Find my id
        print("I am", myid, "of", count);                   // Print a message with theID
        MPI_FINALIZE();                                     // Finish
end
```

- The program is self explanatory with the comments. The initialization of the method is done, followed by which the number of processes is found and displayed along with the ID.
- A call to MPI_RECV has the general form

  MPI_RECV(buf, count, datatype, source, tag, comm, status)
- This function attempts to receive a message that has an envelope corresponding to the specified tag, source, and comm, blocking until such a message is available. When the message arrives, elements of the specified datatype are placed into the buffer at address buf. This buffer is guaranteed to be large enough to contain at least count elements. The status variable can be used subsequently to inquire about the size, tag, and source of the received message.
- Let us see the differences between the PVM and MPI message passing libraries

Sr. No.	PVM	MPI
1.	PVM is built around the concept of dynamic collection of computational resources.	MPI has a focus on message passing and states that the resource management and concept of virtual machine are outside the scope of MPI.
2.	PVM has the capabilities to start and stop a task i.e. spawn and kill dynamically.	MPI-1 has no such processes while MPI-2 has capability to start group of tasks and send signal for killing a process.
3.	PVM is inherently dynamic for resource controlling like for load balancing, task migration and fault tolerance.	MPI lacks such dynamics for the resource management.
4.	PVM provides only simple message passing.	MPI has a much richer source of communication methods than PVM.
5.	PVM supports basic fault notification scheme	MPI has no fault tolerance schemes

6.4.3(A) Message Passing Types in MPI

- Hence the message passing supported by MPI are of two types :
  1. Point to point
  2. Collective
- Fig. 6.4.3 shows the example of point to point message passing mechanisms.

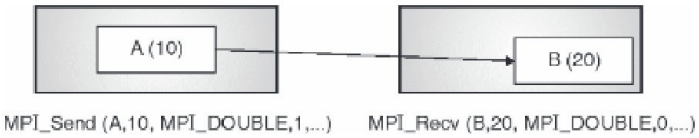


Fig. 6.4.3 : Point-to-Point

- In case of a point to point message passing mechanism the different data-types are supported to have heterogeneity.
- The collective message passing mechanisms can be of various types. For these MPI has various functions for moving the data to different processors according to the required task. There are broadcast, gather and scatter etc. as shown in the Fig. 6.4.4.

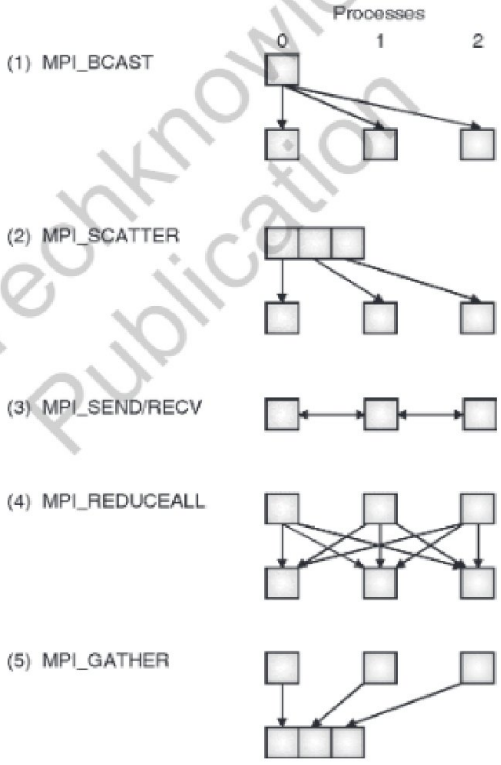


Fig. 6.4.4 : Data movement operations of MPI

- The broadcast (MPI_BCAST) operation transmits the value to all the processes as shown in the Fig. 6.4.5 i.e. A_0 is passed on to all the processors.
- It is used when the data is to be passed on to multiple processes from one of the processes. It is required many times in parallel algorithms when the data is to be passed to multiple processors. Fig. 6.4.5 shows a detailed system of broadcast.

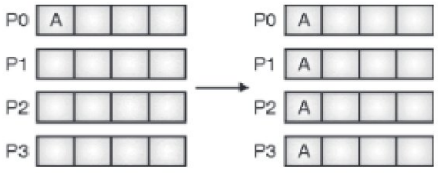


Fig. 6.4.5 : MPI_BCAST function

- The scatter (MPI_SCATTER) operation distributes the elements with one process to all other processes. This is used to scatter the data to multiprocessors.

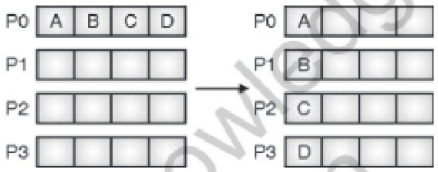


Fig. 6.4.6 : MPI_SCATTER function

- The send receive (MPI_SEND or MPI_RECEIVE) are used to send or receive the message from a particular process whose ID is given along.
- The reduce (MPI_REDUCEALL) is used for various operations like finding the largest element and then distribute the same. Another example of the same is shown in Fig. 6.4.7. The data from each process can be taken with any operation represented as "op" in the Fig. 6.4.7. The operation may be addition, subtraction, multiplication etc.

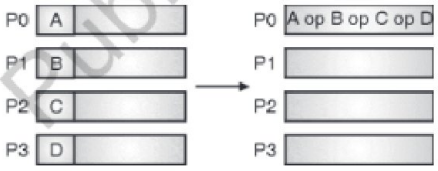


Fig. 6.4.7 : MPI_REDUCEALL function

- The gather (MPI_GATHER) operation gathers the data from different processes into the first processor as shown in the Fig. 6.4.8. Another detailed structure of this is shown in Fig. 6.4.8.

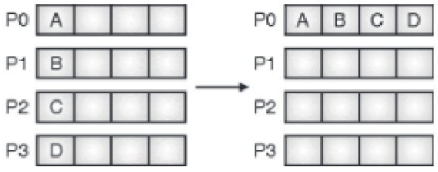


Fig. 6.4.8 : MPI_GATHER function

6.4.3(B) Topology and Embedding

- In MPI library there is a function to create Cartesian topology. The function is:

```
int MPI_Cart_create(MPI_Comm comm._old, int ndims, MPI_Comm *comm._cart)
```

- The parameters required by the function are old communicator to create a new communicator with the given dimensions dims. This makes each processor to be identified in this Cartesian system with a vector of dimensions dims.
- The sending and receiving functions for this topology are performed using the methods that covert from Cartesian system to rank system and vice versa. This is done the following function.

```
int MPI_Cart_coord(MPI_Comm comm._cart, int rank, int maxdims, int *coords);
```

- A very common operation done on the Cartesian topology is shift operation. This can be performed by the following function:

```
Int MPI_Cart_shift(MPI_Comm comm._cart, int dir, int s_step, int *rank_source, int *rank_dest);
```

6.4.3(C) Overlapping Communication with Computation

Overlapping the computation task with the communication task is very important for parallel processing. To support this MPI supports a pair of non-blocking send and receive functions. These functions are:

1. To send, the function is:

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Reguest *request)
```

2. To receive the function is:

```
Int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Reguest *request)
```

6.4.3(D) Collective Communication and Computation Operations

MPI also provides functions to perform collective communication and computation. To perform collective communication and computation, each processor in the system should call the following function

1. for one to all i.e. broadcast

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm comm.)
```

2. for one to one

```
int MPI_Reduce(void *sendbuf,void *recvbuf, int count, MPI_Datatype datatype, MPI_OP op, int target, MPI_Comm comm.)
```

Application of MPI

1. Electromagnetic scattering
2. Groundwater pollution
3. Stellar atmospheres
4. Reactive flows
5. QCD
6. Navier-stokes
7. Convective flows
8. Multi-physics in metal forming.

6.4.4 PThreads (Parallel Threads) in Shared Memory System

- A thread is a set of instructions to be executed within a program. Multithreading is a method of having multiple threads being run concurrently in a system. PThreads or parallel threads are multiple threads running simultaneously or in parallel in a multiprocessor system. Fig. 6.4.9 shows the difference between the concurrent and parallel threads.
- In case of concurrent threads, there are various threads is a system but only one processor to execute them, hence they are executed on a time slicing basis. While in case of parallel threads, there are various processors, each executing a thread and hence no time slicing, thus fast operation.

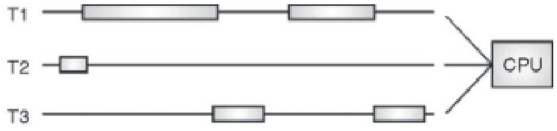
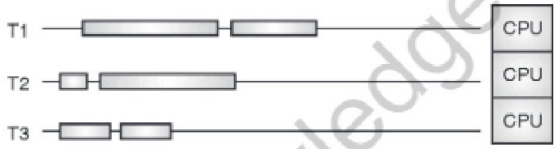


Fig. 6.4.9 (a) : Concurrent run threads



(b) Parallel run threads

Fig. 6.4.9 : Comparison of concurrent and parallel threads

6.4.4(A) POSIX Threads

- The "pthreads.h" library is a set of C function calls that provides a mechanism for writing multithreaded code. There are three major classes of function calls in this library :
  1. Calls to create and destroy threads,
  2. Calls to synchronize threads and lock program resources and
  3. Calls to manage thread scheduling.
- Each thread sees the same global data and files. But, each thread has its own stack and registers as shown in Fig. 6.4.10.

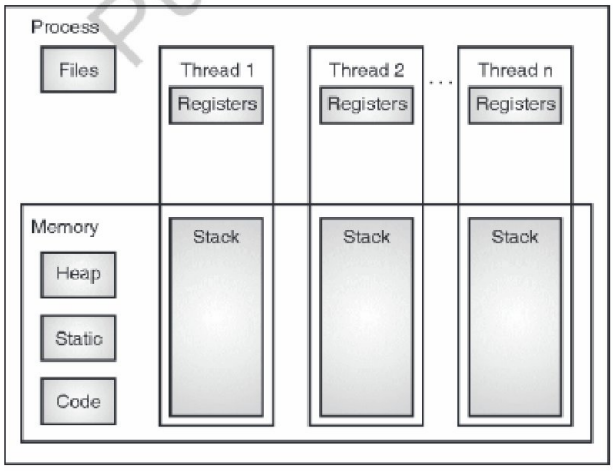


Fig. 6.4.10 : Memory for multiple threads

6.4.4(B) Thread Creation and Destruction

- Pthreads are created using pthread_create().

```
#include <pthread.h>
int pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
                    void *(*thread_function)(void *), void *arguments);
```

- "pthread_t" is an opaque type which acts as a handle for the new thread.
- "thread_id" is the ID for the thread.
- "attributes" is another opaque data type which allows to fine tune various parameters.
- "thread_function" is the function name for which the thread is executing. The thread will terminate when this function terminates.
- "arguments" is a void * pointer which is passed as the argument to the thread_function.
- As discussed above the Pthreads terminate automatically when the function returns, or the thread can call pthread_exit() which terminates the calling thread explicitly. This is shown in the code below.

```
int
pthread_exit (void *status);
```

- "status" is the return value of the thread.
- One Thread can be made to wait on the termination of another by using function pthread_join()as shown below

```
int
pthread_join (pthread_t thread, void **status_ptr);
```

- "status" ptr is the exit status returned.

6.4.4(C) Mutexes

- Mutexes have two basic operations: lock and unlock.
- When a a thread checks for the mutex and finds the mutex is unlocked, then the thread calls lock. This makes the mutex to be locked and the thread continues further operation. But, if the mutex was already locked when the thread checked it, the thread is blocked until the thread 'holding' the lock, unlocks them mutex.

  There are 5 basic functions dealing with mutexes.

1. The method to initialize the mutex is :

```
int pthread_mutex_init (pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
```

2. To lock the mutex, use the following function :

```
int pthread_mutex_lock (pthread_mutex_t *mut);
```

3. To unlock the mutex, use the following function :

```
int pthread_mutex_unlock (pthread_mutex_t *mut);
```

4. To check the lock for its availability and lock if available :

```
int pthread_mutex_trylock (pthread_mutex_t *mut);
```

5. To deallocate the memory and other resources associated with a mutex the following function is to be used :

```
int pthread_mutex_destroy (pthread_mutex_t *mut);
```

6.5 Open MP languages

- Producing a parallel program generally involves three steps :
  - Developing and debugging a sequential program.
  - Transforming the sequential program into a parallel program.
  - Optimizing the parallel program.
- For existing programs, the first step is already done.
- Of course, if the second step is done perfectly then the third one will be unnecessary, but in practice that rarely happens. It's usually much easier-and quicker in the long run, to transform the serial program into a parallel program in the most straightforward way possible, measure its performance, and then search for ways to improve it.
- The language of parallel programming can be quite unclear. Sometimes a program can be irrelevantly adapted to run in parallel because the work it does naturally divides into discrete chunks. Sometimes a serial program must be restructured considerably in order to transform it into a parallel program, reorganizing the computations the program does into units which can be run in parallel.
- Making a parallel version potentially can involve changes to the program structure, or the algorithms it implements, or both.
- The ratio of computation to communication is referred to as **granularity**.
- There are two main challenges facing any parallel programmer :
  1. How to divide the work among the available processors ?
  2. Where to store the data and how to get it to processors that need it ?
- Two completely different approaches to solve these problems have emerged as the main parallel processing paradigms. They are message passing and distributed data structures implemented in virtual shared memory.
- There are various programming languages used for parallel processing. We will be discussing some of them in the subsections here.

6.5.1 Fortran 90

- Fortran has always been used for doing complicated scientific operations It gains the name from its same function i.e. FORmula TRANslation.
- Fortran 90 was developed in 1991 and addresses most of the problems with FORTRAN 77. It incorporates most of the new features of languages developed in newer programming languages. Fortran 95 was a newer version released in 1997 that offered a few new features. Although developed early in the 90's, compilers have only recently become available for widespread use of Fortran 90.
- FORTRAN has always remained as the superior language for numerical, scientific, and engineering applications. With the improvements in Fortran from version 77 to 90, it remains the only language of choice for computational science.
- Fortran 90 is more powerful than C++ in the areas of numerical robustness, data parallelism, data abstraction, and functional programming.
- In Fortran 90 free source form means you are free to type comments and commands wherever you want. An exclamation point ('!'), is used to indicate a comment statement in Fortran 90.
- One major difference in Fortran 90 with respect to quite other programming languages is that it is case insensitive.

- A line of Fortran 90 code can have maximum 132 characters and an ampersand (&) is to be placed at the end of a line to indicate that it continues on the next line. Also another limitation is that there can be a maximum of 39 continuous lines.
- Let us take a simple program example in Fortran

```
PROGRAM test
! This line is a comment
a = 3.0 + 4.0 + &
    13.0
! After execution variable a will have the value 20.0
PRINT* a
END
```

- A Fortran program should have the following form :

1. Heading
2. IMPLICIT NONE
3. Specification section
4. Execution section
5. Internal subprograms
6. END PROGRAM

1. **HEADING :** The heading has the syntax PROGRAM *name*, where *name* is any legal identifier. Program names must begin with a letter, followed by up to thirty letters, digits, or underscores.
2. **IMPLICIT NONE :** This will help us guard against errors that are associated with variable type that may occur due to the default type convention implicit in Fortran.
3. **SPECIFICATION SECTION :** Following the IMPLICIT NONE declaration it is important to declare the environment of the program. All variables and constants need to be explicitly declared in this portion of the program. The data type must be stated for all constants, variables, parameters, etc.
4. **EXECUTION SECTION :** This is the section where the main program operations are specified.
5. **INTERNAL SUBPROGRAMS :** If needed subprograms or functions can be contained within the body of the program .
6. **END PROGRAM :** This is the required part of the program. It indicates to the compiler the end of the program and also halts execution.

Here is a simple example program.

```
PROGRAM introduction                ! heading with name = introduction
IMPLICIT NONE                       ! statement that all data types must be declared
                                    ! Declare variables and constants in the specification section
INTEGER :: age = 32                 ! age is integer type data and has the initial value of 32
CHARACTER (6) :: name = "Harish"    ! name is character type, length 6
                                    ! Now here is the execution section.
                                    ! The output is:  My name is Harish and I am 32 years old.
```

```
PRINT *, "My name is ", name, " and I am ", age, "years old."
                                        ! There are no internal subprograms in this program
                                        ! Here is the END PROGRAM statement.
                                        ! It has been labeled with the program name.
END PROGRAM introduction
```

6.5.1(A) Data types

- Fortran 90 allows manipulations on various types of data. It allows 5 basic data types and the programmer can also define own types. The available data types are :
  1. Real
  2. Integer
  3. Complex
  4. Character
  5. Logical
- All the variables are to be declared in the specification section of the program. Data types are declared with the following syntax :

  TYPE :: var

  Where *TYPE* is the data type (i.e. real, integer, etc) and var is the variable name.

  Examples :

  1. EAL :: temp ! temp is a real variable
  2. CHARACTER :: answer, name ! answer and name are variables of the character type
  3. INTEGER :: A = 4 ! A is integer type and initialized as the value 4
- Real data is a fraction number that can be represented in decimal form as exponents.
- Integers are positive or negative whole numbers (including 0).
- Fortran also allows variables to be declared as complex numbers. Variables such as this are declared with their first value as the real component and the second as the imaginary. The following example shows the declaration and initialization of a complex variable

```
! Declare A as a complex variable
COMPLEX :: A
! Set the value of A to 4.2 -3.1i
A = (4.2, 3.1)
```

- Character is a data type used to store the character of the character set of the Fortran 90. It is to be given in single quotes
- Logical variables have one of two values: **.TRUE. or .FALSE.**

6.5.1(B) Operations

There are various arithmetic operations that can be used with numeric data. They are **+, –, *, /, and** ** (exponential). The order of priority for these operations are essentially the same as typical mathematical rules :

1. Exponents are performed first, from right to left.

2. Multiplications and divisions are performed next, from left to right.
3. Addition and subtraction are last. They are also directed from left to right.

- The programmer can use parentheses to enclose sub-expressions. Sub-expressions are evaluated first according to the previous priority rules.
- Logical expressions are expressions that result in either logical constants (.TRUE. or .FALSE.) or logical variables (variables that can take the values of .TRUE. or .FALSE.). Simple logical expressions consist of the following :

Symbol	Meaning
< or .L.T	Is less than
> or .GT.	Is greater than
== or .EQ.	Is equal to
<= or .LE.	Is less than or equal to
>= or .GE.	Is greater than or equal to
/= or .NE.	Is not equal to

- The following expressions can be evaluated with logical results (.TRUE. or .FALSE.)

  A .NE. 6

  C * A >= 3

- Fortran 90 has a library of many functions available for many common mathematical operations. Some of them are
  - finding absolute value (ABS(x)),
  - exponential (EXP(x)),
  - logarithm (LOG(x)), and
  - square root (SQRT(x)).
- There are also some functions that allow changing in definition of the output type. For example, the default value for INT(x) is integer, but this can be changed as follows :

  INT(x, KIND = *kind_number*)

  The output will be the integer portion of x, but it will be of type *kind_number*.
- The basic requirement of any programming language is being able to read input from keyboard and the output results to the screen. The commands available for reading and writing to file and input-output devices are :
  1. OPEN
  2. CLOSE
  3. READ
  4. WRITE
  5. PRINT

6.5.1(C) Branching Instructions

- Another important construct required to write a program is the branching instructions. These include the selective and the iterative statements.
- Let us first see the selective statements of Fortran 90. It has the following selective statements.

1. IF (IF-THEN)
2. IF-ELSE
3. IF-ELSE IF
4. SELECT CASE

1. IF (IF-THEN)

- The simplest form is the logical IF statement. In general :
- IF (*logical -criteria*) execution statement
- If the *logical-criteria* is TRUE then the execution is performed. If not the execution statement is bypassed. An example :

```
IF (2.0 < x .AND x < 3.0) PRINT *, x
```

- In this case if x is between 2 and 3 then it will be printed.
- Another IF form used when multiple statements are required for the TRUE case is :

```
IF (logical -criteria) THEN

   execution statements

END IF
IF (x  >= 0) THEN

   z = x * y

   PRINT *, "x is a positive number."

END IF
```


2. IF-ELSE

- The ELSE statement allows specification of execution statements for the case that the *logical-criteria* of the IF statement is FALSE.

```
IF (logical -criteria) THEN
   execution statements for true result
ELSE
   execution statements for false result
END IF
```

- If the *logical-criteria* is TRUE then the first set of statements are performed and the second are bypassed. If the *logical-criteria* is FALSE then the first set of statements are bypassed and the second statements performed. As an example:

```
IF (x > 0) THEN
  PRINT *, "The value is greater than zero."
  ELSE

  PRINT *, "The value is not greater than zero."
  END IF
```

3. IF-ELSE IF

– Nested IF statements (IF statements within IF statements)can allow for many selection criteria. For example :

```
 IF (x >= 1) THEN
     PRINT *, "The value is greater than or equal to one."
 ELSE
     IF (x > 0) THEN
        PRINT *, "The value is between zero and one."
      ELSE
        PRINT *, "The value is less than or equal to zero."
     END IF
 END IF
```

– Although nested loops are allowed, they can frequently become very complex if many selection options are desired. **The ELSE IF statement allows for multialternative selection in a more simple and easier to read format**. In general :

```
IF (logical -criteria1) THEN
          execution statements 1
ELSE IF (logical -criteria2) THEN
          execution statements 2
ELSE IF (logical -criteria3) THEN
          execution statements 3
          ....
ELSE
          execution statements n
END IF
```

– The ELSE statement is optional, but it indicates the statements to be performed if none of the IF statements are TRUE. Only one set of execution statements are performed. The expressions are evaluated in order and when a TRUE statement is found, its execution statements will be performed and execution continues with the next statement following the END IF statement. For example:

```
    IF (x > 0) THEN
          PRINT *, "Value is greater than zero."
ELSE IF (x < 0) THEN
          PRINT *, "Value is less than zero."
ELSE IF (x == 1) THEN
          PRINT *, "Value is one."
ELSE
```

```
        PRINT *, "Value is zero."
END IF
```

- Notice if x is equal to one the output of this series of commands will be :
- Value is greater than zero.
- It will not be "Value is one." because the first true IF statement is that it is less than zero. That statement was executed and execution proceeded outside of the IF-ELSE IF construct.

4. Select Case

- The SELECT CASE construct is not as general as the IF constructs, but it is of use for some specific applications. In general :

```
SELECT CASE (selector)
        CASE (list1)
          execution statements 1
        CASE (list2)
          execution statements 2
        ...
        CASE (listn)
          execution statements n
    END SELECT
```

- The *selector* is either an integer, character, or logical expression. **It cannot be real**. *List1* to *listn* are the possible values for the *selector*. Say you wanted to associate output a letter grade for a known a numerical score. Say the numerical score is associated with the real variable named grade:

```
    SELECT CASE (INT(grade))                ! The real value grade is converted to an integer.

    CASE (90:)                  ! 90: indicates values of 90 or above.
          PRINT *, "Your grade is an A."

    CASE (80:89)                ! 80:89 means 80 to 89.
          PRINT *, "Your grade is a B."

        CASE (70:79)
          PRINT *, "Your grade is a C."

        CASE (60:69)
          PRINT *, "Your grade is a D."

        CASE (:59)                      ! 59: indicates 59 or below.
          PRINT *, "Your grade is an F."
    END SELECT
```

- As discussed earlier, besides the selective statements there are also loops required in a programming language. These loops are for repeating a set of statements again and again i.e. iterative statements.
- The various loop statements supported by the Fortran are discussed below:

5. Do Loop

- The Do loop is mainly used as a counter. It will be used to perform the various instructions for given number of times.
- The syntax of the for loop is such that we can give the initial value of the counter variable, the last value or the limit and the step size i.e. the incremental value. If not mentioned the default step size will be one, hence it is optional.
- The syntax of the Do loop is given below :

```
DO control-variable = initial value, limit, step-size
        execution list
    END DO
Here is an example of a counter DO LOOP.
INTEGER :: max = 5
        DO n=1, max
        PRINT *, n, n*2
END DO
```

This output :

```
1  2
2  4
3  6
4  8
5  10
```

- Another example is a loop within a loop as given below :

```
DO n = 5, 1, – 1   !   From 5 to 1 with a step size of minus 1
    DO m = 1, 2
      PRINT *, n, m
    END DO
END DO
```

- Notice how by indenting the loops the order of execution is indicated. The most indented loop is performed in completion first. The output would be :

Output

```
5 1
5 2
4 1
4 2
3 1
3 2
2 1
2 2
1 1
1 2
```

Loops can be labeled as another way to organize your nested loops. The previous nested loops could have been written :

```
outer: DO n = 5, 1, -1   !  From 5 to 1 with a step size of minus 1
         inner: DO m = 1,2
           PRINT *, n, m
           END DO inner
    END DO outer
```

- If you have many nested loops this could assist in helping you and others follow your code.
- Logical loops are loops that require fulfilling a logical criteria for completion of the loop. These can also be called DO-EXIT loops. The general form is :

```
DO
    execution statements
    IF (logical criteria) EXIT
    execution statements
END DO
```

- The execution statements can be before and/or after the EXIT statement. When the IF statement is TRUE repetition is halted. Be careful when determining the *logical criteria* for your IF statement. **If it never becomes TRUE you will be left with an infinite loop**.
- For example :

```
n = 1
    DO
    IF (n > 10) EXIT
    n = n + 1
END DO
```

- You can also use user input to halt repetition. For example :

```
DO
    PRINT *, "Enter the radius of your circle."
    READ *, r
    PRINT *, "Area is ", pi*r*2
    PRINT *, "Do you want to calculate another area?"
    READ *, response
    IF (response == "n") EXIT
    END DO
```

- In this case, the parameter pi must have been defined previously and response would need to be defined as a character variable of length = 1 in the specification section of the program, but you see how the user input was used as a decision criteria.
- A more modern and structured way of realizing a logical loop is with use of the DO WHILE - END DO expression :

```
DO WHILE expression
   execution statements
END DO
```

– Here the execution statements are performed as long as *expression* returns TRUE. When *expression* becomes FALSE the loop is broken and execution continues with the following statements. The equivalence between DO-EXIT and DO WHILE loops is portrayed below :

n = 1
DO
IF (n > 10) EXIT
n = n + 1
END DO
	← these two command list are equivalent →	n = 1
		DO WHILE (n <= 10)
		n = n + 1
		END DO

6.5.2 Occam

– Occam is the name of a parallel programming language which was developed in Great Britain.

– This language is good for the message passing style of parallel programming.

– The **advantage** of Occam is its cost is comparatively less and hence for a modest investment one can write truly parallel programs executing on an group of Transputers.

– The Transputer is a 32-bit microprocessor (20 MHz clock) that provides 10 MIPS (million instructions per second) and 2.0 MFLOPS (million floating point operations per second) processing power with 4K bytes of fast static RAM (Random Access Memory) and concurrent communication capability all on a single chip.

– CSP (Communicating Sequential Processes) is the main basis on which the Occam programming language works for message passing. Using CSP as a basis, the researchers developed an Occam concurrency model. And from the Occam model, they then developed the parallel programming language Occam.

– The name is derived from William of Occam, a philosopher of the thirteenth century. Occam had a philosophical principle of "keep things simple," and hence it is attributed to William. The primary goal of the Occam language is to keep the language simple and hence the name.

– From the Occam model, the researchers also developed a VLSI chip computer to support their concurrency model and this chip is called as the Transputer.

– Although Occam is a high-level language, but for Transputer it is assembly language.

– The Transputer is different from other microprocessors in its machine language. The definition of the operations of the Transputer is in terms of the Occam model and not machine language.

– Since the Transputer is designed to execute Occam, the compiler can generate very efficient and compact machine code.

– Besides being a high performance microprocessor (half the speed of a VAX 8600), the Transputer has on its chip four (4) serial bi-directional links (each 20 Megabits per second) to provide concurrent message passing to other Transputers.

– The "channels" in the Occam language are mapped to these hardware links, which connect through the twisted pairs of wires to other Transputers.

– The Transputer hardware also supports concurrency by time-slicing. It implements concurrency of an arbitrary number of Occam concurrent processes, in round-robin fashion.

– The Occam language and the Transputer hardware are so designed that an program consisting of a collection of concurrent processes may execute on one Transputer in time slicing manner, or can be spread over many Transputers with little or no change in the Occam code to obtain high performance.

6.5.2(A) The Occam Concurrency Model

- As discussed earlier also, the Occam model supports concurrency, i. e., true parallelism on several processors or simulated parallelism on one processor with the help of time-slicing.
- This concurrency model of Occam is based on concurrent processes. In Occam, the communication amongst the concurrent processes is achieved by passing messages along the point to point channels. By, point to point it means that both the source and destination must be at one point or must reside in one concurrent process.
- Fig. 6.5.1 shows how a process P1 can send a message via the channel C to process P2.

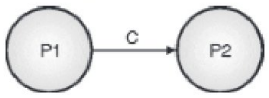


Fig. 6.5.1 : Process P1 sending a message to Process P2 via the channel C

- To reduce the problems caused by interference of other processes while sharing variables between two concurrent processes, all communication between the concurrent processes in Occam take place via the channels.
- Hence the Occam model has no shared variable problem i.e. two concurrent processes can read a variable simultaneously, but one cannot read while the other writes into a shared variable. Thus, Occam reflects the message-passing model of parallel computation that is supported by the Transputer hardware with its local memory (i.e. no global memory) and communication links.
- The communication in the Occam channel is synchronous. When either the sender or the receiver arrives at the place in the code, where communication is required, the first to arrive waits for the other. Once the other process also reaches to the code where communication is to be done i.e. the two processes are synchronized, the message transfer between the two process takes place. Then the two processes continue executing.
- Hence, this communication mechanism does not provides any automatic buffering. If buffering is desired, an intermediate process has to be inserted between the two processes.
- The major **advantage** of this scheme is that one language feature performs both synchronization and message passing.
- Let us see some things related to writing the Occam programs. All reserved words must be in capital letters. Spaces work as delimiters. Each construct must be indented two spaces to show structure.
- Occam is line oriented, which means each statement starts on a new line, possibly indented. Continuation to the next line is possible by breaking an expression at an operator, semicolon or comma.
- Comments are designated by -- to the end of the line.
- The structure of a program is a process with declarations preceding it i.e.

  <declares>

  <process>

- An example can be given as

```
INT j :
SEQ
j := 1
j := j + 1
```

- The Occam "process" can be considered as another name for "statement" in other languages, e. g., C, C++, etc. However, an Occam process is different from the term process as used in most operating systems. For example, two Occam processes need not be "concurrent processes."

- In Occam, there are five primitive processes: assignment, receive, send, SKIP and STOP. The syntax of these statements are given below :

Primitive syntax example

```
assignment <variable> := <expression>
receive <channel> ? <variable>
send <channel> ! <expression>
SKIP
STOP
```

- The above statements used have the following functions :
  - **(a) assignment** - assigns to a variable the value of an expression.
    For example : x := y + 1
  - **(b) receive** - receives value from a channel. Uses "?" to signify a query.
    For example : Ch ? x
  - **(c) send** - sends expression value on a channel. Uses "!" to signify an exclamation.
    For example : Ch ! y + 1
  - **(d) SKIP** - do nothing and terminate the process, i. e., no operation.
    For example : SKIP
  - **(e) STOP** - do nothing and never terminate the process, i.e., never get to the next process.
    For example : STOP
- The notations are straight from Hoare's CSP i.e.'?' for receive and '!' for send.
- In Occam expressions, there is no operator precedence. Thus we need to use parentheses to specify the order of operation.
- For example : x := 2 * y + 1 is illegal. The correct form is using parentheses as in the following :
  x := (2 * y) + 1
- In sends and receives, we can use ";" to separate expressions or variables, as in :
  ch ! x; y; x + y
- Occam is strongly typed language and hence we have to declare every variable. Declarations are of the form :
  <type> <one or more identifiers separated by commas> :

Finally let us discuss about the features of Occam parallel programming language, listed below :

- The Occam programming language is a viable language to express concurrency.
- It is especially valuable for teaching the concepts of parallel algorithms in the message passing paradigm.
- One of Occam's redeeming aspects is its simplicity.
- The Occam concurrency model makes it easy to think and reason about an Occam program as a network of concurrent processes which only passes messages, i.e., the programmer does not have to deal with shared variables and their associated problems.
- In Occam, one expresses concurrency explicitly at the statement level, i.e., using the PAR construct, while in other languages, e.g., Ada, concurrency can only be expressed implicitly at the procedural level.
- Also, in a language like Ada, the "heavy" machinery to support concurrency tends to discourage the programmer from using concurrency. While, with Occam, the PAR is a natural and easy construct to use.

- Another important goal of Occam is security. Security is not easy to achieve in a concurrent system, however, Occam makes it a lot easier than most concurrent languages. In order to support this security, the language features, e. g., pointers, dynamic memory allocation, dynamic process allocation and recursive functions, are not allowed.
- While a concurrent language like Logical Systems C, may be more flexible and support all of these complicated features, but he concurrent C programs are very hard to debug. The security that Occam brings to programming concurrent program is also very important for a student environment where they are learning the concepts of concurrency while mastering the language.

6.5.3 C-Linda

- The Linda model is a general model of parallel computing based on distributed data structures We can use it to implement message passing as well.
- Linda calls the shared data space as the **tuple space**. C-Linda is an implementation of the Linda model using the C programming language, and Fortran-Linda is an implementation of the Linda model using the Fortran programming language. Fig. 6.5.2 shows the tuple space
- It uses a shared memory space and works on heterogeneous networks.
- The processes access the tuple space using some of the function operations that Linda provides. For example, parallel programs using C-Linda are written in C and incorporate these operations as necessary to access tuple space.
- Thus, C-Linda functions as a coordination language, provide the tools and necessary environment to combine different processes to form a complete parallel program.
- The parallel operations in C-Linda are totally different from that of C, providing corresponding capabilities necessary for parallel programs. C-Linda programs make full use of standard C for computation and other non-parallel tasks. C-Linda enables these sequential operations to be divided among the available processors.
- C-Linda is implemented as a precompiler. Hence, the C-Linda programs are always independent of a particular C compiler used for final compilation and linking.
- Linda programmers don't need to worry about how tuple space is set up, where it is physically located, or how data moves between it and running processes; all of this is managed by the Linda software system. Because of this, Linda is logically independent of system architecture, and Linda programs are portable across different architectures, whether they're shared memory computers, distributed memory computers, or networks of workstations.

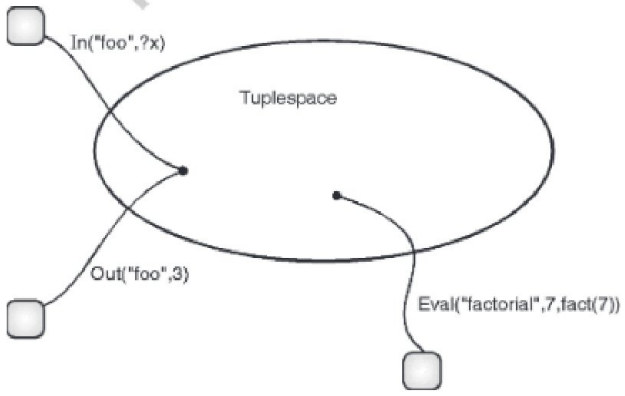


Fig. 6.5.2 : Tuple space

- Fig. 6.5.2 shows an example of Tuple space.
- Data moves to and from tuple space as tuples. Tuples are the data structures of tuple space. A tuple is a sequence of up to 16 typed fields; it is represented by a comma-separated list of items enclosed in parentheses. Here is a simple example :

```
("simple", 1)
```

- This tuple declaration has two fields viz. the character string and an integer. In the above example, both the fields are constant values. Instead of constants, we can also have variables used in tuples as shown in the examples below:

```
("easy", i)
```

- This tuple also has a string in its first field, and the second field is a variable of some numeric data type i.e. the variable i. Linda provides four important operations for accessing tuple space listed in Table 6.5.1.

Table 6.5.1

Operation	Action
out	Places a data tuple in tuple space.
eval	Creates a live tuple (usually starting new process(es)).
in	Removes a tuple from tuple space.
rd	Reads the values in a tuple in tuple space, leaving the tuple there.

- The **out** operation places a data tuple with one string and two integer fields into tuple space as shown in the example below :

```
out("cube", 4, 64);
```

- There are two kinds of tuples: passive tuples or the data tuples, (like the one in above example, which contain static data), and the second kind is the process tuples or the live tuples, which are under active evaluation.
- An eval operation as discussed is used to create a process tuple consisting of the fields specified as its argument. The process tuple then creates another process to evaluate each argument.
- The actual implementation creates a process only for the arguments consisting of a simple function call while all the other fields in the eval are evaluated one by one. When the processes run, the eval's tuple is called as the live tuple; as each process completes, its return values are placed into the corresponding fields. Finally when all the fields are filled and all the processes have completed then the resulting data tuple is placed into tuple space.
- Although it is literally not true that an eval creates the processes which will evaluate its arguments, but it is easier to understand when thought in this way.
- For example, the eval statement given below will result in a process being created to evaluate the function given as the third argument, f(i):

```
eval("test", i, f(i));
```

- Another example, evals can be used to initiate worker processes, as in the following loop :

```
for (i=0;i < NWORKERS; i++)
eval ("worker", worker());
```

- The above loop starts a number of worker processes equal to the variable NWORKERS. In this case, the **eval** simply starts the processes, rather than performing computation and place the result into tuple space. Formally, however, when each worker finishes, a tuple of the form: ("worker", 0) is placed into tuple space.

- The other two operations i.e. the in and rd operations, allow a process to access the data in tuple space.
- The rd operation reads a tuple from tuple space, and then removes the tuple from tuple space. Both rd and in take a template as their argument. The template specifies which type of tuple is to be retrieved.
- Similar to the tuples, templates consist of a sequence of typed fields, some of which hold values that can be constants or expressions resolving to constants while some other that hold variables of placeholders for the data in the corresponding field of the matched tuple in the tuple space.
- The placeholders begin with a question mark and are also known as formals. When it finds a matching tuple, variables used as formals in the template will be assigned the values in corresponding fields of the matched tuple. An example is given below :

```
("simple", ?i)
```

- In this template, the first field is an actual, and the second field is a formal. If this template is used as the argument to a **rd** operation, and a matching tuple is found, then the variable `i` will be assigned the value in the second field of the matched tuple.
- A template matches a tuple when the following three cases are satisfied.
  1. Both of them have the same number of fields.
  2. The types, values and length of all literal values in the template are the same as the those of the corresponding fields in the tuple.
  3. Also the types and lengths of all formals in the template match the types and lengths of the corresponding fields in the tuple.
- Let us see some examples. If the tuple is: ("cube", 8, 512), then the statement: rd("cube", 8, ?i); will match. And assign the value 512 to the variable 'i' considering the variable 'i' is an integer. Also if the value of the variable 'j' is equal to 8, then the statement: in("cube", j, ?i); will match it and i is assigned the value 512 assuming it to be integer, and the tuple is removed from tuple space.
- If more than one matching tuple for a template is present in tuple space, one of the matching tuples will be used, which is non-deterministic i.e. it will not necessarily be the oldest, the most recent, or a tuple specified by any other criteria.
- Programs must be prepared to accept any matching tuple, and to receive equivalent tuples in any order. Similarly, repeated rd operations will often yield the same tuple each time if the tuples in tuple space remain unchanged.
- If no matching tuple is present in tuple space, then both rd and in will wait until one appears; this is called blocking. The routine that called them will pause, waiting for them to return.
- The direction of data movement for the in and out operations is from the point of view of the process calling them and not from the point of view of tuple space. Thus, an out places data into tuple space, and an in retrieves data from it. This is similar to the general use of input and output in conventional programming languages.
- All data operations to and from tuple space occur in this way. Data is placed into tuple space as tuples, and data is read or retrieved from tuple space by matching a tuple to a template, not by, say, specifying a memory address or a position in a list. This characteristic defines tuple space as an associative memory model.
- An example of C-Linda program is given below

```
/* hello.clc: a simple C-Linda program */
 #include "linda.h"
int worker(int num) {
   int i;
```

```
    for (i=0; i<num; i++)
        out("hello, world");
    return 0;
}
int real_main(int argc, char *argv[]) {
    int result;
    eval("worker", worker(5));
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("hello, world");
    in("worker", ? result);
    return 0;
}
```

6.5.3(A) Advantages of C-Linda

- Anonymous and asynchronous communication
- Associative addressing / pattern matching
- Data persistence independent of creator
- Simple API → less and easier coding
- Ease of code transformation

6.5.3(B) Drawbacks of C-Linda

- High system overhead
- Designed as parallel computing model, primarily for LANs. Hence it lacks security model and transaction semantics
- Language specific implementation
- Blocking calls, but no notification mechanism.

6.5.4 CCC

6.5.4(A) Features of CCC

- CCC was the in-house programming language of a Hungarian company (Com-Firm Bt.).
- The language derives the concepts from CA-Clipper, LISP and C++. Using CCC we can develop database-oriented applications extremely fast.
- Since CCC generates and compiles C++ code, it is also ideal for embedded applications.
- Originally the CCC was an acronym for Clipper to C++ Compiler, but now it is just the name.
- CCC is a high-level parallel programming language that supports not only data parallelism but also supports task parallelism. In CCC, data parallelism is implemented in a SIMD (single-instruction-multiple-data) manner. The task parallelism is implemented in MIMD (multiple-instruction-multiple-data) manner.
- The runtime system for CCC is based on a virtual shared memory machine that supports shared memory and dynamic thread creation.

- A CCC program consists of a collection of coordinated concurrent tasks.
- For the implementation of the data parallelism, shared-memory abstraction is provided. This supports concurrent read, concurrent write as well as reduction operations among data-parallel tasks.
- For the communication amongst the tasks in case of task parallelism, both shared-memory and message-passing abstractions are provided.
- These salient features make CCC cover most parallel programming paradigms.

6.5.4(B) Structure of CCC Programming System

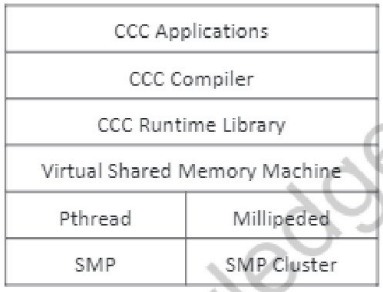


Fig. 6.5.3 : Structure of CCC programming system

- The structure for the CCC programming system is shown in Fig. 6.5.3.
- As shown, the CCC programming system can be divided into five layers. These layers are discussed below :

1. The first layer is the layer of multiprocessors. The growth in the fields of very large silicon integration (VLSI) and communication technologies has made it possible to have symmetric multiprocessors (SMPs) and SMP clusters. The CCC programming is mainly designed for these as the target machines. Most programming languages only support data parallelism and are executed in single-program-multiple-data (SPMD) system instead of SIMD systems for better performance. But, the data parallelism in CCC is specified in machine that supports shared memory and dynamic task creation. Hence SMP are the best target machines for the CCC programming language.
2. The second layer consists of the Pthread and millipede libraries. The Pthread library is a standard library that supports the functionalities for SMP. The millipede library supports the mechanisms for distributed shared memory and dynamic remote task creation that are required for the SMP cluster systems.
3. The third layer of the CCC programming system provides a common interface on top of various libraries that supports shared memory and dynamic task creation. This concept of a virtual shared memory machine makes the CCC compiler and runtime system highly retargetable. To retarget CCC compiler on a different library or machine, this layer is the only layer that needs to be modified and the modification is very easy.
4. The fourth layer is the runtime library of the CCC. This library is required to simplify the code generator of the compiler, and it also consists of a collection of functions that supports the salient features of CCC.
5. The fifth layer is the CCC compiler that compiles the CCC programs.
6. The sixth layer is the CCC applications written by the coder

6.5.4(C) Data Parallel and Control Parallel Constructs (Programming Using Language CCC)

- The language CCC is designed as a extension of the language C that mainly provides concepts for specifying the concurrency, support communication and synchronization of the parallel tasks.

- The CCC design allows us to implement the compiler for CCC as a source-to-source translator and a runtime system. The source-to-source translator will translate the source CCC program into a C program by calling the functions supported by the runtime system. The runtime system will provide mechanisms for shared memory and also for dynamic task creation.
- Let us consider the data parallelism first. The data concurrency concept is specified by the definition of the domain construct and the calling of the data-parallel functions defined in domain construct. The structure of this is shown in the syntax below :

```
domain name [size] {
  data_declarations;
  data_parallel_functions;
}
```

- Each data-parallel function in the above structure represents a collection of size parallel functions. The calling of each data-parallel function concurrently creates tasks of the size given along with the domain name in the above structure.
- The synchronization is implicitly done by the synchronous semantics of the SIMD system. Finally the communication system is implicitly specified by the concurrent read, concurrent write and reduction operations on the variables declares inside the domain as shown above.
- The specifications of the distribution of the data can be given in the specifications for the parameters inside the parallel functions.
- Let us now consider the task parallelism. The concurrency for the task parallelism is specified with the help of the definition of task-parallel functions and the parallel section constructs.
- A task-parallel function i.e. **task func();** is declared by putting the keyword task before the function definition.
- There are two types of parallel sections. The first type of parallel section is used to concurrently call a group of task-parallel functions, shown below:

```
par {
        func_1(...);
        func_2(...);
          ...
        func_n(...);
        }
```


- The above example will execute the n task-parallel functions i.e. func_1, func_2, ..., and func_n as if they are n tasks in parallel.
- Parallel sections will be executed in fork-join form, i.e. the parallel section exits only when all the created tasks exit.
- The second type of parallel section is used to concurrently invoke multiple instances of a group of task-parallel functions. This is shown in the example below :

```
parfor (init_expr; exit_expr; step_expr) {
        func_1;
        func_2;
          ...
        func_n;
        }
```

- The above example will execute multiple instances of the n task-parallel functions i.e. func_1, func_2, ..., and func_n as tasks in parallel.
- The semantics for the control expressions init_expr; exit_expr; and step_expr is the same as it is in the semantics of the for loop in C.
- The communication amongst the tasks for the task parallelism is done by typed channels or asynchronous message queues. There are four types of channels namely: pipes, spliters, mergers, and multiplexers. The messages in these channels can be accessed with the help of the following two functions

```
msg = receive(channel);
send(channel, msg);
```

- Pipes are used for one-to-one communication and are the most basic channels. The remaining three types of channels can be implemented using pipes.
- Spliters are used for one-to-many communication and are useful for producer-consumer applications with single producers.
- Mergers are used for many-to-one communication and are useful for client-server applications with a single server.
- Multiplexers are used for many-to-many communication and are useful for producer-consumer applications with multiple producers or client-server applications with multiple servers.
- These channels provide a simple method for implementing communication in message-passing programming model.
- The synchronization of the tasks in a task parallelism is specified via monitors, whose structure is as shown below :

```
monitor name {
        data_declarations;
        condition_variable_declarations;
        function_definitions;
    }
```

- Monitors are used for implementing both the mutual exclusion and condition synchronization in shared-variable programming model.
- The functions that are defined in monitors are mutually exclusive by default. If a function performs only read operation then a read keyword can be put before the definition of such a function and multiple such functions can be invoked concurrently.
- Three functions namely: wait(cond), signal(cond), and signalall(cond) are provided to manipulate the condition variables. The signal functions have the semantics such that they will continue to execute after receiving the signal.
- Nested monitor calls are allowed in the CCC language. A task releases the monitor exclusion when it makes a nested monitor calls, but it needs to reacquire monitor exclusion when it returns from the call.
- The virtual shared memory system consists of a set of processors each of them with a local memory and a memory shared among the processors in the machine. A list of functions used for accessing the virtual shared memory is given in the Table 6.5.2.

Table 6.5.2 : Functions for the interface of the virtual shared memory

Sr. No.	Function
1	void VSMM_init(int num_of_procs);
2.	void VSMM_final(void);
3.	int VSMM_num_of_procs(void);
4.	int VSMM_my_pid(void);
5.	int VSMM_min_load_proc(void);
6.	void *VSMM_share(int size);
7.	VSMM_task_type VSMM_create(void * (*func) (void *), void *arg, int pid);
8.	VSMM_task_type VSMM_my_tid(void);
9.	void VSMM_exit(void);
10.	void VSMM_join(VSMM_task_type tid);
11	void VSMM_lock_init(VSMM_lock_type lid);
12.	void VSMM_lock(VSMM_lock_type lid);
13.	void VSMM_unlock(VSMM_lock_type lid);
14.	void VSMM_lock_destroy(VSMM_lock_type lid);
15.	void VSMM_rwlock_init(VSMM_rwlock_type rwlid);
16.	void VSMM_rdlock(VSMM_rwlock_type rwlid);
17.	void VSMM_wrlock(VSMM_rwlock_type rwlid);
18.	void VSMM_rwunlock(VSMM_rwlock_type rwlid);
19.	void VSMM_rwlock_destroy (VSMM_rwlock_type rwlid);
20.	void VSMM_cond_init(VSMM_cond_type cid);
21.	void VSMM_wait_lock(VSMM_cond_type cid, VSMM_lock_type lid);
22.	void VSMM_wait_rdlock(VSMM_cond_type cid, VSMM_rwlock_type rwlid);
23.	void VSMM_wait_wrlock(VSMM_cond_type cid, VSMM_rwlock_type rwlid);
24.	void VSMM_signal(VSMM_cond_type cid);
25.	void VSMM_signal_all(VSMM_cond_type cid);
26.	void VSMM_cond_destroy (VSMM_cond_type cid);
27.	void VSMM_barrier_init(VSMM_barrier_type bid, int size);
28.	void VSMM_barrier(VSMM_barrier_type bid);
29.	void VSMM_barrier_final(VSMM_barrier_type bid);
30.	void VSMM_barrier_destroy (VSMM_lock_type lid);

- All the functions listed in the Table 6.5.2 are explained below.
- The virtual shared memory machine is initialized with the function VSMM_init() and is finalized with the function VSMM_final().
- The function VSMM_init() is also used to set the number of processors in the machine.
- The function VSMM_num_of_procs() is used to find the number of processors in the machine.
- The function VSMM_my_pid() returns the name identifier of the processor on which the current task is executing.
- The function VSMM_min_load_proc() returns the name identifier of the processor that has the minimal number of tasks and is useful for load balancing.
- The function VSMM_share() is used to allocate a section of storages from the shared memory.
- The function VSMM_create() is used to create the tasks in the virtual shared memory machine and then each created task is assigned to execute on one of the processors in the machine.
- The function VSMM_my_tid()returns the identifier of the current task.
- The function VSMM_exit() terminates the current task.
- The function VSMM_join()waits for the termination of a specific task.
- The different tasks in the virtual shared memory machine can be synchronized with different synchronization objects, namely mutex locks, read-write locks, condition variables, and barriers. The mutex lock implementation is done with functions VSMM_lock_init(), VSMM_lock(), VSMM_unlock() and VSMM_lock_destroy().
- Similarly the functions VSMM_rwlock_init(), VSMM_rdlock(), VSMM_wrlock(), VSMM_rwunlock() and VSMM_rwlock_destroy() are used for of read-write locks.
- The functions VSMM_cond_init(), VSMM_wait_lock(), VSMM_wait_rdlock(), VSMM_wait_wrlock(), VSMM_signal(), and VSMM_cond_destroy() similarly support for the condition variables.
- And the functions VSMM_barrier_init(), VSMM_barrier(), and VSMM_barrier_destroy() support the functionalities of barriers.
- The implementations of the virtual shared memory machine is done using two libraries Pthread and Millipede which have very similar functions to the ones in the virtual shared memory machine. Based on the above functions, let us see how the different functions are to be called and initialised in the two libraries.
- The details in the implementation using Pthread are as follows :
  1. Since the library is running on a single SMP, the function VSMM_num_of_procs() is defined as 1 (indicating one processor) and both functions VSMM_my_pid() and SMM_min_load_proc() are defined as 0 (as the ID of the processor).
  2. The read-write locks and barriers are not implemented in Pthread. The various functions for read-write locks and barriers are implemented using the mutex locks and the void Pthread. i.e. the condition variables in Pthread.
- The details in the implementation using Millipede are as follows :
  1. To implement the function VSMM_min_load_proc(), an array VSMM_num_of_tasks[] is declared as shared data, which maintains the number of active tasks on each SMP. This array is updated by the execution of the functions namely VSMM_create() and VSMM_exit(), and is read by the function VSMM_min_load_proc().
  2. The function VSMM_create() is implemented using the thread creation function while the function VSMM_share() is implemented using the shared memory allocation function and the shared memory distribution function in Millipede.
  3. The joining of tasks is implemented using the barriers in Millipede.
  4. The read-write locks are also implemented by a set of functions for read-write locks using mutex locks and condition variables in Millipede.

– The CCC runtime library contains a set of functions to implement the various operations of CCC on the virtual shared memory machine. The functions in the library are listed in Table 6.5.3.

Table 6.5.3 : CCC runtime library functions

Sr. No.	Function
1.	typedef struct {...} CCC_dmain_dname_type;
2.	CCC_domain_dname_init(...);
3.	CCC_reduce_op(...);
4.	typedef struct {...} CCC_fname_n_param_type;
5.	CCC_fname_n_pack_arg(...);
6.	CCC_fname_n_unpack_arg();
7.	int CCC_parallel_section_prologue();
8.	CCC_parallel_section_epilogue(num_of_calls);
9.	typedef struct {...} CCC_monitor_mname_type;
10.	typedef struct {...} CCC_channel_tname_type;
11.	void CCC_send_tname(CCC_channel_tname_type, tname);
12.	tname CCC_receive_tname(CCC_channel_tname_type);

– For each domain we need to construct named dname, which the compiler will declare as a structure type CCC_dmain_dname_type for it.

– The compiler will define a function CCC_domain_dname_init(), for it to initialize the mutex lock as well as the barrier.

– For each task of the task-parallel functions named fname and with n parameters, the compiler will make a structure of type CCC_fname_n_param_type for it. Now, to create a task for this task-parallel function, the arguments will be packed into a structure of this type. The compiler will also define two macro definitions to pack and unpack its arguments before and after entering the function listed below in same order.

CCC_fname_n_pack_arg()

CCC_fname_n_unpack_arg()

Review Questions

Q. 1 Explain the Send and receive Operations.

Q. 2 Write a short note on MPI.

Q. 3 Explain Overlapping communication and collective communication.

Q. 4 Write a short note on OpenMP.

❑❑❑