COS464, CONCURRENT PROGRAMMING

O. E. OGUIKE

DEPARTMENT OF COMPUTER SCIENCE

UNN

TABLE OF CONTENTS

1 INTRODUCTION TO CONCURRENCY AND PARALLEL COMPUTING

1.0 Introduction

Concurrency as a computer terminology describes things happening at the same time on the computer system. Whenever things happen at the same time, we are interested in who are doing what at the same time. As a result, two types of concurrency can be identified. The first is quasi or false or pseudo concurrency, when the various peripheral devices and the single processor of a uni-processor computer system are doing things at the same time. It involves the single processor, executing program, at the same time, the peripheral devices will be performing the next input and output operations. It may also involve the various processing elements of the uniprocessor computer performing their respective functions, at the same time, the single processor, executing program. This type of concurrency involves a uniprocessor computer system. The second type of concurrency involves more than one processor of multi-processor computer system/parallel computers. Each of the processors will be executing a part of a program at the same time. True concurrency, therefore, relates to parallel computing, the act of using a parallel computer, whose processors will be executing a part of a program concurrently or in parallel, with the aim of solving a specific problem. This lecture focuses on true concurrency as it relates to parallel computing.

1.1 The Rationale for True Concurrency and Parallel Computing:

Some contemporary computer applications require high speed or high performance computers. Allowing the multiple processors of a parallel computer to execute a piece of a program concurrently or in parallel with the aim of solving a particular problem is aimed at increasing the speed of computer system indefinitely.

In the past, many efforts have been made towards increasing the speed of computer systems, like multiprogramming, pipelining etc. Though these efforts have yielded good results. However, the most effective way of increasing the speed of computer systems is to use a multi-processor computer system or parallel computers, and allow the various processors to execute a part of a program that solves a specific problem concurrently or in parallel. Therefore, increasing the speed of computer system indefinitely aims at developing high performance computers.

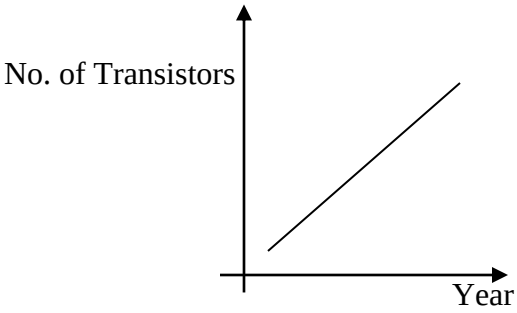
Furthermore, Moore's law can be used to explain the rationale for true concurrent programming. In 1965, Gordon Moore predicted that the numbers of transistors on computers will double every eighteen months Though a lot of computer scientists considered this law/prediction as economic law/prediction. However the effects of Moores law/prediction are as follows:

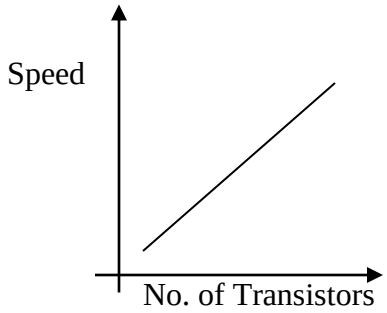
- The speeds of computers will double every eighteen months. The reason for this is that increasing the number of transistors on computers will increase the speed of computers.
- The speed of computers will double as the number of transistors doubles every eighteen months.

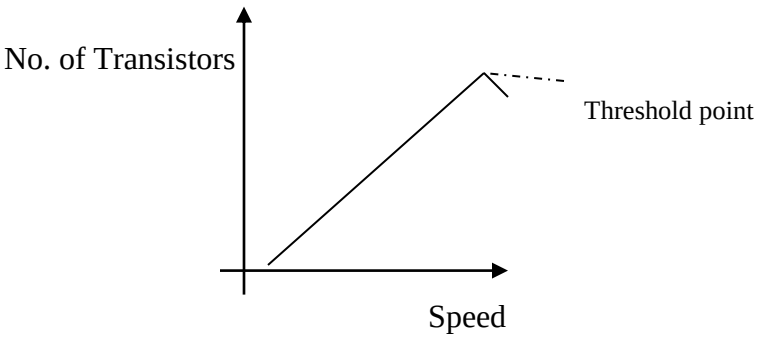
- The size of transistors will continue to shrink(reduce in size), half of its original size every eighteen months.

Though Moore's law and its effects have been fulfilled for some decades, however, recent discoveries show that there is hardware limitation with respect to continued reduction in the size of transistors, in an attempt to increase number of transistors on computer, thereby increasing its speed. This limit or threshold point has been reached; as a result the size of transistors cannot be reduced any more. The reason for this limit is that all semiconductor devices, like transistors are Si based It can be assumed that a circuit element will take at least a single Si atom, and the covalent bonding in Si has a bond length approximately 20nm, hence we will reach the limit of miniaturization very soon. At the moment that limit has been reached. As a result of this limitation, it implies that speed of computers can no longer be realized by increasing the number of transistors. Computer scientists, at the moment are looking into other avenues of increasing the speed of computers. Two of such possible ways are Parallel Computing and Nanocomputing. Parallel Computing is promising, as it has recorded tremendous success, while Nanocomputing remains a technology for the future, two decades from date.

The diagrams below illustrate Moore's Law and the various interpretations of Moore's Law.







At the Threshold point, any further increase in the number of transistors will reduce the speed of the computer. This justifies the need to introduce more than one processor on the computer system.

1.2 True Concurrency Requires Cooperation and Communication:

Because each of the processors executes concurrently or at the same time, with the aim of solving a specific problem, they must cooperate by communicating with each other. Data must be exchanged and communicated among the various processors. It means that the various processors must be networked for the purpose of communication. Determining

the most efficient route of communicating with the processors will help to minimize communication overheads. The root of all successful human endeavors is cooperation, by communicating. Any problem that will be solved requires that the problem be decomposed or split and distributed to the various processors. Each of the processors solves its own problem and communicates the result to others, so that results of processed data can be collated with the aim of producing final result.

1.3 Illustrating with Concept of Division of Labour in Economics:

The concept of division of labour in Economics requires that task be split into various parts, and allocated to various workers. The various workers will perform some of the tasks at the same time, while others will wait until some have been completed. In the course of executing the work, the workers cooperate and communicate with each other. The main advantage of this division of labour is increase in the level of production or increase in the amount of work. The same concept is used in parallel computing, computing task is split among the various processors, and they are allocated to the various processors of the parallel computers. Some of the computing tasks are executed at the same time by the parallel processors, while some will wait until others have completed. The processors communicate and cooperate with each other by exchanging data and synchronizing each other's activities. The result is increase in throughput. However, low productivity can be as a result of poor communication facilities that the workers can use to communicate with each other. The same thing applies to parallel computing. Throughput can be low as a result of inefficient means of communication between the various processors; therefore, the most efficient means of communication between the processors is desired. One way of realizing an efficient means of communication between the processors is to devise the most efficient algorithm that can be used to route the message from one processor to the other.

1.4 Research Area Under Parallel Computing:

Parallel computing, which requires the use of parallel computer to solve problem at reasonable time has defined new research area that have added a parallel dimension to computing. Whatever we can do with the traditional computing, using a uni-processor computer has a parallel dimension, using a parallel computer system. As a result, the following research areas are under parallel computing:

Parallel Architecture, Parallel Algorithm, Parallel/True Concurrent Programming, Parallel Programming Languages, Parallel Computer System Performance Evaluation, etc. Each of these broad research areas will be discussed briefly.

1.4.1 Parallel Architecture:

This considers the various ways of arranging the various components of a parallel computer, like multiple processors, memory, communication network etc with the aim of realizing a functional parallel computer.

1.4.2 Parallel Algorithm:

This involves the analysis and development of the algorithm that will be used to write a parallel or true concurrent programming.

1.4.3 True Concurrent/Parallel Programming:

This research area considers the use of parallel programming construct/control structures to write programs that the parallel computer will execute.

1.4.4 Parallel Programming Languages:

This research area include the design of programming constructs that can be used to write a parallel program or true concurrent program.

1.4.5 Parallel Computer System Performance Evaluation:

This research area examines the performance of parallel computer with the aim of optimizing the performance of the system.

2. PARALLEL ARCHITECTURE

2.0 Introduction:

Writing a true concurrent program or a parallel program requires that we understand the architecture of the computer system that can be used to write the program. Parallel architecture refers to the various ways of organizing or arranging the various components of a parallel computer with the aim of enhancing the performance of the system. These components include the following: memory, parallel processors, peripheral devices, interconnecting network, connecting the various processors etc. This chapter uses Fynn's Taxonomy to examine the various architectures of computer, including parallel computer.

2.1 Fynn's Taxonomy:

Speed as a characteristic of computers has led to an increase in the demand for high performance computers; as a result, there has been a single processor technological leap, from the vacuum tube architecture in the 1940's to the current single processor, RISC architecture. Flynn, has classified the diversities of architecture that have been developed from the 1940 to the present time in 1972, into four broad areas. Flynn's classification of various computer architecture has been captioned **'FLYNN TAXONOMY'** , and it is based on the number of streams of instruction and data that the computer executes at a time. The Flynn's taxonomy for classifying computer architecture follows:

i) SISD, Single Instruction, Single Data
ii) SIMD, Single Instruction, Multiple Data

iii) MISD, Multiple Instruction, Single Data
iv) MIMD, Multiple Instruction, Multiple Data

As I consider each of these classifications, I shall identify the various architecture within each, and discuss in detail those that can be used for concurrent programming.

2.2 SISD: SINGLE INSTRUCTION, SINGLE DATA

All the architectures within the single Instruction, Single Data classification are single processor architecture, it is purely Jon Von Neumann architecture, it executes a single instruction sequentially on single data stream. However, temporal parallelism can be introduced by allowing sequential execution to be over lapped in time by a number of functional unit within the processor, this is pipelining.
An example of computer with this type of architecture is CRAY supercomputer.
An example of architecture that is within this classification is the
CRAY - 1 computers though, a single processor architecture, but it has multiple pipeline that allow both scalar and vector operations to be performed in parallel, this helps to increase the performance of the computer, as a result, many have regarded this as the first true supercomputer.

2.3 SIMD: SINGLE INSTRUCTION, MULTIPLE DATA

This classification of computer architecture includes all computers that execute a single instruction on multiple streams of data. It will require multiple processors that will execute the same instruction on multiple data stream at the same time.
Examples of architecture that is part of this classification are:

2.4 MISD, MULTIPLE INSTRUCTIONS, SINGLE DATA

This classification means that multiple instructions are applied to single stream of data. An example of architecture that is close to this classification is the systolic array architecture. However, because no architecture falls into the MISD classification,

2.5 MIMD: MULTIPLE INSTRUCTIONS, MULTIPLE DATA

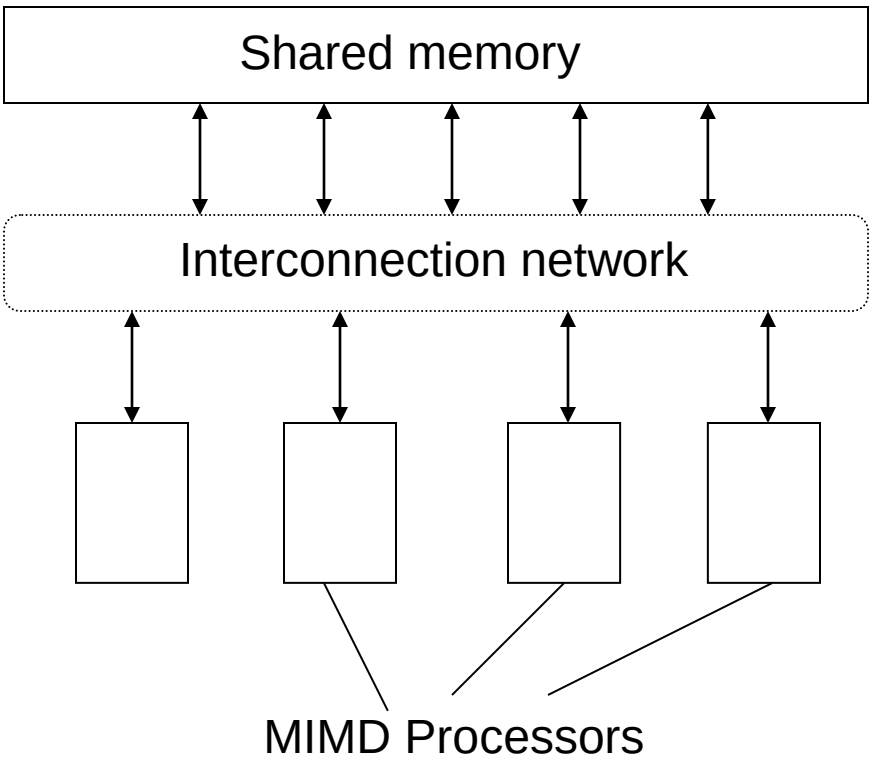
This classification allows each of the processors to apply their instructions to their own data, thus allowing multiple instructions to be executed on multiple data, simultaneously.

- Shared memory, MIMD
- Distributed memory, MIMD

Each of these architectures has its own method of communication between the processor. Each of these architectures will be considered.

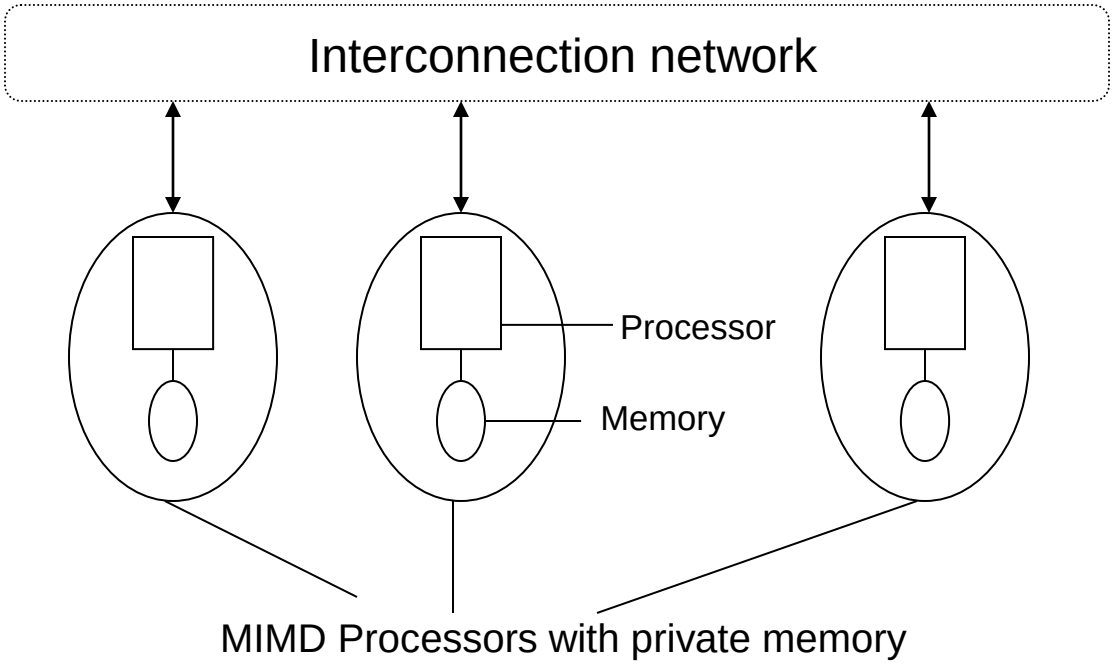
2.5.1 SHARED MEMORY, MIMD

In a shared memory, MIMD, all the processors access a shared memory, therefore, the control that will be used for communication and synchronization will allow exclusive access to update shared variables.
The diagram that follows illustrates further:



2.5.2 DISTRIBUTED MEMORY, MIMD

In a distributed memory, MIMD, each of the multiple processors has its own private memory. Communication between these processors is through an interconnection network. Transputers are devices that combine memory and processors on a single chip. The diagram that follows illustrates further:



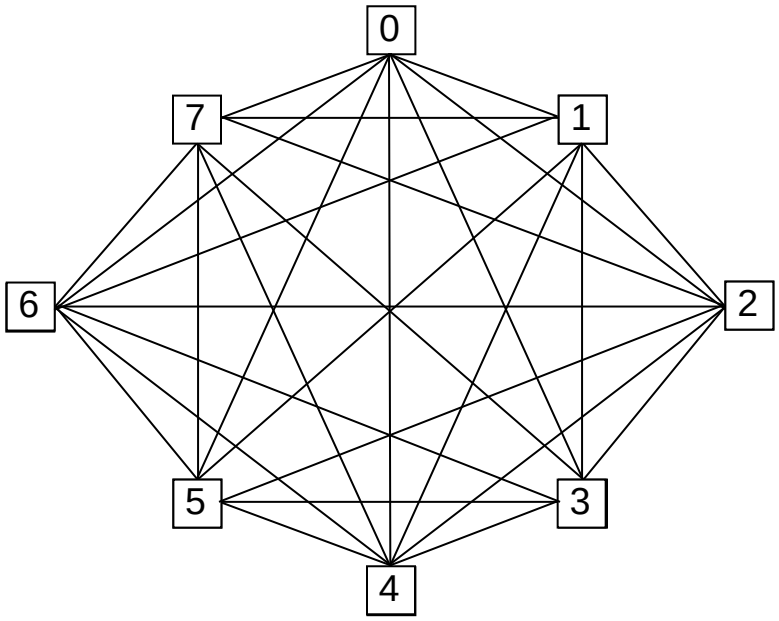
2.5.3 Hybrid MIMD

The two architectures of MIMD can be combined to realize the benefits of the two architectures. In distributed memory MIMD architecture, though each of the processors has its own memory, the concept of virtual memory can be used so that a processor can use the memory of another processor whenever its memory capacity is not sufficient. This is called virtual memory. It is virtual memory because the processor thinks that it is using its own memory, but in reality it is using the memory of another processor. Similarly, in a shared memory MIMD architecture, the common memory of the parallel computer can be divided into memory modules, each module
for each of the processors, thereby allowing the processors to have its own memory while sharing a common memory, this is called interleaved memory MIMD.

2.5.4 INTERCONNECTION NETWORK

Different network configurations can be used to connect the various processors. Among the common network configurations, includes the following:

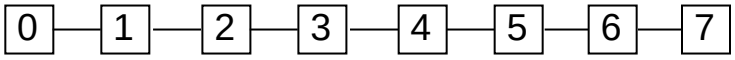
- Fully Connected Network



8-processor fully connected configuration

In a fully connected configuration, every processor is connected to the other processors. A fully connected n- processor configuration will have a maximum of n*(n-1) connections from one processor to the other.

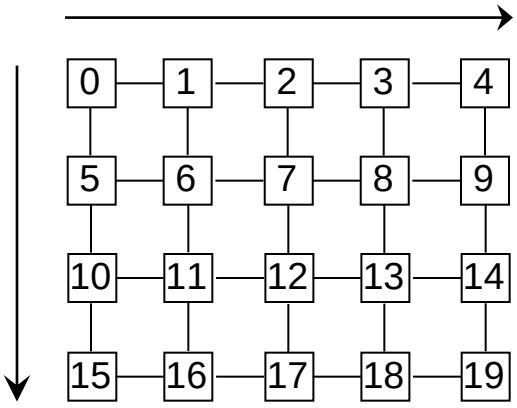
- Chain/Bus



8-processor chain

In a Chain/Bus configuration, the processors are arranged in a linear form. Each processor is connected to two other processors except the processors at the front or back of the chain. The maximum number of connections in such configuration in a n-processor chain is 2*(n-1).

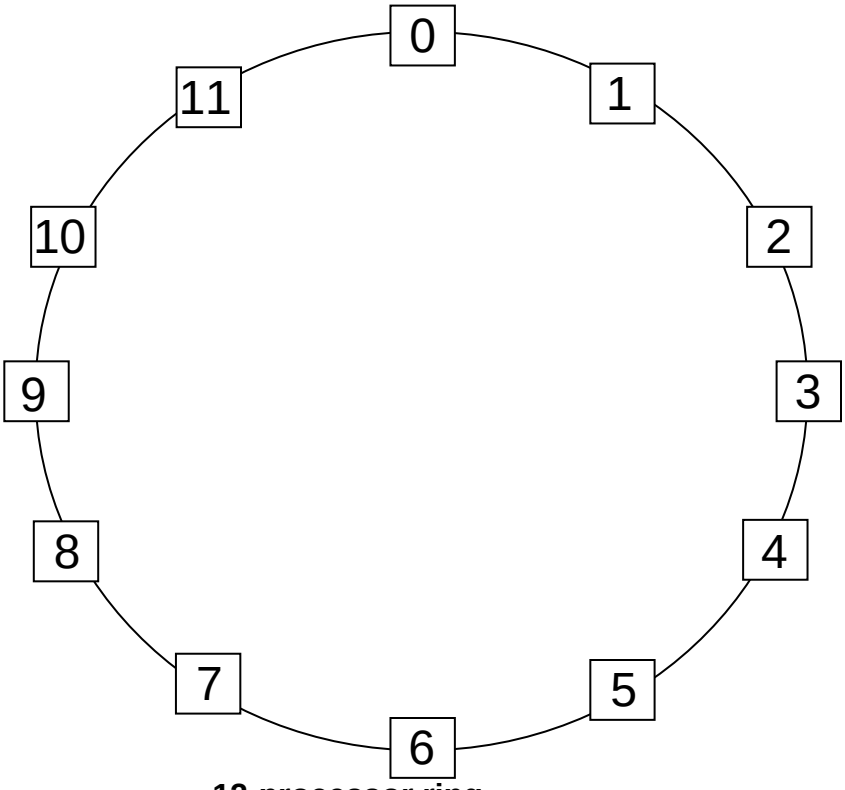
- Two-dimensional Mesh



20-processor mesh

This configuration is known as array. It consists of processors that are arranged in a grid. As an array configuration, a two dimensional mesh of n processors can be decomposed into I rows and j column processors. Each row of the array configuration can be regarded as a horizontal chain, while each column can be regarded as a vertical chain. A two dimensional mesh of I rows and j column of processors will have a maximum of 2*(i-1)*j + 2*(j-1)*i connections, and a total of I*j processors.

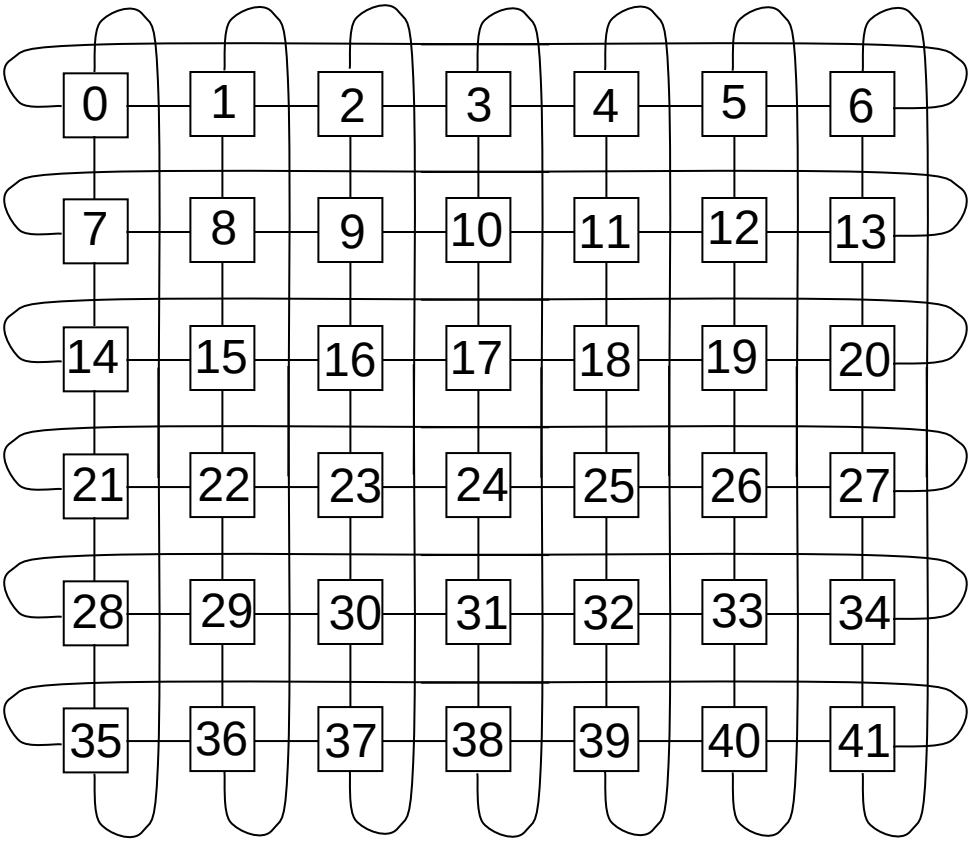
- Ring



12-processor ring

Ring is one of the most common configurations. Like a chain/bus, the processors are arranged in a linear form, and each processor is connected to two other processors for a bi-directional rings where messages travel in both direction of the ring. Unlike a chain/bus, the processors at front and back ends of the chain are connected to each other in both directions. Therefore, for a bi-directional, n-processor ring, the maximum number of connections is 2*n.

- Torus



42-processor torus

A torus configuration can be regarded as a two dimensional mesh, but each row and each column of the array processors is a ring. Therefore the torus configuration can be regarded as a ring of rings of processors. A torus configuration with i rows and j columns of processors will have a total of i*j processors and a total of 2*i*j + 2*j*i = 4*I*j connections.

3. PARALLEL ALGORITHM

3.0 Introduction

Traditionally, algorithm plays an important role in computing. There cannot be any computer program without a corresponding algorithm. The same applies to parallel program; there cannot be any parallel program without a corresponding parallel algorithm. Parallel algorithm, therefore, is the sequence of steps that the parallel processors of a parallel computer will use to solve a particular problem. There are different approaches that can be used to develop parallel algorithm, which include the following:

- Algorithm Decomposition

- Data/Domain Decomposition

3.1 Algorithm Decomposition:

The algorithm decomposition approach to developing parallel algorithm identifies steps in a traditional/sequential algorithm that can be executed concurrently. The steps that can be executed concurrently in a sequential algorithm are used to decompose the sequential algorithm into parallel algorithm. A formal way of verifying steps/statements of a sequential algorithm that can be executed concurrently is the use of Bernstein concurrency conditions. This standard concurrency conditions applies to steps of a sequential algorithm that involve mathematical statements, and it is limited to only two statements. This research work will generalize the Bernstein concurrency conditions so that it can be used to verify if, $n > 2$, different statements of a sequential algorithm can be executed concurrently. Having identified steps/statements of a sequential algorithm that can be executed concurrently, the parallel algorithm can be developed by using a precedence graph, showing the order to execution of the parallel algorithm and the statements that will be executed concurrently. The steps/statements of a parallel algorithm as depict by the precedence graph that can be executed concurrently will be assigned to the various processors for execution at appropriate times. Therefore, the algorithm decomposition approach of developing parallel algorithm decomposes the sequential algorithm by identifying steps/statements of a sequential algorithm that can be executed concurrently.

3.2 Data/Domain Decomposition:

Data/Domain decomposition approach of developing parallel algorithm considers the data requirement of the problem with the aim of decomposing or splitting the data. Each of the processors will apply the same sequential algorithm on different set of data. This approach to parallel algorithm development is ideal when the data requirement of the problem is such that can be split, and it fits into the Sequential Algorithm Multiple Data Model (SAMD)

5. Conditions for Concurrency

5.1 Introduction

The development of parallel algorithms and programs can be complex and difficult, except formal and systematic tools are used in the development process. Two parallel algorithm development approaches can be used to develop parallel algorithms, these are algorithmic and domain (data) decomposition approaches. The algorithmic decomposition approach decomposes the sequential algorithm by identifying steps in the sequential algorithm that can be executed in parallel, while domain (data) decomposition uses the same sequential algorithm, but the data requirement of the problem is decomposed, leading to sequential algorithm multiple data model, SAMD model.

The algorithmic decomposition approach uses a standard and formal tool, Bernstein Concurrency Conditions to identify and decompose statements in the sequential algorithm that can be executed in parallel by the parallel processors. This approach is ideal when the algorithm involves large data structures, like arrays, which can be represented as algebraic expressions. However, the traditional, formal tool, Bernstein Concurrency Conditions is limited to two-processor, parallel computer system.

This chapter, therefore, generalizes the Bernstein Concurrency Conditions as a tool that can be used to develop parallel algorithms for an n – processor, parallel computer [1] (n > 2).
A. J Bernstein established a mathematical rule that must be satisfied before two statements, which are expressed as mathematical expressions be executed in parallel [1]. The rule is based on mathematical set theory and it can be found in Peterson, J. L, Silberschatz, A. (1985). Operating System Concepts, Addison-Wesley Publishing Company. For n = 2 processors of a parallel computer system, such mathematical rule can be used to establish the conditions that must be fulfilled before two statements, S1, S2, can be executed simultaneously.
Before stating the Bernstein Concurrency Conditions, it is important to define the following:

Definition 1.2:
Read Set of a Statement, R(S1):
This is the set of all the variables of the statement, S1 that the statement references, without modifying [1].

Definition 1.3
Write set of a statement, W(S1):
This is the set of all variables of the statement, S1, which the statement modifies during its execution [1].

5.2 Methodology

The mathematical tool used is set theory and its operators, like intersection and union, $\cap$, $\cup$. S1, S2,S3, …Sn are statements of a program. The usual empty set notation is ϕ

5.3 Bernstein Concurrency Conditions

Bernstein Concurrency Conditions for a two processor parallel computer states that for two statements, S1 and S2 of a parallel program to be executed concurrently or simultaneously by a two processor, parallel computer that the following conditions must be satisfied:

$$\left.\begin{aligned} R(S1) \cap W(S2) &= \phi \\ W(S1) \cap R(S2) &= \phi \\ W(S1) \cap W(S2) &= \phi \end{aligned}\right\}$$

5.4 The Generalized Bernstein Concurrency Conditions

Therefore, to obtain a generalization of this condition, means establishing the conditions that must be satisfied before S1, S2, S3, …, Sn statements will be executed concurrently by the n-processors of a parallel computer (n > 2).
It suffices to assume that statements, S1 and S2 can be executed concurrently, statements, S1 and S3 can be executed concurrently, statements, S1 and S4 can be executed concurrently, in general, statements, S1 and Sn can be executed concurrently.

Therefore, using the traditional Bernstein Concurrency Conditions, the following remain true.

$$\left.\begin{array}{l} R(S1) \cap W(S2) = \phi \\ W(S1) \cap R(S2) = \phi \\ W(S1) \cap W(S2) = \phi \end{array}\right\} \quad (1)$$

$$\left.\begin{array}{l} R(S1) \cap W(S3) = \phi \\ W(S1) \cap R(S3) = \phi \\ W(S1) \cap W(S3) = \phi \end{array}\right\} \quad (2)$$

$$\left.\begin{array}{l} R(S1) \cap W(S4) = \phi \\ W(S1) \cap R(S4) = \phi \\ W(S1) \cap W(S4) = \phi \end{array}\right\} \quad (3)$$

In general,

$$\left.\begin{array}{l} R(S1) \cap W(Sn) = \phi \\ W(S1) \cap R(Sn) = \phi \\ W(S1) \cap W(Sn) = \phi \end{array}\right\} \quad (n+1)$$

Combining rules (1), (2), (3) …, (n+1) that have been stated above, the following can be established:

$(R(S1) \cap W(S2)) \cup (R(S1) \cap W(S3)) \cup \ldots \cup (R(S1) \cap W(Sn)) = \phi$
$(W(S1) \cap R(S2)) \cup (W(S1) \cap R(S3)) \cup \ldots \cup (W(S1) \cap R(Sn)) = \phi$
$(W(S1) \cap W(S2)) \cup (W(S1) \cap W(S3)) \cup \ldots \cup (W(S1) \cap W(Sn)) = \phi$

Because intersection is distributive over union, [3], the following hold:

$R(S1) \cap (W(S2) \cup W(S3) \cup \ldots \cup W(Sn)) = \phi$
$W(S1) \cap (R(S2) \cup R(S3) \cup \ldots \cup R(Sn) = \phi$
$W(S1) \cap (W(S2) \cup W(S3) \cup \ldots \cup W(Sn)) = \phi$

These imply the following:

$$\left.\begin{array}{l} R(S1) \cap (\bigcup_{j=2}^{n} W(Sj)) = \phi \\ W(S1) \cap (\bigcup_{j=2}^{n} R(Sj)) = \phi \\ W(S1) \cap (\bigcup_{j=2}^{n} W(Sj)) = \phi \end{array}\right\} \quad I$$

In general, this can be written as follows:

$$R(Si) \cap (\bigcup_{j=2}^{n} W(Sj)) = \phi \quad \{j=1,\ i \neq j\}^n \ (j=1,\ not\ 2)$$

$$W(Si) \cap (\bigcup_{j=2}^{n} R(Sj)) = \phi \quad \{j=1,\ i \neq j\}^n \ (j=1,\ not\ 2)$$

$$W(Si) \cap (\bigcup_{j=1}^{n} W(Sj)) = \phi \quad \{j=1,\ i \neq j\}^n \ (j=1,\ not\ 2)$$

The above constitutes the Generalized Bernstein Concurrency conditions.

5.5 Illustrating with an example:

Consider the following sequential algorithm.

```
S0      Begin
S1              d : = -B ;
S2              e : = B * B ;
S3              z : = 4 * A * C ;
S4              l : = 2 * A ;
S5              k : = e-z;
S6              m : = d + k;
S7              n : = d-k;
S8              x : = m*l;
S9              y : = n*l
S10             End
```

The Generalized Bernstein concurrency conditions will be used to identify steps within the sequential algorithm that can be executed in parallel. Afterwards, the precedence graph will be used to model the parallel algorithm.
Therefore, the formal and novel tool will be used to establish if statements S1, S2, S3, and S4 can be executed in parallel by a four-processor parallel computer system.

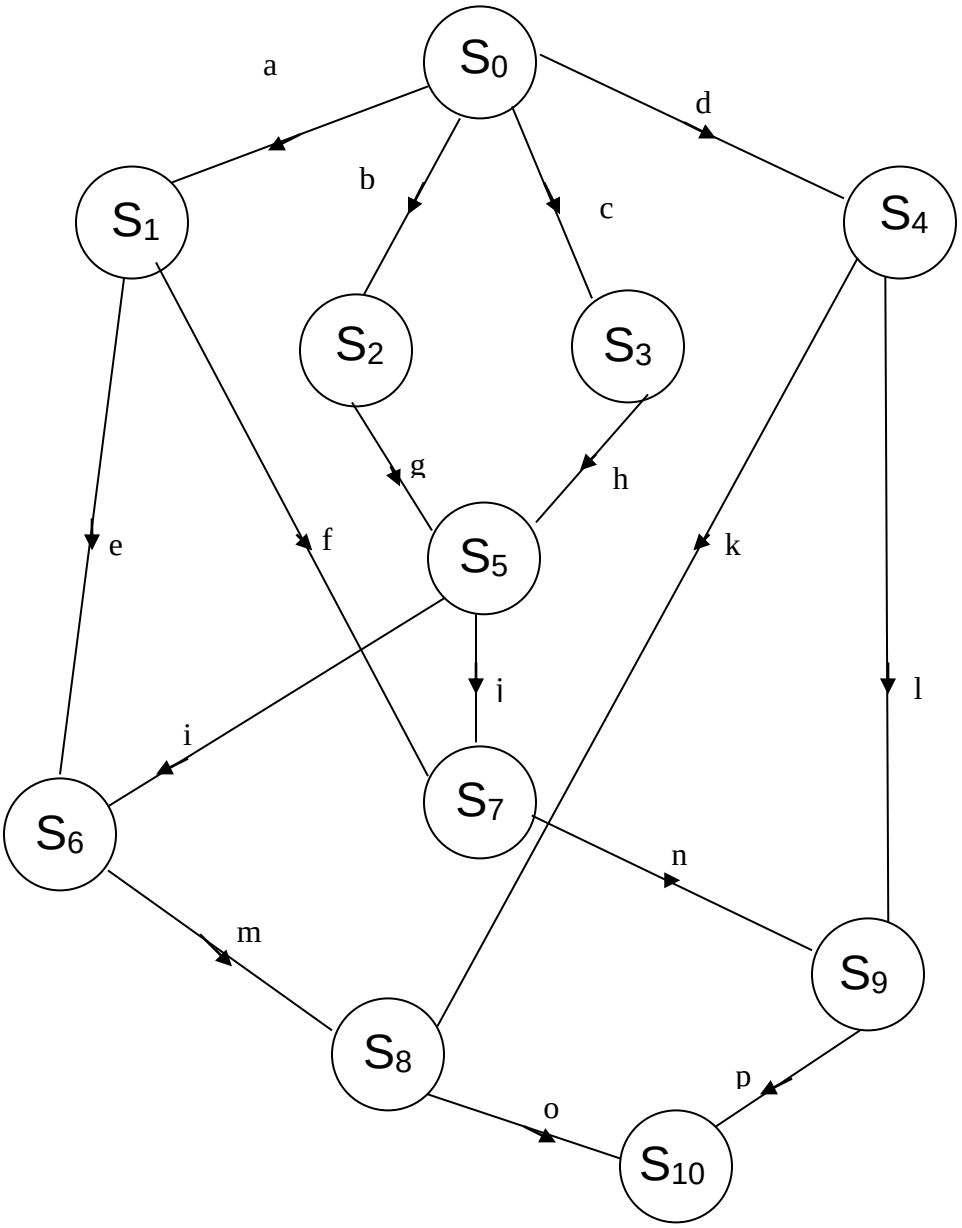
R(S1) = {B}; W(S2) = {e}; W(S3) = {z}; W(S4) = {1}
W(S1) = {d}; R(S2) = {B}; R(S3) = {A,C}; R(S4) = {A}
Therefore,

$$R(S1) \cap (\bigcup_{j=2}^{4} W(Sj)) = \{B\} \cap \{e,z,l\} = \phi$$

$$W(S1) \cap (\bigcup_{j=2}^{4} R(Sj)) = \{d\} \cap \{B,A,C\} = \phi$$

$$W(S1) \cap (\bigcup_{j=2}^{4} W(Sj)) = \{d\} \cap \{e,z,l\} = \phi$$

This means that statements S1, S2, S3 and S4 can be executed in parallel by a four-processor parallel computer system. The traditional Bernstein concurrency conditions can be used to establish that statements S5 and S6 can be executed in parallel. The parallel algorithm can now be modeled, using the precedence graph.


S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p

5.6 Parallel Programming:

The following parallel program can be developed, using the following concurrent control structures:

- FORK and JOIN
- Semaphore with PARBEGIN PAREND

5.6.1 Using FORK and JOIN

```
        count1 := 2;
        count2 := 2;
        count3 := 2;
        count4 := 2;
        count5 := 2;
        count6 := 2;
        S0;
        FORK L1;
        FORK L2;
        FORK L3;
        S1;
        FORK L5;
        GOTO L6;
L1:     S2
        GOTO L4;
L2:     S3;
        GOTO L4;
L3:     S4;
        FORK L7;
        GOTO L8
L4:     JOIN count1;
        S5;
        FORK L5;
        GOTO L6;
L5:     JOIN count2
        S7;
        GOTO L7;
L6:     JOIN count3
        S6
        GOTO L8
L7:     JOIN count4;
        S9;
        GOTO L9
L8:     JOIN count5;
        S8;
L9:     JOIN count6;
        S10
```

5.6.2 Using Semaphore

The parallel program follows, using Semaphore

```
VAR a, b, c, d, e, f, g, h, I, j, k, l, m, n, o  : SEMAPHORE;
(*Initialize all semaphore variables to zero *)
PARBEGIN
        BEGIN S0; V(a); V(b); V(c); V(d) END
        BEGIN P(a); S1; V(e); V(f) END
        BEGIN P(b); S2; V(g) END
        BEGIN P(c); S3; V(h) END
        BEGIN P(d); S4; V(k); V(l) END
        BEGIN P(g); P(h); S5; V(i); V(j) END
        BEGIN P(e); P(i); S6; V(m) END
        BEGIN P(f); P(j) S7; V(n) END
        BEGIN P(m); P(k); S8; V(o) END
        BEGIN P(n); P(l); S9; V(p) END
        BEGIN P(o); P(p) S10 END
PAREND
```

P and V are the two semaphore operations, and S0 and s10 are the initialization and termination codes. This version of parallel program requires a maximum of eleven processors. The program can be restructured as follows:

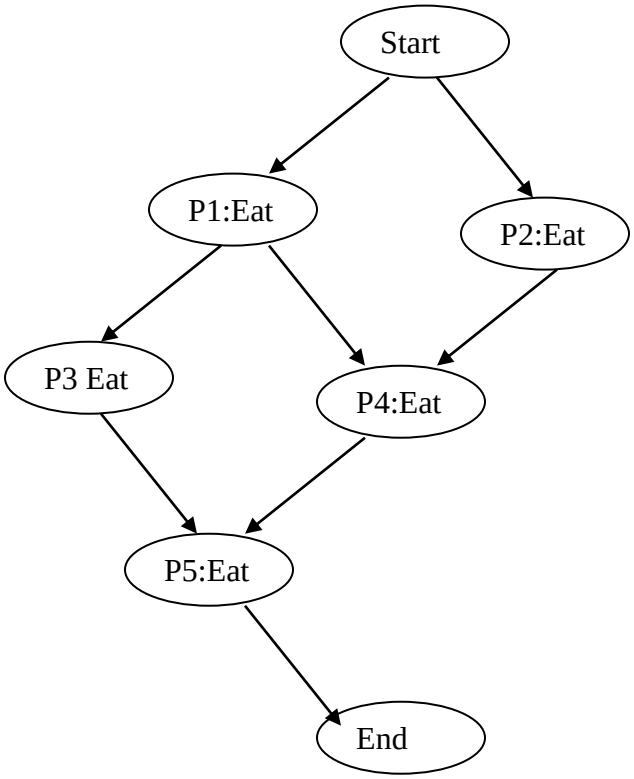
```
VAR a, b, c, d, e, f, g, h, I, j, k, l, m, n, o  : SEMAPHORE;
(*Initialize all semaphore variables to zero *)

S0;
PARBEGIN
        BEGIN S1; V(e); V(f) END
        BEGIN S2; V(g) END
        BEGIN S3; V(h) END
        BEGIN S4; V(k); V(l) END
PAREND
PARBEGIN
        BEGIN P(g); P(h); S5; V(i); V(j) END
        BEGIN P(e); P(i); S6; V(m) END
        BEGIN P(f); P(j) S7; V(n) END
        BEGIN P(m); P(k); S8; V(o) END
        BEGIN P(n); P(l); S9; V(p) END
        BEGIN P(o); P(p) S10 END
PAREND
```

This version of the parallel program requires a maximum of six processors. Both of them are correct parallel programs.

5.7 DINING PHILOSOPHERS PROBLEM:

A classical concurrency problem that illustrates data decomposition approach to parallel algorithm development and the use of shared resources is the dining philosophers problem, which was proposed by E.W Dijkstra. There are five philosophers, each uses this sequential algorithm of eating and thinking in that order. In order to eat, a round dining table has been set for them. Five plates of spaghetti have been placed on the table, and beside each plate are two forks.

Before each philosopher will eat, he/she must hold the two forks that are on the sides of each plate. Each philosopher has equal time to eat. As soon as each philosopher finishes eating, he/she begins to think. The problem, therefore is to devise a synchronization mechanism that the five philosophers will use to eat the spaghetti in order to avoid the following synchronization problem of deadlock and starvation, which are as a result of the use of shared resource, fork. Deadlock occurs when each of the philosophers holds a fork and waits for each other to release the other fork, so that he/she will eat. Starvation occurs when some of the philosophers are not given the opportunity to eat, because some the philosophers refuse to release the fork after eating. Ordering the manner that the five philosophers will use to eat the spaghetti can solve these two undesirable problems. Since for is a limited resource, therefore, all of them can not eat at the same time. A maximum of two philosophers can eat at any given time, therefore, a schedule of eating can be prepared in such a way that the first two philosophers will start eating, as soon as they finish eating, the forks will be released and they will start thinking, while they are thinking, the next two philosophers will be scheduled to starting eating, as soon as they finish eating, their forks will be released, they will start thinking, while they are thinking, the last philosopher will be scheduled to start eating, as soon as he/she finishes eating, he/she will release his/her fork and will begin to think. When the eating and thinking activities of the five philosophers are scheduled in this manner, the problem of deadlock and starvation will be solved. The following precedence graph can be used to show the order the philosophers will use to eat the spaghetti.



5.8 Conclusion

I have been able to generalize the traditional Bernstein Conditions of Concurrency to take care of the conditions that must be satisfied before n parallel processors will execute n different statements in parallel.

References

[1] Peterson, J. L, Silberschatz, A. (1985). Operating System Concepts, Addison-Wesley Publishing Company.

[2] Chien-Min WANG and Sheng.De WANG, Structured Partitioning of Concurrent Program for execution on multiprocessors, J. Parallel Computing.

[3] I.N. Herstein; Topics in Algebra; 1976; John Wiley & sons Inc.

[4] S. Aryan, B. Gaither; Parallel algorithm development workbench; 1998; Proc. Of the 1998 ACM/IEEE conference on Supercomputing