MODULE I

PARALLEL COMPUTING

MODULE I
PARALLEL COMPUTING

INTRODUCTION

The module provides students with fundamentals of distributed systems and parallel computing and state-of-the-art tools that enable students to understand the concepts and principles underpinning distributed systems and utilize industry standard tools. Students are then prepared to specialize further in the field of distributed systems and parallel computing, e.g., in big data analytics or as scientific programmer.

OBJECTIVES

After studying the module, you should be able to:

1. describe the advantages of parallel from sequential computing;
2. parallelize an algorithm using different decomposition techniques;
3. describe at least one design technique for avoiding deadlocks; and
4. describe the key performance challenges in different memory and distributed system topologies.

DIRECTIONS/ MODULE ORGANIZER

There are four lessons in the module. Read each lesson carefully then answer the exercises/activities to find out how much you have benefited from it. Work on these exercises carefully and submit your output to your instructor or to the College of Computer Science office.

In case you encounter difficulty, discuss this with your instructor during the face-to-face meeting. If not contact your instructor at the College of Computer Science office.

Good luck and happy reading!!!

Lesson 1

Parallelism Fundamentals

Parallel Versus Serial Computing

Computer software were written conventionally for serial computing. This meant that to solve a problem, an algorithm divides the problem into smaller instructions. These discrete instructions are then executed on Central Processing Unit of a computer one by one. Only after one instruction is finished, next one starts.

Real life example of this would be people standing in a queue waiting for movie ticket and there is only cashier. Cashier is giving ticket one by one to the persons. Complexity of this situation increases when there are 2 queues and only one cashier.

So, in short **Serial Computing** is following:

1. A problem statement is broken into discrete instructions.
2. Then the instructions are executed **one by one**.
3. Only one instruction is executed at any moment of time.

Look at statement 3. This was causing a huge problem in computing industry as only one instruction was getting executed at any moment of time. This was a huge waste of hardware resources as only one part of the hardware will be running for a particular instruction and of time. As problem statements were getting heavier and bulkier, so does the amount oftime in execution of those statements.

We could definitely say that complexity will decrease when there are 2 queues and 2 cashier giving tickets to 2 persons **simultaneously**. This is an example of Parallel Computing.

Parallel Computing is the use of multiple processing elements simultaneously for solving any problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. It has been an area of active research interest and application for decades, mainly the focus of high-performance computing, but is now emerging as the prevalent computing paradigm due to the semiconductor industry's shift to **multi-core processors**.

Serial computing systems have just one processor. Although modern processors are very fast, performing calculations in milliseconds or less, these systems are still limited in speed. The processor performs one calculation at a time, then loads some new data and performs another

calculation. It does this in a rapid-fire fashion, over and over and over again.

Parallel computing systems, on the other hand, can have many computer processors working in tandem. Part of the computer breaks each big problem down into many smaller calculations. The central processor then assigns each of these smaller calculations to one of many processors. Each processor works on its share of the problem by itself, at the same time as all the other processors. When the processors are done with their small calculations, they report their results back to the central processor, which then assigns them more work. Although a small amount of performance is lost due to the need to coordinate tasks, the overall increase in computing efficiency is very, very large when tackling complex computational projects.

Table 1.1 shows the main differences between Serial and Parallel Computing.

Table 1.1 Serial Computing versus Parallel Computing

SERIAL COMPUTING	PARALLEL COMPUTING
One task is completed at a time and all the tasks are executed by the processor in sequence	Multiple tasks are completed at a time by different processors
There is a single processor	There are multiple processors
Lower performance	Higher performance
Workload of the processor is higher	Workload per processor is lower
Data transfers are in bit by bit format	Data transfers are in byte form (8 bits)
Requires more time to complete the task	Requires less time to complete the task
Cost is lower	Cost is higher

Why Do We Need Parallel Computing?

Some of the reasons why we need to utilize parallel computing are as follows:

- **Saves time and money**. Parallel usage of more resources shortens the task completion time, with potential cost saving. Parallel clusters can be constructed from cheap components**.**
- **Solve Larger Problems**. Complex and large problems that are impractical to solve by a single computer especially with limited memory.
- **Support Non-Local Resources**. Network wide computer resources can be utilized in scarcity at local resources.
- **Parallel computing models the real world**. Things don't happen one at a time, waiting for one event to finish before the next one starts. To crunch numbers on data points in weather, traffic, finance, industry, agriculture, oceans, ice caps, and healthcare, we need parallel computers.

Limitations of Parallel Computing

Parallel computing has also its overheads. Listed below are the current limitations of parallel computing:

- **Transmission Speed**. Transmission speed is relatively low as depends upon, how fast data can move through hardware. Transmission media limitations make data transmission low.
- **Difficult Programming**. It is difficult to write Algorithms and computer programs supporting parallel computing as it requires integration of complex instructions.
- **Communication and Synchronization**. Communication and synchronization between the sub tasks are typically one of the greatest obstacles to get good parallel program performance.

Parallelism Versus Concurrency

In programming, **concurrency** is the composition of independently executing processes, while **parallelism** is the simultaneous execution of possibly related computations. Concurrency is about **dealing** with lots of things at once. Parallelism is about **doing** lots of things at once.

Figure 1.1 illustrates the difference of Parallelism from Concurrency.

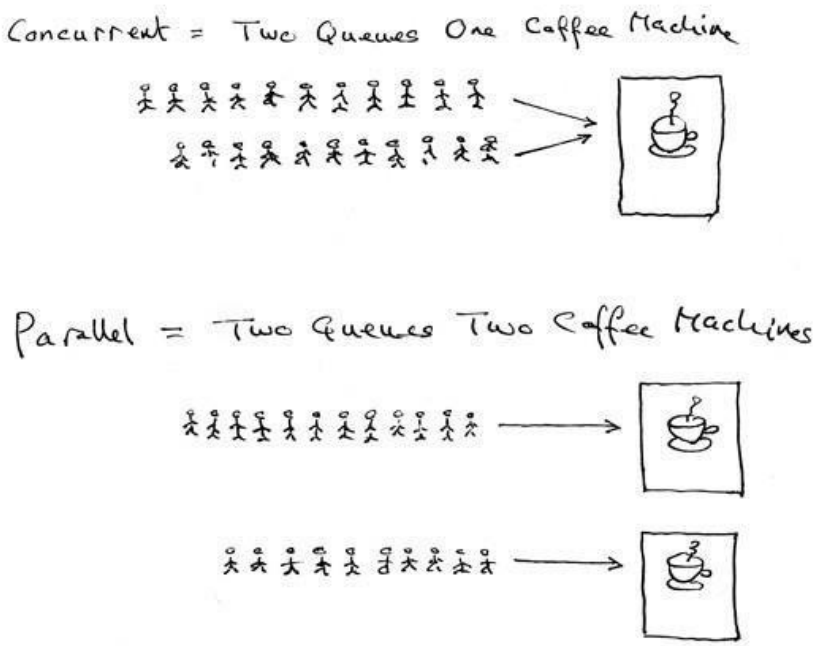


Figure 1.1 Analogy of Processes to Coffee Machine Queues
(https://miro.medium.com/max/600/1*c9hJciosear4PTicZp0ukA.jpeg)

EXERCISE

Demonstrate a real world task that can be solved by serial computing (ex. Sorting a deck of cards). Then, demonstrate how to solve the same task using parallel computing. Indicate how many seconds are consumed per demonstration. Finally, identify which computing paradigm has the least execution time.

THINK!

Give at least three (3) complex problems that cannot be solved by serial computing.

Answer:

Lesson 2

Parallelism Decomposition

Decomposition

In parallel computing, the simultaneous execution of multiple tasks is the key to reducing the time required to solve an entire problem. **Tasks** are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. **Decomposition** is the process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel.

There are three types of decompositions:

1. **Fine-grained decomposition.** A program is broken down to a large number of small tasks. These tasks are assigned individually to many processors. The amount of work associated with a parallel task is low and the work is evenly distributed among the processors which is know as **load balancing**.
2. **Course-grained decomposition.** A program is split into small number of large tasks. Due to this, a large amount of computation takes place in processors. This might result in **load imbalance**, wherein certain tasks process the bulk of the data while others might be idle.
3. **Medium-grained decomposition**. It is a compromise between fine-grained and coarse-grained parallelism, where we have task size and communication time greater than fine-grained parallelism and lower than coarse-grained parallelism.

To further compare the three types of decomposition, consider a 10X10 image that needs to be processed, given that, processing of the 100 pixels is independent of each other.

Fine-grained decomposition: Assume there are 100 processors that are responsible for processing the 10*10 image. Ignoring the communication overhead, the 100 processors can process the 10*10 image in 1 clock cycle. Each processor is working on 1 pixel of the image and then communicates the output to other processors.

Coarse-grained decomposition: If the processors are reduced to 2, then the processing will take 50 clock cycles. Each processor need toprocess 50 elements which increases the computation time, but the communication overhead decreases as the number of processors which sharedata decreases.

Medium-grained decomposition: Consider that there are 25 processors processing the 10*10 image. The processing of the image will now take 4 clock cycles.

Figure 1.2 shows different pseudocodes used by the three types of decomposition.

Fine-grain : Pseudocode for 100 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute element 1; break;
    case 2: Compute element 2; break;
    case 3: Compute element 3; break;
    .
    .
    .
    .
    case 100: Compute element 100;
              break;
  }
}
```

Computation time - 1 clock cycle

Medium-grain : Pseudocode for 25 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-4; break;
    case 2: Compute elements 5-8; break;
    case 3: Compute elements 9-12; break;
    .
    .
    case 25: Compute elements 97-100;
             break;
  }
}
```

Computation time - 4 clock cycles

Coarse-grain : Pseudocode for 2 processors

```
void main()
{
  switch (Processor_ID)
  {
    case 1: Compute elements 1-50;
            break;
    case 2: Compute elements 51-100;
            break;
  }
}
```

Computation time - 50 clock cycles

Figure 1.2 Pseudocodes Used by the Three Types of Decomposition

Decomposition Techniques

As mentioned earlier, one of the fundamental steps to solve a problem in parallel is to split the computations to be performed into a set oftasks. In this section, some commonly used decomposition techniques for achieving parallelism are discussed. These techniques are broadly classified as **recursive decomposition**, **data decomposition**, **exploratory decomposition**, and **speculative decomposition**:

1. **Recursive decomposition.** It is a method for inducing parallelism in problems that can be solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results.

   A classic example of a divide-and-conquer algorithm on which recursive decomposition can be applied is **Quicksort**.

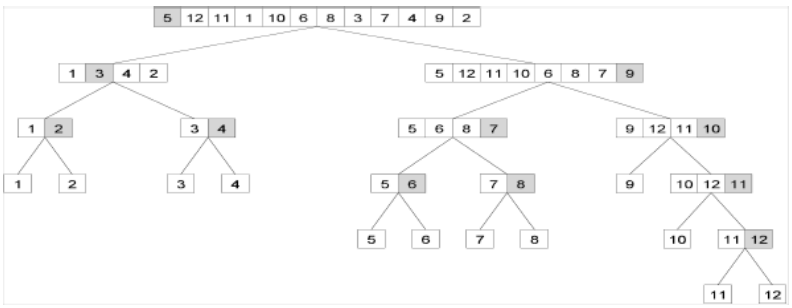


Figure 1.3 The quicksort task-dependency graph based on recursive decomposition

In this example, a task represents the work of partitioning a subarray. Note that each subarray represents an independent subtask. This can be repeated recursively.

2. **Data decomposition.** In this method, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar or are chosen from a small set of operations.

   As an example, let us Consider the problem of multiplying two n x n matrices A and B to yield matrix C. The output matrix C can be partitioned into four submatrices as follows:

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

(a)

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

(b)

Figure 1.4 Partitioning of input and output matrices into 2x2 submatrices

3. **Exploratory decomposition.** It is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, the search space is partitioned into smaller parts, and search each one of these parts concurrently, until the desired solutions are found.

   Exploratory decomposition can be used in a task such as looking for the best move in a Tic Tac Toe game. Each process willlook for possible best moves for the opponent recursively eventually using more processes. When it finds the result, it sends it back to the parent. The parent selects the best move from all ofthe results received from the child.

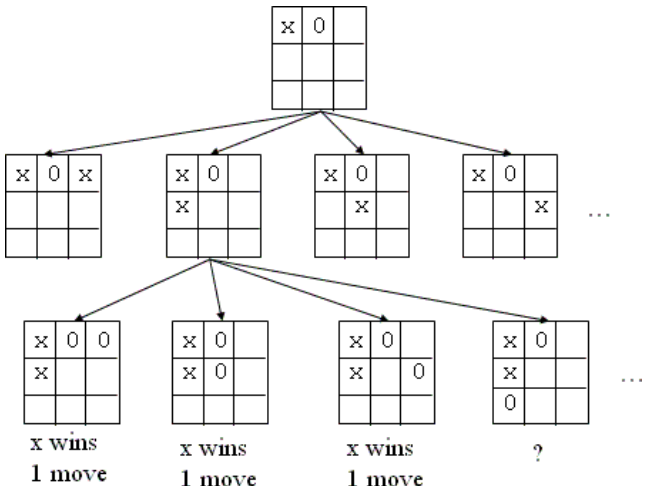


Figure 1.5 Using exploratory decomposition in a Tic Tac Toe Game

4. **Speculative decomposition**. It is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage.

   Consider the situation of a switch statement in a program. Usually, the value of the expression the switch is based on is waited to be known and execute only the case corresponding to it. In speculative decomposition some or all of the cases are executed in advance. When the value of the expression is known, only the results from the computation to be executed in that case is kept. The gain in performance comes from anticipating the possible computations.

Sequential version	Parallel version

compute expr;
switch (expr) {
  case 1:
  compute a1;
  break;
  case 2:
  compute a2;
  break;
  case 3:
  compute a3;
  break;...
}
	Slave(i)
	{
	  compute ai;
	  Wait(request);
	  if (request)
	  Send(ai, 0);
	}
	Master()
	{
	  compute expr;
	  swicth (expr) {
	  case 1:
	  Send(request, 1);
	  Receive(a1, i);
	  ...
	}

Figure 1.6 Speculative Decomposition in Sequential and Parallel programming (https://www.cs.iusb.edu/~danav/teach/b424/b424_20_explor.html)

EXERCISE

In this exercise, the goal is to add a series of 16 random numbers. Addition can only be performed between two numbers at any given time, and it assumed that it takes the same amount of time to add any two numbers. Thus, one addition operation takes one time step. Each number is written down on a separate note card. For further clarification, take note of the following sample scenarios:

Serial case:

One student is given the set of 16 random numbers and is asked to add them up. To add 16 numbers, it takes a total of 15 addition operations.

Parallel case (Two Students):

Each student is given a subset of 8 numbers. Each student adds their 8 numbers together on their own piece of paper. Student 1 then writes their total on a note-card and hands it to student 2. Student 2 then adds their total with the total they received from student 1 to yield the final total. To add 8 numbers, a total of 7 addition operations are required. Since each student performed their additions simultaneously and in parallel, a total of 7-time steps are required. The final global sum requires an additional time step. Therefore 8 total time steps are required.

Questions:

1. What is the total number of time steps for four students?
2. What is the total number of time steps for eight students?

Lesson 3

Communication and Coordination

Communication

Parallel tasks typically need to exchange data so they can stay in sync concerning changes in data values. Processors will rely on software to communicate with each other. Each processor will operate normally and will perform operations in parallel as instructed, pulling data from the computer's memory. Assuming all the processors remain in sync with one another, at the end of a task, software will fit all the data pieces together.

Communications can be accomplished through a **shared memory** or over a **network**. However, as the number of processors in parallel systems increases, the time it takes for a processor to communicate with other processors also increases. When the number of processors is somewhere in the range of several dozens, the performance benefit of adding more processors to the system is too small to justify the additional expense. To get around the problem of long communication times, a **message passing** system was created. In these systems, processors that share data send messages to each other to announce that particular operands have been assigned a new value. Instead of a broadcast of an operand's new value toall processors in the system, the new value is communicated only to those processors that need to know the new value. Instead of shared memory, there is a network to support the transfer of messages between programs. This simplification allows hundreds, even thousands, of processors to work together efficiently in one system.

Types of Communication Between Processors

The types of communication strategies among processors in parallel computing are shown in Figure 3.1:

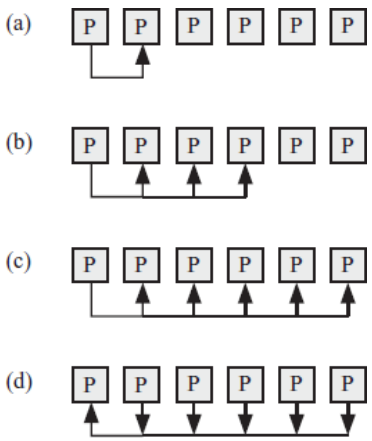


Figure 3.1 The different types or modes of communication among processors: (a) one to one, (b) one to many, (c) broadcast (one to all), and (d) gather and reduce.

1. **One to One (Unicast).** It involves a pair of processors: the sender and the receiver. This mode is sometimes referred to as point - to - point communication. Figure 3.1a shows the one to one mode of communication between processors. The figure only shows communication among a pair of processors, but typically, all processors could be performing the one to one communication at the same time.
2. **One to Many (Multicast).** One to many operation involves one sender processor and several receiver processors. Figure 3.1b shows the one to many mode of communication between processors. The figure only shows communication of one source processor to multiple receiving processors, but typically, all processors could be performing the one to many communication atthe same time.
3. **One to All (Broadcast).** Broadcast operation involves sending the same data to all the processors in the system. Figure 3.1c shows the broadcast mode of communication between processors. This mode is useful in supplying data to all processors. It might also imply one processor acting as the sender and the other processors receiving the data.
4. **Gather.** Gather operation involves collecting data from several or all processors of the system. Figure 3.1d shows the gather mode of communication between processors.
5. **Reduce.** Reduce operation is similar to gather operation except that some operation is performed on the gathered data. Figure 3.1d shows the reduce mode of communication between processors. An example of the reduce operation is when all data produced by all the processors must be added to produce one finalvalue. This task might take a long time when there are many data to be reduced.

Coordination

Coordination of concurrent tasks is called **synchronization**. Synchronization is necessary for correctness. The key to successful parallel processing is to divide up tasks so that very little synchronization is necessary.

The amount of synchronization depends on the amount of resources and the number of users and tasks working on the resources. Little synchronization may be needed to coordinate a small number of concurrent tasks, but lots of synchronization may be necessary to coordinate many concurrent tasks.

A great deal of time spent in synchronization indicates high contention for resources. Too much time spent in synchronization can diminish the benefits of parallel processing. With less time spent in synchronization, better processing speedup can be achieved.

Race Condition

Without coordination among processors, a problem called **race condition** may arise. A race condition is a condition of a program where its behavior depends on relative timing or interleaving of multiple processes. One or more possible outcomes may be undesirable, resulting in a bug.

For example, let as assume that two threads each increment the value of a global integer variable by 1. Typically, the following sequence of operations would take place as shown of Figure 3.2:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Figure 3.2 A Typical Execution of Two Threads

In the case shown above, the final value is 2, as expected. However, if the two threads run simultaneously without communication, the outcome of the operation could be wrong. The alternative sequence of operations as shown in Figure 3.3 demonstrates this scenario:

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Figure 3.3 An Alternative Execution of Two Threads

In this case, the final value is 1 instead of the correct result of 2. This occurs because here the increment operations are not mutually exclusive. Mutually exclusive operations are those that cannot be interrupted while accessing some resource such as a memory location.

To avoid race condition, a synchronization method is needed. Mutual Exclusion is one way of process synchronization making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same things.

Types of Mutual Exclusion Devices

Implementing mutual exclusion can be done using the following devices:

1. **Locks.** These mechanisms enforce limits on access to a resource when there are many threads of execution. Most locking designs block the execution of the thread requesting the lock until it is allowed to access the locked resource.
2. **Readers-Writer Mutex.** These mutexes are typically used for protecting shared data that is seldom updated, but cannot be safely updated if any thread is reading it. The reading threadsthus take shared ownership while they are reading the data. Whenthe data needs to be modified, the modifying thread first takes exclusive ownership of the mutex, thus ensuring that no other thread is reading it, then releases the exclusive lock after the modification has been done.
3. **Semaphores.** These are variables that is non-negative and shared between threads. Semaphores are signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.
4. **Monitors.** These mechanisms provides a structured synchronization mechanism built on top of object-orientedconstructs. Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.
5. **Barriers.** These mechanisms divide a program into phases by requiring all threads to reach it before any of them can proceed to avoid conflicting access to shared data. Code that is executed after a barrier cannot be concurrent with code executed before the barrier.

Deadlocks

While synchronization methods are effective for protecting shared state, they come with a catch. Because they cause processes to wait oneach other, they are vulnerable to **deadlock**, a situation in which two or more processes are stuck, waiting for each other to finish.

As an example, consider two processes that uses locks. Suppose there are two locks, x_lock and y_lock, and they are used as follows:

P1	P2
acquire x_lock: ok	acquire y_lock: ok
acquire y_lock: wait	acquire x_lock: wait
wait	wait
wait	wait
wait	wait
...	...

The resulting situation is a deadlock. P1 and P2 are each holding onto one lock, but they need both in order to proceed. P1 is waiting for P2 to release y_lock, and P2 is waiting for P1 to release x_lock. As a result,neither can proceed.

Generally, deadlocks can occur in in any system that satisfies the four conditions below:

1. **Mutual Exclusion Condition**: a resource is either assigned to one process or it is available
2. **Hold and Wait Condition**: processes already holding resources may request new resources
3. **No Preemption Condition**: only a process holding a resource may release it
4. **Circular Wait Condition**: two or more processes form a circular chain where each process requests a resource that the next process in the chain hold

The Dining Philosophers Problem

The dining philosophers problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both immediate left and right chopsticks of the philosopher is available. In case if both immediate left andright chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

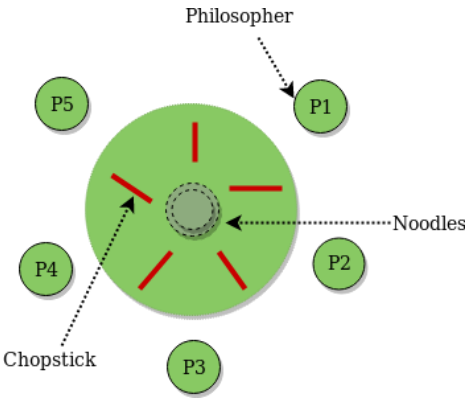


Figure 3.4 The Dining Philosophers Problem

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem. For example, let’s consider P0, P1, P2, P3, and P4 as the philosophers or processes andC0, C1, C2, C3, and C4 as the 5 chopsticks or resources between each philosopher. Now if P0 wants to eat, both resources/chopstick C0 and C1 must be free, which would leave in P1 and P4 void of the resource and the process wouldn't be executed, which indicates there are limited resources(C0,C1..) for multiple processes(P0, P1..)

The solution to the process synchronization problem is Semaphore. A semaphore is an integer used in solving critical sections.

The critical section is a segment of the program that allows you to access the shared variables or resources. In a critical section, an atomic action (independently running process) is needed, which means that only single process can run in that section at a time.

Semaphore has 2 atomic operations: wait() and signal(). If the value of its input S is positive, the wait() operation decrements, it is used to acquire resource while entry. No operation is done if S is negative or zero. The value of the signal() operation's parameter S is increased, it used to release the reource once critical section is executed at exit.

Here's an explanation of the solution:

```
void Philosopher
{
  while(1)
  {
    // Section where the philosopher is using chopstick
    wait(use_resource[x]);
    wait(use_resource[(x + 1) % 5]);
    //  Section where the philosopher is thinking
    signal(free_resource[x]);
    signal(free_resource[(x + 1) % 5]);
  }
}
```

The wait() operation is implemented when the philosopher is using the resources while the others are thinking. Here, the threads use_resource[x] and use_resource[(x + 1) % 5] are being executed.

After using the resource, the signal() operation signifies the philosopher using no resources and thinking. Here, the threads free_resource[x] and free_resource[(x + 1) % 5] are being executed.

In conclusion, no two nearby philosophers can eat at the same time using the aforesaid solution to the dining philosopher problem because this situation causes a deadlock.

Deadlock Detection Using a Resource Allocation Graph

Detecting a deadlock in a parallel system can be done using a Resource Allocation Graph. A **Resource Allocation Graph (RAG)** is the pictorial representation of the state of a system. As its name suggests, the resource allocation graph is the complete information about all the processes which are holding some resources or waiting for some resources.It also contains the information about all the instances of all the resources whether they are available or being used by the processes.

There are two components of the RAG:

1. **Vertices.** These are used to represent a process or a resource. Two type of vertices are used, process vertices and resourcevertices.
    a. **Process Vertices** are represented by **circles.**
    b. **Resource Vertices** are represented by **rectangles.**

    There are two types of resource vertices, single instance and multiple instances. In **single instance** resource type, single dot inside the rectangle is used. The single dot indicates that there is one instance of the resource. In **multiple instance** resource type, multiple dots inside the rectangle is used. Multiple dots indicate that there are various instances of the resources.
2. **Edges.** These are used to represent an assignment of a resource or requesting state of a process. Two types of edges are used, assign edges and request edges.
    a. **Assign edges** represent the allocation of resources to the process. Assign edges are drawn with the help of arrow in which the arrow head points the process, and the process tail points the instance of the resource.
    b. **Request edges** signify the waiting state of the process. Just like in assign edge, an arrow is used to draw arrow edge. Here, the arrow head points the instance of a resource, and tail of the process points to the process.

An example of a RAG is shown in Figure 3.4. Two processes P1, and P2 and two resources which are R1, and R2 can be seen in the figure. This example is a kind of single instance resource type, and it contain a cycle, so there is a deadlock in the system.

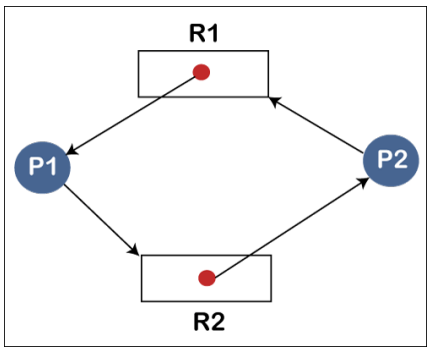


Figure 3.4 A RAG with a Single Instance Resource Type

An Example of a Multiple Instance Resource Type RAG is shown in Figure 3.5. The term multiple instances means the resources are havingmore instances. In the following example, three processes, which are P1, P2, and, P3 and three resources, which are R1, and two instances of R2.

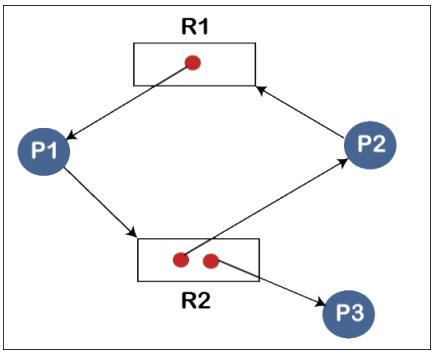
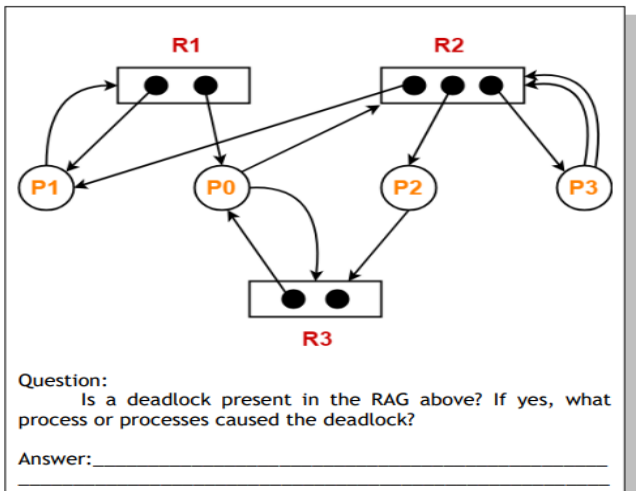


Figure 3.4 A RAG with a Multiple Instance Resource Type

Even though a cycle is present in the graph above, all the processes can be executed successfully because of the extra resource instance of R2. If P3 finished its task, P1 can now have access to R2. If P1 finished its task, P2 can finally access R1 and also finish its task. Thus, no deadlock is present in the system.

EXERCISE



Lesson 4

Parallel Architectures

Shared Memory versus Distributed Memory Based Architecture

Parallel computing architecture can be classified based on the communication model used. The two models are Shared Memory and Distributed Memory:

Shared Memory are the models that rely on the shared memory multiprocessors. Shared memory based parallel programming models communicate by sharing the data objects in the global address space.Shared memory models assume that all parallel activities can access all of memory. Communication between parallel activities is through shared mutable state that must be carefully managed to ensure correctness. Synchronization primitives such as locks are used to enforce thismanagement.

Shared memory models can be further divided into two main classes based upon memory access times which are the Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA).

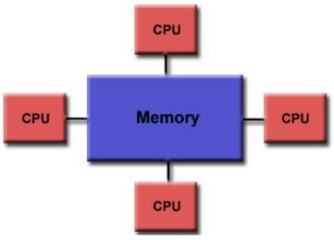


Figure 4.1 Uniform Memory Access (UMA) Model

Figure 4.1 shows the Uniform Memory Access model. In this model, a single memory is used and accessed by all the processors with the help of interconnection network. Each processor has equal memory accessing time (latency) and access speed.

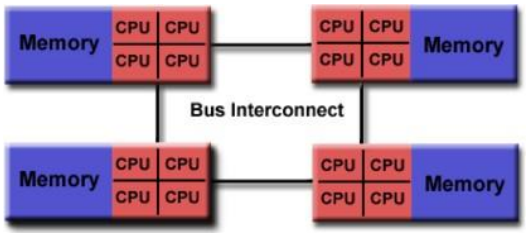


Figure 4.2 Non-Uniform Memory Access (NUMA) Model

Figure 4.2 shows the Non-Uniform Memory Access model. This model is also a multiprocessor model in which each processor is connected with the dedicated memory. However, these small parts of the memory combine to make a single address space. Unlike UMA, the access time of the memory relies on the distance where the processor is placed which means varying memory access time. It allows access to any of the memory location by using the physical address.

Table 4.1 below presents the comparison between UMA and NUMA:

Table 4.1 Key Differences Between UMA and NUMA

UMA	NUMA
Uses one or two memory controllers	Uses multiple memory controllers
Single, multiple and crossbar busses are used for communication	Hierarchical, tree type busses, and network connection are used for communication
Memory accessing time for each processor is the same	Memory accessing time changes as the distance of memory from the processor changes
Suitable for general purpose and time-sharing applications	Suitable for real-time and time-critical centric applications
Work slower than NUMA	Faster than UMA
Have limited bandwidth	More bandwidth than UMA

The advantages of the Shared Memory model are listed below:

1. Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
2. Avoids redundant data copies by managing shared data in main memory in caches
3. There is no need to write explicit code for processor interaction and communication

The disadvantages of the Shared Memory model are as follows:

1. Adding more CPUs can increase traffic on the shared memory path, thus making the speed of processor communication slower
2. It is expensive to design and produce shared memory machines
3. Often encounter data races and deadlocks during programming

On the other hand, **Distributed Memory** based architectures refer to multiprocessor computer systems in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate withone or more remote processors. A communication network is required to connect inter-processor memory in this model. Figure 4.3 shows the Distributed memory model:

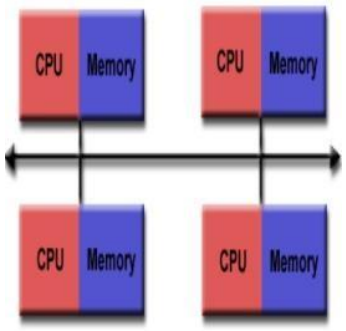


Figure 4.3 Distributed Memory Model

The advantages of Distributed Memory model are enumerated as follows:

1. Increasing the number of processors also increases the size of the memory proportionately
2. Each processor can rapidly access its own memory without affecting the other processors' memory
3. Avoids the data races and as a consequence the programmer is freed from using locks
4. The hardware requirement is low, less complex and comes at very low cost

The disadvantages of Distributed Memory model are listed below:

1. Programmer is responsible for establishing communication between processors

2. Can incur high communication overhead among processors

Flynn's Taxonomy

Parallel computing architecture can be further classified based on the data and the operations performed on this data. This famous processor classification called **Flynn's Taxonomy**, was proposed by Michael J. Flynn, an American professor emeritus at Stanford University. It is composed of four classifications which are described below:

1. **Single Instruction, Single Data (SISD).** It is a sequential computer which exploits no parallelism in either the instruction or data streams. In SISD, machine instructions are processed in a sequential manner and computers adopting this model are popularly called sequential computers. Examples of SISD architecture are the traditional uniprocessor machines like older personal computers, mainframes, and single-instruction-unit-CPU workstations. Figure 4.4 shows the diagram of an SISD model. The speed of the processing element in the SISD model is limited by the rate at which the computer can transfer information internally.

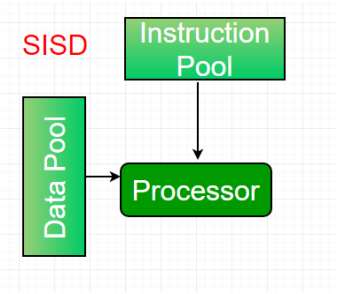


Figure 4.4 Single Instruction, Single Data (SISD) Model
(https://media.geeksforgeeks.org/wp-content/uploads/sisd.png)

2. **Single Instruction, Multiple Data (SIMD).** It is a multiprocessor computer capable of executing the same instruction on all the CPUs but operating on different data streams. Each processor has its own data in a local memory, and the processors exchange data among themselves through simple communication schemes. Many scientific and engineering applications lend themselves to parallel processing using this scheme. Examples of such applications include graphics processing, video compression, and medical image analysis. Figure 4.5 illustrates the SIMD model. The "single" in single-instruction doesn't mean that there's only one instruction unit, as it does in SISD, but rather that there's only one instructionstream, and this instruction stream is executed by multiple processing units on different pieces of data, all at the same time, thus achieving parallelism.

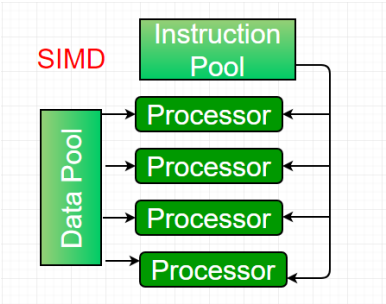


Figure 4.5 Single Instruction, Multiple Data (SIMD) Model
(https://media.geeksforgeeks.org/wp-content/uploads/simd.png)

3. **Multiple Instruction, Single Data (MISD).** It is a multiprocessor computer capable of executing different instructions on different CPUs but all of them operating on the same dataset. This is an uncommon architecture which is generally used for fault tolerance. Figure 4.6 describes the MISD model. A few machines are built with this model, but none of them are available commercially.

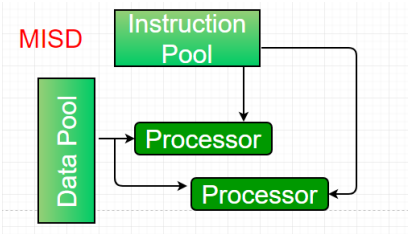


Figure 4.6 Multiple Instruction, Single Data (MISD) Model
(https://media.geeksforgeeks.org/wp-content/uploads/misd.png)

4. **Multiple Instruction, Multiple Data (MIMD).** It is a multiprocessor machine which is capable of executing multiple instructions on multiple data sets. Each CPU in the MIMD model has separate instruction and data streams. Figure 4.7 shows the diagram of an MIMD model. The most general of all of the major classifications, an MIMD machine is capable of being programmed to operate as if it were an SISD, SIMD, and MISD.

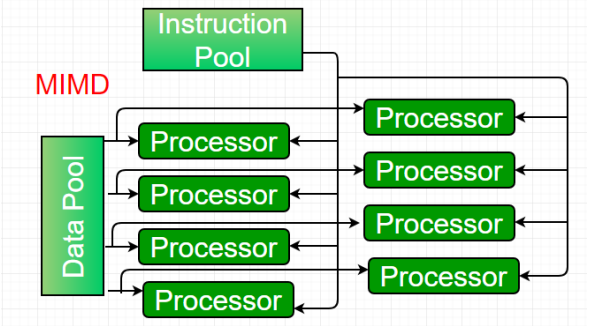


Figure 4.7 Multiple Instruction, Multiple Data (MIMD) Model
(https://media.geeksforgeeks.org/wp-content/uploads/mimd.png)

Flynn's classification can be described by an analogy from the manufacture of automobiles:

- SISD is analogous to the manufacture of an automobile by just one person doing all the various tasks, one at a time.
- MISD can be compared to an assembly line where each worker performs one specialized task or set of specialized tasks on the results of the previous workers accomplishment. Workers perform the same specialized task for each result given to them by the previous worker, similar to an automobile moving down an assembly line.
- SIMD is comparable to several workers performing the same tasks concurrently. After all workers are finished, another task is given to the workers. Each worker constructs an automobile by himself doing the same task at the same time. Instructions for the next task are given to each worker at the same time and from the same source.
- MIMD is like SIMD except the workers do not perform the same task concurrently, each constructs an automobile independently following his own set of instructions.

EXERCISE

Question:

If you were to implement a program that requires several processor to frequently exchange data, what parallel computing architecture would you utilize? Is it shared memory or distributed memory model? Justify your answer in at most ten sentences only.

Answer:

MODULE SUMMARY

In module I, you have learned about the basic information assurance and security principles. You have learned their meanings, importance and kinds. You have also understood the information security needs of certain information systems.

Congratulations! You have just studied Module I. Now you are ready to evaluate how much you have benefited from your reading by answering the summative test. Good Luck!!!

SUMMATIVE TEST

Directions: Answer the following questions concisely. If possible, cite your references to support your answer.

1. What are the two primary reasons for using parallel computing?

   _____
   _____
   _____
   _____
   _____

2. In what instances does a coarse-grained decomposition better than fine-grained decomposition?

   _____
   _____
   _____
   _____
   _____

3. Will adding more processor always speed up a parallel system? Why or why not?

_____
_____
_____
_____
_____

4. Is giving each CPU in an MIMD machine identical instructions to execute cause it to operate like an SIMD machine? Justify your answer.

_____
_____
_____
_____
_____