Using ant colonies to solve multiprocessor task graph scheduling

Appah Bremang

Master Thesis
Computer Engineering
2006
Nr: E3265D

HÖGSKOLAN Dalarna

DEGREE PROJECT
in Computer Engineering

Programme	Reg number	Extent
International Masters of Science In Computer Engineering	E3265D	30 ECTS
Name of student	Year-Month-Day	
Appah Bremang	2006-08-25	
Supervisor	Examiner	
Pascal Rebreyend	Prof. Mark Dougherty & Ernst	
Department of Computer Engineering		
Title		
Using ant colonies for multiprocessor task graph scheduling		
Keywords		
Multiprocessor scheduling problems, ant colony algorithm, ant system, pheromone trail, makespan.		

Abstract

The problem of scheduling a parallel program presented by a weighted directed acyclic graph (DAG) to the set of homogeneous processors for minimizing the completion time of the program has been extensively studied as academic optimization problem which occurs in optimizing the execution time of parallel algorithm with parallel computer.

In this paper, we propose an application of the Ant Colony Optimization (ACO) to a multiprocessor scheduling problem (MPSP). In the MPSP, no preemption is allowed and each operation demands a setup time on the machines. The problem seeks to compose a schedule that minimizes the total completion time.

We therefore rely on heuristics to find solutions since solution methods are not feasible for most problems as such. This novel heuristic searching approach to the multiprocessor based on the ACO algorithm a collection of agents cooperate to effectively explore the search space.

A computational experiment is conducted on a suit of benchmark application. By comparing our algorithm result obtained to that of previous heuristic algorithm, it is evince that the ACO algorithm exhibits competitive performance with small error ratio.

ACKNOWLEDGEMENTS

I give thanks to God, Who has faithfully called, guided and brought me to an expected end of this program. To him be All Glory.

I would like to express my sincere gratitude to my Supervisor, Pascal Rebreyend, for providing a great deal of effort, time, and patient in the completion of this project, without him, the research could not have been completed.

I give my deepest thanks to Prof. Mark Dougherty and Ernst Nordström for their invaluable contribution and advice.

My sincere thanks go to all lecturers in Computer Engineering department for their support in various ways.

Finally to my mother Agnes Gyamena and father R. Adane Appah as well as siblings for their support during my studies.

Table of Contents

1.0 Introduction

As computing system become more complex, so do the application that can run on them. In order to efficiently and effectively map application onto these systems, designers will increasingly rely on heuristic tools since it cannot be solved in the traditional way. On fundamental process of these heuristics is creating a mapping of a behavioral model (ACO) of an application to the computing system. The multiprocessor task schedule problem is *NP-complete*. Although it is possible for formulate and solve the problem using heuristics, the feasible solution space quickly becomes intractable for larger problem instance. In order to address this problem, a range of heuristic methods with a polynomial run time complexity have been proposed. These methods include Ant Colony Optimization, Genetic Algorithm, Tabu Search, Simulated Annealing, Graph Theoretic and Computational Geometry Approaches.
Among them, ant colony optimization is considered in this project work due to its simplicity of implementation and capability of generating reasonably good results at less time.

As the world and therefore all the economic vibrant organizations faces the problem (combinational optimization) of optimizing their scare resources in order to attained the full utilization of such resources, the MPSP and its varying of other scheduling problems addresses such shortfall in the industries. Moreover problems such as this, is hard to be solve in the polynomial time since exact solution methods are unfeasible for most problem instances and heuristic approaches must therefore be employed to find solutions. The developed algorithm can search a wider space for nearly optimal solution to *NP-hard* problems.

The multiprocessor scheduling problem (MPSP) is considered as optimization problem in which the MPSP is a set of tasks with a given processing times and has to be assigned to a set of identical processors in order to minimizing the execution time (thus the full utilization of the time on the processors). This scheduling can be considered as optimal allocation of scarce resources to activities (referred to as tasks) over a period of time.

However it needs to be stated that the processor being a resource can perform at most a activity at any particular time. This problem nature considering are static or in otherwords deterministic and so all the necessary information about the tasks and processors are assumed to be known.

The basic problem that is encountered in solving MPSP is the delay in communication or transmission of data from one task to the other defining a precedence relationship for the set of task, being predecessor and successor. Our concern is assumed in the problem of scheduling dependencies tasks onto multiprocessor system with processors connected in an arbitrary way, while explicitly accounting for the time required to transfer data between the tasks allocated to different processors. The delay in communication therefore occurs whenever two pair of tasks (predecessor, successor) is assigned to different processors.

The communication delay is address by allocation problem as a typical scheduling problem. This employs the program graph as a directed acyclic graph (DAG). Here, the vertices represent the program modules, but a (directed) arc indicates a direct 1-way communication between a predecessor and successor pair of modules. A schedule on the other hand is allocating a time interval on one or many processor to a task (modules) such that, all being equal, the associated constraints and delays in communication are considered with a sole aim of minimizing execution time. This approach is considered in this paper, which is used to model the primary computations and their interdependence. However, their arcs represent functional dependence among primary computations that imply time precedence in parallel scheduling.

In a task graph, a collection of agents cooperate together to search for a good scheduling solution. Both global and local heuristics are combined in a stochastic decision making process in order to effectively and efficiently explore the search space. The quality of the resultant feasible task schedule on the multiprocessor scheduled problem is evaluated using an ant colony optimization.

The main contribution of this work is the formulation of ant colony optimization algorithm that:

- utilizes a hybrid approach combining multi-processor scheduling problem and the developed ant system heuristic;
- dynamically computes local and global heuristics based on the input application to adaptively search the solution space;
- addresses MPSP in the contextual deterministic machine scheduling theory.
- generates consistently good scheduling results over all testing cases compared with a range of other heuristics and demonstrates stable quality over a variety of application benchmark of large size.

In this paper, the investigation of search space characteristics and their relation to the algorithm performance may give useful insights as initial step in addressing these issues.

The rest of the paper is organized as follows. The problem description is described in section 3, together with the model. Section 4 reviews combinatorial optimization and graph problems. Section 5 reviews ant colony optimization. Section 6, discuss basic technique. Section 7, presents ACO for multiprocessor scheduling that includes solution construction as well as algorithm outline. Section 8, discuss pheromone trail reinforcement and pheromone update. Section 9, presents an experimental analysis of these sets. Section 10, concludes the paper.

2.0 Aim and Objective

The idea of this project is to try how ant colonies can be used to solve the multiprocessor task graph scheduling. In this thesis work, an implementation is first done and the experimental result compared with other methods for which we have results on the same benchmarks.

3.0 Problem Description

Considering the following problem; given a set of identical processors, we face a number of independent requests for processing tasks. Each request is characterized by a multi-processor task with (a) its required processing period, (b) required processor for the whole period (c) the corresponding time/cost of processing the task. The objective is to decide which requests to accept and/or as to minimize the total time/cost subject to the constraint that the total number of available processors is fixed.
For instance, consider a multiprocessor schedule problem (MPSP) with different processors but of the same speed and as well as many tasks to be scheduled on the processor in order to minimize the execution time of processing the entire task.

3.1 Definition for MPSP

The MPSP could be defined as: a set of n tasks (t_i) is to be scheduled on a set of m identical processors. Where schedule could be seen as the sequence and time in which the tasks (t_i) are executed with (i = 0….n). A task graph is a weighted DAG with G = (T , E), where T the set of nodes (corresponding to processors) and E is a set of communication edges. W is the set of node weights, and C is the set of edge weights. Given a task graph TG and a number of processors P, whereas MPSP is to distribute tasks (t_i) in TG onto m computational processors, which is fully connected in order for the precedence constraints to be satisfied and the execution time of the task graph minimized. There is no preemption or duplication of task in this case.

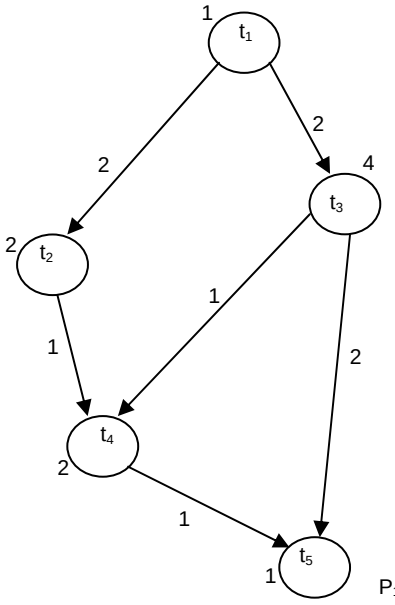
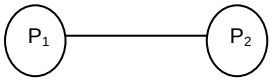


Fig 1a: Example of DAG



Computer/Processo

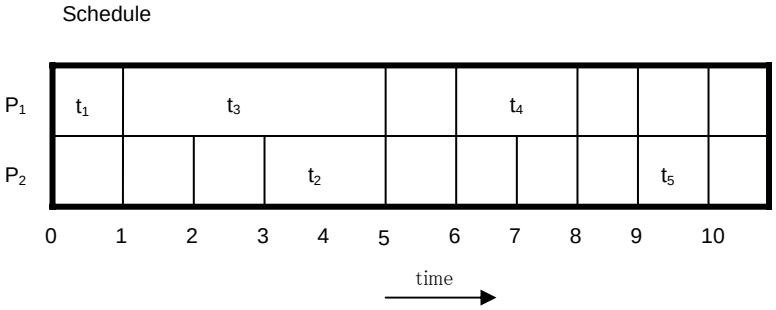


Figure1b: Example of optimal schedule displayed on Gantt chart.

3.2 Model:

The following model is considered in this work to define and approach the MPSP:

In this section, it recalls the graph-based combinatorial formulation of the MPSP problem and then describes the characteristics of the scheduling heuristics used to evaluate the set of test-problem instance.

Let P_i represents the set of processors and T represents the ordered list of tasks which are schedule on the processors P_i**.** Then notation P_i (i = 1,. …,n) refers to the ith processor, P. T = **{**$t_1, t_2 t_3, t_4$ **…**. t_n**}**, represents a set of n-tasks in the system. A task in our case is a job to be run on the processors. The precedence constraints between tasks are modeled using a task graph.

The tasks to be scheduled are represented by a directed acyclic graph (DAG) defined by a tuple G = (T, E, C, L), where T = $\{t_1,...,t_n\}$ denotes a set of tasks; E = $\{e_{ij} / t_{i,}t_j \in T\}$ represents the set of precedence/communication edges where each task node in this case could define a functional unit for the program, which contains information about the computation it needs to perform. However, t_1 and t_n can be considered as two special nodes, which are virtual task nodes. That is they are included for the convenience of having a unique starting and ending point of the task graph; C = $\{c_{ij} / e_{ij} \in E\}$ denotes a set of edge communication costs; and L = $\{l_1,...,l_n\}$ represents the set of task computation times (execution times, length). The communication cost $c_{ij} \in C$ corresponds to the amount of data transferred between tasks t_i and t_j when executed on different processor. When both tasks are assigned to the same processor, the communication cost equals zero. The set E defines precedence relations between tasks t_i and t_j. For a given scheduling on the processor, the execution of a task graph runs in the following way: the task of different precedence levels are sequentially executed from top level down, while tasks in the same precedence level but allocated on different system component (or processor) can run concurrently. A task cannot be executed unless all its predecessors have completed their executed and all relevant data is available. Task Preemption and redundant executions are not allowed in the problem version considered in this paper.

The multiprocessor system is assumed to contain p identical processors with their own local memories. Processors communicate by exchanging messages through bidirectional links of equal capacity. The architecture is modeled by a *distance matrix*. The element (k, l) of the distance matrix D = $[d_{kl}]$ equals the minimum number of links connecting the nodes p_k and p_l. It is also assumed that each processor constituting the multiprocessor system M has I/O processing units for all communication links so that computations and communications can be performed simultaneously.

The scheduling of DAG G onto multiprocessor system consists in determining a processor index and starting-time instant for each task in G in such a way as to minimize a given objective function. An objective function used, represents the completion time of the scheduled task graph (also referred to as makespan, response time or schedule

length). The starting time of the task t_i is determined by completion time of its predecessors and the amount of time needed to transfer the associated data from processors executing these predecessors to the processors p_k and p_l, respectively, may be calculated as

$\gamma_{ij}^{kl} = c_{ij} d_{kl} ccr,$

Where ccr is architecture-dependent and represents the communication-to-communication ratio, defined as the ratio between the transfer time of a unit of data and the time require to perform a single computational operation. However, when l = k, d_{kl} = 0 implying that $\gamma_{ij}^{kl} = 0$.

An edge in E(G) represents a precedence constraint between tasks. It is assured here that the n is the number of nodes in T(G). Then for a given edge *(s, r)* in E(G), s is a predecessor of r, and r is a successor of s.

However, for a node r in T(G), there exist an augmented cost (r) representing the execution time of task r on a processor, and each edge *(r, w)* in E(G) is given augmented cost *c(r, w)* representing the time cost of transmitting from r to w (thus length), that is communication from r to w. This communication from r to w takes zero step if those nodes are assigned to the same processor; otherwise it takes *c(r, w)* steps if they are assigned to different processors.

This algorithm is modeled as a directed acyclic graph set of n tasks $\{t_{i1 \ldots\ldots} t_{in}\}$ positioned at each node while each is associated with a valve which represents the cost of the each of the tasks. For each edge or arc (say; $\{t_{i1,} t_{i2}\}$) scheduled on the processor or find on the directed acyclic graph (DAG) means that the task t_{i2} must received some information or data from t_{i1} before starting the execution of t_{i2}. Moreover, t_{i1} sends this information (or data) at the end of its execution. It is however important to state that, like each node[2], each edge is associated with a valve which represents the communication's cost (or data to be transmitted). If the two corresponding tasks are not on the same processor (for

[2] The term ''node´´ and ''task´´ is use interchangeably throughtout in the work.

instance, t_{i1} to t_{i2}) then communication time will now be the node valve of t_{i1}, the edge valve of data (communication cost{$c(i, j)$}) to be transmitted and the node valve of t_{i2}.

However if two tasks are scheduled on the same processor (for instance, t_{i2} *to* t_{i4}) then the communication cost for processing the two tasks will be zero, the addition of the node valves (thus the cost of executing (d_i) a task on any processor) at each of the two tasks involve. We can therefore infer that there is no preemption or duplication of task in this case of study. The processor could be faced with taking a longer time in executing a task in an attempt of solving the problem without applying any heuristics on the problem. This is because the problem belongs to a set of *NP-hard* problems.

The acyclic digraph *D* shown in figure2 below is composed of six tasks, while the multiprocessor system is composed of three processors fully interconnected. Each d_i indicates the cost of t_i and each c (i_1, i_2) represents the communication times associated to the arc (t_{i1}, t_{i2}). A diagram representing a schedule *S* of the tasks of *D* on the multiprocessor system is also shown. In the diagram, the processor, the introduction data and the duration of each task according to *s* are indicated, as well as the vector representation of the schedule *s*. For instance, t_1, is scheduled on processor P_1 at the time interval [0..1], t_2 is scheduled on processor P_2 at the time interval [3..5], t_3 is scheduled on processor P_3 at the time interval [1..5], and so on.

To describe it further, $c(i, j)$ is the communication cost between task *i* and *j* if they are allocated on different processors and zero if they are allocated on the same processor.
d_i; the cost of execution of task t_i on any processor.

S_ρ : denotes a partial schedule (where, $0 \le \rho \le n$). A schedule S_ρ is a subset of S if the tasks that are scheduled in S_ρ are scheduled in S on the same processor and with the same rank as in S_ρ .

$NT(S_\rho)$: denotes a set of non-scheduled tasks of S_ρ. Given tasks $T = \{t_{i1}, t_{i2}, t_{i3}.....t_{i\rho}\}$ and if the set of task $\{t_{i1}, t_{i2},....,t_{i\rho}\}$ are partial scheduled in S_ρ, then the set $NT(S_\rho)$ = T $\setminus \{t_{i1}, t_{i2},....,t_{i5}\}$ is non-scheduled tasks of S_ρ.

$YT(S_\rho)$: denotes the set of yet to schedule task (or free task) of S_ρ, thus the non-scheduled tasks of a given S_ρ whose all predecessors have already been scheduled. This denotes that, for a specified partial schedule, $YT(S_\rho)$ is a subset of $NT(S_\rho)$.

$ST(t_r)$: denotes the start time of task t_r.

$FT(t_r)$: denotes the finish time of task t_r.

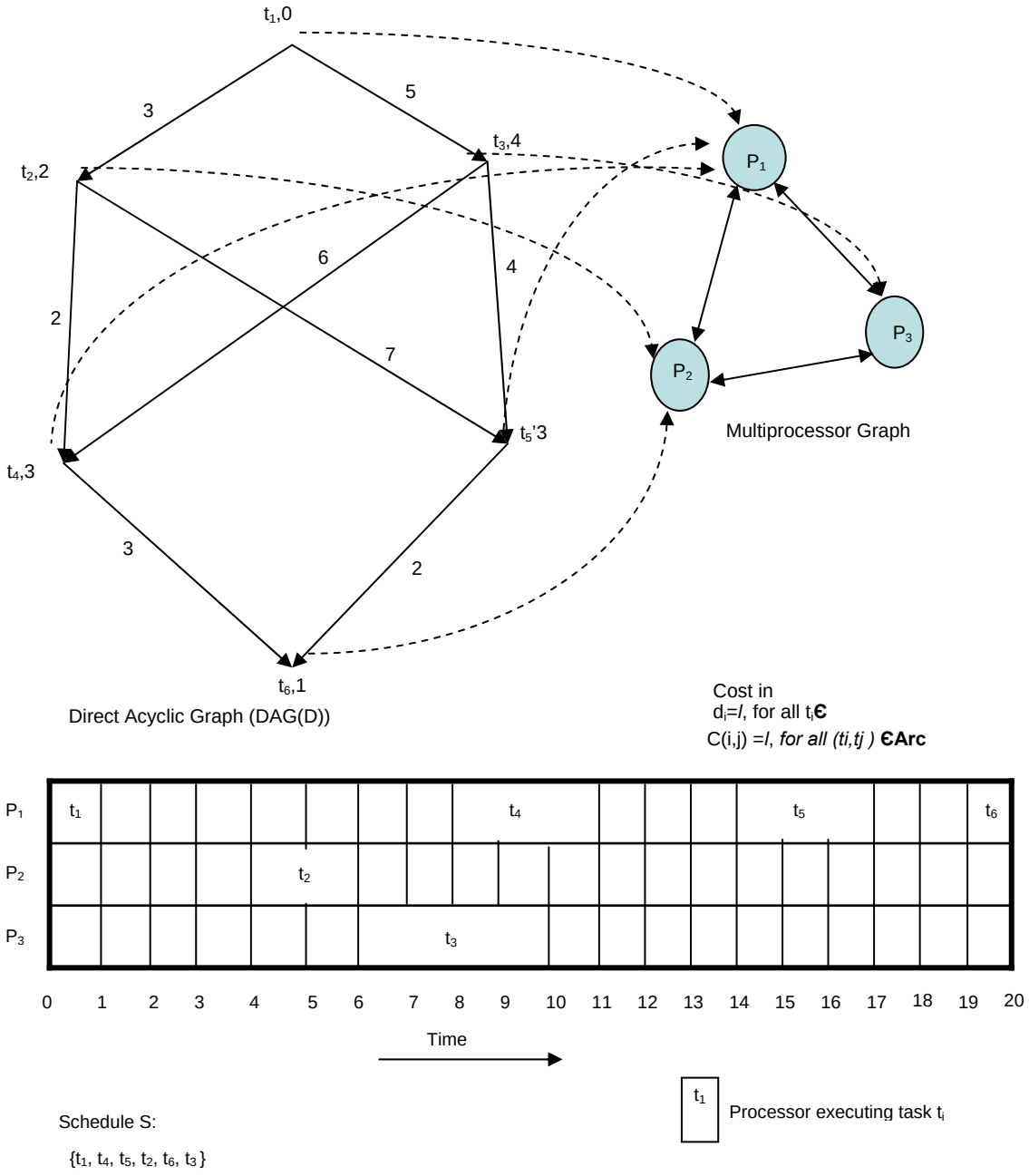
A schedule of G onto P is a function *f* from G(T) to *P* * *I*, with *I* being the set of non-negative integers representing the start time of the tasks: thus, *f(r)* is a pair (*p*, *st*) which implies that task *r* is executed on processor *p* from time *st*. Any task *r* in V(G) is executed on a processor: when *f(r) = (p,st)*, task *r* is executed on processor *p* during time interval $[st, st + \omega(w))$ exactly once without interruption. In the following, we assume that the entry node w_s starts its execution at time zero on processor p_1 and that the execution time of entry and exit tasks is zero while the length of the data transmitted from the entry node and that received by the exit node are both equal to zero.

A scheduled is said to be feasible if and only if, it satisfied the following conditions:

- $\forall r, w \in V(G)$, if $f(r) = (p.st_r)$ and $f(w) = (p.st_w)$, then $st_r + \omega(r) \le st_w$ or $st_w + \omega(w) \le st_r$; thus the executions of two tasks assigned to the same processor need not to be preempted or overlapped.
- For any $(w, y) \in E$, if $f(w) = (p_w, st_w)$ and $f(y) = (p_y, st_y)$, then $st_y \ge st_w + \omega(w) + \lambda(w, y)$, where $\lambda(w, y) = c(w, y)$ if $p_w \ne p_y$ and $\lambda(w, y) = 0$ else the assignment must satisfy the precedence constraint.

Let L_{cp} denote the time cost or length of the longest path in G, where the length of a path from *w* to *y* in G is defined as the total amount of execution costs on the path including end nodes (the length does not include communication costs). A path with length or time cost $L_{cp}(G)$ is referred to as a critical path of G (Discussed later in this work).

In figure 3, an example of MSP represented by the DAG with 6 tasks allocated to 3 processors is shown.



A Partial Schedule S' is given by: $\{t_1, t_2, t_3\}$ *when stopped assigning the tasks at this point. In this case our non-schedule tasks, NT, are* $\{t_4, t_5, t_6\}$ *being the same for yet to schedule tasks, YT (or free tasks).*

Figure2*: Example to illustrate a partial schedule.*

4.0 Overview Combinatorial Optimization and Graph Problems

Introduction to Combinatorial Optimization provides a comprehensive overview of basic optimization technology from Operations Research and Constraint Programming. Concepts covered include: Linear Programming, Duality Theory, Total Unimodularity, Backtracking and Branch-and-Bound, Finite Domain Constraint Programming, and Local Search.

Finding a solution to large combinatorial problems such as multiprocessor scheduling and others is similar in finding a needle in a haystack. A particular class of algorithms, commonly labeled meta-heuristics or combinatorial optimization, such as simulated annealing and tabu search, has been able to provide good enough solutions in reasonably computational time, however (Lockwood and Moore, 1993; Boston and Bettinger , 1998; Baskent and Jordan, 2001). They are designed to solve complex optimization problems where traditional methods have fail to be effective or efficient.

Combinatorial optimization is a broad field, and people come to it with many different perspectives and techniques. Operations Research folk often think of network flow problems and integer programming when they think of combinatorial optimization. Computer Science types often think of heuristics like simulated annealing and genetic algorithms. Artificial Intelligence folks often think of constraint satisfaction, and so on.

A meta-heuristic or combinatorial optimization is defined as an iterative generation process which guides a subordinate heuristic by combining intelligent different concept of exploring and exploiting the search space (Baskent and Jordan, 2001; Beasily et al., 1993). It is based on the idea of making incremental improvements by changing elements of a solution iteratively. While multiprocessor scheduling offers a combinatorially large number of alternatives, many of them represent infeasible solutions and the feasible region is not a continuous space. Thus the strategy is to employ a smart search technique over the solution space. Essentially, a meta-heuristic is a hybrid search technique involving more than one algorithm, tailored to overcome certain ‘traps’ i.e., local optima,

in an extremely large combinatorial solution space. These heuristics have the ability to formulate problem, a problem using discretionary rule that would be difficult to formulate mathematically (Glover and Laguna, 1997). In meta-heuristics parlance, for example, a multiprocessor scheduling designed problem would be represented as either minimizing or maximizing an objective function subject to some constraints.

Meta-heuristics include, but are not limited to: hill climbing or greedy random adaptive search procedures, simulated annealing, genetic algorithms and tabu searches and their hybrids. They basically differ from each other in the use of a move selection and solution mapping procedure.

A graph on the otherhand is a very simple structure consisting of a set of vertices and a family of lines (possibly oriented), called edges (undirected) or arcs (directed), each of them linking some pair of vertices. An undirected graph may for example model conflicts between objects or persons. A directed graph (or digraph) may typically represent a communication network, or some domination relation between individuals, etc.

The famous problem of the bridges of Konigsberg, solved by Euler, is viewed as the first formal result in graph theory. This theory has developed during the second half of the 19th century (with Hamilton, Heawood, Kempe, Kirchhoff, Petersen, Tait), and has boomed since the 1930s (with König, Hall, Kuratowski, Whitney, Erdös, Tutte, Edmonds, Berge, Lovász, Seymour, and many other people). It is clearly related to Algebra, Topology, and other topics from Combinatorics. It applies to, and gets motivating new problems from Computer Science, Operations Research, Game Theory, Decision Theory.

The number of concepts that can be defined on graphs is very large, and many generate deep problems or famous conjectures (for instance the four colour problem). In fact, many of these concepts or theoretical questions arise from practical reasons (and not just from the mathematicians' imaginations) for solving real problems modelled on graphs. Moreover, researchers in Graph Theory try if possible to find efficient algorithms for solving these problems.

The main classical problems in Graph Theory are: flow and connectivity (network reliability), matching (assignment), Eulerian walks (traversing each edge once; more

generally, the "Chinese Postman Problem"), Hamiltonian walks (traversing once each vertex: the "Travelling Salesman Problem"), vertex- or edge-coloring stable, dominating sets. Some of the above (maximum flow, maximum matching, Eulerian walk) can be efficiently solved, while the others are very hard ("NP-complete").
A generalization of the concept of graph, introduced by Claude Berge in 1960, is that of hypergraph, where, simply, the edges may have arbitrary size and not only size two.

However, search methods (stochastic) are a class of search methods which includes heuristics and an element of nondeterminism in traversing the search space. Unlike the search algorithms introduced so far, a stochastic search algorithm moves from one point to another in the search space in a nondeterministic manner, guided by heuristics. The next move is partly determined by the outcome of the previous move. Stochastic search in general, can be said to be incomplete. Stochastic search methods, which although normally does not guarantee completeness, may provide an answer to other applications on delay in decision or communication.

5.0 Overview of Ant Colony Optimization

5.1 *Basics of ACO*

The ACO, in short for ant colony optimization was first introduced proposed by Collorni et al. (1991) as a meta-heuristic scheme for finding near optimal solutions. It has been successfully used to solve many complex problems, such as TSPs, quadratic assignment problems, vehicle routing problems and production scheduling problems, just to name a few.

The ACO simulates the behaviors of real ants moving on weighted connected graph and is also to solve many complex combinatorial optimization problems. The basic algorithm of the ACO introduced by Dorigo et al. (1996 & 1999) is outline (in fig 3) as follows:

```
Algorithm Ant_Colony_Optimization
1. Initialize
        Representing the underlining problem by a weighted connected graph.
        Set initial pheromone for every edge.
2. Repeat
    2.1. For each ant do
            Randomly select a starting node.
            Repeat
                Move to the next node according to a node transition rule.
                Until a complete tour is fulfilled.
     2.2. For each edge do
            Update the pheromone intensity using a pheromone updating rule.
    Until the stopping criterion is satisfied.
3. Output the global best tour.
```

Fig 3: Outline of Basic ACO algorithm.

The ACO was inspired by the ability of real ant colonies to efficiently organize the foraging behavior of the colony using pheromone trails that acts as a means of communication.

The ant colony optimization algorithm is a cooperative heuristic searching algorithm inspired by the ethological study on the behavior of ants. It was observed that ants - who lack sophisticated vision - could manage to establish the optimal path between their

colony and the food source within a very short period of time. This is done by an indirect communication known as *stigmergy* via the pheromone (thus, chemical substance), left by the ants on the paths. Though any single ant moves essentially at random, it will make a decision on its direction biased on the ‘’strength’’ of the pheromone trails that lie before it, where a higher amount of pheromone hints a better path. As an ant traverses a path, it reinforces that path with its own pheromone. A collective autocatalytic behavior emerges as more ants will choose the shortest trails, which in turn creates an even larger amount of pheromone on those short trails, which makes those short trails more likely to be chosen by future ants.

The ACO algorithm is inspired by such observation. It is a population based approach where a collection of agents cooperate together to explore the search space. They communicate via a mechanism imitating the pheromone trails.

The idea of this algorithm is based on using ant colony optimization on the multi-processor schedule to minimize the execution time of the total tasks scheduled on the number of processors. This algorithm is considered in applying on the multi-processor problem because the problem itself is NP-hard which cannot be solved in the traditional way, hence ant colony optimization.

ACO is a population-based algorithm where several artificial ants search for good solutions. Every ant builds up a solution step by step thereby going through several decisions until a solution is found. Ants that found a good solution mark their paths through the decision space by putting some amount of pheromone on the edges of the path. The ants of the next generation are attracted by the pheromone so that they will search in the solution space near good solutions.

The ant colony algorithm works in such a way that; individual ants are simple insects with limited memory and capable of performing simple actions. However, an ant colony expresses a complex collective behavior providing intelligent solutions to problems such as carrying large items, forming bridges and finding the shortest routes from the nest to a food source. It must however be stated that a single ant has no global knowledge about the task it is performing. The ant’s actions are based on local decisions and are usually unpredictable. The intelligent behavior naturally emerges as a consequence of the self-

organization and indirect communication between the ants. This is what is usually called Emergent Behavior or Emergent Intelligence.

The practical example covered in this project involves finding a minimum execution time of the tasks scheduled on the processors. In order to solve this problem, two characteristics of ant's colonies will be particularly useful:

- their ability to find the shortest route between the nest and a food source, which will be used to find and optimized a path in the task graph.
- the simplicity of each individual ant, which will make it easy for us to model the ant colony as a multi-agent system.

More importantly, the ants' search experience can be used to influence in a way reminiscent of Reinforcement Learning [32] the solution construction in future iterations of the algorithm. In addition, the use of a colony of ants give the algorithm increased robustness and in this ACO application the collection interaction of a population of agents is needed to efficiently solve a problem.

5.2 Ants' foraging Behavior and Optimization

Firstly, let's understand the foraging behavior of ants and how they can manage to find the shortest path between the nest and a food source using simply local decisions (thus shortest execution time, when executing the tasks).
An important insight of research on ants' behavior was that most of communication among individuals, or between individuals and the environment, is based on the used of chemicals, named *pheromones* produced by the ants. This is different from, for instances, what happen in human and in other higher species, whose most important senses are visual or acoustic. Particularly important for the social life of an ants' is *trail pheromone*.

Trail pheromone is a specific type of pheromone that ants' use for marking path on the ground, for instance, paths from food sources to the nest. By sensing pheromone trails foragers can follow the path to food discovered by other ants.

Ants use signaling communication system based on deposition of pheromone over the path it follows, marking trail. Pheromone is a hormone produced by ants that establishes a sort of indirect communication among them. Basically, an isolated ant moves at random, but when it finds a pheromone trail there is a high probability that this ant will decide to follow the trail. An ant foraging for food lay down pheromone over its route. When this ant finds a food source, it returns to the nest reinforcing its trail. Other ants in the proximities are attracted by this substance and have greater probability to start following this trail and thereby laying more pheromone on it. This process works as a positive feedback loop system because the higher the intensity of the pheromone over a trail, the higher the probability of an ant start traveling through it. This collective trail-laying and trail-following behavior whereby an ant is influenced by a chemical trail left by other ants is the inspiring source of ACO.

The following instance is taken to give insight of how this process leads the colony to optimize a route, which in our case finds the less execution time. The figure4 illustrates ants foraging behavior and optimization while figure5 demonstrates pheromone evaporation.

Suppose some ants were randomly searching for food when they found two different routes between the nest and the source. Ant need to pick food when they get to the food source and leave food when getting back to the nest. Since for the example route ***B*** (with a length of say 0.5mm) is shorter, the ants on this path will complete the traveling more times and thereby lay more pheromone over it. As the process continues, the pheromone concentration on the trial ***B*** will increase at a higher rate than on ***A*** (with length of say 1.5mm) and soon, even those ants on the route ***A*** will choose to follow the trail ***B***. However, since most ants are no longer traveling through route ***A*** and also due to the

volatile characteristic of the pheromone, the trail ***A*** will start evaporating and soon just the shortest route will remain (*shown at figure5, in the third figure to the right*).

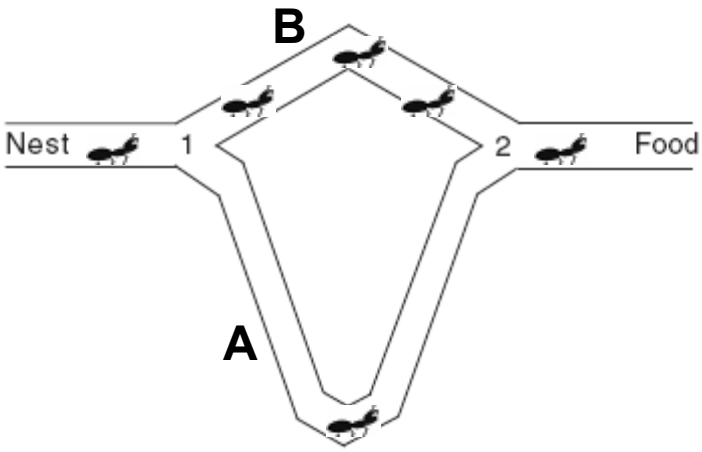


Figure4: *Ants' foraging Behavior and Optimization. More ants using route* ***B*** *to the food source since route* ***A*** *is as double as* ***B****.*

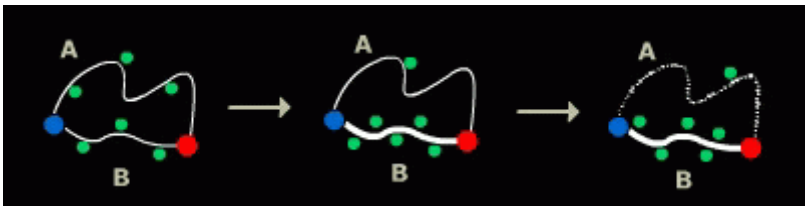


Figure5: *Illustrates how pheromone trails evaporates*

This is because the ants initially picked their route apparently at random and about equal proportion of the ants used each way to the food source. However, as pheromone trial increases, the ants tended to favor the shortest route to the food. Therefore the pheromone trail increases on the shorter route faster than on the any of the routes because the ants using the shorter route to gather food would take a shorter time, hence more ants would cross that path and so more pheromone would be deposited in order to attract other ants on the different routes to the shorter path. This behavior of ants does not allow a single ant to derive this information.

5.3 **Pheromone Trail Evaporation**

Pheromone trail evaporation can be seen as an evaporation mechanism that avoids quick convergence of all the ants towards a suboptimal path. In fact, the decrease in pheromone intensity favors the exploration of different paths during the whole search process. In real ants colonies, pheromone trails also evaporate, however, evaporation does not play an important role in a real ants' shortest path finding. The fact that, on the contrary, pheromone evaporation seems to be important in artificial ants is probably due to the fact that the optimization problems tackled by artificial ants are much more complex than those real ants can solve. A mechanism like, evaporation that, by favoring forgetting of errors or of poor choice done in the pass, allows a continuous improvement of the ''learned'' problem structure seems therefore to be necessary for the artificial ants. Additionally, artificial pheromone evaporation also plays the important function of bounding the less execution time achievable by the pheromone trails.

Evaporation decreases the pheromone trail with exponential speed. In ACO, the pheromone evaporation is interleaved with the pheromone deposit of the ants. After each ant ***k*** has moved to a next node according to the ants' search behavior describes, the pheromone trails are evaporated by applying the following question to all the arcs.

$$\tau_{ij} \leftarrow (1-\ell)\tau_{ij} \qquad (1)$$

When scheduling a task, the agent could initially stores in the pheromone trail, information about the favorability of grouping certain jobs together on a processor since the only salient information used by the agent about the tasks/jobs is their execution time. Thus, scheduling or allocating jobs to processors initially are done at random (or could be favored) that are stored in pheromone trail. It must be emphasis that , because the utilization of a task is tightly couple with a specific processor, it is to realize that there is the need to established a one to one mapping between a pheromone trail and a pair of (*task*, *processor*). Thus the processor identity and the task identity are needed to index the pheromone trail.

6.0 Basic Technique

The list scheduling, denoted by *LS*, is a heuristics strategy that is used in solving MPSP. It is widely used technique that schedules tasks from top to the bottom of the task graph. The idea behind its usage is that it does a topological sort of the dependence DAG and considers when an instruction can be scheduled without causing a stall. It as well, schedules the instruction if it causes no stall and when all its predecessors are already scheduled. One of the critical aspects of a class of scheduling algorithm (*LS*) is how to decide which task is to be scheduled next. This is achieved by assigning priories to the nodes or the edges of the input DAG, and thus the task with the highest priority will be scheduled next. Conversely, the node (or task) with a highest priority is scrutinized for scheduling before a node with a lower priority, however, ties are broken if more than one node has the same priority and here some method is used in addressing. One needs to notice that it will yield a search space of totally n! possible lists, which is simply all the permutations of *n* instructions. List scheduling for all of the permutations considered always yield an optimal solution to MPSP. However, optimal list scheduling is *NP*-completed where heuristics are use when necessary. It worth to say that a task is not ready for scheduling until all its predecessors are scheduled.

The heuristic method adopted for the task-scheduling are based on some priority-ordering scheme, which include two phases: task ordering and task-to-processor allocation. Thus it starts by sorting the tasks according to some priority scheme, followed by the second phase where each task is assigned to the processor which is selected by some scheduling rule. This schedule is referred to as list-scheduling. Task priority here is defined based on task execution times, communication requirements, number of successor tasks among other considered in this work. While the heuristic task assignment rule that could have equally taken into consideration here is breath-first.

The list-scheduling considered in this work is denoted *CPES* and is a modification of the dynamic level scheduling (*DLS*) heuristic. The basic idea of *DLS* is to calculate static task priority based on the *critical path* (*CP*) method and use them to order the tasks and then,

to modify them during the scheduling phase to reflect the partial schedules already generated (the best (task, processor) pair is selected for scheduling). It is important to notice that the calculation of the critical path does not involve communication times, because these depend upon the task-to-processor assignment and schedule. Thus only task execution times l_i are considered. An earliest start (*ES*) heuristic used for computing the start time $st(t_i, p_k)$ for each task t_i and p_k, the task being allocated to the processor with the smallest associated time valve [18]. The valves *ES (est(v))*can be computed in the *O*(n) time.

The dynamic level scheduling (DLS) algorithm are based on the list scheduling approach in which the tasks of the DAG are first arranged as a list such that the ordering of the tasks in the list preserves the precedence constraints. In DLS, the scheduling tasks are constructed by calculating static task priority on the DAG for all the tasks on the top level to the root task based on the critical path method. An optimal ordering of tasks were putted in accordance to the start time $st(t_i, p_k)$ for each task t_i and p_k in order to generate an optimal schedule. In the next step, beginning from the first task in the list, each task is removed and scheduled to a processor that allows an earliest start time. The DLS is a compiled time, static list scheduling heuristic.

However, it worth mention that employing the idea of CP in the DLS causes the algorithm to perform at its best. Although, the algorithm assumes arbitrary computation costs, communication among tasks is ignored here and aim at minimize the number of processor.

Time:=0;
While L ≠ Φ
do for i:=1 to n
do let t_i be the first ready task in L such that
DAG(predecessor(t_i)) ≠ 1 or
(t_1,t_2,.....,t_{i-1}) does not contain a sibling of t_i;
if such an t_i exists
then x(t_i):= time; delete t_i from L;
DAG(predecessor(ti)) - : 1
endif;
enddo ; time := time + 1
enddo ; C_{max} := time

Fig 6. The algorithm assigns schedule times slots to the tasks.

7.0 ACO for Multiprocessor Scheduling

In solving this kind of *NP-hard* problem being face by the multiprocessor schedule problem, we apply ant colony optimization (ACO) algorithm that can aid in finding better minimal execution time and cost effective solutions as fast as possible to resolve possible problems.

This heuristic proposed, thus ACO will try to do a thorough search for solution that will be near optimal in solving the problem that may arise from the delay in communication (the execution time) of executing the tasks scheduled on the processors. Again, this algorithm is proposed in helping the ant system to make decisions in solving the delays that may arise from executing the tasks in order to give fast and cost effective solution in less time.

7.1 Ants' Path-Searching Behavior

Based on the Task Graph (TG) model, our goal is to find a feasible scheduling (G_s) for task graph, which provides the optimal performance subject to the predefined constraints associated to the various node (or tasks).

We therefore introduced heuristic method for solving the multiprocessor schedule problem using the ACO algorithm. Essentially, this ACO algorithm is a multi-agent stochastic decision making process that combines local and global heuristics during the searching process. Each agent traverses and attempts to create a feasible scheduling by selecting the next move probabilistically according to the combined heuristics. The quality of the solution is measured by the overall execution time (or makespan) of the tour, from t_1 and t_n, together with the consideration of the predefined constraints, such as execution cost and edge data to be transmitted (communication cost) and process by the tasks. The quality measurement is used to reinforce the good solutions. The global heuristic information is distributed as pheromone trails on the edges (or E (G)).

However, formulation in a search framework requires the following basic components: representation state, initial state, expansion and goal state with each explained below;

- Representation state: it describes how a search state represents a partial solution. A state in the search space for the scheduling problem is a partial schedule in which a sub-graph of DAG is assigned to a certain number of processor.
- Initial state: Thus the starting state and it is an empty partial schedule.
- Expansion: For expanding a search-state, the first node from the list of ready nodes (the nodes whose predecessors have been scheduled) is selected. The selected node is considered for assignment to each of the available processors. The next node from the list is then selected, and states expansion continues in a similar fashion. The state stops when all of the ready nodes have been considered for assignment.
- Goal state: A goal state is a solution state and hence the terminating point of a search. In this scheduling problem however, it is a complete schedule.

7.2 Solution Construction

A colony of ants concurrently and asynchronously moves through adjacent states of the problem by building paths on TG. They move by applying a stochastic local decision policy that makes use of pheromone trail and heuristic information. They move by applying a stochastic local decision policy that makes use of pheromone trail and heuristic information. By moving, ants incrementally build solutions to the optimization problem. Once an ant has built a solution, or while the solution is being built, the ant evaluates the (partial) solution and deposits pheromone trails on the components or connections it used. This pheromone information will direct the search of the future ants.

Thus, each ant builds, starting from the source node (or task), a solution to the problem by applying a step-by-step decision policy. At each node, local information stored on the node itself or on its outgoing arcs is read (sensed) by the ant and used in a stochastic way to decide which node to move to next. At the beginning of the search process, a constant

amount of pheromone is assigned to all the arcs [Eq1]. When located at node *i* an ant *k* uses the pheromone trails τ_{ij} to compute the probability of choosing *j* as next node.

An ant repeatedly hops from node to node using this decision policy until it eventually reaches the destination node. Due to differences among the ants' paths, the time step at which ants reach the destination node may differ from ant to ant (ants traveling on the shorter paths will reach their destinations faster.

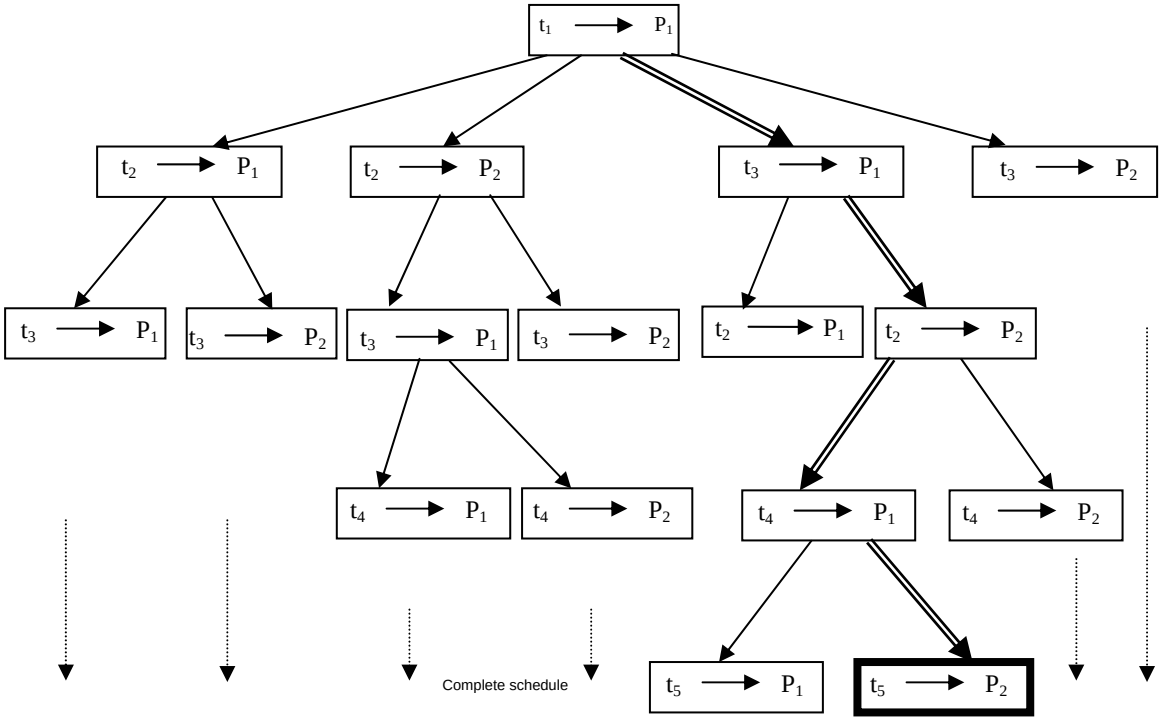


Figure7. The search tree for scheduling the example task graph shown in Fig. 1a, to the 2-processor connected.

On solution construction, assembled set of ready tasks is to be put on the ready processors available. Therefore, the artificial ant stochastically assign the assembled set of ready tasks one by one to the ready processors available, until either all tasks are assigned to some processor, or none of the remaining tasks can be assigned to any processor without exceeding its spare computing capacity. It must however be stated that if an ant stops with at least one unassigned task, the resulted tour is called an infeasible tour. On the otherhand if no stop condition is met, the current tour is called a partial tour.

Once the task to be scheduled next, t_r, has been selected, the set of idle processors M is established, a processor p from this set to which t_r will be scheduled is selected probabilistically with respect to the pheromone between t_r and it, as shown in (equation 4) below;

$$\Pr ob(p) = \frac{[\tau(p, t_r)]^{\alpha}}{\sum_{i=1}^{n} [\tau(p_i, t_r)]^{\alpha}} \forall p_i \in M \qquad (4)$$

However, if no idle processor is available the algorithm simply waits until one becomes available. This process is then repeated until all the tasks, t_i, have been scheduled.

Based on the proceeding observation; an agent[3] in a processor executes each scheduled task in turn until it has completed executing all tasks. Once a task had been fully executed, the next task on the same processor but next on the scheduled is executed by the agent. The agent will repeat this process until it finds the last on the scheduled executed, where it will process and then considered the lowest processing or execution time as solution. In addition, the agent will leave the task it had finished processing and then proceed to the next task based on the intensity of the information stored in the pheromone trail.

Back in the processor, the agent will leave the task scheduled in the first processor and then starts the whole process again on the second processor. However, the two processors' execute its tasks scheduled on it by the agent simultaneously without one preempting the other but along the scheduled. It is however noted that, these actions require only local information and a short memory allowing the agent to recognize its own trial and executes the task next on the scheduled.

[3] The term ''agent'' and ´´ant'' is use interchangeably in this work.

7.3 The algorithm outline

1. Initially, let W_k be the cumulated weight of the complete tour fulfilled by ant *k*, which has selected node *k* as its starting node. Then associated each augmented edge e_{ij} is initialized by a quantity of pheromone, τ_{ij}, defined by

$$\tau_{ij} = \sum_{k=1}^{n} \Delta\tau_{ij}^{k} \qquad (5)$$

where

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{W_k}, & if\ e_{ij} \text{ belongs to the tour performed by ant k;} \\ 0, & \text{otherwise;} \end{cases} \qquad (6)$$

and **Q** is a fixed constraint to control the delivery rate of the pheromone.

the valve of the pheromone on each augmented edge is initially set at the valve τ_0;

2. Put n ants on task node t_0 (thus starting task).
3. Each ant crawls over the *TG* to create a feasible scheduling S^l, where l = 1……n;
4. Evaluate the schedules generated by each of the *n* ants. The quality of a particular scheduling S^l is measured by the overall execution time.
5. Update the pheromone trails on the edges as follows:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \sum_{k=1}^{n} \Delta\tau_{ij}^{l} \qquad (7)$$

where $0 < \rho < 1$ is the pheromone trail evaporation ratio and *n* is the number of distributed ants. The parameter ρ is used to avoid unlimited accumulation of the pheromone trails and enables the algorithm to ´´forget ´´ previously done bad decisions (thus implementing the mean that ´'forgetting´´ solutions which are not reinforced often). On arcs which are not chosen by the ants, the associated pheromone strength decreases exponentially with the number of iterations. $\Delta\tau_{ij}^{k}(t)$ is the amount of pheromone currently laid by ant *k* on the arcs is calculator by Eq(6).

6. If the ending condition is reached, stop and report the best solution found, otherwise go to stop 2.

Step 3 is an important part in the proposed algorithm. It describes how an individual ant ''crawls'' over the TG and generates a solution. In step 3 each ant traverses the graph in the topologically sorted manner in order to satisfy the precedence constraints of task nodes. The trip of an ant starts from t_0 and ends at t_n, the two virtual nodes serving as the nest and the food source respectively. By visiting the nodes in the topologically sorted order, we insured that every predecessor node is visited before we visit the current node and that every incoming edge to the current node has been evaluated.

At each task node t_i where $i \neq n$, the ant makes a probabilistic decision on the allocation for each of its successor task node t_j based on the pheromone on the edge. The pheromone is manipulated by the distributed global heuristic (τ_{ij}) and a local heuristic such as the execution time and the area cost (i.e. communication time of getting a data from one node to its successor node) for a specific assignment of the successor node.

Node transition rule

Moreover, during the running session, the ants moving on the graph travels from node to node through the edges. Because no node can be visited twice, we put the nodes that are already visited in a $tabu_k$ list and mark them as inaccessible to prohibit the ants from visiting any node more than once. For the *k-th* ant on node *i*, the selection of next node *j* to follow is probabilistically determined according to the node transition probability,

$$\phi_{ij}^{k} = \begin{cases} \dfrac{(\tau_{ij})^{\alpha}(\eta_{ij})^{\beta}}{\sum_{h \notin tabu_k}(\tau_{ih})^{\alpha}(\eta_{ij})^{\beta}} & \text{if } j \notin tabu_k; \\ 0, & \text{otherwise;} \end{cases} \qquad (8)$$

where τ_{ij} is the currently pheromone intensity on edge *(i, j)*, η_{ij} is the valve of visibility, defined by 1/*w(i, j)*, α and β are parameters controlling the relative importance of

global and local preference for edge *(i, j)*, and tabu indicates the current set of tasks inaccessible from ant *k*. The visibility is a short-term memory that reflects the local preference for edge *(i, j)*, it is simply determined in a greedy fashion taking into account the local information about the weight *w(i, j)* on edge *(i, j)* only. The probability strategy adopted for ACO to determine the next node to transit to first; the algorithm draws a random number from the interval [0, 1]. Depending on a threshold parameter λ, the algorithm triggers either exploitation or biased exploration strategy. In exploitation (where the random number generated is smaller than or equal to λ), ant *k* will select from the accessible neighbors the node with the largest valve of ϕ_{ij}^{k}. In biased exploration (where the random number generated is greater than λ), the probability of selecting node *j* out of the accessible neighborhood of node *i* is given in Eq.(8). Therefore another random number is generated from the interval [0, 1] to determine, in roulette wheels, which accessible node to visit next.

Here $\eta_{ilk} = 1/d_{ij}$ is local heuristic (heuristic information), If $\alpha = 0$, the selection probability is proportional to $\left[\eta_{ij}\right]^{\beta}$ and the nearest node will more likely be selected; in this case the ant system corresponds to a stochastic classical greedy search. If $\beta = 0$, only pheromone amplification is at work and it leads to the rapid emergence of a *stagnation* situation with the corresponding generation of tours that is strongly suboptimal. Search stagnation is defined as situation whereby all the ants follow the same path and construct the same solution.

It is intuitive to notice that the probability ϕ_{ij}^{k} favors an assignment and a route that yields small execution time, and assignment and route that corresponds with stronger pheromone.

The above decision making process is carried on by the ant until all the task nodes in the graph have been allocated. At the end of the each iteration, the pheromone trails on the edges are updated according to step 5. First, a certain amount of pheromone is evaporated. From optimization point of view, the evaporation step helps to escape from local minimums. Secondly, the *good edges* are reinforced. This reinforcement creates

additional pheromone on the edges that are included on the scheduling solutions that provide shortest execution time for the task graph. In each run of the algorithm, multiple iterations of the above steps are conducted until we reach the ending condition predefined. It worth to state that a schedule *s* is feasible if and only if the algorithm that constructs *s* finishes at iteration $k = n$. This means that all tasks could be scheduled since exactly one task is scheduled at each iteration. The complexity of the algorithm is $O(n^2.m)$ if we stop the algorithm after iteration.

8.0 Reinforcement of Pheromone Trail

In the second stage of our algorithm, the lists constructed by the ants are evaluated and the best solutions after ants had completed traversing the paths are reinforce with pheromone trail in order to generate an optimal solution. Based on this evaluation, the pheromones are adjusted to favor better solution schedules. The hope is that further iterations will benefit from the adjustment and come up with better solution schedule. In generating an optimal solution, the ants were allowed again to traverse. The quality of the scheduling result from ant *h* is judged by the schedule latency L_h. Upon finishing of each iteration, the pheromone trail is updated according to the quality of schedule lists. In the mean time, a certain amount of it will be evaporated.

Thus to make the ants search more competitive, we introduce the *elitist strategy* in order to improve upon the solution quality. It consists in giving the best schedules/tour (called T^{gb}, where gb stands for global-best) a strong additional weight. That is each time the pheromone trails are updated, those belonging to the edges of the global best schedule/tour get an additional amount of pheromone. For these edges, the equation in step 5 of the algorithm above becomes;

$$\Delta\tau_{ij}^{gb}(t) = \begin{cases} e / L^{gb}(t) & \text{if arc e}\left(t_{i,} t_j\right) \in T^{gb} \\ 0 & \text{otherwise} \end{cases} \qquad (9)$$

The arcs of T^{gb} are therefore reinforced with a quantity of $e \cdot 1/L^{gb}$, where L^{gb} is the length of T^{gb} and $\boldsymbol{e}$ is a positive integer.

The two important operations are taken in this pheromone trail updating process. The evaporation operation is necessary for ACO to be effective and diversified to explore different parts of the search space, while the reinforcement operation ensures that the favourable instruction orderings receive a higher volume of pheromone and will have

better chance to be selected in the future iterations of the algorithm. The above process is repeated multiple times until ending condition is reached.

In this extended algorithm, scheduling optimization is formulated as an iterative searching process. Each iteration consists of two stages. First, AS algorithm is applied in which a collection of ants traverse on the DAG with best solutions while it has been reinforce with pheromone to construct instruction lists using global and local heuristics associated with the DAG nodes. Here, to exploit the ant's search experiences the pheromone update is made a function of the solution quality achieved by each particular ant. In this ACO algorithm an elitist strategy whereby the best solutions found during the search strongly contributes to pheromone trail updating.

8.1. Pheromone Updates based on solution quality

In ACO, the ants memorize the nodes they visited during the search path, as well as the cost of the arcs traversed on the weighted graph. They can therefore evaluate the cost of the solutions they generate and use this evaluation to modulate the amount of pheromone they deposit while traversing it. However, making pheromone update a function of the generated solution quality helped in directing future ants more strongly toward better solutions. Further, each ant deposits or removes some amount of pheromone on the visited arcs. This mechanism provides a way of the indirect communications to share the knowledge about searching for good solutions amongst the colony. We prefer the arcs that constitute minimum cost path derived thus far. Therefore at the end of each iteration, we undertake a global pheromone update on the arcs of the best path derived in the most recent iteration. We called such path for simplicity as *global-best* or *iteration-best path*.

However, to keep away from early convergence (also called stagnation), a situation in which all the ants reconstruct the same solution and stop exploring new possibility before some satisfactory solution is found, we perform a local (step-by-step) pheromone update. The local pheromone updating rule is defined as;

$\tau_{ij} = (1-\varphi)\tau_{ij} + \varphi\tau_0$, if (i, j) has been selected by some ant,

where $\varphi(0 < \varphi \leq 1)$ is the parameter controlling the degree of the pheromone decay. The valve of τ_0 is set to be the same as initial pheromone valve of the pheromone trail.

9.0 Experiment and Analysis

This section of the report looks into the evaluating the effectiveness and performance by analysis of the proposed Ant Colony Optimization algorithm. This algorithm was coded in C and compiled with linux. This is employed to develop scheduling algorithms suitable for benchmark applications.

9.1 The testbench

We use as testbench synthetics digraphs generated with ANDES-Synth. The benchmark contains direct acyclic graphs generated from;

1. Bellford	2. Diamond1	3. Diamond2	4. Diamond3	5. Diamond4	6. Gauss
7. Iterative	8. MS-Gauss	9. Prolog	10.fft2-b	11.Gauss	12. Divconq
13.Celbow	14.Cstanford	15.ms-gauss	16.qcd2-b	17.SSC2	18.SSC3
19. SSC4	20. SSC5	21. SSC6	22. SSC7	23.SSC8	24. SSC9

Table1, Digraphs generated with ANDES-Synth used as a test for this algorithm

(A)

1) Bellford: it solves the shortest path problem from all nodes to a single destination in a weighted directed graph. The graph represents the algorithm called Bellman-Ford.

2) Diamond1: thus know as a space-time digraph representing a systolic computation [26].

3) Diamond2: this digraph is known as a systolic matrix multiplication [27].

4) Diamond3: the systolic computation of the transitive closure of a relation on a set of elements is this digraph [28].

5) Diamond4: it's the systolic computation of the transitive closure of a relation on a set of elements [29].

6) Iterative: it's generic iterative algorithm with each iteration being represented in the same level of the digraph. Concerning the next iteration, the immediate successors of a task t_i at kth position are tasked at (k+1)th position.

7) FFT: that's Unidimentional fast fourier transform.

8) Divconq: This stands for divide and conquer algorithm. The digraph has a tree shape.

9) Prolog: The structure of this digraph is obtained at random and corresponds to the resolution of a logic program.

10) MS-Gauss: represents a computation containing successive resolutions of linear systems by Gaussian elimination.

11) Gauss: the task digraph describes the execution of a Gaussian elimination used in the resolution of linear systems.

12) qcd: thus gradient method for linear systems [30].

(B)

13) SSC, Celbow and cstanford: comes from a controller.

Diagram		Number of tasks	Cost of a task t_i	Weight of the arc (t_i, t_j)
bellford	l	992	10000	400
	m	354	10000	400
diamond1	l	1026	10000	400
	m	258	10000	400
diamond2	l	1227	5000	2000//50000
	m	486	5000	2000//50000
diamond3	l	1002	10000	400//800
	m	731	10000	400//800
diamond4	l	1002	5000	2000
	m	731	5000	2000
divconq	l	766	10000	40//80//160//320//640//1280//2560//5120//10240//20480
	m	382	10000	40//80//160//320//640//1280//2560//5120
fft	l	1026	5000	800
	m	194	5000	800
prolog	l	1313	10000	4000
	m	214	10000	4000
iterative	l	938	2500	100//25600
	m	262	2500	100//25600
gauss	l	1227	10000	4000
	m	782	10000	4000
qcd	l	1026	100000	4000
	m	326	100000	4000
ms-gauss	l	1482	1000//128000	400//51200
	m	768	1000//68000	400/51200
celbow	-	103	10//100//320//360//400 280//440//350//120//50 230//240	25
cstanford	-	90	1//5//10//28//57//66/38 15//24//40//32//53//12 39//4//6//111//84//28 29//0	25//10
ssc	5	200	1	25
	6	288	1	25
	7	392	1	25
	9	648	1	25

Table2: *This summarizes the features of each DAG*

The table 2 above shows the size of these graphs, given by the number of tasks, as well as their normalized execution and communication time. The tasks execution times corresponds to the number of integer operations executed, and the communication times corresponds to the number of integers transferred. In order to obtain the actual executions, we multiply the valves in the table by 287 μs. Therefore the communication of L integers cost (129 + 2430450L) μs being result of simulating a IBM SP-1[24][25]. The actual communication times can be obtained by replacing L by the edges' weights. The digraphs were used with two sizes, given as *–m* for middle size while *–l* for the largest size. The tested PC configuration is: Intel Pentium IV, 1.65MHZ and 512M RAM machine running Linux.

9.1.1 Results

The following figures illustrate the evaluation of the performance of most DAG in the suites in 10000seconds. However, due to a limit set, near optimal solution might have arrived with small deviation or error from the optimal.

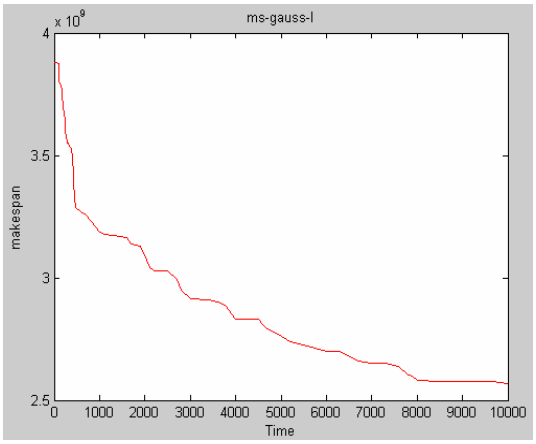


fig10

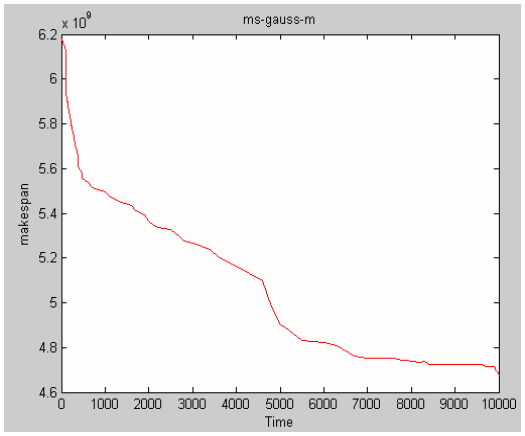


fig11

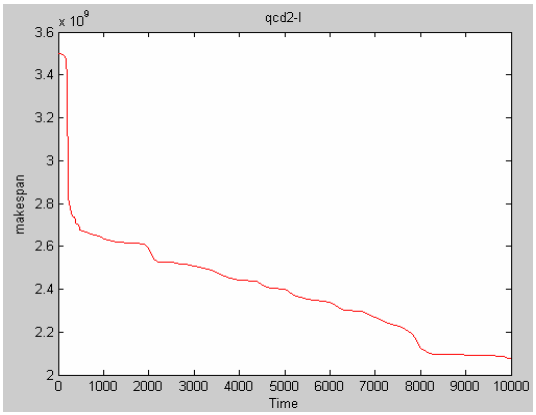


fig12

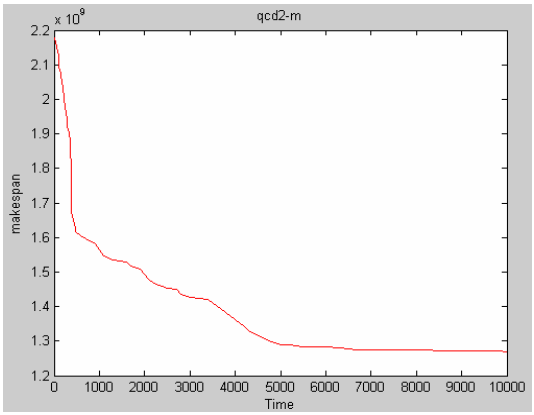


fig13

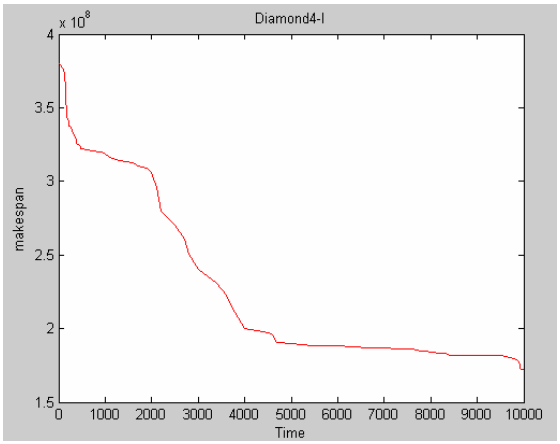


fig14

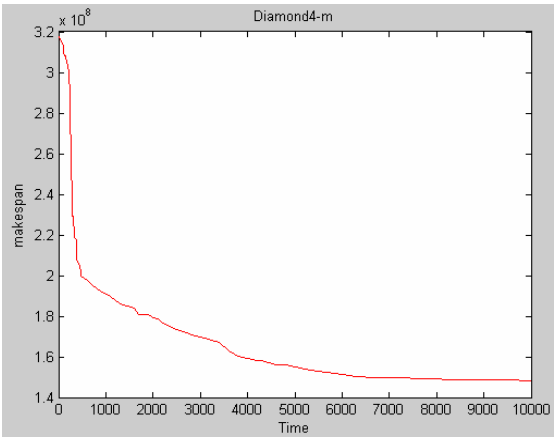


fig15

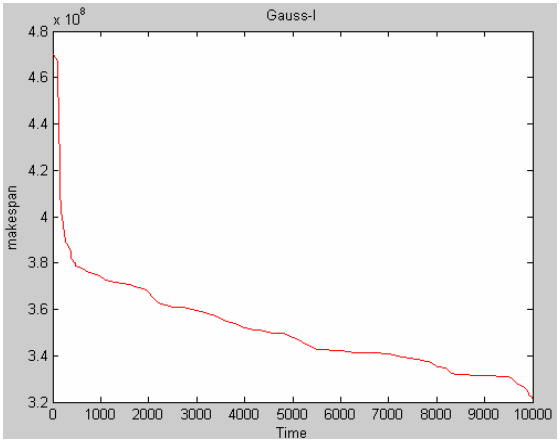


fig16

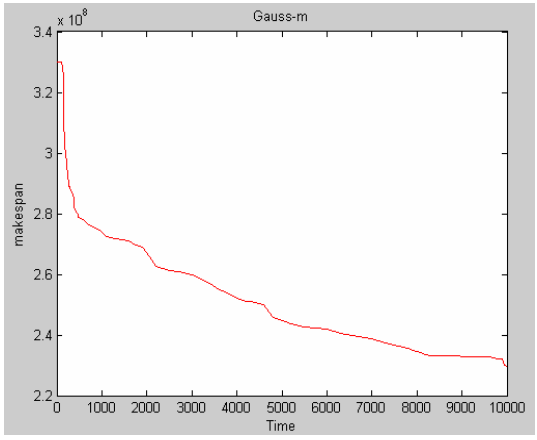


fig17

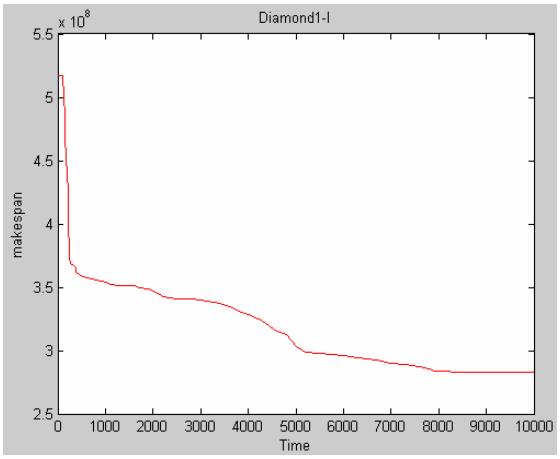


fig18

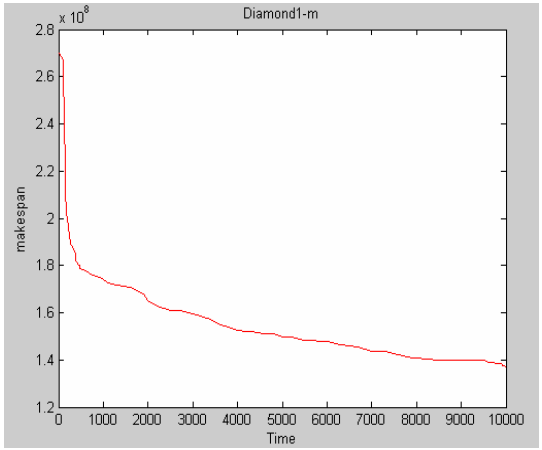
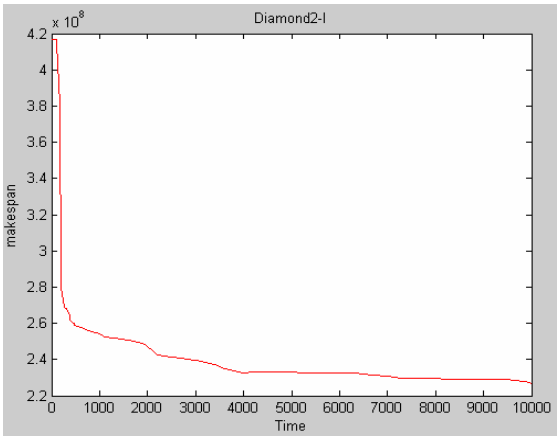


fig19



´fig20

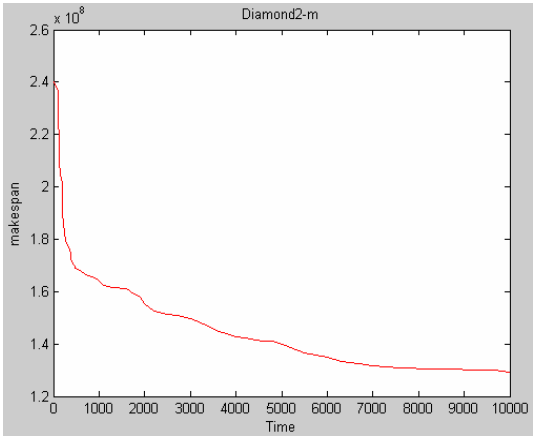


fig21

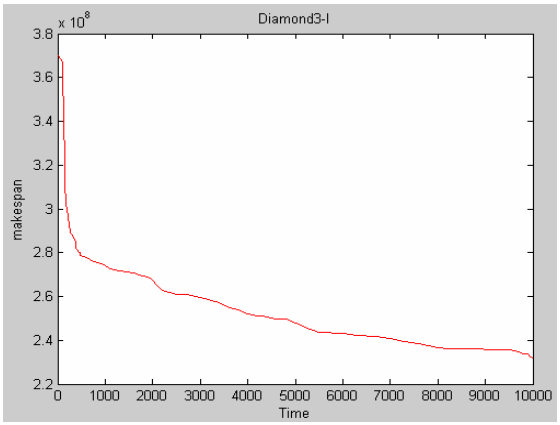


fig22

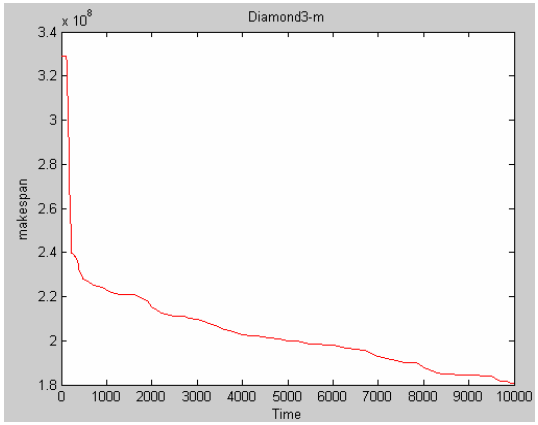


fig23

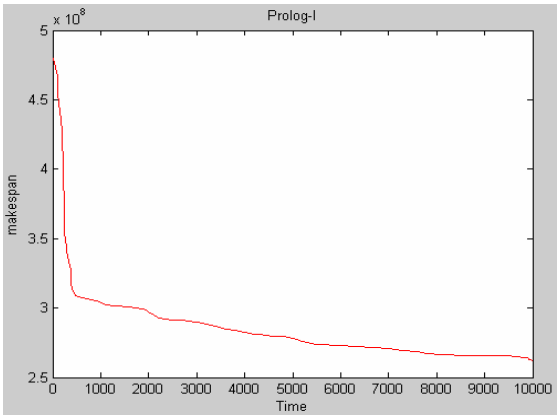


fig24

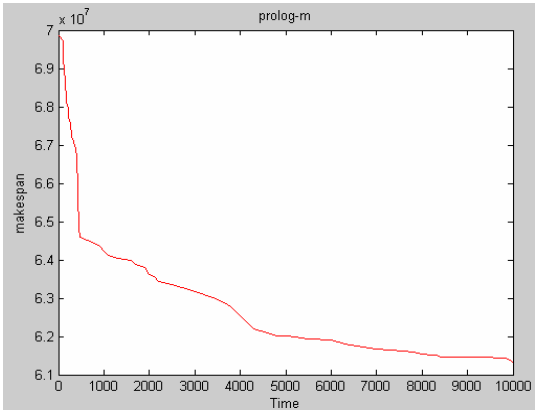


fig25

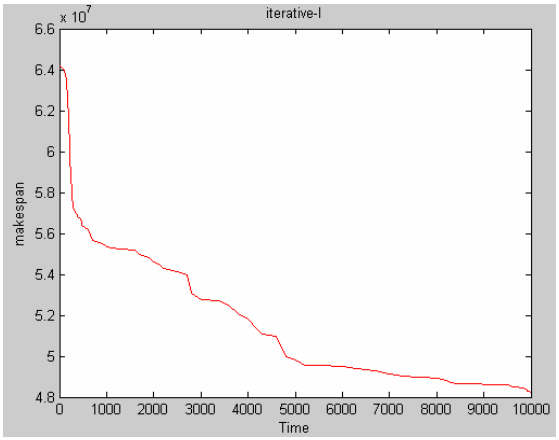


fig26

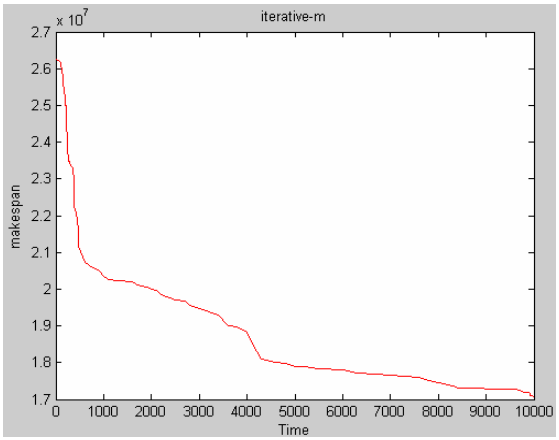


fig27

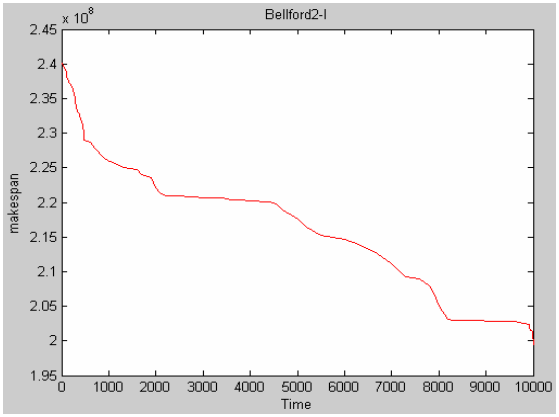


fig28

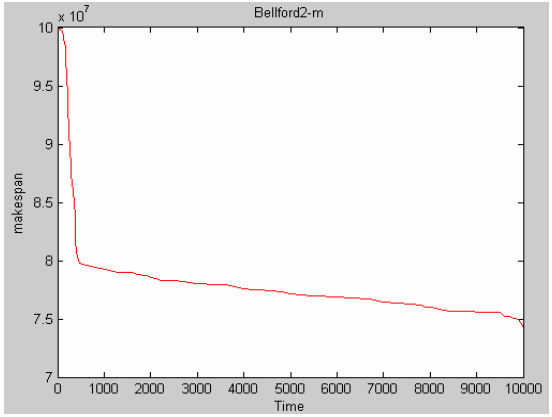


fig29

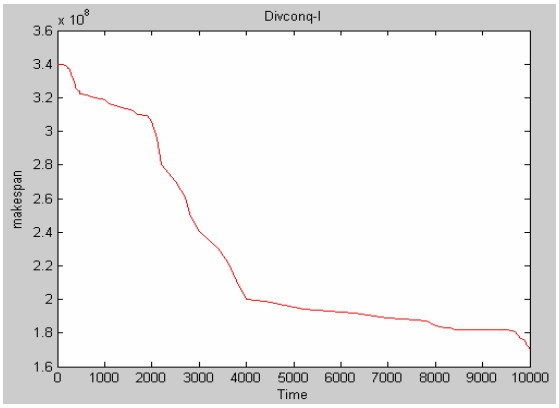


fig30

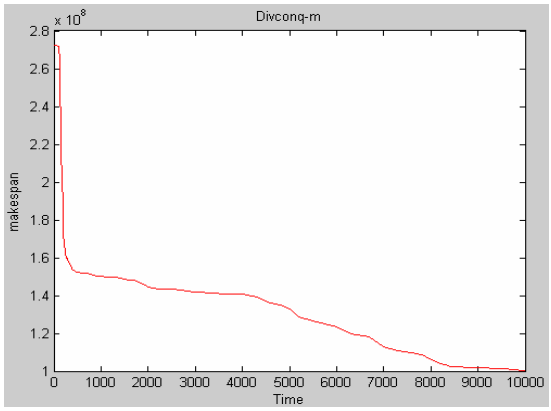


fig31

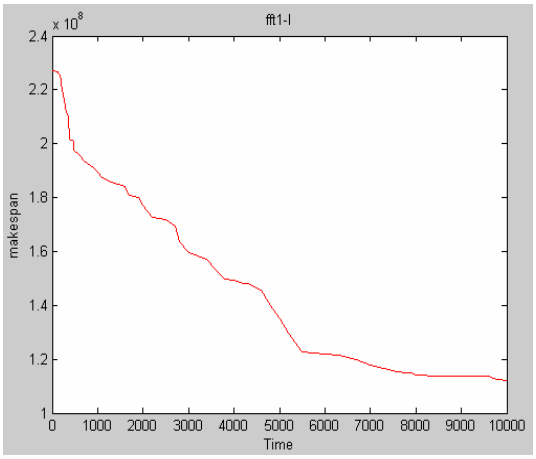


fig32

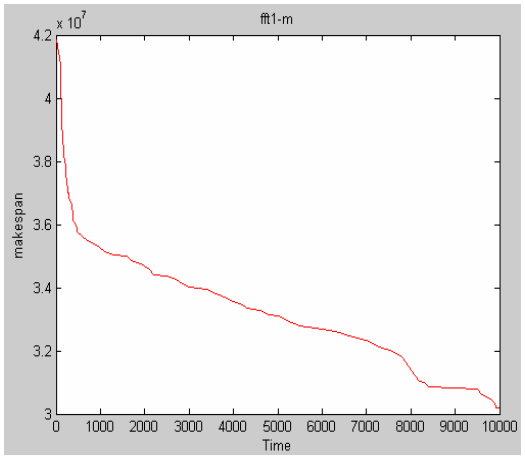


fig33

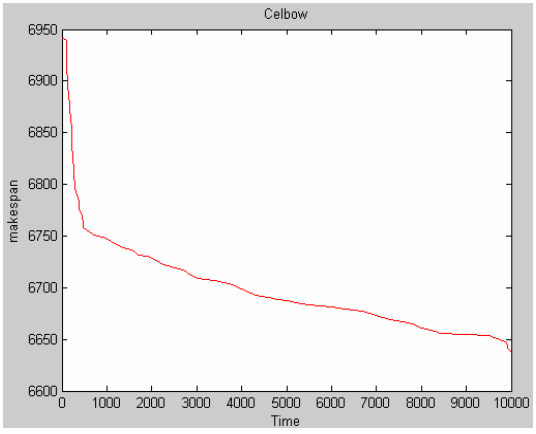


fig34

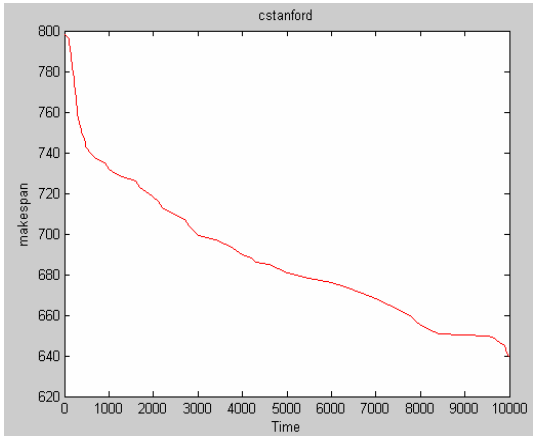


fig35

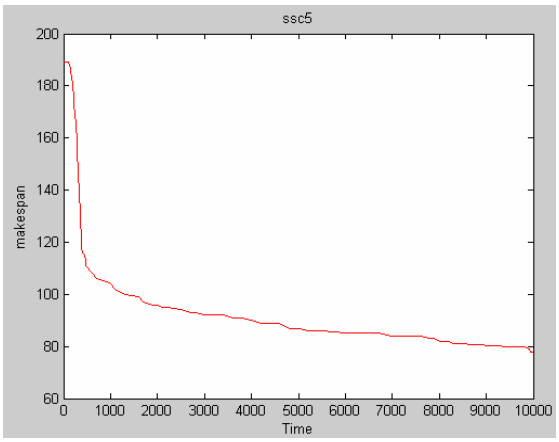


fig36

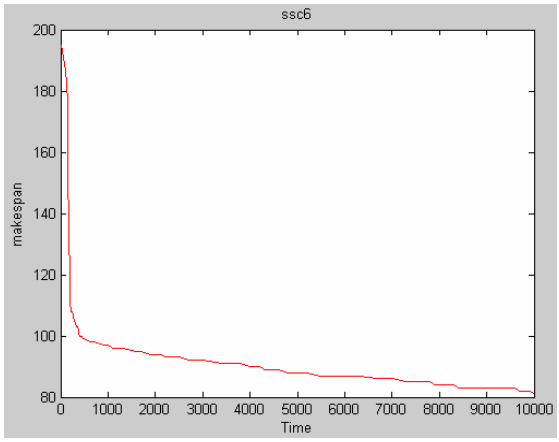


fig37

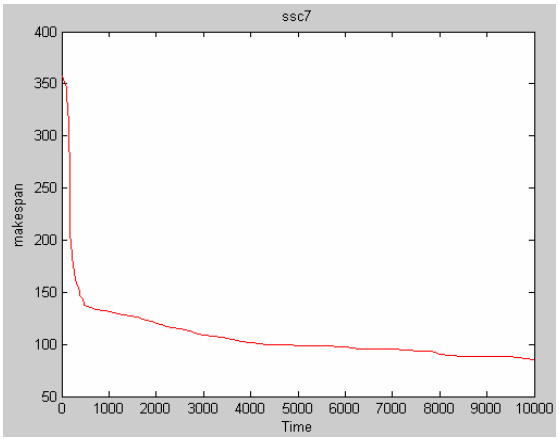


fig38

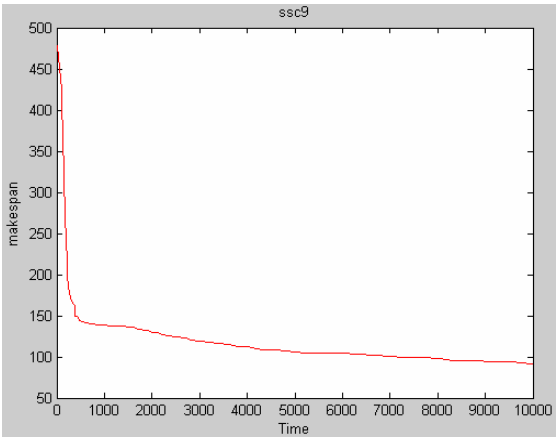


fig39

9.1.2 Result Analysis

From the figures, ACO algorithm found the near optimal execution time throughout in the search. Base on this, the probability of finding the solution with this algorithm for these task graphs is 86.44%. Related to this, we found that for 18 testing examples, or 78% of the testing set, our algorithm discovers the optimal schedule every time in the 10 runs. This indicates that the proposed algorithm is statistically robust in finding close to optimal solution. Also over 90% of the solutions found by the ACO algorithm are within 20% of all possible schedules. The number of solutions drops quickly showing that the ACO algorithm finds very good solutions in almost every run.

For each of the benchmark samples, the proposed algorithm was run with the following parameter choice of the local heuristics. For the choice, we perform 10 runs where in each run, the algorithm is set to iterate for 10000 seconds as stopping condition.

In all the experiment, we test ACO by using different parameter to determine the most advantageous setting as well as to justify the performance of the artificial ants in search for good solution. These settings were obtained from pre-knowledge as well as tuning. The number of ants per iteration set to 6. The evaporation rate, ρ, is configured to be 0.98. The scaling parameters for global and local heuristic are set to be $\alpha = \beta = 1$, the delivery rate Q = 10 and $\tau_0 = 250$. The solution with the best execution time found by the ants is reported as the result of each run.

Among the instances that could not be work out within a shorter time due to a continual decreasing/diminishing at an increasing rate at that stages to get close to an optimal solution, it was work out when the execution time was extended to 10000 seconds after which it start converging. It was therefore realized that, finding close to optimal makespan after the algorithm had iterates within 10000seconds, it start to converge.

Concerning a search for optimal solution, the final result is within 87.14% of the initial look up result. We can as well observe from the graphs (figure 10-37) that the search at all time shows improvement till the ‘good’ solution is found in most cases around 90% through of the calculation.

The table 3 summarized the results for the ACO algorithm comparing with other best known result for benchmarking with running times given in seconds. The meaning of the representation on the table 3, Hybrid Genetic Algorithm (GA) is as follows:

GA is search technique that is designed based on the concept of evolution. A GA is applied with its three genetic search operators – Selection, Crossover and Mutation (explained below) – to transform a population (thus set of individual) of chromosomes (individual) with the objective of improving the quality of the chromosomes. A genetic algorithm is usually employed to determine the optimal solution of specific objective function.

The search space therefore is defined as the solution space so that each feasible solution is represented by a distinct chromosome. Before the search starts a set of chromosomes is randomly chosen from the search space to form the initial population. The three genetic search operations are then applied one after the other to obtain a new generation of chromosomes in which the expected quality over all the chromosomes is better than that of the previous generation. This process is repeated until the stopping criterion is met and the best chromosome of the last generation is reported as the final solution.

Worth to mention that in order to solve a problem with GA, we must know how to generate a solution, to modify it, to mix two solutions and to estimate its quality. However, this is done by applying the following three operations to the population:

- The selection operator: this favors ‘’good’’ individual. It is done by duplicating ‘’good’’ individual in the population in order to suppress ‘’bad’’ ones. The fitness function is employed to compute the quality of an individual.
- The Crossover operator: The aim of this crossover is to mix individual (thus solutions). It is employed with an idea that mixing two good individual might result in generating a better individuals.
- The mutation operator: This is modification of a solution in order to explore all the search space and avoid keeping in a neighborhood of the solution space.

Diagram			ACO	Hybrid GA	c\o change from GA
1	bellford2	*l*	199355100	193794250	2.86
		m	74231330	71936050	3.19
2	diamond1	*l*	283205116	276758950	2.32
		m	136667210	131940750	3.58
3	diamond2	*l*	226689430	218954400	3.53
		m	129123060	127224450	1.49
4	diamond3	*l*	231439840	228590500	1.24
		m	180333810	176982100	1.89
5	diamond4	*l*	172451790	164264000	4.98
		m	137950660	132875550	3.81
6	divconq	*l*	170342660	169043350	0.76
		m	100156230	97307880	2.92
7	fft1	*l*	107711210	102647300	4.93
		m	30173540	29888250	0.95
8	gauss	*l*	321135200	307395800	4.49
		m	229344410	226882900	1.08
9	iterative1	*l*	48212350	47936700	0.57
		m	17072370	16733350	2.02
10	ms_gauss	*l*	2567661200	2559388750	0.32
		m	4674566400	4598559550	1.65
11	prolog	*l*	261845600	258529350	1.28
		m	61322350	60499350	1.36
12	qcd2	*l*	2075873960	2038576050	1.82
		m	1204561020	1173100600	2.68
13	celbow		6637	6630	0.10
14	cstanford		663	627	5.74
15	ssc	*5*	78	74	5.40
16		*6*	81	77	5.19
17		*7*	85	82	3.65
18		*9*	91	89	2.24

Table 3: A benchmark of the ACO algorithm (Parallel Solution)

9.2 Discussion

As seen from the table (a percentage change from GA), the ACO demonstrated competitive results as compared to the best know solution from GA for both large and medium digraphs. The difference of ACO best makespan is not much from the GA and the difference can be attributed to parameter settings, maximum time to iterate and margin of error, evidence by a much smaller deviation over the results.

ACO/TG model makes effective use of the core structural information of the problem. The autocatalytic nature of how the pheromone trails are updated and utilized makes it more attractive in discovering ‘’ideal’’ solutions with short computing time, this very behavior raises stagnation problem. For instance, it is observed that allowing extra computing time after enough iterations of ACO algorithm does not have significant benefit regarding to the solution quality.

The ACO approach which combines a probabilistic search guided by heuristic problem specific information and a simply form of information sharing about good solutions, with global and local heuristic performs creditably in generating the parallel solution. The ants essentially perform an adaptive greedy search of the solution space.

The very important choice when applying ACO is the definition of the intended meaning of the pheromone trails. Explaining these issues with an example, when applying ACO to the MPSP competitive results were obtained when used the absolute position interpretation of the pheromone trails, where $\Delta\tau_{ij}$ is the desirability of putting task *j* on the *ith* position. The best solution found so far and the best solution found in the current iteration is then used to update the pheromone. But before that some of the old pheromone on all the edges according to $\tau_{ij} = (1-\rho) \bullet \tau_{ij}$. The result for this is that the old pheromone should not have a too strong influence on the future. Then for every activity $j \in J$ some amount of pheromone is added to element *(i, j)* of the pheromone

where *i* is the place of activity *j* in the best solution found so far. This is an elitist strategy that leads ants to search near the best found solution.

However, with effect of pheromone update, the pheromone trails on the arcs of the best path evaporates regularly in each iteration and receive additional pheromone whenever ants visit them again. Ants' rapid concentration upon the best path with a large possibility and average pheromone intensity in this case is a compromised result among the evaporation and reinforcement from global updates as well as the decay via local updates. This effect of decay in local pheromone trail update process is to reduce the risk that an ant might mostly follow the same path traverse by its predecessors. Whenever an ant moves along an arc, decay occurs on the traverse arc.

In balancing the exploitation and exploration as any metaheuristic algorithm has to achieve an appropriate balancing between the exploitation of the search experience gather so far and the exploration of unvisited or relatively unexplored search spaces. In ACO, it is typically through the management of the pheromone trails; the pheromone trail induce a probability distribution over the search space and determine which parts of the search space are effectively sampled, that is, in which part of the search space the constructed solutions are located with higher frequency. In line with this, the elitist strategy is enforcing whereby the best solutions found during the search strongly contribute to pheromone trail updating. Also important for the role in the balancing of exploration and exploitation is that of parameters α and β, which determine the relative influence of the pheromone trail and heuristics information. Considering the influence of parameter α, thus the larger the valve of α (α>0) the stronger the exploitation of the search experience. However the pheromone trail is not taken into account when α=0 whereas for α<0, most probable choices done by the ants are those that are less desirable from the point of view of the pheromone trails. Therefore varying α is used to shift from exploration to exploitation and vice visa. The parameter β determines the influence of the heuristic information in a similar way and systematic variation of α and β is useful strategies to balance the exploration and exploitation.

On the part of importance of heuristic information to direct the ants' probabilistic solution construction is important because it gives the possibility of exploiting problem specific knowledge. In this static problem, the heuristic information η is computed once at the initialization time and then is the same throughout the whole algorithm's run. An instance is use in this work (MPSP) application, of the length d_{ij} of the arc connecting tasks *i* and *j* to define heuristic information $\eta_{ij} = 1/d_{ij}$. This static heuristic information has merits as such (i) easy to compute (ii) it has to be computed only once at the initialization time and (iii) in each iteration of the ACO algorithm, it is pre-computed with the valves of $\tau_{ij}(t) \bullet \eta_{ij}^{\beta}$, which result in a significant saving of computation time.

The time complexity of this scheduling under the ACO algorithm is $O(mn^2)$ and requires $O(mn)$ storage space. Thus precedence constraints between jobs (tasks) that have to be respected in every feasible schedule generally increase the computational complexity of a scheduling problem. However, it worth mention that occasionally, their introduction may turn a problem that is solvable within a polynomial time into an *NP*-complete one, on which a good algorithm is highly unlikely to exist.

10.0 Conclusion and Future Work

In this work, we present a novel heuristic searching method for the resource constrained instruction scheduling problem based on the ant colony optimal algorithm. This algorithm works as a collection of agents collaborate to explore the search space. A stochastic decision making strategy is proposed in order to combine global and local heuristics to effectively conduct this exploration. As the algorithm proceeds in finding better quality solution, dynamically computed local heuristics are utilized to better guide the searching process.

The work concerns scheduling tasks on multiprocessors whereby more processors are involved for a single program execution. The goal or aid is to enhance the fastest of the execution of a DAG or program by allotting various tasks to different processors concurrently in order to balance the assignment to each processor so that they can obtain a minimal makespan in completing their processing.

A best found solution that was stable for many iteration has a great influence on the pheromone valves since applying the an elitist strategy when doing the pheromone update, that is pheromone is added after every iteration along the best found solution. Thus, during long runs it can happen that the algorithm converges too early to the best found solution.

The definition of the pheromone trails is crucial and poor choice at this stage of the algorithm design will probably result in poor performance. Fortunately, for many problem instances as this, the intuitive choice is also a very good one.

The parameter settings in ACO is quiet tricky, it means that ACO needs to be tuned, in accordance with the characteristics of the studied problem, to establish the appropriate search strategy. When tuning the parameters, we examine the behaviour of ACO in optimizing the problem. These experiences might however encourage to use the ACO algorithms to deals with other combinatorial optimization problems.

A relative order of the solution components is important due to the role that permutations $\pi = \{1,2.....n\}$ have in the problem. This makes the absolute position based pheromone trail appropriate. Further, the considered algorithm is more effective in finding the near optimal solutions and scales well irrespective of the problem size. It is also shown that with substantial less execution time the proposed method achieve a competitive solution.

10.1 Future Work

Future work needs to be bordered on how ACO can be applied on dynamic and undefined combinational optimization problem. Currently, the large parts around 96% of problems attacked by ACO are static and well-defined combinational optimization problem, that is, problems for which all the necessary information is available and does not change during problem solution.

As a second future finding, an investigation into the effect of α to the stagnation behaviour would interest in order to have a good selection of the trail. Thus, it was observed that for high valves of α the algorithm enters stagnation behaviour very quickly without finding very good solution. We also realised that a high β valve though provide good solutions quickly, but a lower valve provide better results with a longer period of time as a price for the choice. There is a need therefore to experiment with changing large range valves of α and β in order to ascertain which work best for the problem types. However, from the proceeding observations, the issue that deals with the setting of parameter in ACO algorithms must be thoroughly be scrutinized. In our experiement, the parameters given here performed well over a wide range of instances however the solution stands a better chance of improves further when the parameter setting is well addressed. Nevertheless, in other applications adaptive versions which dynamically tune the parameters during algorithm execution may increase algorithm robustness.

Finally, we definitely need a more thorough understanding of the features the successful application of ACO algorithm depend on and how ACO algorithm should be configured for specific problems. Particularly, the following questions need to be examined as future work: The solution components that will yield a better result for the problem instance. What best way the pheromone can be managed. For instance, incorporating different styles of convergence, pheromone updating rules and colony relationship are worthy of further research.

Reference:

[1] H. Kasahara and S. Narita. ‘’Practical multiprocessor scheduling algorithms for efficient parallel processing’’, IEEE Trans. Comput., C-33(11):1023-1023-1029, Nov. 1985.

[2] D.Merkle, M. Middendorf and H. Schmeck. Ant Colony optimization for resource-constrainted project scheduling. In Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), page 893-900. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[3] R. Michel and M. Middendorf. An ACO algorithm for the shortest supersequence Problem. In D. Corne, M. Dorigo and F. Glover, editors, new ideas in optimization, Pages 51-61. McGraw Hill, London, UK, 1999.

[4] T. Stutzle and M. Dorigo. ACO algorithms for the quadratic assignment problem . In D. Corne, M. Dorigo and F. Glover, editors, new ideas in optimization, Pages 33-50. McGraw Hill, London. UK, 1999.

[5] M. Dorigo and T. Stutzle, Ant Colony Optimization, MIT Press, 2004.

[6] S. Baruah, Task partitioning upon heterogeneous multiprocessors, RTAS, 2004.

[7] Gang Wang, Wenrui Gong and Ryan Kastner In Instruction scheduling Using Max - Min Ant System Optimization.Universty of California at Santa Barbara, CA 93106-9560.

[8] T. Stutzle and H. H. Hoos. Max – Min Ant System. Future Generation Computer Systems, 16(8):889-914, 2000.

[9] G. Navarro Varela and M. C. Sinclair. Ant Colony optimization for virtual-wavelength-path routing and wavelength allocation. In proceedings of the 1999 congress on Evolutionary Computation (CEC’99), pages 1809-1816. IEEE Press, Piscataway, NJ, 1999.

[10] J. D. Ullman. ‘’NP-complete scheduling problems’’, Journal of Computer and System Sciences, 10(3): 384-393, Jun. 1975.

[11] T. Yang and A. Gerasoulis. ‘’ DSC: Scheduling parallel tasks on an unbounded number of processors’’, IEEE Trans. Parallel and distributed Systems, 5(9):951-967, sep. 1994.

[12] H. Kasahara and S. Narita. ‘’Practical multiprocessor scheduling algorithms for efficient parallel processing’’, IEEE Trans. Comput., C-33(11):1023-1029, Nov 1985.

[13] K. M. Chandy, C. V. Ramamoorthy and M. J. Gonzalez Jr. ‘’Optimal Scheduling strategies in a multiprocessor systems’’, IEEE Trans. Comput., C-21(2):137-146, 1973.

[14] E. G. Coffman, ‘’ Computer and Job-Shop Scheduling Theory’’, Wiley, New York, 1976.

[15] Marco Dorigo, ‘’ Ant Colony Optimization Metaheuristic: Algorithm, Applications, and advances’’. Technical Report IRIDIA-2000-32.

[16] Hoogeveen, J. Lenstra, B. Veltman, ‘’Three, four, five, six or the complexity of scheduling with communication delays’’, IEEE Transactions on computers 16(1994) 129-137.

[17] T. L. Adam, K. M. Chandy and J. R. Dickson, ‘’A comparison of list schedules for parallel processing systems’’, communication of the ACM 17 17 (1974) 685-690.

[18] Ahmad and Y-K Kwok, ‘’A new approach to scheduling parallel programs using task duplication’’, Proc. Of Int’l Conf. Parallel Processing, vol. II, pp. 47-51, Aug. 1994.

[19] J.Y. Colin and P. Chretienne, ‘’C.P.M. Scheduling with small computation delay and task duplication’’, operation research, pp 680-684, 1991.

[20] H.H Ali and H. El-Rewini, ‘’The Time Complexity of Scheduling interval Orders with Communication Polynomial, ‘’ Parallel Processing Letters, vol. 3, no. 1, 1993, pp. 53-58.

[21] Graham Ritchie, ‘’Static Multi-processor Scheduling with Ant Colony Optimization and Local Search’’, School of Informatics, University of Edinburgh.

[22] M. Fox and D. Long, ‘’Multi-processor scheduling problems in planning. International Conference on AI, IC-AI’01, Las Vegas, 2001.

[23] F. Ducatelle, ‘’Ant Colony Optimisation for bin packing and cutting stock problems. Master’s thesis, University of Edinburgh.

[24] B. Chamaret and P. Rebreyend, ‘’ Static Multiprocessor Task Graph Scheduing: A comparison of several hybrid Genetic Algorithm’’.

[25] R.C.Correa, F. Afonso and P. Rebreyend, ‘’Integration list heuristics into genetic algorithms for multiprocessor scheduling’’.

[26] O. Ibarra and S. Sohn. ‘’ On mapping systolic algorithms onto the hypercube’’, IEEE Transactions on Parallel and Distributed systems, vol. 1, no. 1, pp. 48-63, Jan. 1990.

[27] P.-Z. Lee and Z. Kedam, ‘’Mapping nested loop algorithms into multidimensional systolic array’’, IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 64-76, Jan. 1990.

[28] C.-T. King and W.-H. Chou, L. Ni, ‘’Pipelined data-parallel algorithms: Part II – design’’, IEEE Transactions on Parallel and Distributed Systems, vol. 1, no. 1, pp. 486-499, Oct. 1990.

[29] C. Scheiman and P. Cappello, ‘’A processor-time-minimal systolic array for transitive closure’’, IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 3, pp. 257-269, May. 1992.

[30] D. Bertsekas and J. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall International Editions, 1989.

[31] Y. Zinder, V. H. Do and C. Oguz, Computational Complexity of some scheduling problems with multiprocessor tasks. Doi:10.1016/j.disopt.2005.08.001, august 10, 2005.

[32] R. S. Sutton and A. G. Barto. Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA, 1998.