# UNIT 1 INTRODUCTION TO PARALLEL COMPUTING

**Structure** **Page Nos.**

## 1.0 INTRODUCTION

Parallel computing has been a subject of interest in the computing community over the last few decades. Ever-growing size of databases and increasing complexity of the new problems are putting great stress on the even the super-fast modern single processor computers. Now the entire computer science community all over the world is looking for some computing environment where current computational capacity can be enhanced by a factor in order of thousands. The most obvious solution is the introduction of multiple processors working in tandem i.e. the introduction of parallel computing.

Parallel computing is the simultaneous execution of the same task, split into subtasks, on multiple processors in order to obtain results faster. The idea is based on the fact that the process of solving a problem can usually be divided into smaller tasks, which may be solved out simultaneously with some coordination mechanisms. Before going into the details of parallel computing, we shall discuss some basic concepts frequently used in parallel computing. Then we shall explain why we require parallel computing and what the levels of parallel processing are. We shall see how flow of data occurs in parallel processing. We shall conclude this unit with a discussion of role the of parallel processing in some fields like science and engineering, database queries and artificial intelligence.

## 1.1 OBJECTIVES

After going through this unit, you will be able to:

- tell historical facts of parallel computing;
- explain the basic concepts of the discipline, e.g., of program, process, thread, concurrent execution, parallel execution and granularity;
- explain the need of parallel computing;
- describe the levels of parallel processing;
- explain the basic concepts of dataflow computing, and
- describe various applications of parallel computing;

## 1.2 HISTORY OF PARALLEL COMPUTERS

The experiments with and implementations of the use of parallelism started long back in the 1950s by the IBM. The IBM STRETCH computers also known as IBM 7030 were built in 1959. In the design of these computers, a number of new concepts like overlapping I/O with processing and instruction look ahead were introduced. A serious approach towards designing parallel computers was started with the development of ILLIAC IV in 1964 at the University of Illionis. It had a single control unit but multiple processing elements. On this machine, at one time, a single operation is executed on different data items by different processing elements. The concept of pipelining was introduced in computer CDC 7600 in 1969. It used pipelined arithmatic unit. In the years 1970 to 1985, the research in this area was focused on the development of vector super computer. In 1976, the CRAY1 was developed by Seymour Cray. Cray1 was a pioneering effort in the development of vector registers. It accessed main memory only for load and store operations. Cray1 did not use virtual memory, and optimized pipelined arithmetic unit. Cray1 had clock speed of 12.5 n.sec. The Cray1 processor evloved upto a speed of 12.5 Mflops on 100 × 100 linear equation solutions. The next generation of Cray called Cray XMP was developed in the years 1982-84. It was coupled with 8-vector supercomputers and used a shared memory.

Apart from Cray, the giant company manufacturing parallel computers,Control Data Corporation (CDC) of USA, produced supercomputers, the CDC 7600. Its vector supercomputers called Cyber 205 had memory to memory architecture, that is, input vector operants were streamed from the main memory to the vector arithmetic unit and the results were stored back in the main memory. The advantage of this architecture was that it did not limit the size of vector operands. The disadvantage was that it required a very high speed memory so that there would be no speed mismatch between vector arithmetic units and main memory. Manufacturing such high speed memory is very costly. The clock speed of Cyber 205 was 20 n.sec.

In the 1980s Japan also started manufacturing high performance vector supercomputers. Companies like NEC, Fujitsu and Hitachi were the main manufacturers. Hitachi developed S-810/210 and S-810/10 vector supercomputers in 1982. NEC developed SX-1 and Fujitsu developed VP-200. All these machines used semiconductor technologies to achieve speeds at par with Cray and Cyber. But their operating system and vectorisers were poorer than those of American companies.

## 1.3 PROBLEM SOLVING IN PARALLEL

This section discusses how a given task can be broken into smaller subtasks and how subtasks can be solved in parallel. However, it is essential to note that there are certain applications which are inherently sequential and if for such applications, a parallel computer is used then the performance does not improve.

### 1.3.1 Concept of Temporal Parallelism

In order to explain what is meant by parallelism inherent in the solution of a problem, let us discuss an example of submission of electricity bills. Suppose there are 10000 residents in a locality and they are supposed to submit their electricity bills in one office.

Let us assume the steps to submit the bill are as follows:

1) Go to the appropriate counter to take the form to submit the bill.

2) Submit the filled form along with cash.

3) Get the receipt of submitted bill.

Assume that there is only one counter with just single office person performing all the tasks of giving application forms, accepting the forms, counting the cash, returning the cash if the need be, and giving the receipts.

*This situation is an example of sequential execution*. Let us the approximate time taken by various of events be as follows:

Giving application form = 5 seconds
Accepting filled application form and counting the cash and returning, if required = 5mnts, i.e., 5 ×60= 300 sec.
Giving receipts = 5 seconds.
Total time taken in processing one bill = 5+300+5 = 310 seconds.

Now, if we have 3 persons sitting at three different counters with

i) One person giving the bill submission form

ii) One person accepting the cash and returning,if necessary and

iii) One person giving the receipt.

The time required to process one bill will be 300 seconds because the first and third activity will overlap with the second activity which takes 300 sec. whereas the first and last activity take only 10 secs each. This is an example of a parallel processing method as here 3 persons work in parallel. As three persons work in the same time, it is called temporal parallelism. However, this is a poor example of parallelism in the sense that one of the actions i.e., the second action takes 30 times of the time taken by each of the other two actions. The word 'temporal' means 'pertaining to time'. Here, a task is broken into many subtasks, and those subtasks are executed simultaneously in the time domain. In terms of computing application it can be said that parallel computing is possible, if it is possible, to break the computation or problem in to identical independent computation. Ideally, for parallel processing, the task should be divisible into a number of activities, each of which take roughly same amount of time as other activities.

### 1.3.2 Concept of Data Parallelism

consider the situation where the same problem of submission of 'electricity bill' is handled as follows:
Again, three are counters. However, now every counter handles all the tasks of a resident in respect of submission of his/her bill. Again, we assuming that time required to submit one bill form is the same as earlier, i.e., 5+300+5=310 sec.

We assume all the counters operate simultaneously and each person at a counter takes 310 seconds to process one bill. Then, time taken to process all the 10,000 bills will be $310 \times (9999/3) + 310 \times 1\text{sec}$.
This time is comparatively much less as compared to time taken in the earlier situations, viz. 3100000 sec. and 3000000 sec respectively.

The situation discussed here is the concept of data parallelism. In data parallelism, the complete set of data is divided into multiple blocks and operations on the blocks are applied parallely. As is clear from this example, data parallelism is faster as compared to earlier situations. Here, no synchronisation is required between counters(or processers). It is more tolerant of faults. The working of one person does not effect the other. There is no communication required between processors. Thus, interprocessor communication is less. Data parallelism has certain disadvantages. These are as follows:

i) The task to be performed by each processor is predecided i.e., asssignment of load is static.

ii) It should be possible to break the input task into mutally exclusive tasks. In the given example, space would be required counters. This requires multiple hardware which may be costly.

The estimation of speedup achieved by using the above type of parallel processing is as follows:

Let the number of jobs = m
Let the time to do a job = p

If each job is divided into k tasks,

Assuming task is ideally divisible into activities, as mentioned above then,

Time to complete one task = p/k
Time to complete n jobs without parallel processing = n.p
Time to complete n jobs with parallel processing = $\frac{n * p}{k}$

$$\text{Speed up} \quad \frac{\text{time to complete the task if parallelism is not used}}{\text{time to complete the task if parallelism is used}}$$

$$= \frac{np}{n\frac{p}{k}}$$

$$= k$$

## 1.4 PERFORMANCE EVALUATION

In this section, we discuss the primary attributes used to measure the performance of a computer system. Unit 2 of block 3 is completely devoted to performance evaluation of parallel computer systems. The performance attributes are:

i) **Cycle time(T):** It is the unit of time for all the operations of a computer system. It is the inverse of clock rate (l/f). The cycle time is represented in n sec.

ii) **Cycles Per Instruction(CPI):** Different instructions takes different number of cycles for exection. CPI is measurement of number of cycles per instruction

iii) **Instruction count($I_c$):** Number of instruction in a program is called instruction count. If we assume that all instructions have same number of cycles, then the total execution time of a program
= number of instruction in the program*number of cycle required by one instruction *time of one cycle.

Hence, execution time $T=I_c$*CPI*Tsec.

Practically the clock frequency of the system is specified in MHz. Also, the processor speed is measured in terms of million instructions per sec(MIPS).

## 1.5 SOME ELEMENTARY CONCEPTS

In this section, we shall give a brief introduction to the basic concepts like, program, process, thread, concurrency and granularity.

### 1.5.1 The Concept of Program

From the programmer's perspective, roughly a program is a
well-defined set of instructions, written in some programming language, with defined sets of inputs and outputs. From the operating systems perspective, a program is an executable file stored in a secondary memory. Software may consist of a single program or a number of programs. However, a program does nothing unless its instructions are executed by the processor. Thus a program is a passive entity.

### 1.5.2 The Concept of Process

Informally, a process is a program in execution, after the program has been loaded in the main memory. However, a process is more than just a program code. A process has its own address space, value of program counter, return addresses, temporary variables, file handles, security attributes, threads, etc.

Each process has a life cycle, which consists of creation, execution and termination phases. A process may create several new processes, which in turn may also create a new process. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.
ii) **Allocating an address space:** There are two ways to allocate address space to processes: sharing the address space among the created processes or allocating separate space to each process.
iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.
iv) **Passing the process description to the process scheduler:** once, the three steps of process creation as mentioned above are completed, the information gatherd through the above-mentioned steps is sent to the process scheduler which allocates processor(s) resources to various competing to-be-executed processses queue.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process state, state transition and scheduling policy.

A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single processor or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.
- **Terminated:** The process has finished execution.

At any time, a process may be in any one of the above mentioned states. As soon as the process is admitted into job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. If the process is completely executed then it is terminated and we say that it is in terminated state. However, the process may return to ready state due to some interrupts or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown in *Figure 1*:

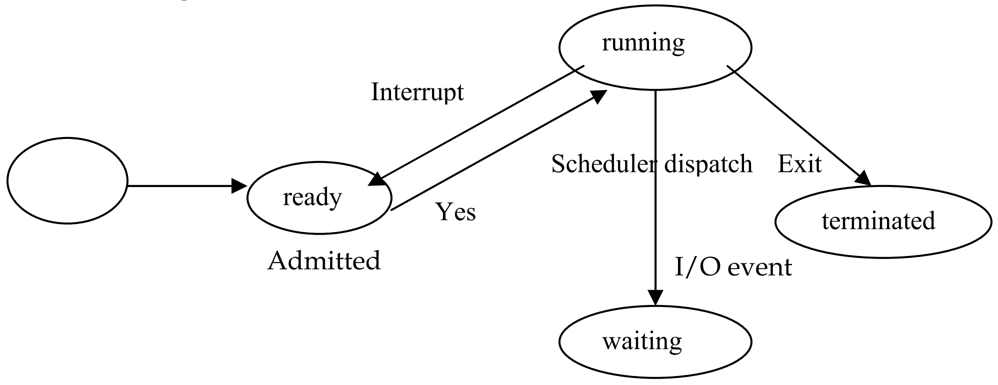

**Figure1: Process state transition diagram**

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. For example, to select a process to be executed, one of the secheduling policy may be: First In First Out(FIFO).

When the process finishes execution it is terminated by system calls like *abort*, releasing all the allocated resources.

### 1.5.3 The Concept of Thread

Thread is a sequential flow of control within a process. A process can contain one or more threads. Threads have their own program counter and register values, but they are share the memory space and other resources of the process. Each process starts with a single thread. During the execution other threads may be created as and when required. Like processes, each thread has an execution state (running, ready, blocked or terminated). A thread has access to the memory address space and resources of its process. Threads have similar life cycles as the processes do. A single processor system can support concurrency by switching execution between two or more threads. A multi-processor system can support parallel concurrency by executing a separate thread on each processor. There are three basic methods in concurrent programming languages for creating and terminating threads:

- **Unsynchronised creation and termination:** In this method threads are created and terminated using library functions such as CREATE_PROCESS, START_PROCESS, CREATE_THREAD, and START_THREAD. As a result of these function calls a new process or thread is created and starts running independent of its parents.

- **Unsynchronised creation and synchronized termination:** This method uses two instructions: FORK and JOIN. The FORK instruction creates a new process or thread. When the parent needs the child's (process or thread) result, it calls JOIN instruction. At this junction two threads (processes) are synchronised.

- **Synchronised creation and termination:** The most frequently system construct to implement synchronization is

COBEGIN…COEND. The threads between the COBEGIN…COEND construct are executed in parallel. The termination of parent-child is suspended until all child threads are terminated.

We can think of a thread as basically a lightweight process. However, threads offer some advantages over processes. The advantages are:

i) It takes less time to create and terminate a new thread than to create, and terminate a process. The reason being that a newly created thread uses the current process address space.

ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.

iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share among other entities the address space. So, data produced by one thread is immediately available to all the other threads.

### 1.5.4 The Concept of Concurrent and Parallel Execution

Real world systems are naturally concurrent, and computer science is about modeling the real world. Examples of real world systems which require concurrency are railway networks and machines in a factory. In the computer world, many new operating systems support concurrency. While working on our personal computers, we may download a file, listen to streaming audio, have a clock running , print something and type in a text editor. A multiprocessor or a distributed computer system can better exploit the inherent concurrency in problem solutions than a uniprocessor system. Concurrency is achieved either by creating simultaneous processes or by creating threads within a process. Whichever of these methods is used, it requires a lot of effort to synchronise the processses/threads to avoid race conditions, deadlocks and starvations.

Study of concurrent and parallel executions is important due to following reasons:

i) Some problems are most naturally solved by using a set of co-operating processes.
ii) To reduce the execution time.

The words "concurrent" and "parallel" are often used interchangeably, however they are distinct.

Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has dual nature; it is sequential in a small time scale, but simultaneous in a large time scale. In our context, a processor works as server and process or thread works as client. Examples of concurrent languages include Adam, concurrent Pascal, Modula-2 and concurrent PROLOG).
Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. Examples of parallel languages includes Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

### 1.5.5 Granularity

Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems, parallel parts are relatively small and that means more frequent communication. They have low computation to communication ratio and require high communication overhead. In coarse-grained systems parallel parts are relatively large and that means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amount of computational work is done. They have high computation to communication ratio and imply more opportunity for performance increase.

The extent of granularity in a system is determined by the algorithm applied and the hardware environment in which it runs. On an architecturally neutral system, the granularity does affect the performance of the resulting program. The communication of data required to start a large process may take a considerable amount of time. On the other hand, a large process will often have less communication to do during processing. A process may need only a small amount of data to get going, but may need to receive more data to continue processing, or may need to do a lot of communication with other processes in order to perform its processing. In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.

### 1.5.6 Potential of Parallelism

Problems in the real world vary in respect of the degree of inherent parallelism inherent in the respective problem domain. Some problems may be easily parallelized. On the other hand, there are some inherent sequential problems (for example computation of Fibonacci sequence) whose parallelization is nearly impossible. The extent of parallelism may be improved by appropriate design of an algorithm to solve the problem consideration. If processes don't share address space and we could eliminate data dependency among instructions, we can achieve higher level of parallelism. The concept of speed up is used as a measure of the *speed up* that indicates up to what extent to which a sequential program can be parallelised. Speed up may be taken as a sort of degree of inherent parallelism in a program. In this respect, Amdahl, has given a law, known as Amdahl's Law, which states that potential program speedup is defined by the fraction of code (P) that can be parallelised:

$$\text{Speed up} = \frac{1}{1-P}$$

If no part of the code can be parallelized, P = 0 and the speedup = 1 i.e. it is an inherently sequential program. If all of the code is parallelized, P = 1, the speedup is infinite. But practically, the code in no program can made 100% parallel. Hence speed up can never be infinite.

If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

If we introduce the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{Speed up} = \frac{1}{{}^{P}\!/_{N} + S}$$

Where P = parallel fraction, N = number of processors and S = serial fraction.

The *Table 1* shows the value of speed up for different values N and P.

Table 1

| N | Speedup P = .50 | Speedup P = .90 | Speedup P = .99 |
|---|---|---|---|
| 10 | 1.82 | 5.26 | 9.17 |
| 100 | 1.98 | 9.17 | 50.25 |
| 1000 | 1.99 | 9.91 | 90.99 |
| 10000 | 1.99 | 9.91 | 99.02 |

The *Table 1* suggests that speed up increases as P increases. However, after a certain limits N does not have much impact on the value of speed up. The reason being that, for N processors to remain active, the code should be, in some way or other, be divisible in roughly N parts, independent part, each part taking almost same amount of time.

**Check Your Progress 1**

1) Explain the life cycle of a process.

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………

2) What are the advantages of threads over processes?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………

3) Differentiate concurrent and parallel executions.

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………

4) What do understand by the granularity of a parallel system?

…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………………..

## 1.6 THE NEED OF PARALLEL COMPUTATION

With the progress of computer science, computational speed of the processors has also increased many a time. However, there are certain constraints as we go upwards and face large complex problems. So we have to look for alternatives. The answer lies in parallel computing. There are two primary reasons for using parallel computing: save time and solve larger problems. It is obvious that with the increase in number of processors working in parallel, computation time is bound to reduce. Also, they're some scientific problems that even the fastest processor to takes months or even years to solve. However, with the application of parallel computing these problems may be solved in a few hours. Other reasons to adopt parallel computing are:

i) **Cost savings:** We can use multiple cheap computing resources instead of paying
ii) heavily for a supercomputer.

iii) **Overcoming memory constraints:** Single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle. So if we combine the memory resources of multiple computers then we can easily fulfill the memory requirements of the large-size problems.

iv) **Limits to serial computing:** Both physical and practical factors pose significant constraints to simply building ever faster serial computers. The speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light ($3*10^8$ m/sec) and the transmission limit of copper wire ($9*10^8$ m/sec). Increasing speeds necessitate increasing proximity of processing elements. Secondly, processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be made. It is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

## 1.7 LEVELS OF PARALLEL PROCESSING

Depending upon the problem under consideration, parallelism in the solution of the problem may be achieved at different levels and in different ways. This section discusses various levels of parallelism. Parallelism in a problem and its possible solutions may be exploited either manually by the programmer or through automating compilers. We can have parallel processing at four levels.

### 1.7.1 Instruction Level

It refers to the situation where different instructions of a program are executed by different processing elements. Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe. The earliest use of instruction level parallelism in designing PE's to enhance processing speed is pipelining. Pipelining was extensively used in early Reduced Instruction Set Computer (RISC). After RISC, super scalar processors were developed which execute multiple instruction in one clock cycle. The super scalar processor design exploits the parallelism available at instruction level by enhancing the number of arithmetic and functional units in PE's. The concept of instruction level parallelism was further modified and applied in the design of Very Large Instruction Word (VLIW) processor, in which one instruction word encodes more than one operation. The idea of executing a number of instructions of a program in parallel by scheduling them on a single processor has been a major driving force in the design of recent processors.

### 1.7.2 Loop Level

At this level, consecutive loop iterations are the candidates for parallel execution. However, data dependencies between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

Example: In the following loop in C language,

```
for (i=0;  i <= n; i++)
A(i) = B(i)+ C(i)
```

Each of the instruction A(i) =B(i)+C(i) can be executed by different processing elements provided there are at least n processing elements. However, the instructions in the loop:

```
for ( J=0, J<= n, J++)
A(J) = A(J-1) + B(J)
```

cannot be executed parallely as A(J) is data dependent on A(J-1). This means that before exploiting the loop level parallelism the data dependencies must be checked:

### 1.7.3 Procedure Level

Here, parallelism is available in the form of parallel executable procedures. In this case, the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

### 1.7.4 Program Level

This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by operating the system at this level.

### Check Your Progress 2

1) What are the advantages of parallel processing over sequential computations?
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
………………………………………………………………………………………
2) Explains the various levels of parallel processing.
…………………………………………………………………………………………
…………………………………………………………………………………………
…………………………………………………………………………………………
……………………………………………………………………………

## 1.8 DATAFLOW COMPUTING

An alternative to the von Neumann model of computation is the dataflow computation model. In a dataflow model, control is tied to the flow of data. The order of instructions in the program plays no role on the execution order. Execution of an instruction can take place when all the data needed by the instruction are available. Data is in continuous flow independent of reusable memory cells and its availability initiates execution. Since, data is available for several instructions at the same time, these instructions can be executed in parallel.

For the purpose of exploiting parallelism in computation Data Flow Graph notation is used to represent computations. In a data flow graph, the nodes represent instructions of the program and the edges represent data dependency between instructions. As an example, the dataflow graph for the instruction z = w × (x+y) is shown in *Figure 2*.

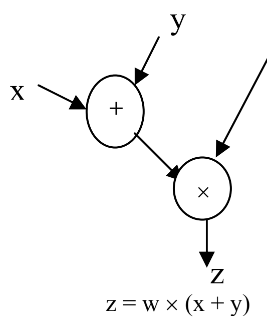

**Figure 2: DFG for z = w × (x+y)**

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each of its input edges. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

The advantage of the dataflow concept is that nodes of DFG can be self-scheduled. However, the hardware support to recognize the availability of necessary data is much more complicated than the von Neumann model. The example of dataflow computer includes Manchester Data Flow Machine, and MIT Tagged Token Data Flow architecture.

## 1.9 APPLICATIONS OF PARALLEL PROCESSING

Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world. In the natural world, it is quite common to find many complex, interrelated events happening at the same time. Examples of concurrent processing in natural and man-made environments include:

- Automobile assembly line
- Daily operations within a business
- Building a shopping mall
- Ordering an aloo tikki burger at the drive through.

Hence, parallel computing has been considered to be “the high end of computing” and has been motivated by numerical simulations of complex systems and “Grand Challenge Problems” such as:

- Weather forecasting
- Predicting results of chemical and nuclear reactions

- DNA structures of various species
- Design of mechanical devices
- Design of electronic circuits
- Design of complex manufacturing processes
- Accessing of large databases
- Design of oil exploration systems
- Design of web search engines, web based business services
- Design of computer-aided diagnosis in medicine
- Development of MIS for national and multi-national corporations
- Development of advanced graphics and virtual reality software, particularly for the entertainment industry, including networked video and multi-media technologies
- Collaborative work (virtual) environments

### 1.9.1 Scientific Applications/Image processing

Most of parallel processing applications from science and other academic disciplines, are mainly have based upon numerical simulations where vast quantities of data must be processed, in order to create or test a model. Examples of such applications include:

- Global atmospheric circulation,
- Blood flow circulation in the heart,
- The evolution of galaxies,
- Atomic particle movement,
- Optimisation of mechanical components.

The production of realistic moving images for television and the film industry has become a big business. In the area of large computer animation, though much of the work can be done on high specification workstations, yet the input will often involve the application of parallel processing. Even at the cheap end of the image production spectrum, affordable systems for small production companies have been formed by connecting cheap PC technology using a small LAN to farm off processing work on each image to be produced.

### 1.9.2 Engineering Applications

Some of the engineering applications are:

- Simulations of artificial ecosystems,
- Airflow circulation over aircraft components.

Airflow circulation is a particularly important application. A large aircraft design company might perform up to five or six full body simulations per working day.

### 1.9.3 Database Query/Answering Systems

There are a large number of opportunities for speed-up through parallelizing a Database Management System. However, the actual application of parallelism required depends very much on the application area that the DBMS is used for. For example, in the financial sector the DBMS generally is used for short simple transactions, but with a high number of transactions per second. On the other hand in a Computer Aided Design (CAD) situation (e.g., VLSI design) the transactions would be long and with low traffic rates. In a Text query system, the database would undergo few updates, but would be required to do complex pattern matching queries over a large number of entries. An example of a computer designed to speed up database queries is the Teradata computer, which employs parallelism in processing complex queries.

### 1.9.4 AI Applications

Search is a vital component of an AI system, and the search operations are performed over large quantities of complex structured data using unstructured inputs. Applications of parallelism include:

- Search through the rules of a production system,
- Using fine-grain parallelism to search the semantic networks created by NETL,
- Implementation of Genetic Algorithms,
- Neural Network processors,
- Preprocessing inputs from complex environments, such as visual stimuli.

### 1.9.5 Mathematical Simulation and Modeling Applications

The tasks involving mathematical simulation and modeling require a lot of parallel processing. Three basic formalisms in mathematical simulation and modeling are Discrete Time System Simulation (DTSS), Differential Equation System Simulation (DESS) and Discrete Event System Simulation (DEVS). All other formalisms are combinations of these three formalisms. DEVS is the most popular. Consequently a number of software tools have been designed for DEVS. Some of such softwares are:

- Parsec, a C-based simulation language for sequential and parallel execution of discrete-event simulation models.
- Omnet++ a discrete-event simulation software development environment written in C++.
- Desmo-J a Discrete event simulation framework in Java.
- Adevs (A Discrete Event System simulator) is a C++ library for constructing discrete event simulations based on the Parallel DEVS and Dynamic Structure DEVS formalisms.
- Any Logic is a professional simulation tool for complex discrete, continuous and hybrid systems.

## 1.10 INDIA'S PARALLEL COMPUTERS

In India, the development and design of parallel computers started in the early 80's. The Indian Government established the Centre for Development of Advanced Computing (CDAC) in 1988 with the aim of building high-speed parallel machines. CDAC designed and built a 256 processors computer using INMOS T8000 series processors in 1991. The other groups which developed parallel machines were at Centre for Development of Telematics, Bhabha Atomic Research Centre, Indian Institute for Sciences, Defence Research and Development Organisation. The systems developed by these organisations are said to be the state of the art of parallel computers. It is generally agreed that all the computers built by 2020 will be inherently parallel.

## India's Parallel Computer

Next, we enumerate sailent features of various generations of parallel systems developed in India.

*Sailent Features of PARAM series:*

PARAM 8000 CDAC 1991: 256 Processor parallel computer, INMOS 8000 transputer as processing element. Peak performance of 1 Gigaflop. Application software weak.

PARAM 8600 CDAC 1994: PARAM 8000 enhanced with Intel i860 vector microprocessor. One vector processor for 4 INMOS 8000. Vectorized Fortran. Improved software for numerical applications.

PARAM 9000/SS CDAC 1996 : Used Sunsparc II processors and an interconnection switch made of INMOS transputers.

*Salient Features of MARK Series:*

Flosolver Mark I NAL 1986: Used 4 Intel 8086 processors with 8087 co-processors. Proof of concept design.

Flosolver Mark II NAL 1988: 16 Intel 80386 processor and 80387 floating-point processor connected to Multibus II backplane bus for interprocessor communication. Used for solving fluid dynamics problems using Fortran.

Flosolver Mark III NAL 1991: 8 Intel i860 vector processors connected using message passing co-processor on a back plane bus. i860 were rated at 80 Mflops peak. Fluid dynamics Codes were optimized for the architecture.

*Salient Features of ANUPAM Series:*

ANUPAM Model 1 BARC 1993: 8 Intel i860 processors connected to a Multibus II. Eight such clusters connected by using 16-bit SCSI interface. Used Front-end processor to allocate tasks to the parallel computing cluster. One parallel program at a time could be run. Fortran environment.

ANUPAM Model 2 BARC 1997: DEC Alpha processors connected by ATM switch in a cluster. DEC Unix environment. High Performance Fortran compiler to run data parallel programs.

## 1.11 PARALLEL TERMINOLOGY USED

Some of the more commonly used terms associated with parallel computing are listed below.

*Task*

A logically discrete section of a computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

*Parallel Task*

A task, some parts of which can be executed by more than one multiple processor at same point of time (yields correct results)

*Serial Execution*
Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, even most of the parallel tasks also have some sections of a parallel program that must be executed serially.

*Parallel Execution*

Execution of sections of a program by more than one processor at the same point of time.

*Shared Memory*

Refers to the memory component of a computer system in which the memory can accessed directly by any of the processors in the system.

*Distributed Memory*

Refers to network based memory in which there is no common address space for the various memory modules comprising the memory system of the (networked) system. Generally, a processor or a group of processors have some memory module associated with it, independent of the memory modules associated with other processors or group of processors.

*Communications*

Parallel tasks typically need to exchange data. There are several ways in which this can be accomplished, such as, through a shared memory bus or over a network. The actual event of data exchange is commonly referred to as communication regardless of the method employed.

*Synchronization*

The process of the coordination of parallel tasks in real time, very often associated with communications is called synchronisation. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's execution time to increase.

*Granularity*

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

***Coarse Granularity:*** relatively large amounts of computational work are done between communication events

***Fine Granularity:*** relatively small amounts of computational work are done between communication events

*Observed Speedup*
Observed speedup of a code which has been parallelized, is defined as:

$$\frac{\texttt{wall-clock time of serial}}{\texttt{wall-clock time of parallel}}$$

Granularity is one of the simplest and most widely used indicators for a parallel program's performance.

*Parallel Overhead*

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

Task start-up time
Synchronisations
Data communications
Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.

*Massively Parallel System*

Refers to a parallel computer system having a large number of processors. The number in 'a large number of' keeps increasing, and, currently it means more than 1000.

*Scalability*

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in (parallel) speedup with the addition of more processors. Factors that contribute to scalability include:

*Hardware* - particularly memory-cpu bandwidths and network communications:

- *Application algorithm*
- *Parallel overhead related*
- *Characteristics of your specific application and coding*

## Check Your Progress 3

1) Explain dataflow computation model.
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
…………………………………………………………………………………….
2) Enumerate applications of parallel processing.
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………
……………………………………………………………………………………………

## 1.12 SUMMARY

In this unit, a number of introductory issues and concepts in respect of parallel computing are discussed. First of all, section 1.2 briefly discusses history of parallel computing. Next section discusses two types of parallelism, viz, temporal and data parallelisms. The issues relating to performance evaluation of a parallel system are discussed in section 1.4. Section 1.5 defines a number of new concepts. Next section explains why parallel computation is essential for solving computationally difficult problems. Section 1.7 discusses how parallelism can be achieved at different levels within a program. Dataflow computing is a different paradigm of computing as compared to the most frequently used Von-Neumann-architecture based computing. Dataflow computing allows to exploit easily the inherent parallelism in a problem and its solution. Issues related to Dataflow computing are discussed in section 1.8. Applications of parallel computing are discussed in section 1.9. India's effort at developing parallel computers is briefly discussed in section 1.10. A glossary of parallel computing terms is given in section 1.11.

## 1.13 SOLUTIONS/ANSWERS

### Check Your Progress 1

1) Each process has a life cycle, which consists of creation, execution and termination phases of the process. A process may create several new processes, which in turn may also create still new processes, using system calls. In UNIX operating system environment, a new process is created by *fork* system call. Process creation requires the following four actions:

i) **Setting up the process description:** Setting up the process description requires the creation of a *Process Control Block* (PCB). A PCB contains basic data such as process identification number, owner, process status, description of the allocated address space and other implementation dependent process specific information needed for process management.
ii) **Allocating an address space:** There are two ways to allocate address space to processes; sharing the address space among the created processes or allocating separate space to each process.
iii) **Loading the program into the allocated address space:** The executable program file is loaded into the allocated memory space.
iv) **Passing the process description to the process scheduler:** The process created is then passed to the process scheduler which allocates the processor to the competing processes.

The process execution phase is controlled by the process scheduler. Process scheduling may be per process or per thread. The process scheduling involves three concepts: process states, state transition diagram and scheduling policy.
A process may be in one of the following states:

- **New:** The process is being created.
- **Running:** The process is being executed on a single or multiple processors.
- **Waiting:** The process is waiting for some event to occur.
- **Ready:** The process is ready to be executed if a processor is available.

- **Terminated:** The process has finished execution.

At any time a process may be in any one of the above said states. As soon as the process is admitted into the job queue, it goes into ready state. When the process scheduler dispatches the process, its state becomes running. When the process is completely executed then it is terminated and we say that it is in the terminated state. However, the process may return to ready state due to some interruption or may go to waiting state due to some I/O activity. When I/O activity is over it may go to ready state. The state transition diagram is shown below in the Figure 3:

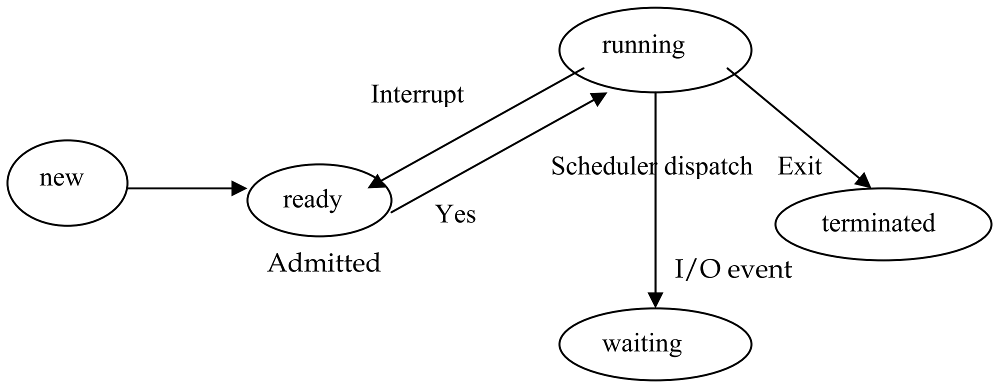

**Figure 3: Process state transition diagram**

The scheduling policy may be either pre-emptive or non pre-emptive. In pre-emptive policy, the process may be interrupted. Operating systems have different scheduling policies. One of the well-known policies is First In First Out(FIFO) to select the process to be executed.When the process finishes execution it is terminated by system calls like *abort*.

2) Some of the advantages that threads offer over processes include:

i) It takes less time to create and terminate a new thread than it takes for a process, because the newly created thread uses the current process address space.
ii) It takes less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
iii) Less communication overheads -- communicating between the threads of one process is simple because the threads share the address space, in particular. So, data produced by one thread is immediately available to all the other threads.

3) The words “concurrent” and “parallel” are often used interchangeably, however they are distinct. Concurrent execution is the temporal behaviour of the N-client 1-server model where only one client is served at any given moment. It has a dual nature: it is sequential in a small time scale, but simultaneous in large time scale. In our context the processor works as a server and a process or a thread works as a client. For facilitating expression of concurrent programs, number of concurrent languages are available including Ada, concurrent pascal, Modula-2 and concurrent PROLOG.

Parallel execution is associated with the N-client N-server model. It allows the servicing of more than one client at the same time as the number of servers is more than one. For facilitating expression of parallel programs, a number of parallel languages are available including: Occam-2, Parallel C and strand-88.

Parallel execution does not need explicit concurrency in the language. Parallelism can be achieved by the underlying hardware. Similarly, we can have concurrency in a language without parallel execution. This is the case when a program is executed on a single processor.

4) Granularity refers to the amount of computation done in parallel relative to the size of the whole program. In parallel computing, granularity is a qualitative measure of the ratio of computation to communication. According to granularity of the system, parallel-processing systems can be divided into two groups: fine-grain systems and coarse-grain systems. In fine-grained systems parallel parts are relatively small and which means more frequent communication. Fine-grain processings have low computation to communication ratio and require high communication overhead. In coarse grained systems parallel parts are relatively large and which means more computation and less communication. If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation. On the other hand, in coarse-grain parallel systems, relatively large amounts of computational work is done. Coarse-grain processings have high computation to communication ratio and imply more opportunity for performance increase.

## Check Your Progress 2

1) Parallel computing has the following advantages over sequential computing:

i) Saves time
ii) Solves larger problems.
iii) Large pool of memory.

2) Levels of parallel processing:

We can have parallel processing at four levels.

i) **Instruction Level:** Most processors have several execution units and can execute several instructions (usually machine level) at the same time. Good compilers can reorder instructions to maximize instruction throughput. Often the processor itself can do this. Modern processors even parallelize execution of micro-steps of instructions within the same pipe.

ii) **Loop Level:** Here, consecutive loop iterations are candidates for parallel execution. However, data between subsequent iterations may restrict parallel execution of instructions at loop level. There is a lot of scope for parallel execution at loop level.

iii) **Procedure Level:** Here parallelism is available in the form of parallel executable procedures. Here the design of the algorithm plays a major role. For example each thread in Java can be spawned to run a function or method.

iv) **Program Level:** This is usually the responsibility of the operating system, which runs processes concurrently. Different programs are obviously independent of each other. So parallelism can be extracted by the operating system at this level.

## Check Your Progress 3

1) An alternative to the von Neumann model of computation is dataflow computation model. In a dataflow model control is tied to the flow of data. The order of instructions in the program plays no role in the execution order. Computations take place when all the data items needed for initiating execution of an instruction are available. Data is in continuous flow independent of reusable memory cells and its availability initiates execution. Since data may be available for several instructions at the same time, these instructions can be executed in parallel.

The potential for parallel computation, is reflected by the dataflow graph, the nodes of which are the instructions of the program and the edges of which represent data dependency between instructions. The dataflow graph for the instruction z = w × (x+y) is shown below.
.

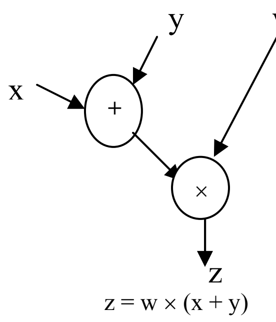

**Figure 4: DFG for z = w × (x+y)**

Data moves on the edges of the graph in the form of data tokens, which contain data values and status information. The asynchronous parallel computation is determined by the firing rule, which is expressed by means of tokens: a node of DFG can fire if there is a token on each input edge. If a node fires, it consumes the input tokens, performs the associated operation and places result tokens on the output edge. Graph nodes can be single instructions or tasks comprising multiple instructions.

2) Please refer Section 1.9.