Task Scheduling On Multiprocessor System Using Ant Colony Optimization

B. Karthik[1], S. Javeed Basha[2], B. Abdul Rahim[3], D. Vishnu Vardhan[4]

Department of ECE, AITS, Rajampet.[1,2,3]

Department of ECE, JNTUACE, Kalikiri.[4]

***Abstract:**Ant Colony Optimization algorithm, is a viable method for solving hard combinatorial optimization problems. The behaviour of ant colonies is very interesting and highly structured due to the coordinated interactions among the ant colonies. Among the ant colonies the communication is limited and therefore the coordinated interactions must be a simple flow of informative data. In this paper we investigate the implications in the study of antcolony behaviour can have on problem solving and optimization by using generic ant colony algorithm and modified ant colony algorithm with travelling sales man problem.*

Keywords: Scheduling, Optimization, Ant colony optimization, TSP, Multiprocessor system.

I.INTRODUCTION

In Engineering and research, the **ant colony optimization (ACO)** algorithmisa probabilistic technique used for solving computational problems which can be reduced for finding good paths through graphs.

The **ant colony algorithm** is a member of family having swarm intelligent methods, and it constitutes some meta heuristic optimizations. Initially proposed by Marco Dorigo in 1992 in his PhD thesis, the first algorithm was aiming to search for an optimal path in a graph, based on the behavior of ants seeking a path between their colony and a source of food. The original idea has since diversified to solve a wider class of numerical problems, and as a result, several problems have emerged, drawing on various aspects of the behavior of ants.

SCHEDULING AND ITS TYPES

In computing, **scheduling** is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance and share system resources effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (executing more than one process at a time) and multiplexing (transmit multiple data streams simultaneously across a single physical channel)[8].

The scheduler is concerned mainly with the throughput (the total number of processes that complete their execution per time unit), latency (specifically the *turnaround time*, as a total time between submission of a process and its completion, and the response time, as a time from submission of a process to the first time it is scheduled), *fairness* (equal CPU time to each process, or more generally appropriate times according to the priority and workload of each process), and *waiting time* (the time the process remains in the ready queue). In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the concerns mentioned above, depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks can also be distributed to remote devices across a network and managed through an administrative back end[10].

Process scheduler

The process scheduler is a part of the operating system that decides which process runs at a certain point in time

Long-term scheduling

The long-term scheduler, or admission

scheduler, decides which jobs or processes are to be admitted to the ready queue (in main memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is eitherauthorized or delayed by the long-term scheduler.

Medium-term scheduling

The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes.

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes is to be executed (allocated a CPU) after a clock interrupt, an I/O interrupt, an operating system call or another form of signal.

Scheduling disciplines

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers(to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms.

- First in first out
- Earliest deadline first
- Shortest remaining time
- Fixed priority pre-emptive scheduling
- Round-robin scheduling
- Multilevel queue scheduling
- Manual scheduling

II.OPTIMIZATION

Although the word "Optimization" shares the same route as "optimal", it is rare for the process of optimization to produce a truly optimal system. The optimized system will be typical only be optimal in only one application or for one audience[2]. One might reduce the amount of time at the price of making it consume more memory. In an application where memory space is at premium, one might deliberately chooses lower algorithm in order to use less memory. Often there is "no one size fits all" design which works well in all cases. So engineers make trade of to optimize the attributes of greatest interest. Additionally, the effort required to make a piece of software completely optimal incapable of any further improvement of is almost always more than is reasonable for the benefits that would be accrued. So the process of optimization may be halted before a completely optimal solution has been reached. Fortunately, it is often the cases that the greatest improvements come early in the process[2].

ANT COLONY OPTIMIZATION:

Ant Colony Optimization (ACO) is a recently proposed metaheuristic approach for solving hard combinatorial optimization problems. The inspiring source of ACO is the pheromone trail laying and following behavior of real ants which use pheromones as a communication medium. In analogy to the biological example,ACO is based on the indirect communication of a colony of simple agents, called (artificial) ants, mediated by (artificial) pheromone trails. The pheromone trails in ACO serve as a distributed, numerical information which the ants use to probabilistically construct solutions to the problem being solved and which the ants adapt during the algorithm's execution to reflect their search experience. The first example of such an algorithm is Ant System (AS) which was proposed using as example application the well known Traveling Salesman Problem (TSP) .Despite encouraging initial results, AS could not compete with state-of-the-art algorithms for the TSP. Nevertheless, it had the important role of stimulating further research on algorithmic variants which obtain much better computational performance, as well as on applications to a large varietyof different problems. In fact, there exists now a considerable amount ofapplications obtaining world class performance on problems like the quadratic assignment,vehicle routing, sequential ordering, scheduling, routing in Internet-like networks, and so on .Motivated by this success, the ACOmetaheuristic has been proposed as a common framework for the existingapplications and algorithmic variants. Algorithms which follow the ACO metaheuristicwill be called in the following ACO algorithms[1].

Current applications of ACO algorithms fall into the two important problem classes of static and dynamic combinatorial optimization problems. Static problems are those whose topology and cost do not change while the problems are being solved. This is the case, for example, for the classic TSP, in which city locations and intercity distances do not change during the algorithm's run-time. Differently, in dynamic problems the topology and costs can change while solutions are built. An example of such a problem is routing in telecommunications networks, in which traffic patterns change all the time. The ACO algorithms for solving thesetwo classes of problems are very similar from a high-level perspective, but they differsignificantly in implementation details. The ACO metaheuristic captures thesedifferences and is general enough to comprise the ideas common to both applicationtypes.

The (artificial) ants in ACO implement a randomized construction heuristic which makes probabilistic decisions as a function of artificial pheromone trails and possibly available heuristic information based on the input data of the problem to resolved. As such, ACO can be interpreted as an extension of traditional construction heuristics which are readily available for many combinatorial optimization problems. Yet, an important difference with construction heuristics is the adaptation of the pheromone trails during algorithm execution to take into account the cumulated search experience.

Multiprocessor Operating System Types

Let us now turn from multiprocessor hardware to multiprocessor software, in particular, multiprocessor operating systems. Various organizations are possible. Below we will see three of them.

Each CPU Has Its Own Operating System

The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each CPU its own private memory and its own private copy of the operating system. In effect, the *n* CPUs then operate as *n* independent computers. One obvious optimization is to allow all the CPUs to share the operating system code and make private copies of only the data.

This scheme is still better than having *n* separate computers since it allows all the machines to share a set of disks and other I/O devices, and it also allows the memory to be shared flexibly. For example, if one day an unusually large program has to be run, one of the CPUs can be allocated an extra large portion of memory for the duration of that program. In addition, processes can efficiently communicate with one another by having, say a producer be able to write data into memory and have a consumer fetch it from the place the producer wrote it.

Still, from an operating systems' perspective, having each CPU have its own operating system is as primitive as it gets.

It is worth explicitly mentioning four aspects of this design that may not be obvious. First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's tables.

Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes. If a user logs into CPU 1, all of his processes run on CPU 1. As a consequence, it can happen that CPU 1 is idle while CPU 2 is loaded with work.

Third, there is no sharing of pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.

Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones. Thus it can happen that a certain disk block is present and dirty in multiple buffer caches at the same time, leading to inconsistent results. The only way to avoid this problem is to eliminate the buffer caches. Doing so is not hard, but it hurts performance considerably.

Master-Slave Multiprocessors

For these reasons, this model is rarely used any more, although it was used in the early days of multiprocessors, when the goal was to port existing operating systems to some new multiprocessor as fast as possible. Here, one copy of the operating system and its tables are present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called **master-slave** since CPU 1 is the master and all the others are slaves.

The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track of ready processes. When a CPU goes idle, it asks the operating system for a process to run and it is assigned one. Thus it can never happen that one CPU is idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.

The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

Symmetric Multiprocessors

Our third model, the **SMP** (**Symmetric Multi-Processor**), eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it. When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call.

This model balances processes and memory dynamically, since there is only one set of operating system tables. It also eliminates the master CPU bottleneck, since there is no master, but it introduces its own problems. In particular, if two or more CPUs are running operating system code at the same time, disaster will result. Imagine two CPUs simultaneously picking the same process to run or claiming the same free memory page. The simplest way around these problems is to associate a mutex (i.e., lock) with the operating system, making the whole system one big critical region. When a CPU wants to run operating system code, it must first acquire the mutex. If the mutex is locked, it just waits. In this way, any CPU can run the operating system, but only one at a time.

This model works, but is almost as bad as the master-slave model. Again, suppose that 10% of all run time is spent inside the operating system. With 20 CPUs, there will be long queues of CPUs waiting to get in. Fortunately, it is easy to improve. Many parts of the operating system are independent of one another. For example, there is no problem with one CPU running the scheduler while another CPU is handling a file system call and a third one is processing a page fault.

This observation leads to splitting the operating system up into independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it. In this way, far more parallelism can be achieved. However, it may well happen that some tables, such as the process table, are used by multiple critical regions. For example, the process table is needed for scheduling, but also for the fork system call and also for signal handling. Each table that may be used by multiple critical regions needs its own mutex. In this way, each critical region can be executed by only one CPU at a time and each critical table can be accessed by only one CPU at a time.

Most modern multiprocessors use this arrangement. The hard part about writing the operating system for such a machine is not that the actual code is so different from a regular operating system. It is not! The hard part is splitting it into critical regions that can be executed concurrently by different CPUs without interfering with one another, not even in subtle, indirect ways. In addition, every table used by two or more critical regions must be separately protected by a mutex and all code using the table must use the mutex correctly.

Furthermore, great care must be taken to avoid deadlocks. If two critical regions both need table *A* and table *B*, and one of them claims *A* first and the other claims *B* first, sooner or later a deadlock will occur and nobody will know why. In theory, all the tables could be assigned integer values and all the critical regions could be required to acquire tables in increasing order. This strategy avoids deadlocks, but it requires the programmer to think very carefully which tables each critical region needs to make the requests in the right order.

As the code evolves over time, a critical region may need a new table as it did not required previously. If the programmer is new and does not understand the full logic of the system, then the temptation will be to just grab the mutex on the table at the point it is needed and release it when it is no longer needed. How much reasonable this may appear, it may lead to deadlocks, which the user will perceive as the system is freezing. Getting it right is not easy and keeping it right over a period of years in the face of changing programmers is very difficult.

Multiprocessor Scheduling

On a uniprocessor, scheduling is one dimensional. The only question that must be answered (repeatedly) is: "Which process should be run next?" On a multiprocessor, scheduling is two dimensional. The scheduler has to decide which process to run and which CPU to run it. This extra dimension greatly complicates scheduling on multiprocessors.

Another complicating factor is that in some systems, all the processes are unrelated whereas in others they come in groups. An example of the former situation is a timesharing system in which independent users start up independent processes. The processes are unrelated and each one can be scheduled without regard to the other ones.

An example of the latter situation occurs regularly in program development environments. Large systems often consist of some number of header files containing macros, type definitions, and variable declarations that are used by the actual code files. When a header file is changed, all the code files that include it must be recompiled. The program *make* is commonly used to manage development. When *make* is invoked, it starts the compilation of only those code files that must be recompiled on account of changes to the header or code files. Object files that are still valid are not regenerated.

The original version of *make* did its work sequentially, but newer versions designed for multiprocessors can start up all the compilations at once. If 10 compilations are needed, it does not make sense to schedule 9 of them quickly and leave the last one until much later since the user will not perceive the work as completed until the last one finishes. In this case it makes sense to regard the processes as a group and to take that into account when scheduling them.

III.ALGORITHM

AboutAnt colony algorithm:

The ant colony algorithm is an algorithm for finding optimal paths that is based on the behavior of ants searching for food.

At first, the ants wander randomly. When an ant finds a source of food, it walks back to the colony leaving "markers" (pheromones) that show the path has food. When other ants come across the markers, they are likely to follow the path with a certain probability. If they do, they then populate the path with their own markers as they bring the food back. As more ants find the path, it gets stronger until there are a couple streams of ants traveling to various food sources near the colony[1]. Because the ants drop pheromones every time they bring food, shorter paths are more likely to be stronger, hence optimizing the "solution." In the meantime, some ants are still randomly scouting for closer food sources. A similar approach can be used find near-optimal solution to the travelling salesman problem[3].

Once the food source is depleted, the route is no longer populated with pheromones and slowly decays.

Because the ant-colony works on a very dynamic system, the ant colony algorithm works very well in graphs with changing topologies. Examples of such systems include computer networks, and artificial intelligence simulations of workers[5][6].

Steps of ant colony algorithm:

- ***Step 1: Ants information:*** Here the number of nodes, iterations, ants and distance is to be considered. The equidistance between two nodes is to be considered as distance = √(x1-x2) ^2-(y1-y2) ^2

- ***Step 2: Primary placing:*** Here random placing to make a path takes place
- ***Step 3 : Ants cycle*** : Here an ant tour gets completed and different paths to reach destination are generated
- ***Step 4 : Ants cost*** : Among all the paths the best path is selected else set back to primary placing
- ***Step 5 : Ants trace update*** : the best path chosen is updated and set back to step 1 and the process continues again

Algorithm for ACO with TSP(Travelling Sales man problem):

Step 1: Initialize

Step2: Place each ant in a randomly chosen city

Step3: Chose next city for each ant

Step4: More cities to visit:

- If "yes" goto step3
- Else goto step 5

Step5: Return to the initial cities

Flowchart:

The flow chart for ACO with TSP follows the basic algorithm of ACO. The best criteria is to find the best and shortest path to visit all the cities with minimal time[3]. This is the basic principle to schedule the task in a multi processor system

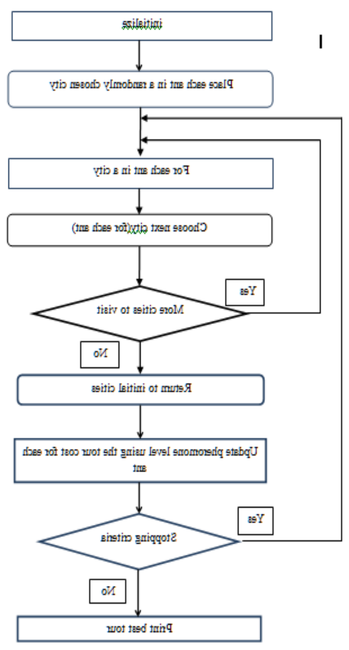

initialize
Place each ant in a randomly chosen city
For each ant in a city
Choose next city(for each ant)
More cities to visit
Yes
No
Return to initial cities
Update pheromone level using the tour cost for each ant
Stopping criteria
Yes
No
Print best tour

Fig 2.2: Flow chart for ACO with TSP

Explanation

The flow chart begins with creating number of ants required. The ants will randomly choose a path. Since ants cannot visualize them, therefore deposit pheromone, a chemical deposition on their way to destination. The other ants thereby recognize the way and reach the destination. Ants smell the chemical deposition, if they found that pheromone is going to get evaporated the succeeding ant will deposit the chemical deposition again. This how they make their way to destination in order to find their food.

First to reach the destination they randomly chose different paths. If at all they find any obstacle they chose different paths .once after completing their tour to reach the destination, the shortest path to reach the destination will be chosen[9].

The natural behavior of ants can be shown as follows:

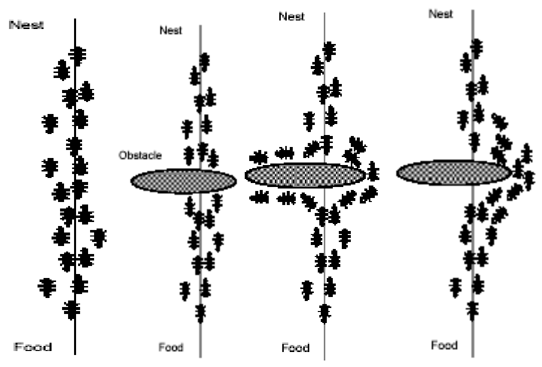


fig 3.1: Natural behavior of ants

Applications:

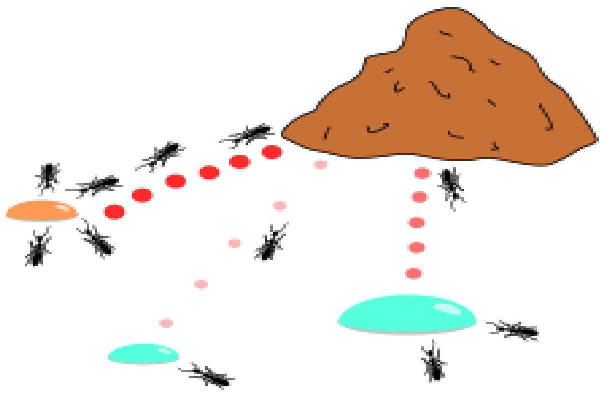


Fig 3.2: The ants prefer the smaller drop of honey over the more abundant, but less nutritious, sugar

Ant colony optimization algorithms have been applied to many combinatorial optimization problems, ranging from quadratic assignment to protein folding or routing and a lot of derived methods have been adapted to dynamic problems in real variables, stochastic problems, multi-targets and parallel implementations. It has also been used to produce near-optimal solutions to the traveling salesman problem. They have an advantage over simulated annealing and genetic algorithm approaches of similar problems when the graph may change dynamically; the ant colony algorithm can be run continuously and adapt to changes in real time. This is of interest in network routing and urban transportation systems[7].

The first ACO algorithm was called the Ant system and it was aimed to solve the travelling salesman problem, in which the goal is to find the shortest round-trip to link a series of cities[4]. The general algorithm is relatively simple and based on a set of ants, each making one of the possible round-trips along the cities. At each stage, the ant chooses to move from one city to another according to some rules:

1. It must visit each city exactly once;
2. A distant city has less chance of being chosen (the visibility);
3. The more intense the pheromone trail laid out on an edge between two cities, the greater the probability that that edge will be chosen;
4. Having completed its journey, the ant deposits more pheromones on all edges it traversed, if the journey is short;
5. After each iteration, trails of pheromones evaporate.

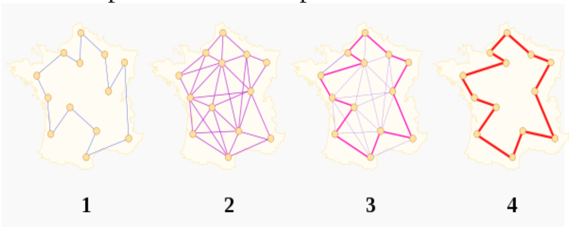


Fig 3.3: Routing techniques

Scheduling problems

- Job-shop scheduling problem (JSP)
- Open-shop scheduling problem (OSP)
- Permutation flow shop problem (PFSP)
- Single machine total tardiness problem (SMTTP)
- Single machine total weighted tardiness problem (SMTWTP)
- Resource-constrained project scheduling problem (RCPSP)
- Group-shop scheduling problem (GSP)
- Single-machine total tardiness problem with sequence dependent setup times (SMTTPDST)
- Multistage Flow shop Scheduling Problem (MFSP) with sequence dependent setup/changeover times

Vehicle routing problems

- Capacitated vehicle routing problem (CVRP)
- Multi-depot vehicle routing problem (MDVRP)
- Period vehicle routing problem (PVRP)

- Split delivery vehicle routing problem (SDVRP)
- Stochastic vehicle routing problem (SVRP)
- Vehicle routing problem with pick-up and delivery (VRPPD)
- Vehicle routing problem with time windows (VRPTW)
- Time Dependent Vehicle Routing Problem with Time Windows (TDVRPTW)
- Vehicle Routing Problem with Time Windows and Multiple Service Workers (VRPTWMS)

Assignment problems

- Quadratic assignment problem (QAP)
- Generalized assignment problem (GAP)
- Frequency assignment problem (FAP)
- Redundancy allocation problem (RAP)

Set problems

- Set cover problem (SCP)
- Partition problem (SPP)
- Weight constrained graph tree partition problem (WCGTPP)
- Arc-weighted l-cardinality tree problem (AWlCTP)
- Multiple knapsack problem (MKP)
- Maximum independent set problem (MIS)

Device Sizing Problems in Nano electronics Physical Design

- Ant Colony Optimization (ACO) based optimization of 45 nm CMOS based sense amplifier circuit could converge to optimal solutions in very minimal time.

- Ant Colony Optimization (ACO) based reversible circuit synthesis could improve efficiency significantly.

IV.EXPERIMENTATION RESULTS

In this project the simulation is carried out for the optimum task allocation of processors using Ant Colony Optimization(ACO) based on Travelling Salesmen Problem(TSP). For this work 13 processors are concerned for optimizationswhich are referred as nodes. The simulation is carried according to the algorithm as described in chapter 3. The main parameters are alpha, beta& M and they can be elaborated as,

- Alpha = order of effect of ants' sight, the number of ants represents number of processor in alpha is used to monitor the status of the processor.
- Beta= order of trace's effect, this represents total number of processors and task allocate to each processor considering the time constraints.
- M= number of ant, where number of ants represents numbers of processors in a multiprocessor system.

Later on the simulation results are obtained by varying the alpha, beta &M to observe the characteristics of the ACO algorithm that how the effects will take place in task scheduling for Multi-processor system

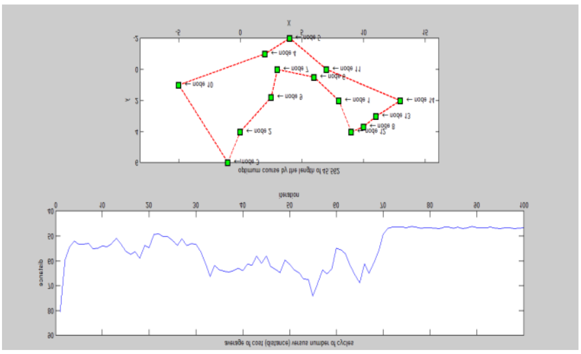
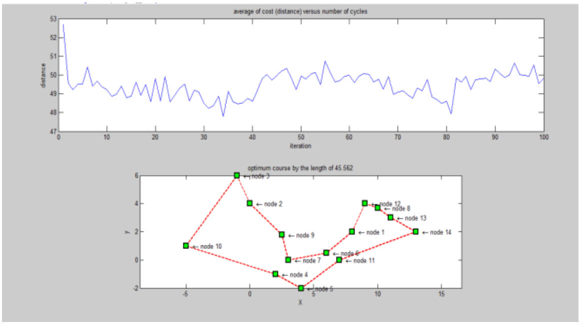


Fig (4.1): Alpha=1



Fig(4.2): Alpha=10

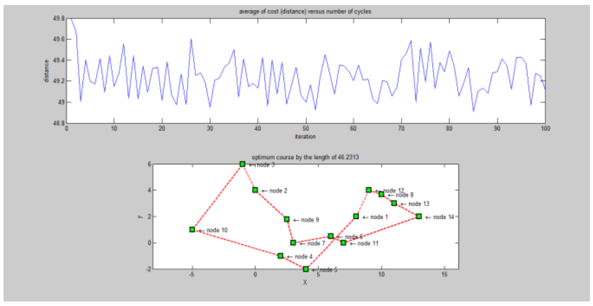


Fig (4.3): Alpha=50

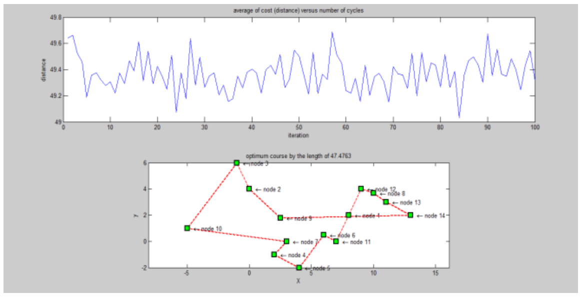


Fig (4.4): Alpha=100

For initial values of alpha =1 and m= 100 , by considering different values of beta the results are obtained

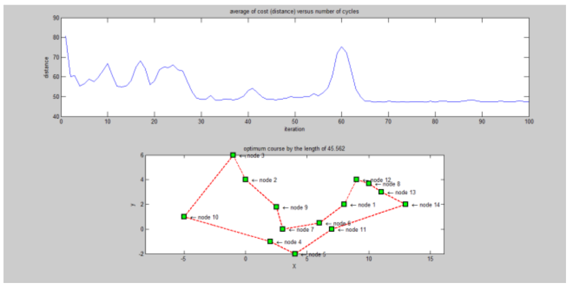


Fig (4.5): Beta=5

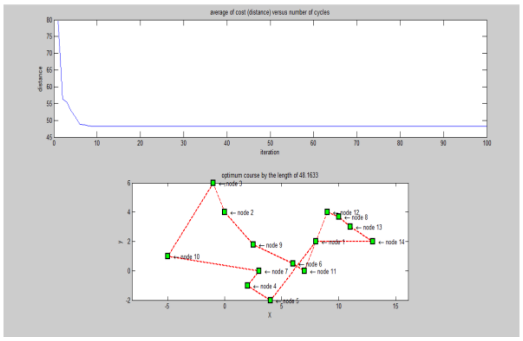


Fig (4.6): Beta=15

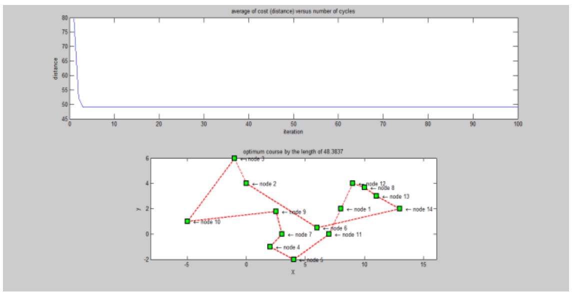


Fig (4.7): Beta=50

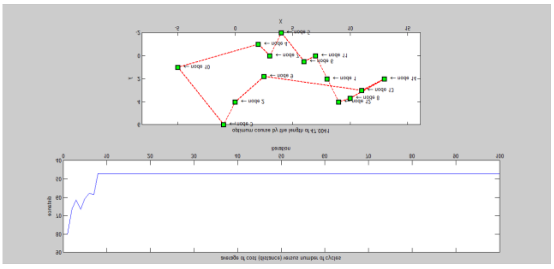


Fig (4.8): Beta=60

By considering different values of ants i.e.., nodes in case of multiprocessor system the results are obtained

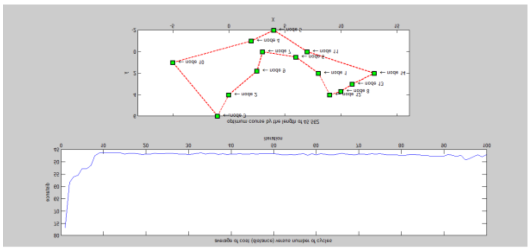


Fig (4.9): m=100

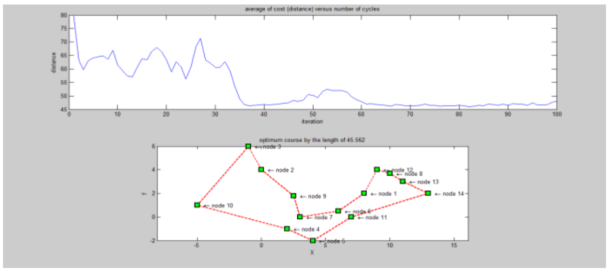


Fig (4.10): m=400

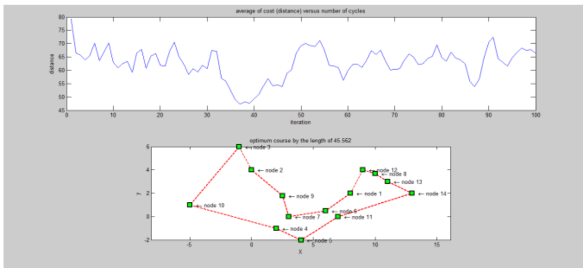


Fig (4.11): m=800

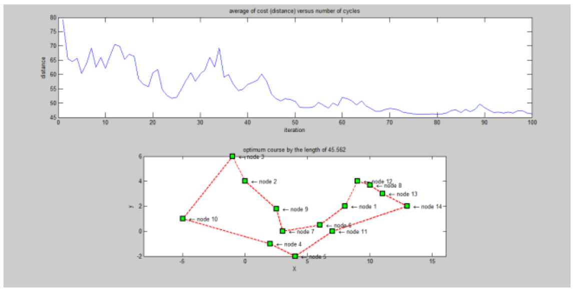


Fig (4.12): m=1000

V.CONCLUSION

Based on the experiments, we can conclude that the quality of solutions depends on the number of ants. The lower number of ants allows the individual to change the path much faster. The higher number of ants in population causes the higher accumulation of pheromone on edges, and thus an individual keeps the path with higher concentration of pheromone with a high probability. The final result differs from optimal solution by 12 km (deviation is less than 0-9 %). The great advantage over the use of exact methods is that ACO algorithm provides relatively good results by a comparatively low number of iterations, and is therefore able to find an acceptable solution in a comparatively short time, so it is useable for solving problems occurring in practical applications.

The most widely studied problems using ACO algorithms are the traveling salesman problem and the quadratic assignment problem (QAP). Applications of ACO algorithms to the TSP have been reviewed in this paper. As in the TSP case, the rest application of an ACO algorithm to the QAP has been that of Ant System. In the last two years several ant and ACO algorithms for the QAP have been presented by Maniezzo and Colorni, Maniezzo, Gambardella, Taillard, and Dorigo, and StÄutzle. Currently, ant algorithms are among the best algorithms for attacking real-life QAP instances.

REFERENCES

1. A. Colorni, M. Dorigo et V. Maniezzo, *Distributed Optimization by Ant Colonies*, actes de la première conférenceeuropéenne sur la vie artificielle, Paris, France, Elsevier Publishing, 134-142, 1991.
2. . Dorigo, *Optimization, Learning and Natural Algorithms*, PhD thesis, Politecnico di Milano, Italy, 1992.
3. T. Stützle et H.H. Hoos, *MAX MIN Ant System*, Future Generation Computer Systems, volume 16, pages 889-914M, 2000
4. M. Dorigo et L.M. Gambardella, *Ant Colony System : A Cooperative Learning Approach to the Traveling Salesman Problem*, IEEE Transactions on Evolutionary Computation, volume 1, numéro 1, pages 53-66, 1997.
5. Gupta, D.K.; Arora, Y.; Singh, U.K.; Gupta, J.P., "Recursive Ant Colony Optimization for estimation of parameters of a function," Recent Advances in Information Technology (RAIT), 2012 1st International Conference on , vol., no., pp.448-454, 15–17 March 2012
6. Gupta, D.K.; Gupta, J.P.; Arora, Y.; Shankar, U., "Recursive ant colony optimization: a new technique for the estimation of function parameters from geophysical field data," Near Surface Geophysics , vol. 11, no. 3, pp.325-339
7. M. Zlochin, M. Birattari, N. Meuleau, et M. Dorigo, *Model-based search for combinatorial optimization: A critical survey*, Annals of Operations Research, vol. 131, pp. 373-395, 2004.
8. V.K.Ojha, A. Abraham and V. Snasel, ACO for Continuous Function Optimization: A Performance Analysis, 14th International Conference on Intelligent Systems Design and Applications (ISDA), Japan, Page 145 - 150 978-1-4799-7938-7/14 2014 IEEE
9. M. Dorigo, V. Maniezzo, et A. Colorni, *Ant system: optimization by a colony of cooperating agents*, IEEE Transactions on Systems, Man, and Cybernetics--Part B , volume 26, numéro 1, pages 29-41, 1996.
10. D. Martens, M. De Backer, R. Haesen, J. Vanthienen, M. Snoeck, B. Baesens,*Classification with Ant Colony Optimization*, IEEE Transactions on Evolutionary Computation, volume 11, number 5, pages 651—665, 2007.