Chapter 1

Introduction

1.1 Parallel Processing

There is a continual demand for greater computational speed from a computer system than is currently possible (i.e. sequential systems). Areas need great computational speed include numerical modeling and simulation of scientific and engineering problems. For example; weather forecasting, predicting the motion of the astronomical bodies in the space, virtual reality, etc. Such problems are known as grand challenge problems. On the other hand, the grand challenge problem is the problem that cannot be solved in a reasonable amount of time [1].

One way of increasing the computational speed is by using multiple processors in single case box or network of computers like cluster operate together on a single problem. Therefore, the overall problem is needed to split into partitions, with each partition is performed by a separate processor in parallel. Writing programs for this form of computation is known as parallel programming [1]. How to execute the programs of applications in very fast way and on a concurrent manner? This is known as parallel processing. In the parallel processing, we must have underline parallel architectures, as well as, parallel programming languages and algorithms.

1.2 Parallel Architectures

The main feature of a parallel architecture is that there is more than one processor. These processors may communicate and cooperate with one another to execute the program instructions.

There are diverse classifications for the parallel architectures and the most popular one is the Flynn taxonomy (see Figure 1.1) [2].

1.2.1 The Flynn Taxonomy

Michael Flynn [2] has introduced taxonomy for various computer architectures based on notions of Instruction Streams (IS) and Data Streams (DS). According to this taxonomy, the parallel architectures could be classified into four categories; Single Instruction Single Data (SISD), Multiple Instruction Single Data (MISD), Single Instruction Multiple Data (SIMD), and Multiple Instruction Multiple Data (MIMD).

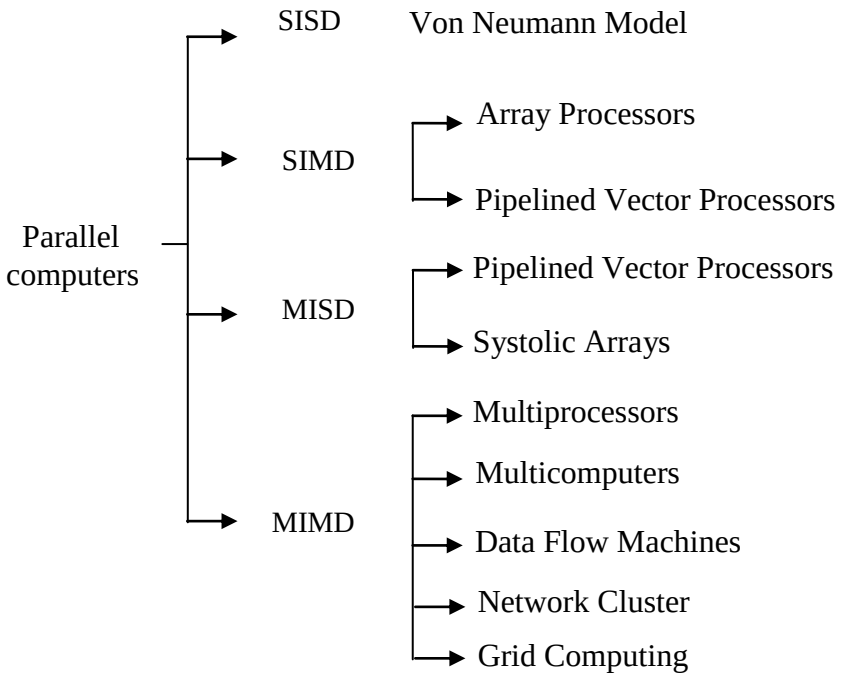

Figure 1.1: Taxonomy of Parallel Processing Architectures

Single Instruction Single Data (SISD)

According to this category, only one instruction stream is executed on one data stream each time. Conventional sequential machines are considered SISD architecture. The prototype of the sequential machine is shown in Figure 1.2.

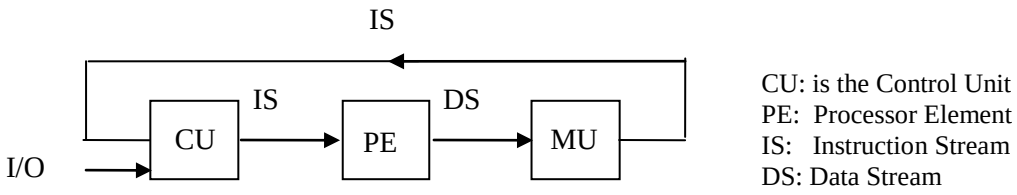

Figure 1.2: SISD Uniprocessor Architecture.

Single Instruction Multiple Data Stream (SIMD)

The SIMD architectures consist of **N** processors (see Figure 1.3). The processors operate synchronously, where at each cycle all processors execute the same instruction, each on a different datum. Some popular commercial SIMD architectures are ILLIAC IV, DAP, and connection machine CM-1 and CM-2. The SIMD architectures can also support vector processing, which can be accomplished by assigning vector of elements to individual processors for concurrent computation [3].

SIMD machine is a synchronized machine, where all of the processors in the machine execute only one instruction at a time and the fasting processor has to wait the slowest one before it is starting executing next instruction. This type of synchronization is called Lockstep [3]. These architectures are used mostly for problems having high degrees of data parallelism.

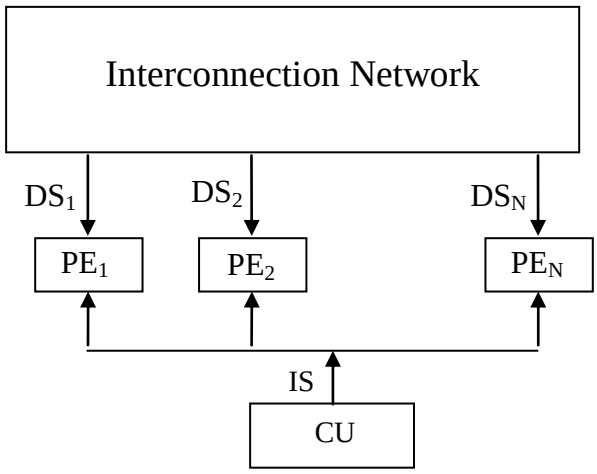

Figure 1.3: SIMD Architecture

Multiple Instructions Single Data (MISD)

The MISD architectures consist of **N** processors each with its own control unit sharing a common memory where data reside (see Figure 1.4) [3].

The architecture of the MISD computers fall into two different categories:

1. A class of machines that would require distinct processors that would receive distinct instructions to be performed on the same data.
2. A class of machines such that data flows through a series of processors (i.e. pipelined architectures). On type of this machine is a systolic array.

A systolic array is formed with a network of functional units which are locally connected. This array operates synchronously with multidimensional pipelining. Therefore, this class of multidimensional pipelined array architectures is designed for implementing fixed algorithms and it offers good performance for special applications, like signal and image processing, with simple, regular and modular layouts.

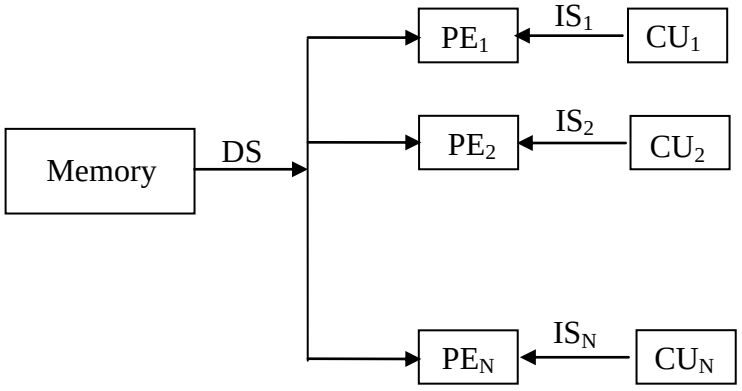

Figure 1.4: MISD Architecture

Multiple Instructions Multiple Data (MIMD)

This class of architectures is the most general and powerful one, where most of practical applications need MIMD machines [3]. The MIMD architectures consist of **N** processors. Each processor operates under the control of an instruction stream issued by its control unit (see Figure 1.5). All processors are potentially executing different subproblems on different data while solving a single problem. This means that the processors typically operate asynchronously. The communication between processors is preformed through a

shared memory or an interconnection network. MIMD computers with a shared common memory are often referred to as multiprocessors (or tightly coupled machines), while those with an interconnection network connecting processors are known as multicomputers (or loosely coupled machines) [4].

Gordon Bell [5] has provided taxonomy of MIMD machines (see Figure 1.6). He considers shared-memory multiprocessors as having a single address space (i.e. global address space) and multicomputers use distributed memories as having multiple address space (i.e. local address space).

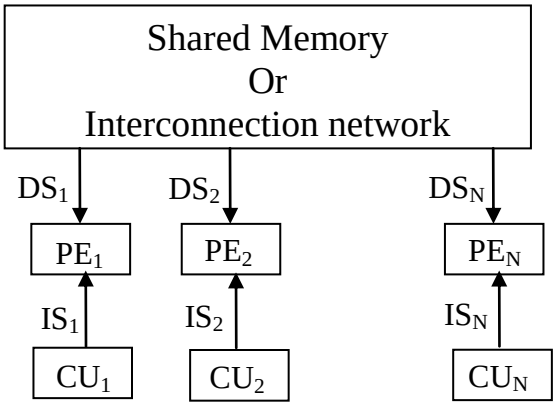

Figure 1.5: MIMD Architecture

Multicomputer Architectures.

According to Bell' taxonomy of MIMD architectures, the multicomputer architecture is classified into distributed multicomputer architectures and central multicomputer. Distributed multicomputer architecture (see Figure 1.7) consists of multiple computers interconnected by a message-passing network. Each computer consists of a processor, local memory, and attached I/O peripherals. All local memories are private and not accessible by other processors. The communication between processors is achieved through a message-passing interface protocol, via a general interconnection network. The disadvantage of these architectures is that a programming model imposes a significant burden on the programmer, which induces considerable software overhead. On the other hand, these architectures are claimed to have better scalability and cost-effectiveness [6].

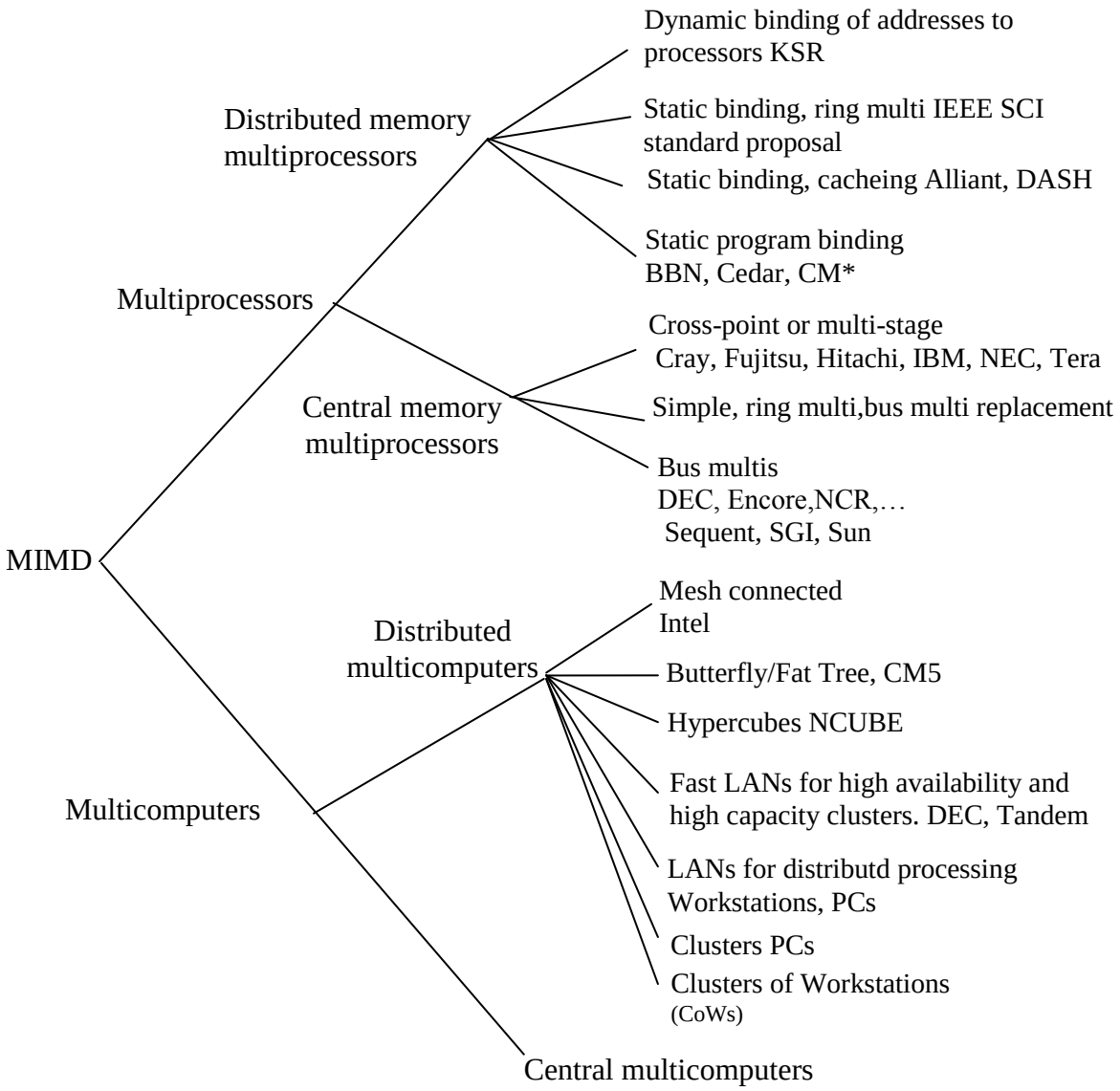

Figure 1.6: Bell's Taxonomy of MIMD Computers

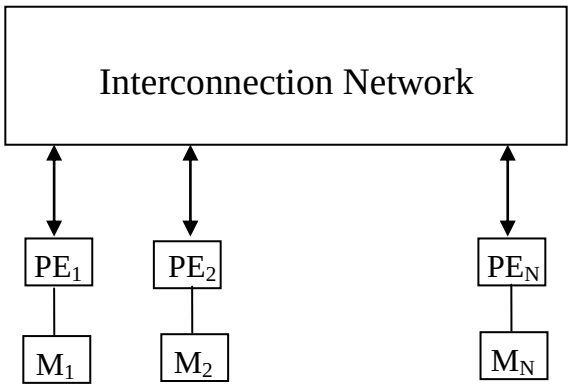

Figure 1.7: Generic Model of Message-Passing Multicomputer

According to central multicomputers (see Figure 1.8), there is one computer is considered as a master one where all of the database as well as the application programs are resident their. Before executing a program the master computer has to distribute the program partitions into the other processors (slaves). After finishing executing the program, the final results from the slaves are received and organized in the master. This type of multicomputers is referred as farm model.

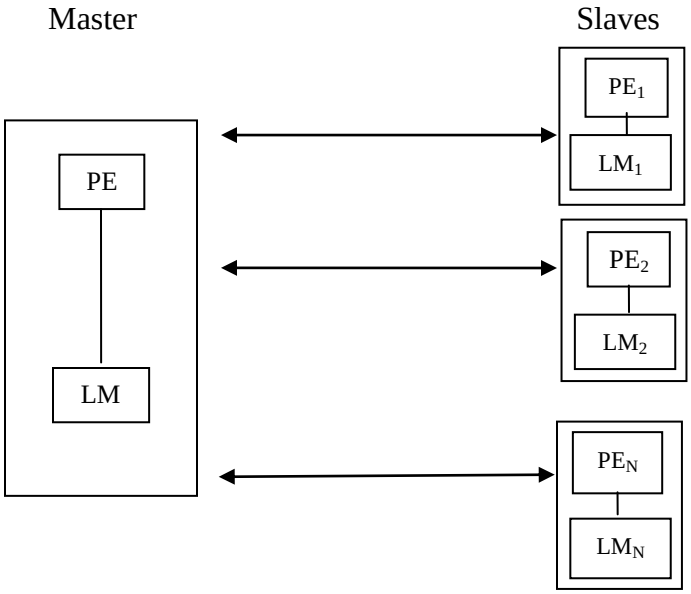

Figure 1.8: Central Multicomputer

Multiprocessor Architecture

Multiprocessor architecture called shared memory architecture, where a single global memory is shared among all processors (i.e. global space memory model). Consequently, the multiprocessor is classified into two category; shared memory multiprocessor and distributed shared memory (DSM) [6]. According to shared memory multiprocessor, only one memory is accessible by all processors equally (see Figure 1.9). The main advantage of these architectures is the simplicity in designing algorithms for them and transparent data access to the user. However, it suffers from increasing contention in accessing the shared memory, which limits the scalability [7].

A relatively new concept - (DSM) - tries to combine the advantages of the multicomputer and multiprocessors architectures [8].

The DSM architecture can be generally viewed as a set of nodes or clusters, connected by an interconnection network (see Figure 1.10). Each clusters organized around a shared bus contains a physically local memory module, which is partially or entirely mapped to the DSM global address space. Private caches attached to the processors are inevitable for reducing memory latency. Regardless of the network topology a specific interconnection controller within each cluster is needed to connect it into the system. Therefore, the DSM architecture logically implements the shared memory model in physically distributed memory architecture [8].

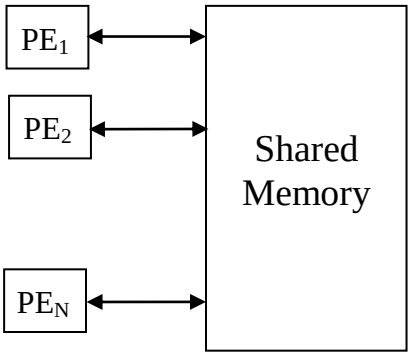

Figure 1.9: Generic Model of Shared Memory Architecture.

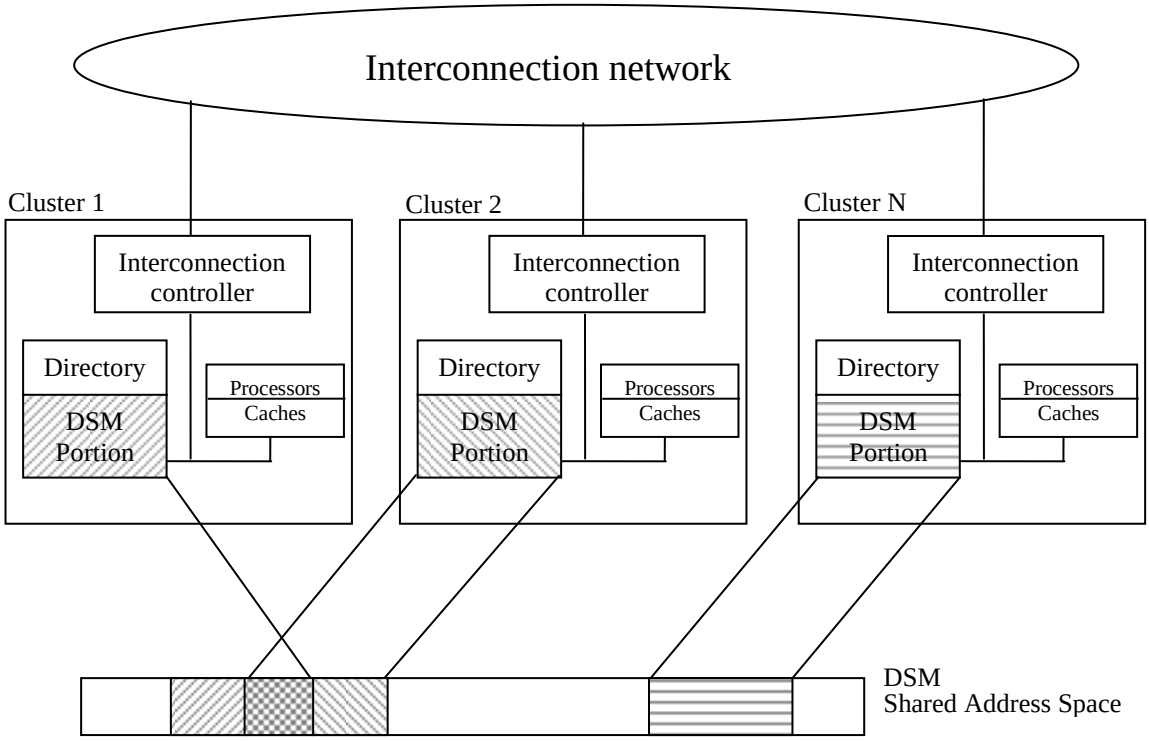

Figure 1.10: Structure and Organization of a DSM System.

1.3 The Parallel Algorithms

Generally, a parallel algorithm can be defined as a set of instructions that may be executed in parallel concurrently and may, connect, with each other in order to solve a given problem [3]. The term task or process may be defined as a part of a program that can be executed on a processor. The main phases of a parallel algorithm include four main phases; partitioning, communication, agglomeration, and mapping phase (see Figure 1.11). [3].

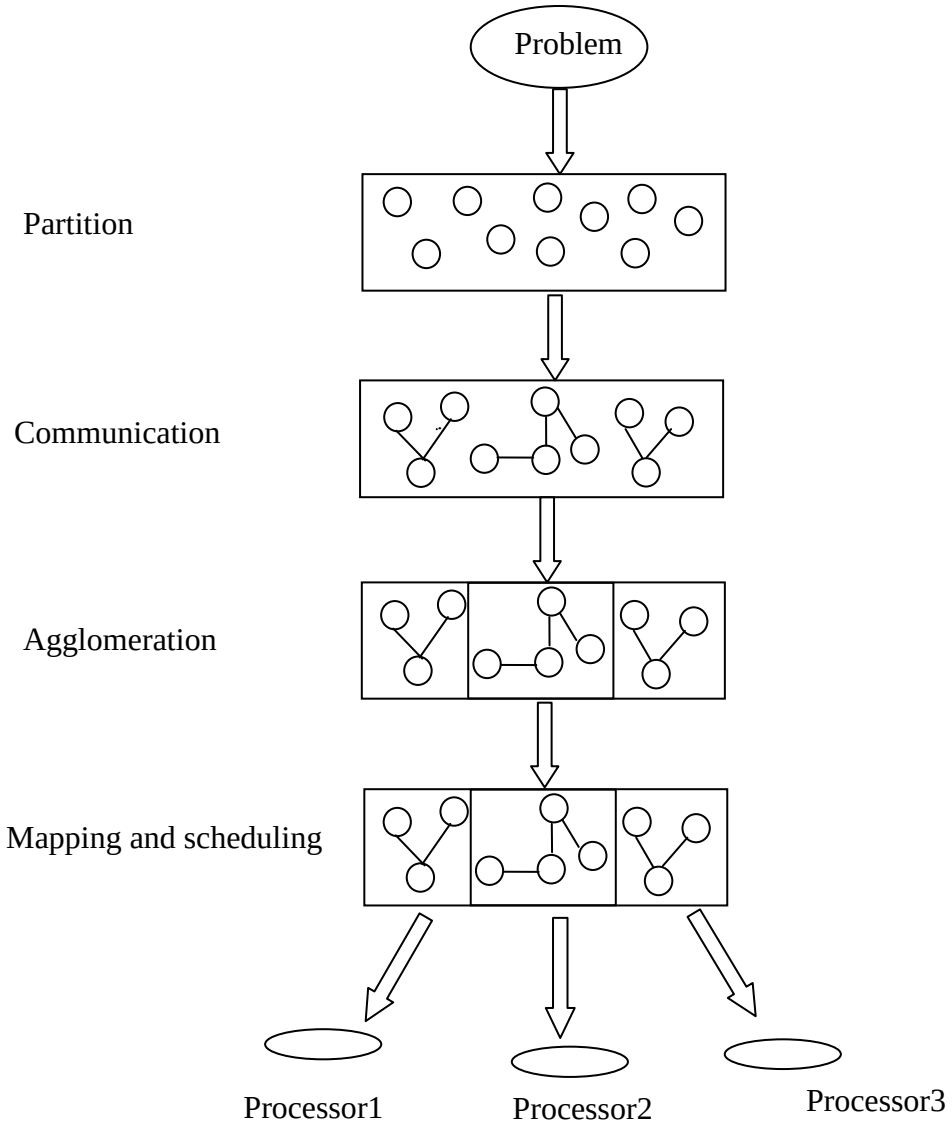

Figure 1.11: Different Phases of a Parallel Algorithm

1.3.1 Partitioning Phase

The partitioning phase is intended to expose opportunities for parallel execution by decomposing computations of assigned problem into small tasks. A good partitioning method divides both the computation and the data associated with a problem into little instructions [3].

1.3.2 Communication Phase

The communication phase is enables an appropriate communication structure and algorithms to coordinate task execution. To allow computation to proceed, the data must be transferred between tasks, which are the outcome of the communication phase of a design. There are two communication structures: channel structure and message-passing structure [3]. In the channel structure, there is a link (directly or indirectly) between tasks that require data with other tasks that possess those data. In the message passing structure, the message is sent and received on these channels.

1.3.3 Agglomeration phase

The first two phases of the design process concern about dividing the computations into a set of tasks and communication to provide data required by these tasks. The agglomeration phase is intended to evaluate the tasks and the required communication with respect to the performance and implementation costs. For example, one critical issue influencing the parallel performance is the communication cost. The performance might be improved by reducing the amount of time spent communicating (i.e. sending less data) [3].

1.3.4 Mapping and Scheduling Phase

The mapping and scheduling phase aims to specify where each task has to be executed in which processor and determine a sequence for their execution such that the processor utilization is maximized and the communication costs, as well as, execution time are minimized [3].

If these four phases (problems) are not properly handled, parallelization of an application may not be beneficial. A large research efforts addressing theses problems has been

reported in the literature [9,10]. Our work is concerned mainly to study and implement the mapping and schedule phase. Through the thesis, we refer to this phase as task scheduling problem.

Two important issues are involved in the task scheduling problem: enhancing concurrency and increasing locality. Enhancing concurrency deals with placing tasks that can be executed simultaneously on different processors, while increasing locality refers to placing tasks that likely communicate frequently on the same processor [3].

1.4 The Task Scheduling Problem

The task scheduling problem concerns with determine which computational tasks will be executed on which processor and at what time. On the other hand, the scheduling and allocation of multiple interacting tasks of a single parallel program is a highly important issue since an inappropriate scheduling of tasks can fail to exploit the true performance of the system and can offset the grain from parallelization [10]. Therefore, this problem has received considerable attention in recent years [11, 12].

The main objective of the task scheduling is to assign tasks to available processors such that precedence requirements between tasks are satisfied and, in the same time, the overall completion time which known as schedule length (or make span) is minimized [13]. The task scheduling techniques are mainly classified as static and dynamic. In static scheduling, the characteristic of a parallel program, including task processing times, data dependencies and synchronization, are known before program execution [10]. In dynamic scheduling algorithm, few assumptions about the parallel program can be made before execution, and thus, scheduling decisions have to be made on-the-fly [13]. On the other hand, the scheduling problem could be classified into preemptive and nonpreemptive scheduling [13]. The preemptive scheduling permits a task to be interrupted and removed from the processor under the assumption that it will eventually receive the execution time it requires. This interruption of tasks contributes to system overhead. With nonpreemptive scheduling, a task cannot be interrupted once it has begun execution in a specific processor. If there are no precedence relations among the tasks forming a

program, the problem is known as task allocation problem [13]. Our research work is focused on static and nonpreemptive scheduling.

1.4.1 The Problem Model

The model of the underline parallel system to be considered in this research work could be described as follows [14]:

The architecture is a network of arbitrary number of homogeneous processors. Assume P = {P_1, P_2, P_3... P_m} denotes the set of *m* processors. Let a task graph G be a Directed, Acyclic Graph (DAG) composed of N nodes $n_1, n_2, \ldots, n_N$, each node terms a task of the graph which in turn is a set of instruction that must be executed sequentially without preemption in the same processor. A node[1] has one or more inputs. When all inputs are available, the node is triggered to execute. A node with no predecessor is called an entry node and a node with no successor is called an exit node. The weight of node n_i is called the computation cost of a node n_i and is denoted by $weight(n_i)$. The graph also has E directed edges, where each edge $e(n_i, n_j) \in E$ representing a partial order among the tasks. The partial order introduced a precedence-constrained DAG and implies that if $n_i \rightarrow n_j$, then n_j is a successor of n_i, which cannot be started until its predecessor n_i finishes. The weight on an edge is called communication cost of the edge and is denoted by $c(n_i, n_j)$. This cost is incurred if n_i and n_j are scheduled on different processors and is considered to be zero if n_i and n_j are scheduled on the same processor. Let $p(n_i)$ and $Sc(n_i)$ be the set of immediate predecessors and the set of successors of the node n_i respectively. Where, $p(n_i) = \{n_j : e(n_j, n_i) \in E\}$ and $Sc(n_i) = \{n_j : e(n_i, n_j) \in E\}$. If a node n_i is scheduled to processor P, the start time and finish time of the node is denoted by $ST(n_i)$ and $FT(n_i)$ respectively. After all nodes have been scheduled, the schedule length is defined as $\max_i\{FT(n_i)\}$ across all processors. The objective is to find an assignment and the start times of the tasks to processors such that the schedule length is minimized and, in the same time, the precedence constrains are preserved. A Critical Path

[1] task and node are exchangeable through the thesis

(CP) of a task graph is defined as the path with the maximum sum of node and edge weights from an entry node to an exit node. The nodes lies on CP are denoted as CPNs, and the communication-to-computation-ratio (CCR) of a parallel program is defined as its average edge weight divided by its average task weight.

The task scheduling in general is known to be NP-complete problem except for some special cases. The suggested taxonomy of the static task scheduling algorithms is represented in Figure 1.12. According to this taxonomy, scheduling methods are divided into optimal solution and non optimal solution.

According to the non optimal solutions, many heuristics have been suggested to tackle the problem under more pragmatic situations. These heuristics algorithms could be divided in two categories: greedy and non greedy (iterative). The greedy algorithms are initialized by a partial solution and search to extend this solution until a complete task scheduling is achieved [15]. At each step, one task assignment is done and it cannot be changed in the remaining steps.

The iterative algorithms are initialized by a complete scheduling and search to improve it by moving a task from one processor to another or by exchanging the mapping of two tasks [15].

1.4.2 Optimal Task Scheduling Algorithms

The optimal solution is known in few restricted cases, such as polynomial – time algorithms which described as Hu algorithm [13], Coffman and Graham algorithm [16], and Papadimitriou and Yannakakis algorithm [13].

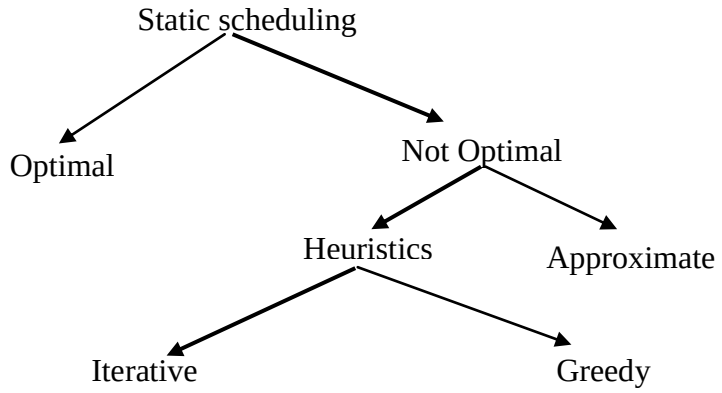

Figure 1.12: A Partial Taxonomy of Static Scheduling Methods

1. HU Algorithm

Hu [13] devised a linear-time algorithm to scheduling problem called level algorithm. The algorithms is used to solve the schedule length in linear time when the task graph is either an in-forest, i.e., each task at most one immediate successor, or an out-forset, i.e., each task has at most one immediate predecessor. The communication between tasks is ignored and all tasks have unit computations.

The algorithm begin with calculate the level of each node n_i which is defined as the maximum number of nodes (including n_i) on any path from n_i to the exit node. The pseudo code of the algorithm is as follow:

HU Algorithm

Step1: The level of each node in the DAG is calculated and used as each nodes priority.

Step2: **When** the processor become available. Assign it the ready node with the highest priority.
IF the number of nodes in a level is greater than the number of processors in the system,
Then schedule the tasks using round robin fashion.

The complexity of the algorithm is O(v) where v is the number of tasks in DAG[13].

Example

By applying HU algorithm to the DAG shown in Figure 1.13a using three processors, the list of read tasks according to its level is { 1,2,3,4,5,6,7,8,9}. The generated schedule length is shown in Figure 1.13b.

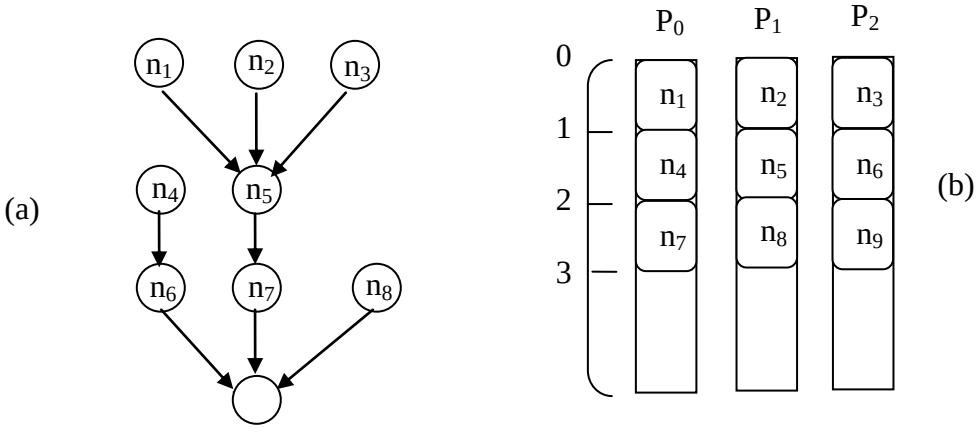

Figure1.13: (a) A Simple In-Forest Task Graph; (b) The Optimal Schedule of the Task Graph using Three Processors System

2. Coffman and Graham Algorithm

Coffman and Graham [16] devised a quadratic-time algorithm for scheduling an arbitrary structured DAG with unit-weighted tasks and zero-weighted edges to a two-processors system. The algorithm starts by assigning Labels to DAG's tasks starting with the exit task. After that, a list of tasks is constructed by sorting them in descending order. Schedule each task to one of the two processors that allows earliest start time. The pseudo code of the algorithm is as follow:

Coffman and Graham Algorithm

Step1: Assign the number 1 to one of the exit tasks
Step2: Let labels $1,2,\ldots, j-1$ have been assigned. Let S be the set of unassigned tasks with no unlabeled successor.
Select an element of S to be assigned label j.
For each task t in S,
Define $l(t)$ as follow: Let $n_1, n_2, \ldots, n_k$ be the immediate successors of t. Then $l(t)$ is the decreasing sequence of integers formed by ordering the set $\{L(n_1), L(n_2), \ldots, L(n_k)\}$. Let t be an element of S such that for all t' in S, $l(t) \le l(t')$ (lexicographically). Define $L(t)$ to be j.
End For
Step3: When all tasks have been labeled, use the list $(T_n, T_{n-1}, \ldots, T_1)$ where for all i, $1 \le i \le n$, $L(T_i) = i$ to schedule the tasks.

The complexity of the algorithm is $O(v^2)$ where v is the number of tasks in DAG [16].

Example:

Consider the task graph shown in Figure 1.14(a). The two terminal task are assigned the label 1,2 respectively. The set S of unassigned tasks with no unlabeled successors becomes $\{4,5\}$. Also it can be noticed that $l(4) = \{6\}$ and $l(5) = \{7,6\}$. Since {6} < {7, 6} (lexicographically), assign label 3, 4 as shown in Figure 1.14a. Begin the schedule by assigning the tasks with no predecessors after that assign the task in decreasing order of its label. Figure 1.14b shows the output schedule on two processor.

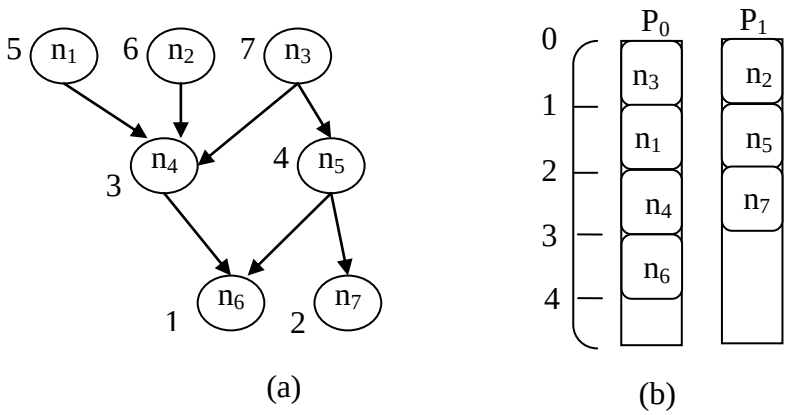

Figure1.14: (a) A Simple Task Graph with Unit-Tasks and No-Cost Communication Edges; (b) The Optimal Schedule of the Task Graph in a Two-Processors System

3. Papadimitriou and Yannakakis Algorithm

Optimal scheduling algorithm has also been addressed by Papadimitriou and Yannakakis [13]. They designed a linear-time algorithm to tackle the scheduling problem of an interval-ordered DAG with unit-weight nodes to an arbitrary number of processors. In an interval-ordered DAG, two nodes are precedence-related if and only if the nodes can be mapped to non-overlapping intervals on the real line. The pseudo code of the algorithm is as follow:

Papadimitriou and Yannakakis Algorithm

1. The number of successors of each node is used as its priority.
2. Whenever a processor becomes available, assign it the ready task with high priority.

This algorithm solves the scheduling problem for interval order (V, A) in $O(|A| + |v|)$.

Example:

Consider the problem of scheduling the interval order given in Figure 1.15a on two identical processors. The result according to Papadimitriou and Yannakakis Algorithm is shown in Figure 1.15b.

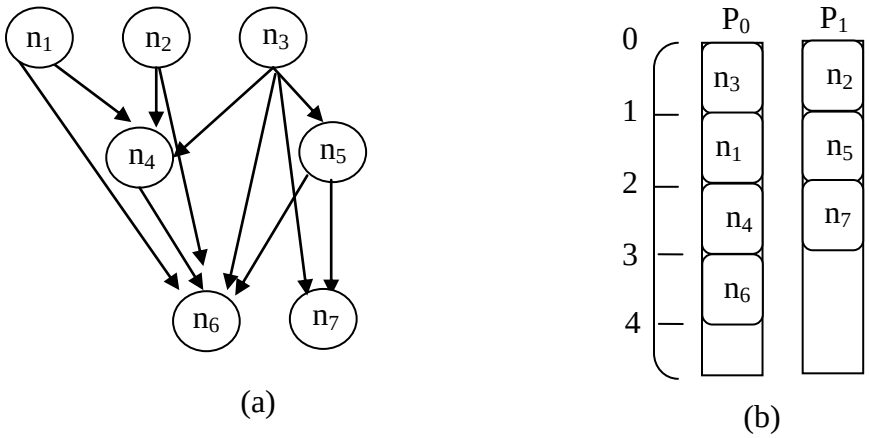

Figure 1.15: (a) A Unit-Computational Interval Ordered DAG; (b) An Optimal Schedule of The DAG.

1.5 Performance Criteria

The performance of task scheduling algorithms of parallel systems is generally evaluated and measured by means of some criteria (speedup, efficiency, normalized schedule length)[6, 17, 18].

1.5.1 Speedup

Speedup is a good measure for the execution of an application program on a parallel system. The speedup relates to the time for executing the program on a single processor to the time for executing the same program on a parallel system. Linear speedup means that the value of speedup increases as the number of processors in the parallel systems increases [6]. It is known that linear speedup does not occur in distributed memory architectures because of the communication overhead. Assume T(1) is the time required for executing a program on a uniprocessor computer and T(P) is the time required for executing the same program on a parallel computer containing P processors. Thus the speedup can be estimated as:

$$S(P) = \frac{T(1)}{T(P)}$$

Speedup Folklore Theorem [17]:

For a given computational problem, the speedup provided by a parallel program using P processors, over the fastest possible sequential program for the problem, is at most equal to P.

$$1 < S(P) \le P .$$

1.5.2 Efficiency

Efficiency is an indication to what percentage of a processor's time is being spent in useful computation. The efficiency of a parallel computer containing **P** processors can be defined as

$$E(P) = \frac{S(P)}{P}$$

Since $1 < S(P) \le P$, then $1/P < E(P) \le 1$.

The maximum efficiency is achieved when all P processors are fully utilized throughout the execution [6].

1.5.3 Normalized Schedule Length

The main performance measure of task scheduling algorithms is the schedule length. The Normalized Schedule Length (NSL) of an algorithm is defined as:

$$NSL = \frac{S\_Length}{\sum_{n_i \in CP} weight(n_i)},$$

Where S_Length is the maximum finish time of each task and *weight* (n_i) is the execution time of the node n_i. The sum of computation costs on the CP represents a lower bound on the schedule length [18]. Such lower bound may not always be possible achieve, and the optimal schedule length may be larger than this bound.

1.6 Objective of the Thesis

The main objective of the thesis is developed and implemented task scheduling algorithms using genetic approach to improve the performance of genetic algorithms, as well as, the heuristic one. So two genetic algorithms have been developed and

implemented: Critical Path Genetic Algorithms (CPGA) and Task Duplication Genetic Algorithm (TDGA).

1.7 Organization of the Thesis

The organization of the thesis is as follow: In chapter two, some of the early and recent task scheduling algorithms that were developed in the literature are shortly reviewed. Chapter three include the first proposed algorithm called Critical Path Genetic Algorithm (CPGA). In this chapter, we present briefly the basic idea of the algorithm with some definitions, followed by the details of the algorithm. Finally, our CPGA compared with Modified Critical Path (MCP) algorithm. In chapter four, we present the second proposed algorithm Task Duplication Genetic Algorithm (TDGA). The TDGA is based on task duplication technique in an attempt to reduce the communication delays and then minimize the overall execution time. Therefore, the performance of the genetic algorithm is increased. The performance of our TDGA is compared with a common heuristic task scheduling technique based on task duplication (DSH).

1.8 Conclusion

A survey of main types of the parallel architectures, as well as, the main conceptes of the task scheduling has been discussed. In the next chapter, a complete survey about main task scheduling algorithms will be discussed. One of these approaches is using Genetic Algorithms (GAs) which is considered in this research work. Also the principle of GAs will be included in the next chapter.