Comparative study of Genetic Crossover operators for Task Partitioning Problem

Dissertation submitted to the Jawaharlal Nehru University in partial fulfillment of the requirement for the award of the degree of

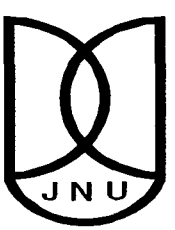
MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE AND TECHNOLOGY

By

Sunil Kumar Bharti



SCHOOL OF COMPUTER AND SYSTEMS SCIENCES

JAWAHARLAL NEHRU UNIVERSITY

NEW DELHI-110067, INDIA

2008

CERTIFICATE

This is to certify that this dissertation entitled **"Comparative study of Genetic Crossover Operators for Task Partitioning Problem"** submitted by **Mr. Sunil Kumar Bharti**, to the School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi, for the award of degree of **MASTER OF TECHNOLOGY**, is a bonafide work carried out under my supervision.

This study pertaining in this dissertation is original and has not been submitted, in part or in full, to any other University or Institution for the award of any other degree.

8/8/08

Prof. Parimala N.
(Dean of School of Computer and Systems Sciences, JNU)

29.7.08

Dr. D. P. Vidyarthi
(Supervisor)

DECLARATION

This is to certify that the dissertation titled **"Comparative Study of Genetic Crossover Operators for Task Partitioning Problem"**, which is being submitted to the **School of Computer & System Sciences, Jawaharlal Nehru University, New Delhi,** in partial fulfillment of the requirements for the award of **Master of Technology in Computer Science & Technology** is a bonafide work carried out by me. This research work has been carried out the under the supervision of **Dr. D. P. Vidyarthi**.

This research work is original and has not been submitted, in part or in full, to any other University or Institution for the award of any other degree.

S. K. Bharti

SUNIL KUMAR BHARTI
M.Tech, SC & SS, JNU,
New Delhi - 110067.

In Loving Memory

of my father and elder brother.....

ACKNOWLEDGEMENTS

First and foremost, with great pleasure and sense of obligation, I express my heartfelt gratitude to my supervisor, **Dr. D. P. Vidyarthi**, who gave me valuable guidance and support. I would also like to express my great thanks for his unlimited help and his support, for insightful discussion on the ideas of this dissertation. I am also grateful for his wisdom, understanding, and suggestions throughout the course of this study. Working under his guidance has always been a fruitful and unforgotten experience which is very much valuable gift to flourish my incoming life.

With a blend of gratitude, delight and gratification, I convey my indebtness to all those who have directly or indirectly contributed to the successfully completion of my dissertation.

I owe my heartfelt gratitude to **Prof. Parimala N.**, Dean, School of Computer & System Sciences, Jawaharlal Nehru University, New Delhi, for his kind and active cooperation during the course of study. I would also like to express my gratefulness to the entire faculty and staff of SC&SS for their cooperation during the course of study.

I am thankful to my parents, without whose blessing and support it would not possible to complete this work. I also extend my thanks to all of my classmate and my senior Mr. Arun and my room partner Mr. Deepak S. Nikarthil for their warmth, care and morale support.

S. K. Bharti

(Sunil Kumar Bharti)

ABSTRACT

Genetic algorithm (GA) is highly used for solving optimization problem for which no straight forward solution exists. GA is based on the evolutionary ideas of natural selection and genetics. It is an adaptive heuristic search algorithm. GAs mimics the principle of the "survival of the fittest" among individuals over successive generation towards solving a given problem.

Genetic algorithm consists of few operators e.g. selection, crossover and mutation. Of these crossover operator is used for mating and reproduction and therefore is very important operator. Three types of crossover operators, discussed in the literature, are Partial Matched Crossover (PMX) Ordered Crossover (OX) and Cycle Crossover (CX).

Parallel/distributed system allows the concurrent execution of tasks. Execution scenario in both Parallel and Distributed system are almost similar. The difference between these systems lies in sharing. A parallel system also called a multiprocessor system has a shared memory and clock whereas in a distributed systems memory and the clock is distributed. Though it is a collection of autonomous computer it gives the appearance of single system. The main advantages of theses systems are resource sharing, openness, higher throughput and shorter response time, etc.

Task scheduling in parallel/distributed system is comprised of two steps: partitioning the task into sub-tasks (modules) and allocating these sub-tasks onto different processing nodes of the system. Both these steps are an NP-hard problem. Task partitioning is done for grouping of the modules in a single entity. It is called Load. If the grouping is done in a proper manner, the load will be even and thus the allocation will also be stable.

In this dissertation, we have compared three crossover operators mentioned above on the task partitioning problem. The grouping of the modules of the task in done in such a manner that these groups of the task can be allocated in its entirety onto the processors of the machine. This grouping will be done in such a manner so that the cumulative

execution time of the modules of the group approaches to same for all the groups. The work tests the performance of the three crossover operators PMX, OX and CX for this problem. The programs written for the dissertation has been implemented in C language.

First chapter is an introductory chapter which discusses the genetic algorithm, operators used in GA, parallel and distributed system and models of distributed system.

Second chapter contains Genetic Algorithm in detail.

Third chapter discusses about the task partitioning problem of Parallel/distributed system.

Fourths chapter elaborates the operators used in GA specially three Crossover operators. As our study is based on these operators, we have emphasized on these operators.

Fifth chapter discusses the experiments performed, observations and the conclusion.

List of Figures and Table

TABLE OF CONTENTS

Chapter 1

Introduction

This dissertation work measures the efficiency of three genetic crossover operators over a problem of task partitioning for the tasks submitted to a Parallel/Distributed Computing System for its execution. The emphasis is on the crossover operators and the problem has been formulated around the modules grouping of the task. The imperative here is to discuss both the genetic algorithm and the parallel/distributed computing system.

1.1 Genetic Algorithm

Genetic algorithms (GAs) are evolutionary algorithm derived from the Darwin's theory of natural genetics. It is based on the principle of "Survival of the Fittest". A best solution is derived from the number of solutions in GA. There are many search techniques available for problem solving [3, 4, 5, 16, 17, 18]; these search techniques are:

- Calculus Based Techniques (Fibonacci, Sorting)
- Enumerative Techniques (DFS, BFS, etc)
- Guided Random Search Techniques (Hill Climbing, Simulated Annealing, Evolutionary Algorithms, etc.)

Evolutionary Computation has two components: Genetic Programming and Genetic Algorithms. Genetic algorithm generates population of potential solution and explores the best solution of the problem. The father of the GA is John Holland, who along with his student DeJong introduced it in the year by 1975 at the University of Michigan. He was influenced by the natural system and in his effort to propose the GA he thought of mimicking the natural system. He thought to involve computing in natural system and derived that as the natural system evolves any computation may also evolve. This was the basis for the development of the GA. Most of the terminology in GA has been borrowed from genetic engineering and also it uses the biological concept of "Natural Selection" and "Genetic Inheritance". For most of the optimization problems, where a little

knowledge about problem solving approach is available, GA is beneficial. It is widely used for the NP-class of problems (both NP-Complete and NP-Hard problems).

The GA, as a problem solving tool, is different in comparison to some other search techniques described as follows [16].

- GA is applied on the coding of the parameter sets not only on the parameters.
- Populations consisting of solutions are used for searching in GA, so it is a multipoint search in parallel.
- The original information is used in GA; no other secondary information, knowledge or consequential information is needed.
- GA uses probabilistic transition instead of deterministic transition.

GA has a chromosome (genotype) that consists of genes. Genes are the fundamental instructions for building an organism. The complete chromosome derives the solution of the problem and a particular chromosome (phenotype) becomes the target solution of the problem. The pseudo-code for the GA has been given below.

Genetic Algorithm ( )

1. *Generate the population of solutions randomly.* *//using any approach*
2. *Evaluate the fitness value of each individual against the fitness function.*
3. *Repeat while termination condition does not hold*

{ *Randomly pick two parents from the population*
Perform the crossover over the parents to produce offspring // based on the probability of applying crossover
Mutate each offspring // probability of applying mutation is often very less
Evaluate the fitness of each new individual
}

1.1.1 Search Space

A big search space is created in terms of the feasible solution of population. In search space each individual is represented by finite length vectors. Though these can be any alphabets but normally GA uses binary number alphabet {0,1}. An individual of population (each chromosome) generate a fitness value of each solution. The most favorable fitness score is required. GA maintains a population of 'n' chromosomes and its fitness score. The parents use reproductive map to produce offspring. In reproduction the parents with high fitness score are given more opportunity than others and they arrive to replace old one in population which is least fit among reproduced solution. [5, 10, 18]

It is expected that every new generation after reproduction gives better solution than previous solutions and successive generations improves the solution which saturates after certain generations. It is said to be the convergence of the solution.

1.1.2 Genetic Operators

Genetic algorithms have the following components

Initial Population: Consisting of potential solution to the problem.
Selection: the mechanism of selecting the parent for the reproduction.
Fitness Function: Parents are selected according to their fitness value based on the fitness function.
Crossover: mating between individuals, so that the parents may produce children.
Mutation: the random alteration in genetic population.
Reproduction: Generation of new offspring from old one.

1.1.3 Effect of Operators

The selection operator selects the better population from the old population and combined with crossover operator it produces better but sub-optimal solution. The combination of selection and crossover operator creates parallel hill climbing algorithms to solve the problem. The number of crossover mechanism has been suggested by the researchers. These have its own advantages and disadvantages depending on the problem

to be solved. The current work is an attempt to evaluate the various crossover operators for the task partitioning problem of parallel/distributed system.

1.2 Parallel System

The parallel systems have been developed with the intention of using multiprocessor for solving a single problem. Thus the objective is to explore the parallelism present in the job, design the modules accordingly and then to utilize the number of processors available in parallel system to execute the job. Parallel systems can theoretically multiply the computational power of a single processor by a large factor. To achieve maximum efficiency of these large systems the workload has to be distributed equally throughout the network. The uniform distribution of the load among the various processing nodes maximizes resource utilization and enhances the total throughput of the system. Ideally, the speed up expected from a parallel system consisting of "n" processor system is n. Though, due to practical implementation, this is never achieved.

Parallel system is single computer with multiple processors which are used for executing more than one job/subjobs/instruction simultaneously. Execution of more than one process on parallel system is known as parallel processing. Parallel system is also referred as multiprocessor system. In parallel processing task is divided into number of modules (sub tasks) and is executed on the processors of the system. Some of the advantages of parallel processing are higher throughput, computation power and better price/performance ratio than a single-processor system. The parallel systems are classified into two main categories; General purpose and Special purpose computer systems. General purpose computer system is Pipeline computers, Asynchronous multiprocessors and Data flow computers. Special purpose computers are Synchronous multiprocessors (Array Processors), Systolic Array and Neural Network.

1.3 Distributed System

A Distributed System is an environment that provides a single system image with the distribution of the memory and the clock amongst the computing nodes of the DCS. To

explore distributed computing system, a task is divided in to the smaller components, called modules, so that they can execute concurrently and hence can utilize the system for parallel execution. It is similar to the parallel system with the difference that it is decentralized i.e. there is no central components. In distributed computing system, all processing nodes have different clock, and their own memory. Distributed computing system are scalable meaning thereby that it can be scaled up or down as per the requirement of the application to be executed on the DCS. The executable components, located on the nodes of the DCS, communicate and coordinate their actions by passing messages [1]. The advantages of the distributed system are information and resource sharing, higher reliability, higher throughput, shorter response time, extensibility etc. The performance of the distributed computing system is obviously better than the uni-processor system and generally measured in terms of the performance of execution of jobs. Due to rapid advances in VLSI technology and the increase in the development of distributed computing system, software for these systems became complicated due to the additional complexity. It is because of the existence of concurrency, communication, synchronization and non-determinism in its execution.

A number of methods for developing distributed computing system software have been presented in the literature [1-4, 6]. These methods are classified into two categories. The first category involves a series of top-down decomposition and partitions of the distributed computing system software in order to derive a set of distributed software component [1-4]. However none of them provide efficient analysis technique for verification of generated distributed computing system software. The second category is bottom up oriented [6]. They may produce the whole distributed computing system software structure in a synthetic way which, under some limited synthesizing rules, guarantees a certain degree of correctness in the system behavior and properties.

No matter which of the approach is used, the entire distributed computing system software development process is still similar to the central software development system. It may consist of successive phases: requirement analysis, design, implementation, testing and validation. This is not a straight forward process and feedback path do exist among

successive phases so that the previous steps can be repeated to eliminate effects or errors found in the latter phase. Most of these defects can be traced back to the requirement and design phase. As the size of the distributed computing system continue to grow, the relationship among concurrency, contention, synchronization and communication among all processes become more and more complicated.

Based on the memory sharing, distributed computing system is classified of two types.

1.3.1 Distributed Memory

In Distributed Memory System, each processing node has own memory which is not shared by any other processing node. These nodes may interact with each other by message passing i.e. if any node requires any data stored on the memory of other node then the first node just sends a message to the second node about the data that it requires. In reply, the second node sends him the required information. The distributed memory organization is elaborated in Fig. 1.1. The performance of the distributed memory system depends on the communication link through which the message is transmitted. The distributed memory systems are becoming the architecture of choice for many supercomputer applications. The reason is that they are inherently scalable and provide reliability, inexpensive computing cycles compared with the traditional uni-processor or the shared memory multiprocessor supercomputer.

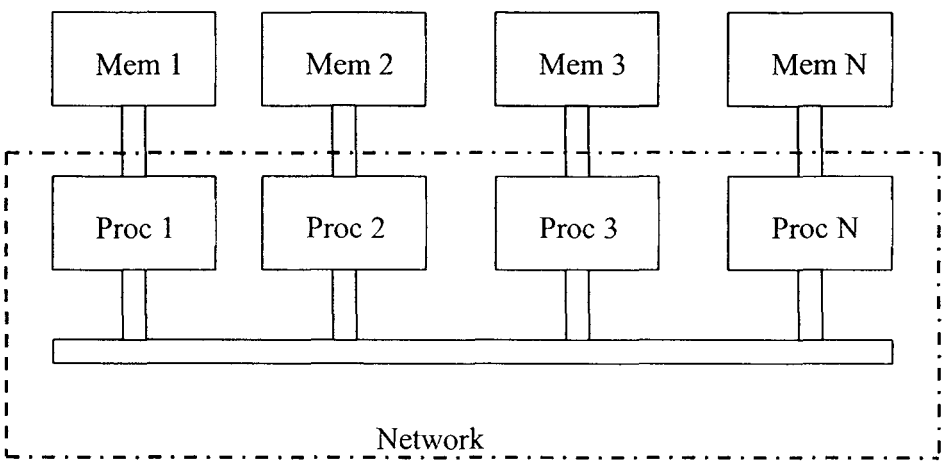


Fig. 1.1 Distributed Memory Systems

High speed I/O for the distributed memory machines is difficult because the entire system, including the architecture, software, programming models and applications have been designed to work in a distributed fashion and any form of serialization is avoided, if possible. The connection to the network, however, is inherently centralized and this creates problems in the following areas.

1. Distribution of work over large number of relatively slow processing nodes is source of power of distributed memory machines but communication protocol processing does not parallelize well.
2. The application will often have to manage multiple connections and this involves scheduling resources in both distributed memory system and on the network interface. This is a complicated task and there is a conflict between using general purpose solutions on one hand and providing mechanisms that are for the specific application on the other.
3. The communication software has to perform scatter and gather operations to collect or distribute the data that makes up the data stream because data that is sent or received over the network is typically distributed over the private memories of the nodes.

1.3.2 Distributed System Models

The basic models of distributed system are [12, 13]:

- Minicomputer Model- In minicomputer model of distributed system number of minicomputers are connected to a network, each with some terminals.
- Workstation Model- Many work station are connected to network and make distributed system to useful when user uses remote workstation.
- Workstation server Model- Its is very advantageous due to sharing of several servers like file server, printer server etc.,
- Processor Pool Model- Pool of processors connected to network.
- Hybrid Model

1.4 Organization of the Thesis

Second chapter of our dissertation describes Genetic Algorithm in detail.

Third chapter gives the idea of Task Partitioning Problem, Task Preprocessing and the proposed work for our dissertation.

Fourth chapter gives basic knowledge of three crossover operator PMX, OX and CX.

Fifth chapter contains the experimental results and the conclusion.

Chapter 2

Genetic Algorithms

Scientists looked in other directions after their disillusion with classical and neo-classical attempts at modeling intelligence. Two main fields derived were connectionism (neural networking, parallel processing) and evolutionary computing. AI systems are mostly static. Most of the AI problems can usually solve only one given specific problem, since their architecture was designed for that specific problem. Thus, if the problem were to be changed, these systems could hardly adopt them. Even if it adopts so, it will be less efficient. Genetic algorithms (GA) came as the natural solution to such problems. [4, 5]

2.1 Biological Inspiration of Genetic Algorithms

Genetic algorithms are search methods mimicking the processes found in natural biological evolution. Evolution is the change in the inherited traits of a population from one generation to the next. In nature organisms such as animals or plants produce a number of offspring. These offspring are almost, but not entirely, like themselves. This variation may be due to sexual reproduction (offspring have some characteristics from each parent) or due to mutation (random changes). Some or more of these offspring may survive to produce offspring of their own. The better offspring are likely to survive and over the generations these become better and better. Genetic algorithms (GA) use the same process for evolution. Mostly Genetic algorithms (GA's) are used to solve optimization problems. It is based on the biological concept of "Natural Selection" and "Genetic Inheritance" which is given by Darwin in 1859. The word "Survival of the Fittest" best define this algorithm as given by Darwin. [3, 4, 5, 16, 17, 18]

2.1.1 Vocabulary of GA

Most of the vocabulary of GA has been derived from genetic engineering. Gene is a symbol or bit in a string, the smallest unit of information and fundamental building blocks of chromosome. Each gene in a chromosome represents each variable to be optimized. A Chromosome is a string of genes, and an individual representing a candidate solution of the optimization problem. Genotype is entire combination of genes. The complete chromosome derives the solution of problem with the help of a particular chromosome (phenotype). Phenotype is an organism made by genotype which contains all the information and take part to generate fitness function. Population is the total number of chromosomes available for the test [4, 5].

2.2 Genetic Algorithms (GA)

GA is developed by John Holland (1975) and his student DeJong (1975) at the University of Michigan, to understand the adaptive process of natural systems and to design artificial systems software that retains the robustness of natural systems. These algorithms search or operate on a given population of potential solutions to find those that approach close to the solution as per the given specification or criteria (fitness function).GA is beneficial where we have little knowledge about problem solving approach, mostly for NP-Hard problems (e.g. TSP).[5, 16] GA is a blend of exploitation and exploration.

In simple genetic algorithms, randomly generated solution strings are formed into a population. These strings are generated using any method e.g. greedy. These strings represent a variety of solutions for a given problem. These solutions are typically encoded in some manner on the strings in some defined format. The strings are decoded and then evaluated according to a fitness function. Individuals are then selected to undergo reproduction to produce offspring (individuals for the next generation). Selection is done by assigning a higher probability of selection to those parents who are deemed to have higher fitness according to the fitness function. The process of reproduction consists of two operations. Firstly, selected solution strings are recombined using a recombination

operator, where two or more parent solution strings provide elements of their string to generate a new solution. Secondly mutation is applied to the offspring. The mutation operator affects only a small amount of the genetic material in the offspring solution strings. Following the generation of a complete population of offspring solution, the offspring population replaces the parent population.

The flowchart for the simple GA is given in fig.2.1.

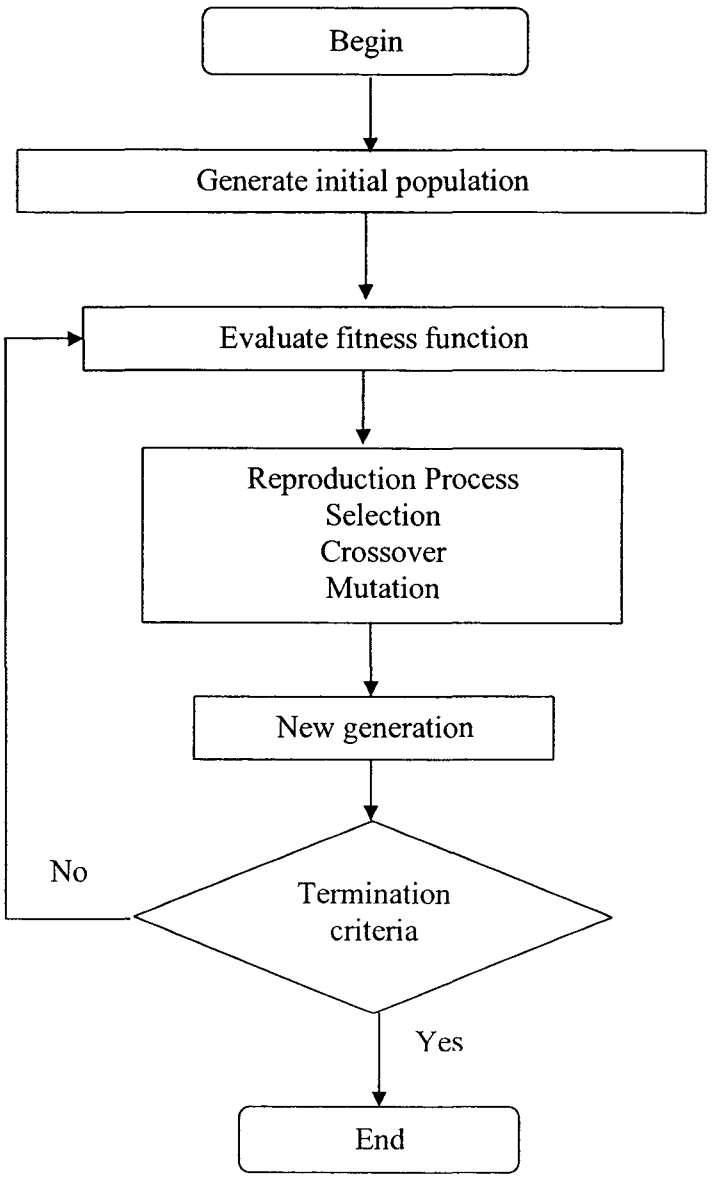


Fig. 2.1 Genetic algorithm

The pseudo-code for simple GA is as follows:

```
Simple GA ()
{
        Initialize population;
        Do {
                Perform Crossover and Mutation;
                Evaluate population;
                Reproduction;
            }
        Unless the result converges
}
```

2.2.1 Search Space

If we are solving a problem, and are not sure about the best solution then will look for the solution, which is the best among others. The space of all feasible solutions is called search space (also state space). Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. GA maintains population of 'n' chromosomes with there fitness score. The parents are able to select their mate on the basis of their fitness score or any other things. In reproduction highly fit fitness score solutions are given more opportunity and they arrive to replace old one in population which is least fit among reproduced solution. [5, 10,16] Thus search space in the solution space comprised of population.

2.2.2 Genetic operators

There are three fundamental genetic operators selection, crossover and mutation. [5, 18]

- **Selection:** Selection operation selects individuals for survival based on the probabilistic function of fitness.

- **Crossover:** Crossover operation combines two individuals to create new individuals for possible inclusion in next generation.
- **Mutation:** Mutation operation does random changes in genetic population to produce offspring.

Selection

Selection is a procedure of picking best individuals from a pool of population based on their fitness function value to produce offspring. It is an important step in GA. Many different approaches for selection schemes have been proposed. One of them is Fitness-proportionate selection scheme with roulette-wheel method and mostly used in GA. In roulette-wheel method the chance of an individual's being selected is proportional to the amount by which its fitness is greater or less than its competitors' fitness. Each individual gets a slice of the wheel. More fit ones get larger slices than less fit ones. The wheel is then spun and an individual "owns" the section on which it lands. Suppose that there are four populations A, B, C and D having their fitness value 4, 6, 2 and 8 respectively. The roulette wheel for selection of these populations is as given in Fig.2.2. [3, 17] Obliviously D has the more probability to be chosen than A, B and C and this has wider allocation in wheel.

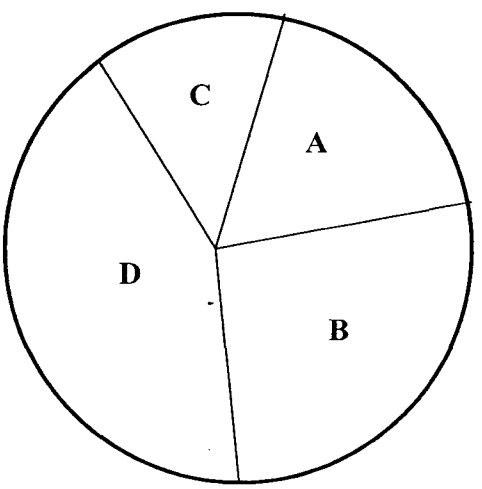


Fig2.2 Roulette Wheel representation of Selection Operation

Crossover

Crossover is a concept of genetics and has analogy of sexual reproduction. Crossover chooses two individuals and swap segments of their code (bit- value). It produces offspring that are combinations of their parents. Common forms of crossover are single-point crossover, two point crossover (also called multi point crossover) and uniform crossover. In **single-point crossover** one common site in the chromosomes is chosen randomly. Chromosomes of the parents are cut at that point and the resulting sub-chromosomes are swapped. [5, 10, 16, 18]

Parent 1: 11000101 010**11000 01101010**
Parent 2: 00100100 101**11001 01111000**

Offspring 1: 11000101 010**11001 01111000**
Offspring 2: 00100100 101**11000 01101010**

Fig2.3 Single Point Crossover Operator

In **two-point crossover** two bits are selected and the sub-string between the bits is swapped. In **Uniform crossover** each gene of the offspring is selected randomly from the corresponding genes of the parents. Single-point and two-point crossover produce two offspring, whilst uniform crossover produces only one.

Mutation

Mutation is asexual reproduction and it introduces randomness into population. This operator randomly flips or alters one or more bit values at randomly selected locations in a chromosome. Mutation generates low ranked children, which are eliminated in reproduction process. Sometimes however, the mutation may introduce a better individual with a new property into. This prevents process of reproduction from degeneration.

2.2.3 Fitness function

Fitness function is evaluation function that determines what solutions are better than others. The purpose of fitness function is parent selection and a measure for convergence. Fitness function is the most critical part of a GA next to coding. In any optimization problem the fitness is replaced by the objective function. The fitness is to be designed properly for the GA to be effective.

2.2.4 Advantages of GA

For any problem when the search space is very big the straight forward algorithms becomes more exhaustive. GA comes into effect to solve such problems. The solution obtained using GA are close to the optimal solution. The other advantages are:

- Only primitive procedures like "cut" and "exchange" of strings are used for generating new genes from old, it is easy to handle large problems simply by using long strings.
- only values of the objective function for optimization are used to select genes, this algorithm can be robustly applied to problems with any kinds of objective functions, such as nonlinear, indifferentiable, or step functions.
- The genetic operations are performed at random and also include mutation; it is possible to avoid being trapped by local-optima.

Chapter 3

Task Partitioning Problem

Genetic algorithms are quite often used successfully to solve the NP-complete, NP-hard problems of combinatorial optimization. Task partitioning problem of Parallel/Distributed Systems is also falls in the same category so GA has been used to solve such problems. Implementation of any algorithm in parallel/distributed environment, if cleverly done, accelerates calculations considerably and makes it possible to solve large instances of the problem relatively fast. Given a task to be executed on parallel/distributed system requires partitioning of the task into number of modules (sub-tasks). It is done so, as not the task as a whole, rather the modules of the task becomes the execution entity. This is called the task partitioning and is the pre-processing step of the task allocation problem. [6, 7, 28]

3.1 Task Preprocessing

Task preprocessing is a method to analyze the task for its possible mapping to the processing nodes of the parallel/distributed system and specify an order to follow to execute these tasks in a synchronized manner. Preparation of the raw data for taking further decision is known as task preprocessing. There are many approaches to preprocess data. Task preprocessing is required to extract the information before allocating a task on parallel/distributed system. The extracted information helps to take further decision for allocation. Preprocessing can be performed at compile time or/and at run time. [2, 23]

In compile time preprocessing, the analysis and information extraction is done at compile time. The information that is supplied by the program is extracted at compile time. In runtime preprocessing approach two steps are involved. First to analyze the data, optimize it, and generate information for the second step. In second step actual computation is performed. Further, there are four phases in preprocessing. It is sensing, recognition, decision, and acting. [8]

3.2 Task Partitioning

Decomposition, mapping and tuning is three required steps to program execution in parallel/distributed computing system. Partitioning or decomposition is the first step towards designing a parallel program. This requires breaking the problem into discrete chunks of work that can be distributed on multiple processors. There are several ways to decompose/partition a computational task into parallel tasks. [6, 7]

3.2.1 Perfect Decomposition

In perfect decomposition, the task is divided into number of sub-tasks (modules) and these modules either do not require a communication or a little communication among each other. This is called perfect decomposition. Least amount of effort is required for this type of decomposition. Perfect decomposition can not be applied when dependency between processes exist. The π composition is a good example of a perfect decomposition in which only the partial sums need to be communicated. [7]

3.2.2 Domain decomposition

The main feature of domain decomposition is regularity of data structure. In this type of decomposition data associated with a problem is decomposed. Each parallel task then works on a portion of data. Normally the domain decomposition is applied when the domain and data contingent are non-regular. Static data structure, for example, matrix factorization for solving a large finite difference problem on a system with a regular network topology. Dynamic data structure tied to a single entity, for example, in a many body problem, subsets of bodies can be distributed to different nodes. [7, 28]

Three major steps are specified below to decompose the domain of a given applications.

- At various nodes sub domain of data are distributed.
- Restrict the computation so that each node updates its own sub domains of data.
- Put the communication in node programs.

3.2.3 Control decomposition

When the domain of data is not suitable for domain decomposition and irregular behavior of data structure and domain is found, then control decomposition is used. Control decomposition technique is used for artificial intelligence and symbolic processing problem. Computation and data structure is the key for the control decomposition. Due to dependency on data structure and computation, interface between different functional modules is needed. Functional decomposition and Manager-worker approach are two famous strategies for control decomposition. [7, 28]

In **functional decomposition** approach, the focus is on the computation rather than on the data manipulated by the computation. The problem is decomposed according to the work (computation) that must be done. Each task then performs a portion of the overall work. This strategy for control decomposition works well on those problems which can be split into different task. This method is used in signal processing, climate modeling and eco-system modeling. [28]

In **Manager-worker approach**, divide and conquer method is applied. In this approach the task partitioning problem is divided into different size subtask. Among these subtasks one of the subtasks plays the role of a manager and others as workers. The manager is responsible for dispatching the modules to the workers. [28]

The current work uses the functional decomposition approach for a given task graph in the task partitioning problem. Regrouping of modules will be done with genetic crossover operators to minimize the differences among the group execution time sums. [28]

3.1 The Model

In the proposed problem, we consider the n modules of a task; each represented by a distinct integer number from the range {1, 2, ---- n}. These modules are partitioned into k groups of equal or unequal length. Length indicates the number of modules in a group. The idea, borrowed from the traveling salesman problem (TSP), is applied. Three types of genetic crossover operator PMX (Partially Matched Crossover), OX (Order Crossover)

and CX (Cycle Crossover) are applied. PMX is most suited crossover for this type of problem although order crossover (OX) and cycle crossover (CX) are also applied.

The Evaluation function is defined as follows [28].

Let us assume that there are 'm' modules in a group and e_j is the execution time of j^{th} modules. S_i is the sum of the execution time of this i^{th} group then

$$S_i = \sum_{j=1}^{m} e_j$$

Difference in sum of the execution times of i^{th} and j^{th} group is

$D_{ij} = S_i - S_j$

Where i= 1, 2, ----------k
And j = 1, 2, 3--------k - j

These differences should approach to zero. The reason is that when the D_{ij} will be approaching to zero all the groups will be of equal load in terms of time. The there crossover operators are applied until the evaluation function reaches almost to zero. The problem is given in the form of a task graph which consists of modules. Task graph is a kind of enumeration which represents control flow of a program. In the task graph, the flow of control is represented by directed edges.

For example in figure 3.1 a task graph is given consisting of nine modules. If we form three groups of the given task graph, it will consists of {1, 2, 5, 8}, {3, 6, 9} and {4, 7}. If we assume the execution times of these modules to be uniform and unit then the sum of the execution times of the groups will be 4, 3 and 2 respectively. Though, in practice, these times will not be uniform and that is where the possibility of applying GA for this partitioning problem is.

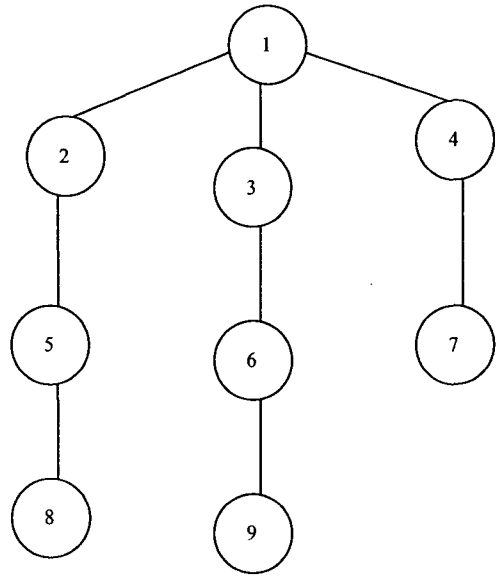


Fig. 3.1 Task Graph

The initial population may be formed randomly for the task graph. Then the GA will be applied for making D_{ij} to approach to zero. There crossover operators, mentioned in the next chapter, will be used individually on same problem and the study will be made on the three crossover operators.

We will perform the experiment with various sets of data (i.e. task graph) to reach to a conclusion.

Chapter 4

Comparative Study of the Crossover operators

My experiments compared the following permutation crossover operators: Partially Matched Crossover (PMX) [16], Order Crossover (OX) [16], and Cycle Crossover (CX) [16] over the task partitioning problem of Distributed system. We have tested effectiveness of these operators on task partitioning.
Below are these operators in general.

4.1 PMX- Partially Mapped Crossover

Goldberg and Lingle in 1985 proposed the partially mapped crossover (PMX) operator. [16, 18] This operator is designed to work with a path representation and is devised to prevent repetition of values in the offspring. This is done by mapping a portion of one parent on to the portion of the other parent. Following this any remaining information is exchanged.

PMX first selects two random crossover points in the parent; these crossover points define the matching section. The corresponding crossover points are then reproduced on the second parent. For example, in fig. 4.1 depicting PMX, crossover points are selected between the third and fourth, and between the sixth and seventh elements in the parents. The first step is to define the mappings for PMX, in this case 5 ↔ 2, 6 ↔ 4 and 7 ↔ 8. Following this the second step is to exchange substrings between the two crossover points in each parent to opposite offspring (i.e. parent 1 to offspring 2 and parent 2 to offspring 1).

P1 = (0 8 4 | 5 6 7 | 1 2 3 9)
P2 = (6 7 1 | 2 4 8 | 3 5 9 0)

O1 = (x x x | 2 4 8 | x x x x)
O2 = (x x x | 5 6 7 | x x x x)

Fig - 4.1 PMX example-Crossover points

To complete the offspring, step 3 requires that the offspring be filled. The ith offspring (where i = 1,2) being filled with the remaining elements from parent i. In this case the first element in offspring 1 will be 0. The second would be 8; however this is already present in the offspring string and as a result will be replaced according to the mapping defined above, 7 ↔ 8 (see Fig - 4.2).

P1 = (0 8 4 | 5 6 7 | 1 2 3 9)
P2 = (6 7 1 | 2 4 8 | 3 5 9 0)

O1 = (0 **8 4** | 2 4 8 | 1 **2** 3 9)

O2 = (**6 7** 1 | 5 6 7 | 3 **5** 9 0)

Fig – 4.2 PMX- filling offspring

This process of inserting an element, checking the validity of the string and correcting offending elements through the mappings that were defined is repeated until the offspring are completely filled. This produces offspring that have no replicated elements (see Fig-4.3). PMX is a very popular crossover technique for problems that have validity constraints. It has been applied extensively to the traveling salesman problem (TSP).

O1 = (0 7 6 2 4 8 1 5 3 9)

O2 = (4 8 1 5 6 7 3 2 9 0)

Fig – 4.3 Completed offspring produced by PMX

4.1.1 PMX Algorithm

The algorithm for PMX operation is as follows.

1. Start with initial population.
2. Select two crossover points randomly.
3. Replace bits with blank spaces which do not reside between crossover points.
4. Swap the bits of offspring between crossover points.
5. Fill the blank spaces created in step 3 with the help of initial population offspring (parent) with no conflict.
6. Fill remaining blank spaces with exchange operation.
7. Find new population.

4.2 OX – Order Crossover

The Order Crossover was devised by Davis in 1985. [16] This operator starts in a similar fashion as PMX. Start with previous example taken to illustrate PMX. Initially two random crossover points are selected. In the above case these are after position 3 and 6. The substrings are selected, and then copied to the offspring as in the example (Fig. 4.4)

P1 = (0 8 4 | 5 6 7 | 1 2 3 9)

P2 = (6 7 1 | 2 4 8 | 3 5 9 0)

O1 = (x x x | 2 4 8 | x x x x)

O2 = (x x x | 5 6 7 | x x x x)

Fig – 4.4 OX crossover points

The process starts at the position just after the last crossover point (in this case position 7).Order crossover uses a sliding motion to fill the holes left by swap operation. The remaining positions in offspring are determined as, for offspring 2 remaining order is determined by parent 2. Nonduplicative bits are copied from parent 2 to the offspring 2 beginning at the position following the second cross point. Both the parent 2 and the offspring 2 are traversed circularly from that point. A copy of the parent's next

nonduplicative bit is placed in the next available child position [18].The string after second cross point in parent 2 would provide 3 without any confliction and copy 3 to offspring 2 at position 7. The next bit in order for parent 2 is 5 but it is already present in offspring 2 so this bit is skipped and the bit in the next position is taken. The bit 9, 0 do not have confliction so copied these in position 8, 9 respectively in offspring 2. Traversing parent 2 circularly, the bits (6, 7, 1, 2, 4, 8), skipping 6, 7 because these are present in offspring 2 at position 5, 6 so we left with bits (1, 2, 4, 8), copied these to offspring 2 at position 10, 1, 2, 3 respectively. Offspring 1 is completed in a similar way based on the order of bits in parent 1.

O1 = (5 6 7 | 2 4 8 | 1 3 9 0)

O2 = (2 4 8 | 5 6 7 | 3 9 0 1)

Fig – 4.5 Completed offspring produced by OX

Where PMX preserves the absolute position of a bit within strings, OX preserves the order of bits in the permutation.

4.2.1 OX Algorithm

The OX algorithm can be listed as follows.

1. Take parent P1 and P2 as initial population.
2. Select two crossover points randomly.
3. Swap the substring between crossover points and leave other positions blank.
4. Start with last crossover point of parent P2 and copy nonduplicative bits in offspring O2 in same order as P2.
5. If confliction (bits already present in offspring after swapping)
   Then discard that bits and traverse next bit circularly in P2 and copy non conflicting bit next position in offspring preserving the order.
6. Do step 4 and 5 until all blank spaces filled in offspring.
7. For second offspring start with last crossover point of P1 and do step 4 to 6.

4.3 CX - Cycle Crossover

The cycle crossover operator CX was developed by Oliver and Holland in 1987. [16] It is designed so that two conditions are satisfied. First every element position in an offspring retains a value that has a corresponding position in a parent; and second the offspring must be a permutation. With these two conditions, CX cycles through both the parents selecting elements to be placed in the offspring. Consider the following example:

P1 = (0 8 4 5 6 7 1 2 3 9)
P2 = (6 7 1 2 4 8 3 5 9 0)

From the conditions outlined above, the first element in the offspring must either be 0 or 6 (the left most element in parent 1 and 2) selected at random. Let us select 0 for this example. We start from the first position of offspring 1.

O1 = (0 x x x x x x x x x)
O2 = (x x x x x x x x x x)

Then offspring 2 may only have a 6 in the first position, because we do not want new values to be introduced there.

O1 = (0 x x x x x x x x x)
O2 = (6 x x x x x x x x x)

Since 6 is already fixed for offspring 2 now, we have to keep it in the same position for offspring 1 in order to guarantee that no new position for the 6 are introduced. We have to keep the 4 in the fifth position of offspring 2 automatically for the same reason.

O1 = (0 x x x 6 x x x x x)
O2 = (6 x x x 4 x x x x x)

This algorithm is continued, and results in the selection of bits 4, 1, 3 and 9. The cycle finally terminates in this example when bit 9 is selected from parent 1 and the corresponding parent 2 bit is 0, which is already present in the offspring i.e. we have completed a cycle.

O1 = (0 x 4 x 6 x 1 x 3 9)
O2 = (6 x 1 x 4 x 3 x 9 0)

This completes our first cycle. For the second cycle, we can start with a value from parent 2 and insert it into offspring 1.

O1 = (0 x 4 x 6 x 1 x 3 9)
O2 = (6 x 1 x 4 x 3 x 9 0)

After applying same algorithm to detect a cycle, we end up with the following.

O1 = (0 7 4 x 6 8 1 x 3 9)
O2 = (6 8 1 x 4 7 3 x 9 0)

This is our second cycle. For next cycle, we can start with parent 1. Starting with 5 and applying same algorithm we end up with the following.

O1 = (0 7 4 5 6 8 1 2 3 9)
O2 = (6 8 1 2 4 7 3 5 9 0)

This is our third cycle and no positions are left, so we end up with our final offspring's.

In the previous example only three cycles are required to complete the offspring. The CX operator, like the OX operator, retains the relative ordering information of the alleles. Of the three operators, CX and OX were found to be least disruptive from the point of view of relative ordering. [16]

4.3.1 CX Algorithm

1. Take parent P1 and P2 as initial population.
2. Make a cycle of bits from P1 in the following way.
   (a) Start with the first bit of P1.
   (b) Look at the bit at the same position in P2.
   (c) Go to the position with the same bit in P1.
   (d) Add this allele to the cycle.
   (e) Repeat step (b) through (d) until you arrive at the first bit of P1.
3. Put the bits of the cycle in the first child on the positions they have in the first parent.
4. Take next cycle from second parent.
5. Repeat the process (2) to (4) unless all the bits are filled in both the strings.

Chapter 5

Experimental Observation & Conclusion

The task partitioning problem in Parallel/Distributed System is very important problem for the scheduling aspect of these systems. We have evaluated the three genetic crossover operators, as mentioned in the chapter 4, with respect to the task partitioning problem. We have compared the performance of these operators over the task partitioning problem.

We have taken the varying parent size population and applied three crossover operators, Cycle Crossover (CX), Partially Matched Crossover (PMX) and Order Crossover (OX) one by one to compare the execution time they take for producing the offspring.

Let us consider a population of two P1 and P2 as given below. It consists of 10 modules as numbered, i.e. it is a population of size 10.

P1 = (0 8 4 | 5 6 7 | 1 2 3 9)
P2 = (6 7 1 | 2 4 8 | 3 5 9 0)

The vertical line " | "indicates the initial grouping of the modules. We will apply the genetic algorithm after this. The three crossover operators of GA, used in the study to explain the research, give diverse explanation with respect to the size of the population of the parents. They offer different characteristic for the efficiency of the GA. The experiment chosen to show my observation is using the three different operators to understand the difference in the execution time in which they produce their results. We first applied PMX operator to find the grouping. We also calculated the time taken when using PMX. Thereafter, we applied OX operator for the grouping of the modules. We then applied CX operator. We obtained the time taken in each case.

We designed the experiment with varying size of the population. To start with we took the minimum number of parent size ten and increased the parent size in each successive experiment. We tested against all the operators. The execution times of the modules have

been generated randomly and are in the range of depending on the number of modules in that task. Note that in all the experiment, our fitness function is the same which equalizes the load between the different groups of the modules.

As we get our first set of results, we would increase our parent size to forty, the results under this observation differs from the earlier observation. The CX under this parameter gives the execution time of one hundred fifty while PMX takes two hundred twenty two and OX shorten to one hundred and fifty nine. This shows that there is no linearity in the results of the three operators as they wary with the parent size.

In another observation about while increasing the parent size to eighty the results show much different than the earlier observations. The CX here gives much different count reducing down to one hundred sixteen which actually less than result under the parent size of forty. Even with the operator PMX the execution time with eighty has slightly increase by eight compared to the parent size of forty, taking only two hundred thirty one to complete the execution. While the operator OX this time increases by seven rather than decreasing in the execution time by taking one hundred sixty six. This proves that there is no fixed time for execution according to the operators; they vary according to the size of parent and the offspring.

In further experiment observation of the execution time for the offspring under the three different operators under different conditions, with increase in the parent size the operator gives different calculations of execution time. They don't actually increase their execution time but it varies with the size in many cases they increase with a small increase but in some cases they decrease in the execution time as given in the table. Thus we can say that there is no fixed execution time for the operators considering the parent size. All these observations are shown in the table 1.

Table 1 Three Crossover operators

Parent Size	Time taken in msec.		
	CX	PMX	OX
10	86	152	194
40	150	222	159
80	116	231	166
150	156	239	196
200	134	179	139
500	112	131	141
600	131	162	159
800	159	181	191
1000	121	140	149

The graph for the three operators has been plotted as shown in Fig. 5.1. On the X-axis, we represent the size of the parent and on the y-axis the execution time taken by the program using different operators have been shown.

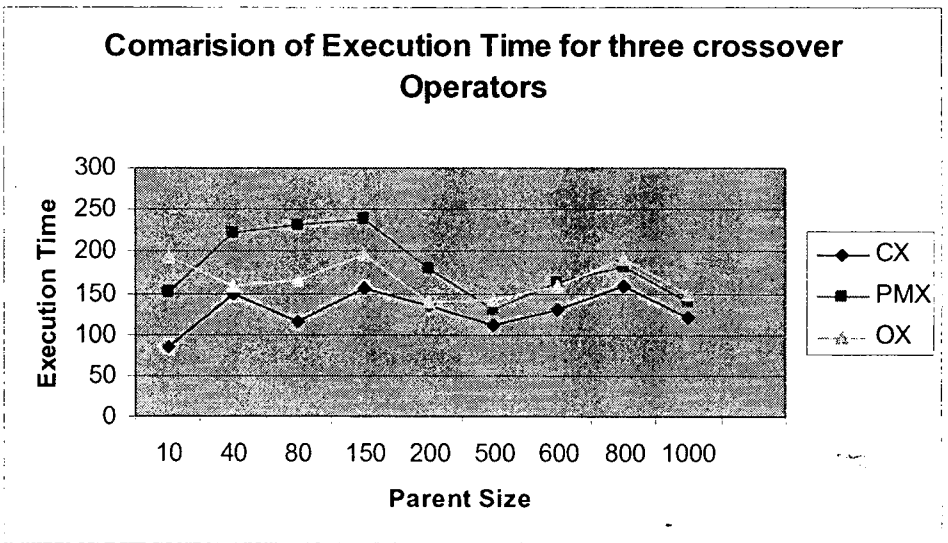


Fig. 5.1 Performance of the crossover operators

5.1 Observations

The obvious from the table and the graph is that OX operator performs better than the PMX operator for the smaller population sizes. For the bigger population size, both the OX and PMX performs neck to neck.

CX operator outperforms both the OX and PMX operator. It is evident from the graph that at no point of time the curve for the CX operator goes above the OX or the PMX. Thus a conclusion can be derived that out of three crossover operators, the performance of CX operator is best.

Graph in Fig. 5.1 represents observation for all the three PMX, OX and CX operators. X-axis represent parent size i.e. the number of modules in the task and Y-axis represent the time taken by the algorithm using PMX, OX and CX operators respectively. We conducted the experiment initially for less numbers of modules and the performance was same for PMX and OX. Increasing the number of modules, OX started performing better than PMX. But both became same when it reached to the number of modules in the range of 300 and above. CX was much better than both PMX and OX since beginning.

It should be kept in mind that this performance is for the task partitioning problem in Parallel/Distributed System. The performance may vary for any other problem which is to be solved using GA. However, the conclusion can be drawn that CX performs better than other two crossover operators.

5.2 Conclusion

The work conducted towards the dissertation comprised of partitioning the task into modules and then grouping these modules. The groupings of these modules are done so that load (cumulative modules) is uniform. This is done by minimizing the difference between the total times taken by each group. It is obvious that the result is better when the number of module in the task is very large. The performance of PMX operator and

OX operator are almost same when the number of modules in the task is very large. However, OX operator performs better for the modules in the range of 30 to 300. Both provide a better task sequence for execution in multiprocessor and multi-computer system. Applying CX (circular crossover) operator on the same problem solution refines drastically. CX outperforms than PMX and OX operators. This particular observation of execution of task modules in parallel and distributed system provides an insight for the development of a better task partitioning module in the task scheduler for a parallel and distributed computing environment.

We observe task partitioning and its execution in parallel/distributed system. The proposed work is based on the decomposition and the execution of task in parallel/distributed system. Our plan is to observe the behavior of these operators for many more problems before it can be established that CX is better than PMX and OX.

References

1) "A compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures,", G.C. Sih and E.A. Lee, IEEE Trans. Parallel and Distributed Systems, vol. 4, no. 2, pp. 175-186, Feb. 1993.

2) " A comparison of heuristics for scheduling DAGS on multiprocessor" C. McCreary, A Khan, J. Thompson, and M. McArdle, 8th International Parallel Processing Symposium, 446-451, 1994.

3) "A Genetic Algorithm for Multiprocessor Scheduling", Edwin S. H. Hou, Member, IEEE, Ninvan Ansari, Member, IEEE, and Hong Ren, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS. VOL. 5, NO. 2, FEBRUARY 1994.

4) "A Genetic Algorithm Tutorial" by Darrell Whitley-Computer Science Department Colorado State University; Fort Collins, CO 80523

5) "An Introduction to Genetic Algorithm", by Melanie Mitchell, PHI, 2005.

6) "Adaptive load sharing m homogeneous distributed systems"- EAGER, D. L., LAZOWSKA, E. D., AND ZAHORIAN, J. IEEE Trans. Software Engineering. 12 (1986), 662-675.

7) "Advance Computer Architecture" Kai Hwang, Tata McGraw Hill Ed, 2001.

8) "An architecture for distributed agent-based data preprocessing", Petteri Nurmi, Michael Przybilski, Greger Linden and Patrik Floreen Helsinki, Institute for Information Technology HIIT, Basic Research Unit Department of Computer Science, P.O. Box 68.

9) "An introduction to genetic-based scheduling in parallel processor systems."- A. Y. Zomaya, F. Ercal, and S. Olariu, editors,Solutions to Parallel and Distributed Computing Problems, chapter 5, pages 111–133. John Wiley and Sons, 2001.

10) "An Overview of Genetic Algorithms: Part 2, Research Topics" by David Beasley- Department of Computing Mathematics, University of Wales College of Cardiff, Cardiff, CF2 4YN, UK; David R. Bully-Department of Electrical and Electronic Engineering, University of Bristol, Bristol, BS8 1TR, UK; Ralph R. Martinz-Department of Computing Mathematics, University of Wales College of Cardiff, Cardiff, CF2 4YN, UK; University Computing, 1993, 15(4) 170-181.

11) "Communication Contention in Task Scheduling"- Oliver Sinnen and Leonel A. Sousa, Senior Member, IEEE, IEEE Transaction on Parallel and Distributed System Vol. 16, No-6, June 2005.

12) "Distributed system Concept and Design", George Coulouris, Jean Dollimore and Tim Kindberg, Third Edition, Sixth Indian Reprint 2004, Publisher-Pearson Education (Singapore) Pte Ltd.

13) "Distributed System - Principles and Paradigms" by Andrew S. Tenenbaum, Maarten Van Steen, Pearson Education, Revised Edition 2006.

14) "Dynamic Scheduling of Computer Tasks Using Genetic Algorithms"- C.A Gonzalez Pico and R.L Wainwright; Proc. First IEEE Conf. Evolutionary Computation, IEEE World Congress Computational Intelligence, vol. 2,1994, pp. 829-833.

15) "EFFICIENT MULTIPROCESSOR SCHEDULING BASED ON GENETIC ALGORITHMS" E. S. H. Hou, R. Hong, and N. Ansari Department of Electrical and Computer Engineering New Jersey Institute of Technology Newark, NJ 07102,087942-6004/90/1100-1239, 1990 IEEE.

16) "Genetic Algorithm - in search, optimization & machine learning", by David E. Goldberg, Pearson Education, 2006.

17) "Genetic Alorithms- simulating nature's methods of evolving the best design solution", Cezary Z. Janikow and Daniel St. Clair; Feb-March 1995 0278-6648/95- 1995 IEEE.

18) "Genetic Algorithms: Theory and Application", Lecture Notes, Third Edition –Winter 2003/2004 by Ulrich Bodenhofer.

19) "Genetic Scheduling for Parallel Processor Systems: Comparative Studies and Performance Issues", Albert Y. Zomaya, Senior Member, IEEE, Chris Ward, and Ben Macey; IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 10, NO. 8, AUGUST 1999.

20) "Observation on Using Genetic Algorithms For Dynamic Load-Balancing", Albert Y. Zomaya, And Yee-Hwei, IEEE Transactions On Parallel And Distributed Systems, Vol. 12, No. 9, September 2001.

21) "Optimal scheduling strategies in a multiprocessor system" C. V. Ramamoorthy et al IEEE Trans. Comput., vol. C-21, Feb.1972, pp. 137-146.

22) "Parallel Computing" by Moreshwer R. Bhujade, New Age International Publishers, 2003.

23) "Parallelization of Fine-grained Irregular DAGs" by Frederic T. Chong, Shamik D. Sharmay, Eric A. Brewer and Joel Saltzx -Massachusetts Institute of Technology ,University of Maryland, College Park, University of California, Berkeley.

24) "Practical Multiprocessor scheduling algorithms for efficient processing"- H Kasahara and S. Narita; IEEE Trans, Comput. Vol. C-33 no. 11, pp. 1023-1029, Nov 1984.

25) "Preemptive Scheduling for Distributed Systems" by - Donald McLaughlin- Department of Computer Science and Engineering. Arizona State University, and Shantanu Sardesai and Partha Dasgupta- Tandem Computers Inc. 19333 Vallco Parkway, Cupertino, CA.

26) "Scheduling a Job Mix in a Partitionable Parallel System" by - Helen D. Karatza, Department of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece and Ralph C. Hilzer, Computer Science Department,California State University, Chico, Chico, California 95929-0410 USA; Proceedings of the 35thAnnual Simulation Symposium (SS.02)- 2002 IEEE.

27) "Task Assignment in a Distributed System: Improving Performance by Unbalancing Load", Mark E. Crovella Department of Computer Science Boston University Boston, , Mor Harchol-Balter Laboratory for Computer Science MIT, NE43-340 Cambridge, MA 02139, Cristina D. Murtaz Department of Computer Science Boston University Boston, MA 02215; October 31, 1997 BUCS-TR-1997-018.

28) "Task partitioning using Genetic Algorithm" by Anil Kumar Tripathi, Deo Prakash Vidyarthi and A. N. Mantri, Proceeding of International Conference on Cognitive System, New Delhi, December 1997, pp. 248-254.

Annexure

Program of the comparison of three Genetic Crossover Operators for Task Partitioning Problem in "C"

```
#include<stdio.h>
#include<stdlib.h>      /*HEADER FILES*/
#include<time.h>

struct parent           /*STRUCTURE FOR TEMPORARY PARENT 2*/
        {
        int info;
        struct parent *p_next;
        };

typedef struct parent p_node;

struct index            /*STRUCTURE FOR TEMPORARY PARENT 1*/
        {
        int pno;
        struct parent *i_p_next;
        struct index *i_next;
        };
typedef struct index i_node;
i_node *start1=NULL;

struct parent2
        {
        int gene2;
        struct parent2 *next2;
        };
typedef struct parent2 node2;

struct parent1   /*STRUCTURE TO REPRESENT FIRST PARENT*/
        {
        int gene1;
        int flag;
        struct parent1 *next1;
        node2 *link;
        };
typedef struct parent1 node1;

node1 *start=NULL; /*START POINTER FOR LINK LIST*/

struct data      /*STRUCTURE FOR LINK LIST FOR OX OPERATOR*/
```

```
        {
        int t;
        struct data *d_next;
        };

typedef struct data d_node;

d_node *d_start1=NULL;

d_node *d_start2=NULL;  /*LINK LIST HEADER FOR OX OPERATOR*/

d_node *m,*n,*x,*y;

static double time_co=0,time_pmx=0,time_ox=0;  /*GLOBAL VARIABLES FOR TIME*/

void node_delete(d_node*);      /*OX OPERATOR SUPPORT NODE DELETION
FUNCTION*/

void display1(); /* FUNCTION TO DISPLAY THE PARENTS*/

void galgo(); /*FUNCTION TO CYCLE CROSSOVER WITHOUT SWAPPING*/

void galgoswap(); /*FUNCTION TO CYCLE CROSSOVER WITH SWAPPING*/

void  delete_list();  /*DELEATING LINK LIST NODES*/

float crossover(p_node*,p_node*); /*CYCLE CROSSOVER FUNCTION*/

float pmx(p_node*,p_node*);     /*PMX FUNCTION*/

float ox(p_node*,p_node*);      /*OX FUNCTION*/

void create(int,int,int);       /*PEREMANT PARENT'S MAPPING FUNCTION*/

void display();                 /*DISPLAY THE PARENTS*/

void main()
{
int pno,size,ch;
float pmxt,oxt,cxt;      /*VARIABE DECLARATION*/
i_node *p,*q;

clrscr();
                            /*MENU PRINTING CODE*/
printf("\n\n PRESS       (1) MANUAL PARENT GENERATION");
printf("\n       (2) AUTOMATED PARENT GENERATION");

printf("\n\n ENTER YOUR CHOICE:");
scanf("%d",&ch);

if(ch==1 || ch==2)
```

```
        {
        printf("\n\n ENTER NO OF PARENTS:");   /*PARENT NUMBER ENTRY*/
        scanf("%d",&pno);

        printf("\n\n ENTER PARENT SIZE(MAX 1000):"); /*PARENT SIZE ENTRY*/
        scanf("%d",&size);
        }

create(pno,size,ch); /*MAPPING PARENTS*/

for(p=start1;p!=NULL;p=p->i_next)
        for(q=p->i_next;q!=NULL;q=q->i_next)
        {

        printf("\n\n OPERATING ON PARENT(%d) & PARENT(%d)",p->pno,q->pno);
        cxt=crossover(p->i_p_next,q->i_p_next);
        pmxt=pmx(p->i_p_next,q->i_p_next);          /*CALLING DIFFRENT OPERATORS*/
        oxt=ox(p->i_p_next,q->i_p_next);
        clrscr();

        printf("\n\n\n");
        printf("\n\n\t\t OPERATING ON PARENT(%d) & PARENT(%d)",p->pno,q->pno);
        printf("\n\n\n\t------------BEST MODULE TIME COMPARASION-----------");
        printf("\n\n\t------------TIME COMPLEXTITY IN MILLISECOND---------");
        printf("\n\n\t PARENTS SIZE : %d  NUMBER OF PARENTS : %d\n",size,pno);
        printf("\n\t-----------------------------------------------------");
        printf("\n\t       OPERATINON                TIME          ");
        printf("\n\t-----------------------------------------------------");
        printf("\n\n\t CYCLE CROSSOVER                    %f  ",time_co);
        printf("\n\n\t PMX                                  %f  ",time_pmx);
        printf("\n\n\t OX                                   %f  ",time_ox);
        printf("\n\t-----------------------------------------------------");

        printf("\n\n\n");
        printf("\n\n\t\t OPERATING ON PARENT(%d) & PARENT(%d)",p->pno,q->pno);
        printf("\n\n\n\t -------------TOTAL EXECUTION TIME----------------");
        printf("\n\n\t------------TIME COMPLEXTITY IN MILLISECOND---------");
        printf("\n\n\t PARENTS SIZE : %d  NUMBER OF PARENTS : %d\n",size,pno);
        printf("\n\t-----------------------------------------------------");
        printf("\n\t       OPERATINON                TIME          ");
        printf("\n\t-----------------------------------------------------");
        printf("\n\n\t CYCLE CROSSOVER                    %f  ",cxt);
        printf("\n\n\t PMX                                  %f  ",pmxt);
        printf("\n\n\t OX                                   %f  ",oxt);
        printf("\n\t-----------------------------------------------------");
        getch();

        }
}
```

```
void create(int pno,int size,int ch)
{
int arr[1000],i,j,k,pp;
i_node *p,*q;
p_node *r,*s;


for(i=0;i<pno;i++)
        {

        if(ch==2)
        {
        for(j=0;j<size;j++)
        arr[j]=j+1;

        for(j=0;j<size;j++)
                {
                randomize();
                k=(random(size)+j+1+i+rand())%size;
                if(j!=k)
                        {
                        arr[j]=arr[j]+arr[k];
                        arr[k]=arr[j]-arr[k];
                        arr[j]=arr[j]-arr[k];
                        }
                }
         }
        else if(ch==1)
                {
                printf("\n\n ENTER %d PARENT",i+1);
                for(j=0;j<size;j++)
                        {
                        printf("\n %d ELEMENT: ",j+1);
                        scanf("%d",&pp);
                        arr[j]=(int)pp;
                        }
                }
        else
                {
                printf("\n\n WRONG ENTRY!");
                exit(1);
                }

           p=(i_node*)malloc(sizeof(i_node));
           if(p==NULL)
                {
                printf("\n\n MEMORY OVERFLOW");
                exit(1);
                }
           if(start1==NULL)
                start1=p;
```

```
            else
                q->i_next=p;

            p->i_next=NULL;
            p->pno=i+1;
            p->i_p_next=NULL;
            q=p;

            for(j=0;j<size;j++)
            {
                r=(p_node*)malloc(sizeof(p_node));
                if(r==NULL)
                        {
                        printf("\n\n MEMORY OVERFLOW");
                        exit(1);
                        }
                if(p->i_p_next==NULL)
                        p->i_p_next=r;

                r->info=arr[j];
                r->p_next=NULL;
                if(j!=0)
                        s->p_next=r;
                s=r;
            }
        }

}

void display()
{
i_node *p=start1;
p_node *r;

while(p!=NULL)
        {
        r=p->i_p_next;
        printf("\n\n PARENT(%d)\n",p->pno);
        while(r!=NULL)
                {
                printf(" %d",r->info);
                r=r->p_next;
                }
        p=p->i_next;
        }
}

float crossover(p_node *parent1,p_node *parent2)
{
int i=0,j=0,at=0,x,xx;
double ptime,mod_time[50],xxx;
```

```
node1 *p,*q;
node2 *r,*s;
clock_t sta,en,st1,en1;
time_t t;
        /* Creating Link List for Parents*/
                i=0;
                while(parent1!=NULL)
                        {
                                if(i==0)
                                {
                                 p=(node1*)malloc(sizeof(node1));
                                 r=(node2*)malloc(sizeof(node2));
                                 if(p==NULL||r==NULL)
                                 printf("\n MEMORY OVERFLOW");
                                 else
                                        {
                                        start=p;
                                        p->flag=0;
                                        p->gene1=parent1->info;   r->gene2=parent2->info;
                                        p->next1=start;     r->next2=NULL;
                                        p->link=r;
                                        q=p;              s=r;
                                         }
                                }
                                else
                                {
                                 p=(node1*)malloc(sizeof(node1));
                                 r=(node2*)malloc(sizeof(node2));
                                if(p==NULL||r==NULL)
                                        printf("\n MEMORY OVERFLOW");
                                 else
                                        {
                                        p->flag=0;
                                        p->gene1=parent1->info;   r->gene2=parent2->info;
                                        p->next1=start;     r->next2=NULL;
                                        q->next1=p;         s->next2=r;
                                        p->link=r;
                                        q=p;              s=r;
                                         }

                                } i++;
                        parent1=parent1->p_next;
                        parent2=parent2->p_next;
                        }

                        display1();

printf("\n\n----------CYCLE CROSSOVER----------");
sta=clock();
i=1;
while(j!=3)
```

```
        {
        for(p=start;p->next1!=start;p=p->next1)
                        {
                        if(p->flag==0)
                                {

//printf("\n-------------------------------------------------");
                                i++;
                                i=i%2;
                                //printf("\n Cycle: %d",tcount+1);
                                switch(i)
                                        {
                                        case 0: st1=clock();
                                                galgo();
                                                en1=clock();
                                                mod_time[at++]=((en1)-(st1)/CLK_TCK);
                                                break;
                                        case 1: st1=clock();
                                                galgoswap();
                                                en1=clock();
                                                mod_time[at++]=((en1)-(st1)/CLK_TCK);
                                                break;
                                        }
//printf("\n-------------------------------------------------");

                        break;
                                }
                        }
                if(p->next1==start)
                j=3;
                }
en=clock();
display1();
//printf("\n\nTIME COMPLEXTITY IN MILLISECOND");
printf("\n-------------------------------------------------");
ptime=(double)((en)-(sta)/CLK_TCK);
delete_list();

        printf("\n\n----------TIME COMPLEXTITY IN MILLISECOND---------");
        printf("\n--------------------CYCLE CROSSOVER-----------------");
        printf("\n        CYCLE'S                                 TIME          ");
        printf("\n-------------------------------------------------");
        st1=99999;
        time_co=0;
        for(i=0;i<at;i++)
        for(j=i+1;j<at;j++)
        {
        xxx=(double)mod_time[i]-mod_time[j];
        printf("\n  %d CYCLE(%f) - %d CYCLE(%f)     :%f
",i+1,mod_time[i],j+1,mod_time[j],xxx);
        if(st1>xxx)
```

```
                {
                st1=xxx;
                x=i+1;
                xx=j+1;
                }
        }
        printf("\n-----------------------------------------------------");
        if(st1==99999)
        st1=0;
        time_co=(double)st1;
        printf("\n\n DEFERENCE IN EXECUTION TIME BETWEEN  CYCLE'S %d & %d :%f
ms",x,xx,time_co);
        getch();
return(ptime);
}

void display1()
{
node1 *p=start;
printf("\n\n\n");
if(start==NULL)
printf("\n EMPTY LINK LIST");
else
        {
        printf("\n\n PARENT1:  ");
        do
        {
        printf(" %d",p->gene1);
        p=p->next1;
        }while(p!=start);
        }
p=start;
if(start==NULL)
printf("\n EMPTY LINK LIST");
else
        {
        printf("\n\n PARENT2:  ");
        do
        {
        printf(" %d",p->link->gene2);
        p=p->next1;
        }while(p!=start);
        }
}


void galgo()
{
int child1,child2;
node1 *p=start,*q;
```

```
while(p->flag!=0 && p->next1!=start)
p=p->next1;

while(p->flag==0)
        {
        p->flag=1;
        child2=p->link->gene2;
        for(q=p->next1;q->gene1!=child2 && q!=p;q=q->next1);
        if(q->gene1==child2)
                p=q;
        }
}

void galgoswap()
{
node1 *p=start,*q;
int item;

while(p->flag!=0 && p->next1!=start)
p=p->next1;

while(p->flag==0)
        {
        p->flag=2;
        item=p->gene1;
        p->gene1=p->link->gene2;
        p->link->gene2=item;
        for(q=p->next1;q->link->gene2!=item && q!=p;q=q->next1);
        if(q->link->gene2==item)
                p=q;
        }
}

void delete_list()
{
node1 *p=start,*q;
printf("\n\n\n");
if(start==NULL)
printf("\n EMPTY LINK LIST");
else
        do
        {
        free(p->link);
        q=p;
        p=p->next1;
        free(q);
        }while(p!=start);
start=NULL;
}

float pmx(p_node *parent1,p_node *parent2)
```

```
{
int i=0,j,up,low,hold,x,xx,at=0;
double ptime,xxx,mod_time[50];
node1 *p,*q;
node2 *r,*s;
clock_t sta,en,st1,en1;
time_t t;
        /* Creating Link List for Parents*/
                i=0;
                while(parent1!=NULL)
                        {
                                if(i==0)
                                {
                                 p=(node1*)malloc(sizeof(node1));
                                 r=(node2*)malloc(sizeof(node2));
                                 if(p==NULL||r==NULL)
                                 printf("\n MEMORY OVERFLOW");
                                 else
                                        {
                                        start=p;
                                        p->flag=++i;
                                        p->gene1=parent1->info;   r->gene2=parent2->info;
                                        p->next1=start;     r->next2=NULL;
                                        p->link=r;
                                        q=p;              s=r;
                                         }
                                }
                                else
                                {
                                 p=(node1*)malloc(sizeof(node1));
                                 r=(node2*)malloc(sizeof(node2));
                                if(p==NULL||r==NULL)
                                        printf("\n MEMORY OVERFLOW");
                                 else
                                        {
                                        p->flag=++i;
                                        p->gene1=parent1->info;   r->gene2=parent2->info;
                                        p->next1=start;     r->next2=NULL;
                                        q->next1=p;         s->next2=r;
                                        p->link=r;
                                        q=p;              s=r;
                                         }

                                }
                        parent1=parent1->p_next;
                        parent2=parent2->p_next;
                        }

                        display1();

printf("\n\n----------PMX----------");
```

```
printf("\n\n ENTER LOWER BOUND:");
scanf("%d",&low);
printf("\n\n ENTER UPPER BOUND:");
scanf("%d",&up);
i=1;
sta=clock();
st1=clock();
p=start;
while(p->flag<low && p->next1!=start)
        p=p->next1;

while(p->flag<=up && p->next1!=start)
        {
        p->gene1=p->gene1 + p->link->gene2;
        p->link->gene2=p->gene1 - p->link->gene2;
        p->gene1=p->gene1 - p->link->gene2;
        p=p->next1;
        }
en1=clock();
mod_time[at++]=((en1)-(st1)/CLK_TCK);
st1=clock();
p=start;
while(p->flag<low && p->next1!=start)
        p=p->next1;


while(p->flag<=up)
        {
        if(i%2==1)
        hold=p->gene1;
        else
        hold=p->link->gene2;
        q=p;

        while(q->flag<=up&& q->next1!=start)
                q=q->next1;
        while(q->flag<low || q->flag>up)
                {
                if(q->gene1==hold)
                q->gene1=p->link->gene2;

                if(q->link->gene2==hold)
                 q->link->gene2=p->gene1;
                q=q->next1;
                }

        if(i%2==0)
        p=p->next1;
        i++;
        }
en1=clock();
```

```
mod_time[at++]=((en1)-(st1)/CLK_TCK);
en=clock();
display1();
//printf("\n\nTIME COMPLEXTITY IN MILLISECOND");
printf("\n\n-----------------------------------------");
ptime=((en)-(sta)/CLK_TCK);
delete_list();

        printf("\n\n----------TIME COMPLEXTITY IN MILLISECOND---------");
        printf("\n-------------------------PMX-------------------------");
        printf("\n    GROUP OF MODULES                                TIME          ");
        printf("\n-------------------------------------------------");
        st1=99999;
        time_pmx=0;
        for(i=0;i<at;i++)
        for(j=i+1;j<at;j++)
        {
        xxx=mod_time[i]-mod_time[j];
        printf("\n  %d GROUP(%f) - %d GROUP(%f)     :%f
",i+1,mod_time[i],j+1,mod_time[j],xxx);
        if(st1>=xxx)
                {
                st1=xxx;
                x=i+1;
                xx=j+1;
                }
        }
        printf("\n-------------------------------------------------");
        if(st1==99999)
        st1=0;
        time_pmx=st1;
        printf("\n\n BEST PERFORMANCE MODULE GROUPS %d & %d :%f
ms",x,xx,time_pmx);
        getch();
return(ptime);
}



float ox(p_node *parent1,p_node *parent2)
{
int i=0,j,up,low,hold,bit,at=0,x1,xx;
double ptime,xxx,mod_time[50],temp;
node1 *p,*q;
node2 *r,*s;
clock_t sta,en,st1,en1,st2,en2;
time_t t;

        /* Creating Link List for Parents*/
                i=0;
                while(parent1!=NULL)
```

```
{
        if(i==0)
        {
         p=(node1*)malloc(sizeof(node1));
         r=(node2*)malloc(sizeof(node2));
         if(p==NULL||r==NULL)
         printf("\n MEMORY OVERFLOW");
         else
                {
                start=p;
                p->flag=++i;
                p->gene1=parent1->info;   r->gene2=parent2->info;
                p->next1=start;     r->next2=NULL;
                p->link=r;
                q=p;              s=r;
                 }
        }
        else
        {
         p=(node1*)malloc(sizeof(node1));
         r=(node2*)malloc(sizeof(node2));
        if(p==NULL||r==NULL)
                printf("\n MEMORY OVERFLOW");
         else
                {
                p->flag=++i;
                p->gene1=parent1->info;   r->gene2=parent2->info;
                p->next1=start;     r->next2=NULL;
                q->next1=p;         s->next2=r;
                p->link=r;
                q=p;              s=r;
                 }

                }
      parent1=parent1->p_next;
      parent2=parent2->p_next;
      }

      display1();

printf("\n\n----------OX----------");
printf("\n\n ENTER LOWER BOUND:");
scanf("%d",&low);
printf("\n\n ENTER UPPER BOUND:");
scanf("%d",&up);
i=1;
sta=clock();
st1=clock();
p=start;
while(p->flag<=up && p->next1!=start)
        p=p->next1;
```

```
while(p->flag!=up)
        {
        m=(d_node*)malloc(sizeof(d_node));
        x=(d_node*)malloc(sizeof(d_node));
        if(m==NULL || x==NULL)
                {
                printf("\n\n MEMORY OVERFLOW");
                exit(1);
                }
        m->d_next=NULL;
        x->d_next=NULL;
        m->t=p->gene1;
        x->t=p->link->gene2;
        if(d_start1==NULL)
                d_start1=m;
        else
                n->d_next=m;
        if(d_start2==NULL)
                d_start2=x;
        else
                y->d_next=x;
        y=x;
        n=m;
        p=p->next1;
        }
        m=(d_node*)malloc(sizeof(d_node));
        x=(d_node*)malloc(sizeof(d_node));
        if(m==NULL || x==NULL)
                {
                printf("\n\n MEMORY OVERFLOW");
                exit(1);
                }
        m->d_next=NULL;
        x->d_next=NULL;
        m->t=p->gene1;
        x->t=p->link->gene2;
        if(d_start1==NULL)
                d_start1=m;
        else
                n->d_next=m;
        if(d_start2==NULL)
                d_start2=x;
        else
                y->d_next=x;

st2=clock();
p=start;
while(p->flag<low && p->next1!=start)
        p=p->next1;
```

```
while(p->flag<=up && p->next1!=start)
        {
        p->gene1=p->gene1 + p->link->gene2;
        p->link->gene2=p->gene1 - p->link->gene2;
        p->gene1=p->gene1 - p->link->gene2;
        p=p->next1;
        }
en2=clock();
temp=((en2)-(st2)/CLK_TCK);
mod_time[at++]=temp;
p=start;
while(p->flag<low && p->next1!=start)
        p=p->next1;

while(p->flag<=up)
        {
        hold=p->gene1;
        bit=p->link->gene2;
        m=d_start1;
        while(m!=NULL)
                {
                if(d_start1->t==hold)
                        {
                        d_start1=m->d_next;
                        free(m);
                        break;
                        }
                if(m->d_next->t==hold)
                        {
                        node_delete(m);
                        break;
                        }
                m=m->d_next;
                }

        x=d_start2;
        while(x!=NULL)
                {
                if(d_start2->t==bit)
                        {
                        d_start2=x->d_next;
                        free(x);
                        break;
                        }
                if(x->d_next->t==bit)
                        {
                        node_delete(x);
                        break;
                        }
                x=x->d_next;
                }
```

```
        p=p->next1;
        }

p=start;
while(p->flag<=up)
        p=p->next1;

m=d_start1;
x=d_start2;
while(p->flag<low||p->flag>up)
        {
        if(m!=NULL)
                p->gene1=m->t;
        if(x!=NULL)
                p->link->gene2=x->t;
        m=m->d_next;
        x=x->d_next;
        p=p->next1;
        }
en1=clock();
mod_time[at++]=((en1)-(st1)/CLK_TCK)-temp;
en=clock();
display1();
//printf("\n\nTIME COMPLEXTITY IN MILLISECOND");
printf("\n\n-----------------------------------------------");
ptime=((en)-(sta)/CLK_TCK);
delete_list();

        printf("\n\n------------TIME COMPLEXTITY IN MILLISECOND---------");
        printf("\n---------------------------OX-------------------------");
        printf("\n        GROUP OF MODULES                        TIME          ");
        printf("\n-------------------------------------------------------");
        st1=99999;
        time_ox=0;
        for(i=0;i<at;i++)
        for(j=i+1;j<at;j++)
        {
        xxx=mod_time[i]-mod_time[j];
        printf("\n  %d GROUP(%f) - %d GROUP(%f)     :%f
",i+1,mod_time[i],j+1,mod_time[j],xxx);
        if(st1>xxx)
                {
                st1=xxx;
                x1=i+1;
                xx=j+1;
                }
        }
        printf("\n-------------------------------------------------------");
        time_ox=st1;
        printf("\n\n BEST PERFORMANCE CYCLE'S %d & %d :%f ms",x1,xx,time_ox);
```

```
        getch();

return(ptime);
}

void node_delete(d_node *n)
        {
        d_node *m=n,*p;
        n=n->d_next;
        m->d_next=n->d_next;
        free(n);
        }
```