MODULE 2: GOALS OF PARALLELISM
WEEK 2

Learning Outcomes:

After completing this course you are expected to demonstrate the following:

1. Explain the different goals of parallelism in controlling access to shared resourses as throughput versus concurrency.

A. Engage

Trivia

General-Purpose Computing On Graphics Processing Units

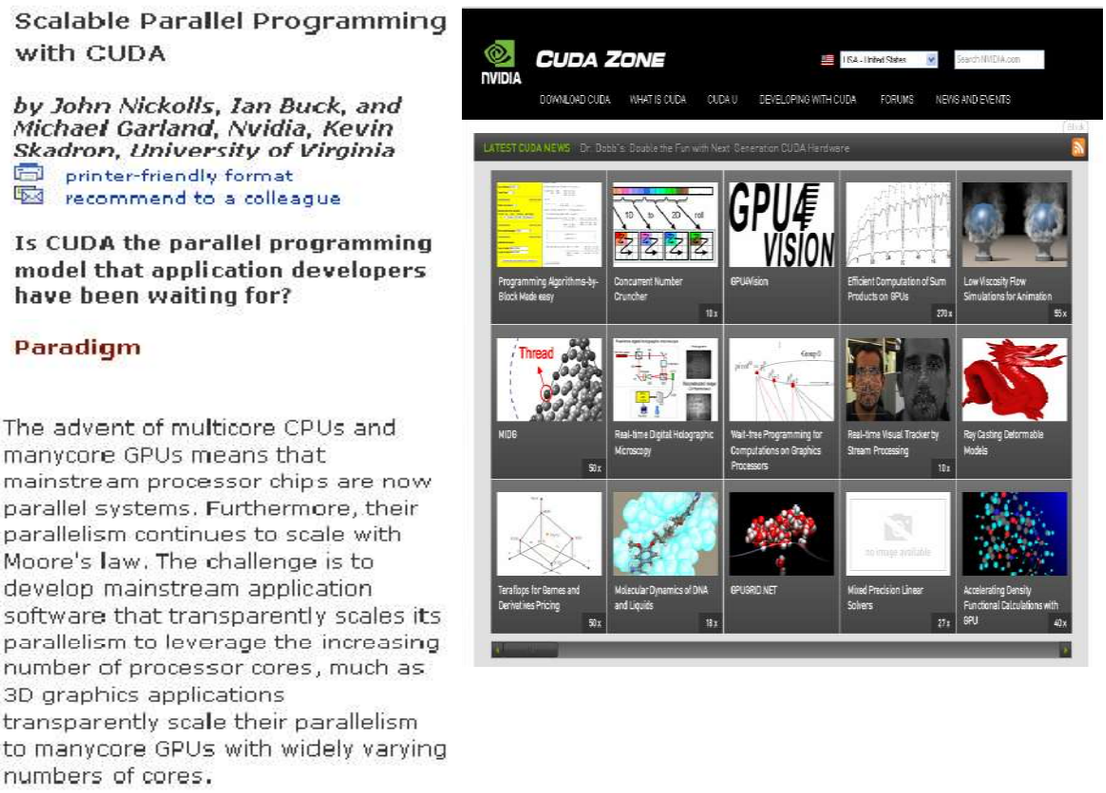

Figure 2.1 Taken From http://www.acmqueue.org 04/08

B. Explore

Slideshare Title: **Parallel Processing**

Slideshare Link: https://www.slideshare.net/mobile/rajshreemuthiah/parallel-processing-180676361

C. Explain

In computers, parallel computing is closely related to parallel processing (or concurrent computing). It is the form of computation in which concomitant ("in parallel") use of multiple CPUs that is carried out simultaneously with shared-memory systems to solving a supercomputing computational problem. Parallelism is the process of large computations, which can be broken down into multiple processors that can process independently and whose results combined upon completion. Parallelism has long employed in high-performance supercomputing.

Parallel processing generally implemented in the broad spectrum of applications that need massive amounts of calculations. The primary goal of parallel computing is to increase the computational power available to your essential applications. Typically, This infrastructure is where the set of processors are present on a server, or separate servers are connected to each other to solve a computational problem.

In the earliest computer software, that executes a single instruction (having a single Central Processing Unit (CPU)) at a time that has written for serial computation. A Problem is broken down into multiple series of instructions, and that Instructions executed one after another. Only one of computational instruction complete at a time.

D. Elaborate

Introduction

Parallelism vs. Concurrency

What is concurrency and how is it different from parallelism?

Concurrency - capable of operating at the same time.

It appears that **"concurrency"** and **"parallelism"** are synonyms. But there is a subtle difference between the two. We reserve "parallelism" to refer to situations where actions truly happen at the same time, as in four processes (tasks) executing on four processing agents (CPUs) simultaneously. "Concurrency" refers to both situations -- ones which are truly parallel and ones which have the illusion of parallelism, as in four processes (tasks) time-sliced on one processing agent (CPU).

Unfortunately, the two terms "concurrency" and "parallelism" are not used consistently in computer literature. Further, even though many times the term "concurrency" or "concurrent" would be more appropriate, "parallel" is used since it is a current buzz word with more appeal. Therefore, we hear of "parallel programming languages" and "parallel architectures" as opposed to "concurrent programming languages" and "concurrent architectures," which would be more accurate. In this book we will use parallelism for simultaneous actions and concurrency if there is a possibility of illusion of simultaneous actions.

Levels of Parallelism

In modern computers, parallelism appears at many levels. At the very lowest level, signals travel in parallel along parallel data paths. At a slightly higher level of functionality, functional units such as adders replicate components which operate in parallel for faster performance. Some computers, e. g., the Cray-1, have multiple functional units which allow the operations of addition and multiplication to be done in parallel. Most larger computers overlap I/O operations, e. g., adisk read, for one user with execution of instructions for another user. For faster accesses to memory, some computers use memory interleaving where several banks of memory can be accessed simultaneously or in parallel.

At a higher level of parallelism, some computers such as the Cray Y-MP have multiple Central Processing Units (CPUs) which operate in parallel. At an even higher level of parallelism, one can connect several computers together, for example in a local area network. It is important

tounderstand that parallelism is pervasive through all of computer science and used across many levels of computer technology.

Why Use Parallelism?

The main reason to use parallelism in a design (hardware or software) is for higher performance or speed. All of today's supercomputers use parallelism extensively to increase performance. Computers have increased in speed to the point where computer circuits are reaching physical limits such as the speed of light. Therefore, in order to improve performance, parallelismmust be used.

Raw speed is not the only reason to use parallelism. A computer designer might replicate components to increase reliability. For example, the Space Shuttle's guidance system is composed of three computers that compare their results against each other. The Shuttle can fly with just one of the computers working and the other two as backups. A system of this nature will be more tolerant to errors or faults and is, therefore, called fault tolerant.

Parallelism might be used to decentralize control. Rather than one large computer, a bank might have a network of smaller computers at the main and branch offices. This distributive approach to computing has the advantage of local control by the branch managers as well as gradual degradation of services if a computer should fail.

Parallelism (really concurrency) is an important problem solving paradigm. Nature is parallel. As you talk with your friends, your heart pumps, your lungs breathe, your eyes move and your tongue wags, all in parallel. Many problems are inherently parallel and to solve them sequentially forces us to distort the solution. For example, consider the actions of a crowd of people waiting for and riding on an elevator (activities). Individuals at different floors may push a button (an event) at the same time as someone inside the elevator. To model properly the behavior of the elevator, for example, by a computer program, you must deal with these parallel activities and events.

Why Study Parallel Processing?

In recent years, parallel processing has revolutionized scientific computing and is beginning to enter the world of every day data processing in the form of distributed databases. Scientific programmers need to understand the principles of parallel processing to effectively program the computers of the future.

All of today's supercomputers rely heavily on parallelism. Parallelism is used at the software level as well as in the architectural design of hardware. The United States' success in building most of the supercomputers has become an issue of national pride and a symbol of technological leadership. The race to hold its lead over the Japanese and European competition is intense. In order to compete in a world economy, countries require innovative scientists, engineers and computer scientists to utilize the supercomputers in an effective manner.

The use of parallel programming has seen a dramatic increase in recent years. Not only may the principles of parallelism be used to increase the performance of hardware, but the ideas of parallelism may also be incorporated into a programming language. Using such a language is called parallel programming.

Parallel programming - programming in a language which has explicit parallel (concurrent) constructs or features.

These new parallel programming languages reflect several important developments in computer science. First, the realization that the real world is parallel, especially if the problem has asynchronous events. To properly model the world, we need expressibility beyond what is available in current sequential programming languages. Second, parallelism (or better, concurrency) is a fundamental element of algorithms along with selection, repetition and recursion. The study of parallel algorithms is an interesting topic in its own right. Third, parallelism is an important abstraction for the design of software and understanding of computation. In fact, one can treat all of sequential programming as a special case of parallel programming. Research exploring these issues in parallel programming is very active at universities today.

What is Parallel Processing?

We have used the phrase "parallel processing" several times in the text so far. Now it is time to distinguish the phrase from several others, notably "parallel programming" and "parallel computing." Unfortunately, the discipline of computer science has problems with the meaning of terms. There are no generally accepted definitions of parallel processing, etc. Therefore, we need to describe how we will use these terms.

Parallel processing - all the subject matter where the principles of parallelism are applicable.

Parallel processing is a sub-field of computer science which includes ideas from theoretical computer science, computer architecture, programming languages, algorithms and applications areas such as computer graphics and artificial intelligence.

What is a parallel computer?

If we say that a parallel computer is one that uses parallelism then we must include all computers! All computers use parallelism at the bit level for data paths. But clearly, we say some machines are parallel ones and other are not. To distinguish, we need a test. We propose a test that is imprecise, but which is generally close to common usage. If a user can write a program which can determine if a parallel architectural feature is present, we say the parallelism is visible. For example, one cannot write a program to determine the width of a data bus. But, one can write a program to determine if vectorization, as on the Cray-1, is available.

Parallel computing - computing on a computer where the parallelism is "visible" to the applications programmer.

The parallel features could be constructs in the language like PAR in the language Occam or could be extensions to a traditional sequential language, e. g., FORTRAN 77 or C, using library routines.

Note, that one could be engaged in parallel computing but not parallel programming, for example, a programmer writing traditional FORTRAN 77 on a Cray-1. Here there are no parallel features in the FORTRAN, but the user still must be aware of the parallel architecture of the machine for effective performance. We will explore this more later.

Parallel Computing vs. Distributed Computing

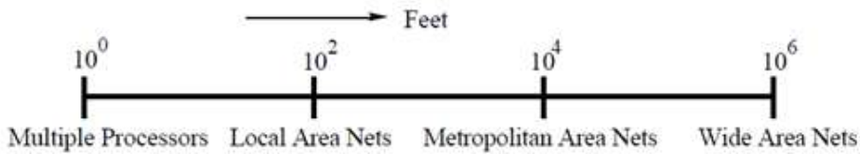

Figure 2.2 Spectrum of Parallel and Distributed Computing

In parallel computing, we limit the situations to solving one problem on multiple processing elements, for example, eight processors of a Cray Y-MP. The processors are tightly coupled for fast coordination. In distributed computing, the processors are loosely coupled, usually separated by a distance of many feet or miles to form a network; the coordination is usually for network services, e. g., a database search request; and not the solution to a single problem. A major concern in the study of parallel processing is performance of algorithms, whereas a major concern in the study of distributed computing is dealing with uncertainty. The student of distributed systems has to realize that the exact state of the whole system cannot be instantaneously computed. Every distributed operation may involve network traversals, which cannot be performed instantaneously [Plouzeau, 1992]. Nevertheless, we must be aware that parallel computing/distributed computing is really a continuum of networks based on the average distance between nodes.

There is no reason why parallel computing could not be performed on a Local Area Network (LAN) of workstations, e. g., ten SPARC Stations. If the solution to one end-user problem is spread over many workstations, we would say we are engaged in parallel computing. However, the communication over a network is much slower than between tightly coupled processors.

Therefore, in order to be effective, the solution will require very little communication compared to the computation.

Two Themes of Parallelism: Replication and Pipelining

One way to organize activities in parallel is to replicate agents to do part of the activities. If we want to lay a brick wall faster, we hire more brick layers.

Replication - organizing additional agents to perform part of the activity.

This form of parallelism is easy to understand and is widely used. We replicate data wires to allow signals to travel in parallel. We replicate functional units as in the Cray-1. We replicate memorybanks in an interleaving scheme. We replicate CPUs as in the Cray Y-MP, which can have up to eight CPUs.

The second way to organize activities in parallel is to specialize the agents into an assembly line. Henry Ford revolutionized industry by using assembly line methods in producing his lowpriced Model T Ford (1908). Instead of each worker building a car from scratch, the worker specialized in doing one task very well. The cars ride on a conveyor belt which moves through the factory. Each worker performs his or her task on a car as it moves by. When finished, the worker works on the next car on the conveyor belt. As long as the assembly line moves smoothly with no interruptions, a finished car is rolled off the line every few minutes. For historical reasons, the idea of assembly line is called “pipelining” in computer science.

Pipelining - organizing agents in an assembly line where each agent performs a specific part of the activity.

Replication and pipelining (assembly lining) are the two ways of organizing parallel activities. We will see these two themes used over and over again as we study parallel processing.

Speedup

How much faster are the parallel organizations? If it takes one unit of time for one bricklayer to build a wall, how long for n bricklayers? Let us assume the best case where the bricklayers do not interfere with each other, then n bricklayers should build the wall in 1/n time units.

speedup = the time for the sequential case / the time for the parallel case

Here the sequential case is the time for one bricklayer and the parallel case is the time for n bricklayers.

$$\text{speedup}_{\text{replication}} = \frac{1}{(\frac{1}{n})} = n$$

Therefore, with our assumptions, the n bricklayers are n times faster than one.

In general, with n replications, we have the potential for a speedup of n.

Let us assume a mythical assembly line where four tasks need to be performed to construct a widget.

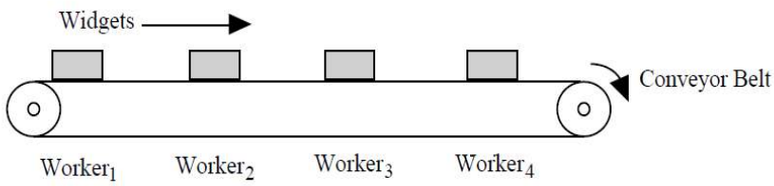

Figure 2.3

Let us further assume that each of the four tasks takes T time units. In the first T time unit, worker1 performs task1 on widget1. In the second T time unit, worker1 performs task1 on the second widget while worker2 performs task2 on widget1 and so on.

After 4T time units, the first widget rolls off the conveyor belt. After that a widget rolls off the conveyor belt every T time unit. Let us assume we want to manufacture 10 widgets, what is the speedup of the assembly line over one worker doing all four tasks?

Timeone worker = 10·4·T = 40T

It takes the worker 4T to build each widget and he has 10 to build.

On the assembly line, it takes 4T for the first widget to ride the length of the conveyor belt and one T for each widget after that.

Timeassembly line = 4·T + (10 - 1)·T

$$\text{speedup}_{\text{assembly line}} = \frac{\text{time for one worker}}{\text{time for assembly line}} = \frac{40 \bullet T}{13 \bullet T} = 3.07$$

For building the ten widgets, the four workers on the assembly line are a little over three times faster than the one worker.

Let us assume then we want to manufacture k widgets:

Timeone worker = k·4·T

Timeassembly line = 4·T + (k - 1)·T

What is the asymptotic behavior of the speedup if we produce many widgets? Or as k approaches ¥?

Simplifying the two expressions:

$$\text{speedup}_{\text{assembly line}} = \frac{k \bullet 4 \bullet T}{4 \bullet T + (k - 1) \bullet T} = \frac{4 \bullet k}{3 + k}$$

Divide the top and bottom by k.

$$\text{speedup}_{\text{assembly line}} = \frac{4}{\frac{3}{k} + 1}$$

$$\lim_{k \to \infty} \frac{4}{\frac{3}{k} + 1} = 4$$

as the $\frac{3}{k}$ term will go to zero as $k \to \infty$.

Therefore, for a four station assembly line (pipeline), the asymptotic speedup is four.

Generalizing: Assuming equal time at each station, n stations in an assembly line will have an asymptotic speedup of n.

There are many different ways of connecting processors together. The main difference between the approaches is whether the memory is distributed or shared. Most architectures use a shared memory arrangement whereby each processor can access data from a common memory store e.g. workstations, vector processors like the YMP, C90. This arrangement is shown in figure 2.4 for a SM-MIMD system. More recently the martket has seen the highly successful introduction of Symmetric Multi-Processing (SMP) machines e.g the Silicon Graphics Power Challenge and Origin series. The SMP system is ofthe SM-MIMD class whereby each processor has access to a global memory. Distributed memory (DMMIMD) architectures are now quite common and their typical arrangement is shown in figure 2.5. A typical SIMD arrangement is shown in figure 2.6 but in recent years, SIMD systems have undergone a rapid decline.

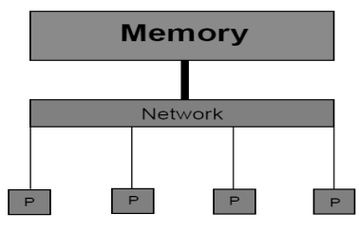

Figure 2.4 Typical Shared Memory (SM-MIMD) Layout

1. **Shared Memory**
   - Global memory which can be accessed by all processors of a parallel computer.
   - Data in the global memory can be read/write by any of the processors.
   - Examples: Sun HPC, Cray T90

Shared Memory Arragement

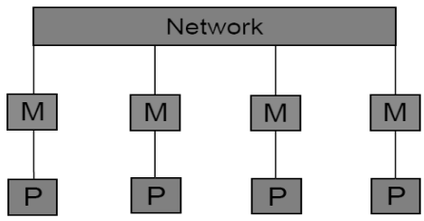

Figure 2.5 Typical Distributed Memory (DM-MIMD) Layout

2. **Distributed Memory**
   - Each processor in a parallel computer has its own memory (local memory); no other processor can access this memory.
   - Data can only be shared by message passing
   - Examples: Cray T3E, IBM SP2

Introduction to Parallel Computing

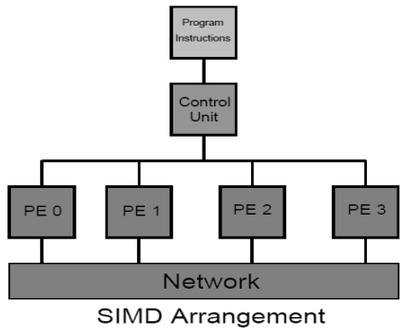

Figure 2.6 Typical Distributed SIMD Layout

3. **SIMD (Single-Instruction Multi-Data)**
   - All processors in a parallel computer execute the same instructions but operate on different data at the same time.
   - Only one program can be run at a time.
   - Processors run in synchronous, lockstep function
   - Shared or distributed memory
   - Less flexible in expressing parallel algorithms, usually exploiting parallelism on array operations, e.g. F90
   - Examples: CM2, MsPar

4. **MISD (Multiple-Instruction Single-Data)**
   - Special purpose computer

5. **SISD (Single-Instruction Single-Data)**
   - Serial computer

Any communication between the processors of a shared memory system is via the global memory and network, which could be a high speed bus. One problem that can arise with this system is that more than one processor may wish to access the same memory location. This is known as ***memory contention***. A time delay is then incurred until the memory is free. Another problem area is ***bus contention***, which can become severe with large numbers of processors. Another method of obtaining a completely connected system is to use a ***crossbar switch***, as illustrated in figure 2.7. Here, each processor has access to allthe memory units with fewer connection lines. However, the number of switches required to connect ***p*** processors to ***m*** memory modules is ***p x m***. Memory contention will also occur if 2 or more processorstry to access the same memory. In practice, these problems prevent very large systems being built.

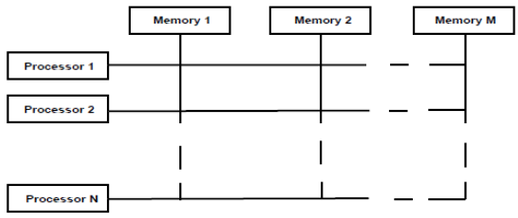

Figure 2.7 Crossbar Switch Network

As previously mentioned, distributed memory systems need to communicate with each other. Therefore, any inter-processor communication is done by message passing. Figure 2.8 shows the interconnect topology for a ***completely connected system***. A completely connected system of ***p*** processors would eachrequire ***p – 1*** connections, which is impractical for large systems.

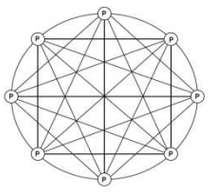

Figure 2.8 Completely Connected System

Figure 2.9(a) shows a ring network. This also utilises a high speed bus approach. Each message placed onthe ring by a processor would contain a destination address and a source address. A cluster of workstations could use this type of arrangement with Ethernet providing the ring network. A popular connection scheme is to employ a mesh arrangement. The simplest example is a ***linear array***, as illustrated in figure 2.9 (b). In this arrangement, each processor is connected to its nearest neighbour (except at the end).

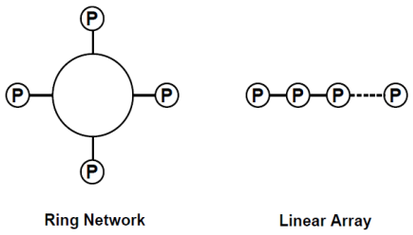

Figure 2.9 The Ring Network(a) and the Linear Array(b)

This has the advantage of simplicity, the disadvantage is that data may need to be passed through ***p – 1*** processors. Joining the two end processors to make a simple toroid would enhance its flexibility. Theobvious extension of the linear array is a 2–D or 3–D mesh-connected array. The Intel Paragon employs a 2–D mesh and the Cray T3D and T3E have a 3–D toroid configuration. Whilst the simplicity is retained, the disadvantage of having to pass messages through intermediate processors remains.

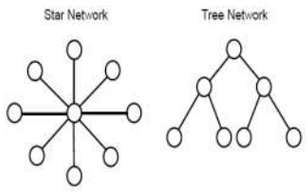

Figure 2.10 Star and Tree Networks

One of the more interesting connection topologies is that of the hypercube. The main market vendorsf or this connection scheme are Intel and nCube. One of the advantages of this approach is that the communication length (which depends strongly upon the routing technology) only grows as ***log2 (p)*** as the number of processors increase. The disadvantage is that the complexity and number of lines from each processor also increases and a practical size limit to the cube will be reached. The hypercube arrangement has certainly lost its popularity and the foregoing reason may have been a contributing factor as massively parallel processing became a reality. Other connection networks have been proposed, such as the star and tree networks illustrated in figure 2.10, but are not used in many practical applications.

Main Reasons to use Parallel Computing is that:

1. Save time and money.
2. Solve larger problems.
3. Provide concurrency.
4. Multiple execution units

Types of Parallel Computing

1. Bit-level parallelism

In the Bit-level parallelism every task is running on the processor level and depends on processor word size (32-bit, 64-bit, etc.) and we need to divide the maximum size of instruction into multiple series of instructions in the tasks. For Example, if we want to do an operation on 16-bit numbers in the 8-bit processor, then we would require dividing the process into two 8 bit operations.

2. Instruction-level parallelism (ILP)

Instruction-level parallelism (ILP) is running on the hardware level (dynamic parallelism), and it includes how many instructions executed simultaneously in single CPU clock cycle.

3. Data Parallelism

The multiprocessor system can execute a single set of instructions (SIMD), data parallelism achieved when several processors simultaneously perform the same task on the separate section of the distributed data.

4. Task Parallelism

Task parallelism is the parallelism in which tasks are splitting up between the processors to perform at once.

E. Evaluate

ASSESSMENT:
Instruction: Answer the questions below using the Answer Sheet (AS) provided in this module.

CONTENT FOR ASSESSMENT:
(2-points each)

1. Capable of operating at the same time.
2. Programming in a language which has explicit parallel (concurrent) constructs or features.
3. Organizing additional agents to perform part of the activity.
4. Organizing agents in an assembly line where each agent performs a specific part of the activity.
5. A Global memory which can be accessed by all processors of a parallel computer

References:

1. ***Introduction to Computer parallel Computing*** *https://ecomputernotes.com/fundamental/introduction-to-computer/parallel-computing*
2. ***What is Parallel Computing?*** *– Definition by Dinesh Thakur Category: Introduction to Computer*
3. ***Introduction to the Principles of Parallel Computation*** *By Dr. Daniel C. Hyde Department of Computer Science Bucknell University Lewisburg, PA 17837 hyde@bucknell.edu*

Facilitated By:		
Name	:	
MS Teams Account (email)	:	
Smart Phone Number	:	