Nouman Butt

March 2009

MASTER THESIS 2009

HÖGSKOLAN
Dalarna

BEE COLONIES APPLIED TO MULTIPROCESSOR SCHEDULING

BY

NOUMAN BUTT

SUPERVISED BY: MR. PASCAL REBREYEND

Dalarna University
Röda vägen 3S-781 88
Borlänge Sweden

Tel: +46(0)23 7780000
Fax: +46(0)23 778080
http://www.du.se

HÖGSKOLAN Dalarna

DEGREE PROJECT

Dept. of Computer Engineering

THESIS TITLE	**BEE COLONIES APPLIED TO MULTIPROCESSOR SCHEDULING**
NAME OF STUDENT	**NOUMAN BUTT**
REGISTRATION NUMBER	
EXTENT	**15 ECTS**
DATE (yyy-mm-dd)	**2009-03-05**
PROGRAM	**APPLIED ARTIFICIAL INTELLIGENCE**
SUPERVISOR	**MR. PASCAL REBREYEND**
EXAMINER	**MR. MARK DOUGHERTY**
DEPARTMENT	**DEPARTMENT OF COMPUTER ENGINEERING, HÖGSKOLAN DALARNA SWEDEN**

Dalarna University
Röda vägen 3S-781 88
Borlänge Sweden

Tel: +46(0)23 7780000
Fax: +46(0)23 778080
http://www.du.se

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

This work is done under the supervision of Mr. Pascal Rebreyend, a professor in Högskolan Dalarna, whose constructive comments and guidance have led to a substantial improvement in the presentation of my work and the report as well.

1.0 ABSTRACT

In order to achieve the high performance, we need to have an efficient scheduling of a parallel program onto the processors in multiprocessor systems that minimizes the entire execution time. This problem of multiprocessor scheduling can be stated as finding a schedule for a general task graph to be executed on a multiprocessor system so that the schedule length can be minimize [10]. This scheduling problem is known to be NP- Hard.

In multi processor task scheduling, we have a number of CPU's on which a number of tasks are to be scheduled that the program's execution time is minimized. According to [10], the tasks scheduling problem is a key factor for a parallel multiprocessor system to gain better performance. A task can be partitioned into a group of subtasks and represented as a DAG (Directed Acyclic Graph), so the problem can be stated as finding a schedule for a DAG to be executed in a parallel multiprocessor system so that the schedule can be minimized. This helps to reduce processing time and increase processor utilization. The aim of this thesis work is to check and compare the results obtained by Bee Colony algorithm with already generated best known results in multi processor task scheduling domain.

2.0 PREVIOUS WORK

Bee colony optimization is almost a recent technique of optimizing solutions like swarm intelligence and a lot of work is in progress nowadays. Chin Soon Chong and Malcolm Yoke Hean Low [13] in their work in "A Bee Colony Optimization Algorithm to Job Shop Scheduling" have compared bee colony to ant colony algorithms. Again these both scientists in their work [14] "Using a Bee Colony Algorithm for neighborhood search in Job Shop Scheduling problems" used self-organization of honey bee colony for solving job shop scheduling problems and the algorithm was founded on neighborhood search based foraging. Li-Pei Wong, Malcolm Yoke Hean Low, and Chin Soon Chong in their work in [11] "A Bee Colony Optimization Algorithm for Traveling Salesman Problem" tried to solve the TSP using bee colony. Other than these, there are a lot of other scientists too who have contributed in this domain with excellence.

3.0 INTRODUCTION

Scheduling a set of dependent or independent tasks for parallel execution on a set of processors is an important and computationally complex problem. Parallel program can be decomposed into a set of smaller tasks that generally have dependencies. The goal of task scheduling is to assign tasks to available processors such that precedence requirements between tasks are satisfied and the overall time required to execute all tasks, the make span, is minimized. There are various variants of this problem, depending on whether we consider communication delays or not, whether the multiprocessor systems are heterogeneous or homogeneous and other considerations. Various studies have proven that finding an optimal schedule is an NP-complete problem even in the simplest forms [3].

Since finding an optimal solution is not feasible, a large number of algorithms were proposed which attempt to obtain a near-optimal solution for various variants of the multiprocessor task scheduling problem. These algorithms usually trade the computational complexity of the scheduling algorithm itself to the quality of the solution. Algorithms based on complex, iterative search can usually (but not always) outperform simple one-pass heuristics, but their computational complexity makes them less scalable [3].

Let a (homogeneous) multiprocessor system be a set of '*m*' identical processors, $m > 1$. Each processor has its own memory and each pair of processors communicate exclusively by message passing through an interconnection network. Additionally, let a parallel program be a set of communicating tasks to be executed under a number of precedence constraints. To each task is associated a cost, representing its execution time. A weighted acyclic task digraph can be used to represent the tasks and the precedence constraints. In order to be executed, each task of a given parallel program must be scheduled to some processor of a given multiprocessor system. Consequently, tasks that communicate in the parallel program may be scheduled to different processors, which lead these processors to communicate during the execution of the parallel program. In general, these communications slow down the execution of the parallel program. Considering these communications and the precedence constraints between tasks, it follows those different schedules of each task satisfying the precedence constraints lead to different execution times of the parallel program.

4.0 BEE COLONY

According to [11], the foraging behavior in a bee colony remains mysterious for many years until von Frisch translated the language embedded in bee waggle dances [12]. In his series of experiments, he also discovered that bees actually own color vision and attracted by scent deposited by their hive mates.

To examine how bees communicate via waggle dance, suppose a bee has found a rich food source. Upon its return to the hive, it starts to dance in a figure eight pattern: a straight waggle run followed by a turn to the right back to the starting point, and then another straight waggle run followed by a turn to the left and back to the starting point again. The bee usually repeats these for a few times. Remarkably, via the dance, the bee has actually informed its hive mates about the direction and distance of the food source. The direction is expressed via the angle of dance relative to the sun position whereas the distance is expressed through the length of the straight waggle run. This coding and decoding process will eventually bring more bees towards the new food discovery.

In Bee Colony model, the colony consists of three groups of bees: employed bees, onlookers and scouts. It is assumed that there is only one artificial employed bee for each food source. In other words, the number of employed bees in the colony is equal to the number of food sources around the hive. Employed bees goes to their food source and come back to hive and dance on the dance floor. The employed bee whose food source has been abandoned becomes a scout and starts to search for finding a new food source. Onlookers watch the dances of employed bees and choose food sources depending on dances.

In Bee Colony model which is an evolutionary and population based algorithm, the position of a food source represents a possible solution to the optimization problem and the nectar amount of a food source corresponds to the quality (fitness) of the associated solution.
The pseudo code for the bee's algorithm in its simplest form according to [11] is:

```
1. procedure BCO
      a. Initialize_Population ( )
            i. while stop criteria are not fulfilled, do
                  1. while all bees have not built a complete path, do
                        a. Observe_Dance ( )
                        b. Forage_ByTransRule ( )
                        c. Perform_Waggle_Dance ( )
            2. end *while*
        ii. end *while*
    2. *end procedure* BCO
```

5.0 ALGORITHM AND DESIGN

5.1 Potential Challenges

During my work, I came across a lot of challenges regarding the implementation of Bee Colony in real or my scenario. The main challenge for the person implementing swarm intelligence algorithms is to understand the algorithm correctly and then map it in his/her own environment which sometimes is hard to do. Implementing an algorithm in programming domain gives rise to lot of logical questions which need to be answered before implementing.

5.2 Greedy Implementation

Before implementation of Bee Colony in my work, I scheduled the tasks using Greedy algorithm and got some results. In this section I'm going to present the pseudo code of greedy algorithm. The basic idea of greedy algorithm here is to check each task on every CPU and calculate the time. The CPU producing the least time when a particular task executes on it; will be the CPU on which the task will be executed finally.
The pseudo code of the algorithm is as follows:

Greedy Pseudo Code:

```
1. After reading a file, randomly fill all tasks in an array
2. WHILE all tasks are not scheduled, DO
    2.1. Select first task from array, let it be 'Incomp_Task'
    2.2. IF 'Incomp_Task' is scheduled, THEN
        2.2.1. Select next task from the random array
    2.3. ELSE
        2.3.1. IF Incomp_Task has a predecessor, THEN
            2.3.1.1.      IF predecessors of Incomp_Task are complete, THEN
                2.3.1.1.1.    Compute FOR each CPU, the earliest starting time of Incomp_Task
                2.3.1.1.2.    IF no. of CPU's giving least time is more than 1, THEN
            2.3.1.1.2.1. Execute Incomp_Task randomly on any CPU giving least time
        2.3.1.1.3. ELSE
            2.3.1.1.3.1. Execute Incomp_Task on CPU giving least starting time
        2.3.1.1.4. END IF
        2.3.1.1.5. Go to step (2.1.)
      2.3.1.2. ELSE
        2.3.1.2.1. Get all predecessors of Incomp_Task
        2.3.1.2.2. Select a new Incomp_Task from all predecessors
        2.3.1.2.3. Go to step (2.3.1.)
      2.3.1.3. END IF
    2.3.2. ELSE
      2.3.2.1. Go to step (2.3.1.1.1.)
    2.3.3. END IF
  2.4. END IF
3. END WHILE
4. Compute make span
```

5.3 Bee Colony Implementation and Pseudo Code

The algorithm shown below is just the Bee Colony part i.e. the re-ordering of the array of random tasks used in greedy part to know which tasks to schedule first and which in the last. The difference between this Bee Colony algorithm and the above Greedy algorithm is that the Greedy algorithm selected tasks randomly to schedule while this Bee Colony will select on tasks on the basis of some criteria. The least make span selected at the last step of the algorithm below, will be the make span of the final selected order of tasks produced by Bee algorithm and also this order of tasks will be the output to greedy.

```
1. procedure BCO
  1.1. randomly fill all tasks in an array
  1.2. select employed bees i.e. no. of employed bees equals to the no. of tasks
```

```
1.3. select tasks having no predecessor from all tasks in list 'A'
1.4. count all tasks of list 'A', let count be 'x'
1.5. FOR i = 1 to x,
    1.5.1.    after observing waggle dance by onlooker bees, chose a
              task to execute from list 'A'
    1.5.2.    execute task of list 'A' i.e. A [ i ]th task
    1.5.3.    WHILE all tasks are not scheduled, DO
        1.5.3.1.  update list 'A' with tasks whose predecessors are either
                  scheduled or not exist
        1.5.3.2.  execute the task from list 'A' having least execution time
        1.5.3.3.  Go to step (1.5.2.)
    1.5.4.    END WHILE
    1.5.5.    compute make span of this iteration and save it in list 'B'
    1.5.6.    perform waggle dance by employed bees
    1.5.7.    i = i + 1
1.6. END FOR
1.7. select the least make span from list 'B'
1.8. select the order of tasks producing the least make span. This order of tasks
     will be the input of greedy part.
2. end procedure BCO
```

5.4 Explanation

When greedy algorithm was implemented, the order of tasks was taken as random. Now the only thing left is just to re-order the tasks by using Bee Colony algorithm so that we could generate a set of feasible solutions rather than using a pseudorandom approach and also we could know which tasks to execute first and which in the last.

Each time the algorithm runs, it runs or starts from a task having no predecessors among other tasks because it's impossible to start from a task having predecessors. Hence the stopping criteria of the above Bee Colony algorithm is when all independent tasks or tasks having no predecessors are not finished or when the no. of iterations exceeds the no. of tasks having no predecessors. There are a lot of ways to implement Bee Colony algorithm among which I made an attempt to implement it by taking help from [15]. In [15], the no. of bees recruited by authors are equal to the no. of cities to visit. In the same way, I have recruited the no. of bees equal to the no. of tasks I have. In order to understand the implementation of Bee Colony optimization, the writers Li-Pei Wong, Malcolm Yoke Hean Low, and Chin Soon Chong in their article "*Bee colony optimization with local search for TSP*" [15] have used two basic parameters in order to visit from city 1 to city 2. These two parameters are:

- Arc fitness
- Distance

The above writers used two sets of tables, one containing the preferred path of cities and the other containing the next city to visit. In their article, the table containing the preferred path is obtained after the communication among bees after a series of waggle dance performed. The more the arc fitness and the less the distance between two cities, the highest is the priority of a particular city to visit. On the basis of this strategy, I have tried to develop a strategy based on the above article and also on Bee Colony. Since I have tasks to schedule on a number of CPU's and unlike cities in TSP of [15], tasks are dependent on each other, so I have made the following parameters from which a bee will decide or prioritize which task to visit first. Those three parameters are:

1. Look for independent tasks
2. Tasks with less execution time
3. Tasks with dependencies

Since it is impossible to start a tour of tasks from a dependent task, therefore the priority of independent tasks are a bit higher than the dependent tasks that's why I put priority of independent tasks on top. The priority function in my problem is based on the above three parameters. Allow me to explain it with an example.

Suppose I have nine tasks with dependencies among them with execution time shown below in figure 1. Here the circles represent the task name and the numbers in the brackets represent the execution time of each task. The arrows are representing that the task is sending data to other task, for e.g. task 'A' has an execution time of '7' and is sending data to 'D' or 'D' is dependent on task 'A'.

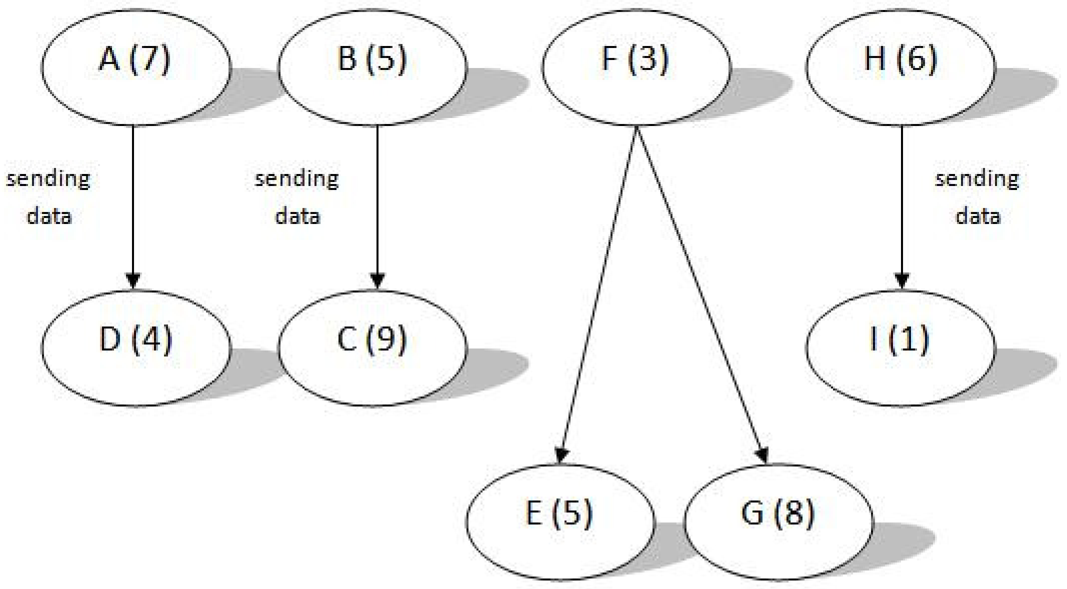


Fig. 1: Tasks and dependencies among them

Since in the above case, there are nine tasks therefore there will be nine employed bees that will search for the path. After returning to its hive, the bees will perform a waggle dance to show the other Onlooker bees which task to chose first. The onlooker bee will then be equipped with a set of moves to start, which are observed from the dance. These set of moves are shown below in table 1. According to the above three strategies I mentioned, the bee will look for the independent tasks first because it's not possible to execute a task if its predecessor isn't scheduled yet. So in order to avoid this situation, the bee will look for the independent tasks first. After getting all the independent tasks, the new table will look like this:

Table 1:

FREE TASKS	EXECUTION TIME
A	7
B	5
F	3
H	6

The Free Tasks column is the same as independent tasks column but with a different name just for the ease of understanding. In the above table, the free tasks column contains all the tasks which are not dependent on any other task or whose predecessors don't exist. Since it is the first iteration, therefore the bee will choose the ***first*** free task to execute or add in its memory and this task will be the task "A". After executing task "A", a new table is created i.e. table 2 and the bee will be left with the following situation:

Table 2:

FREE TASKS	EXECUTION TIME
B	5
D	4
F	3
H	6

As can be seen in table 2, a new task is added in free tasks column i.e. task "D", because after execution of task "A", task "D" is now available to execute. From this point and on, the bee will look for the tasks having less execution time because of our defined strategy. The task with less execution time is task "F". This task will be now executed on a CPU giving the least time. After the execution of task "F", table 2 will be updated again and will be looking like this:

Table 3:

FREE TASKS	EXECUTION TIME
B	5
D	4
H	6
E	5
G	8

The new added tasks are "E" and "G" as can be seen in table 3. Now again the bee will look for the task with less execution time and now it will be task "D". Now task "D" will be executed and this process will go on until all the tasks are completed.

After execution of every task, this one iteration is completed and the bee has developed the following order or sequence of execution of tasks:

A --> F --> D --> B --> E --> H --> I --> G --> C

After execution of every task, the make span of the above order and other orders will be calculated and let suppose the total make span after execution of all tasks in the first iteration is ‘x1’. In the same way, bees will again perform an iteration starting from table 1. This iteration will again start after performing the waggle dance by employed bees and observing dance by onlooker bees. In the beginning of the first iteration, the bees executed task “A”, so now in the second iteration the bees will execute or select the second task in table ‘**1**’ i.e. task “B”. After executing task “B” from the figure shown above, the new table will look like this:

Table 4:

FREE TASKS	**EXECUTION TIME**
A	7
C	9
F	3
H	6

As can be seen in table 4, after executing task “B”, a new task is ready to execute i.e. task “C”. In the same way, the bees will again make tables like they did before and will create an order of tasks and calculate the make span after execution of every task. Let the make span of this second iteration is ‘x2’.

In the above mentioned example, bees will have maximum four iterations because in table 1, there are four independent or free tasks to start with, hence bees will get a total of four make spans i.e. ‘x1’, ‘x2’, ‘x3’, and ‘x4’. After getting all the make spans, we will select the least make span among those four make spans and let suppose that make span is ‘x3’. This order of tasks executed in the case of make span ‘x3’ will be our order of tasks to schedule tasks using Bee Colony on CPU’s.

6.0 IMPLEMENTATION DETAILS

This work of mine is developed in Visual C++ in a .net 2005 environment as a development tool. A list of tour or order of tasks created in the beginning symbolizes the dances by bees. After each waggle dance, a new table is created for bees containing the new set or tasks to complete.

My implementation contains three basic parameters as discussed above, by which a bee selects a task to complete or execute first.

As stated above, before implementing Bee Colony, I've implemented and checked the results using greedy algorithm. The greedy algorithm is used in a way such that I selected each task among whole tasks, and executed it on every processor. The processor which is giving the least starting time of the selected task will be the processor on which the selected task will be scheduled. In this way all of the tasks were scheduled using this least starting time technique and calculated the make span.

7.0 RESULTS AND ANALYSIS

In order to evaluate the performance of the Bee Colony algorithm, I used some 32 benchmark files for testing. Also, to compare the results obtained by Bee Colony, I have made use of some best known results already generated by some other methods used by different authors. Below is the table of the results I've got from both Greedy and Bee Colony:

Table 5: Results of make spans

GRAPH		BEST KNOWN RESULTS	GREEDY	BEE COLONY
Bellford	m	71 936 050	101 720 650	100 402 450
	l	193 794 250	326 431 650	297 577 300
Diamond-1	m	131 940 750	**131 940 750**	134 422 800
	l	276 758 950	**269 700 750**	275 520 000
Diamond-2	m	127 224 450	184 456 550	228 420 450

	l	218 954 400	333 438 400	345 007 300
Diamond-3	m	176 982 100	189 753 550	191 047 350
	l	228 590 500	256 848 600	252 454 000
Diamond-4	m	132 875 550	**126 227 800**	156 345 400
	l	164 264 000	**150 804 350**	182 521 100
Divconq	m	97 307 880	136 341 910	150 908 180
	l	169 043 350	240 759 590	287 153 410
Fft	m	29 888 250	37 489 200	39 260 550
	l	102 647 300	138 297 600	134 239 350
Gauss	m	226 882 900	254 241 900	248 501 900
	l	307 395 80	345 623 200	345 699 650
Iteratif	m	16 733 350	23 961 200	33 722 500
	l	47 936 700	65 576 200	79 642 500
ms-gauss	m	4 598 550	4 605 112 750	4 605 112 750
	l	2 559 388 750	2 567 263 050	2 572 987 200
prolog	m	60 499 350	83 841 600	66 086 450
	l	258 529 350	380 674 800	261 475 800
qcd	m	1 173 100 600	1 176 047 050	1 176 047 050
	L	2 038 576 050	2 041 522 500	2 041 522 500
elbow	m	6630	6734	6708
stanford	m	627	703	686
ssc	5	74	169	144
	6	77	196	146
	7	82	194	149
	8	86	213	183
	9	89	264	213

In the table above, the '*Graph*' column is containing the names of the benchmark files provided to me by my supervisor. The column '*Best Known Results*' is containing the results obtained by different scientists in their respective work and are the best optimized results so far in this particular problem domain. The other two columns are '*Greedy*' and '*Bee Colony*' respectively containing the results I've got after implementing both algorithms as discussed earlier.

Unfortunately, the results obtained from Bee Colony are not very promising when compared to the best known results as can be seen from the table above, but when it comes to the comparison between Bee Colony and Greedy results, both algorithms have produced good results in some of their respective benchmark files or in other words we cannot say for sure that any particular algorithm is producing good results. The shaded boxes in the '*Greedy*' and '*Bee Colony*' columns are containing the best result when both algorithms were compared.

From the above table, we have found an important finding that in some of the benchmark files, Greedy algorithm has outperformed the best known results, and while on the other hand, Bee Colony didn't. These outperformed results can be seen in the '*Greedy*' column with red colored values.

For analysis purposes, we compared both results obtained from both algorithms with the number of tasks or jobs we have in our bench mark files, the table below shows the results we got:

Table 6: Results of make spans with no. of Tasks

GRAPH		NO. OF TASKS	GREEDY	BEE COLONY
Bellford	m	354	101 720 650	100 402 450
	l	992	326 431 650	297 577 300
Diamond-1	m	258	**131 940 750**	134 422 800
	l	1026	**269 700 750**	275 520 000
Diamond-2	m	262	184 456 550	228 420 450
	l	1227	333 438 400	345 007 300
Diamond-3	m	731	189 753 550	191 047 350
	l	1002	256 848 600	252 454 000
Diamond-4	m	731	**126 227 800**	156 345 400
	l	1002	**150 804 350**	182 521 100
Divconq	m	382	136 341 910	150 908 180
	l	766	240 759 590	287 153 410
Fft	m	194	37 489 200	39 260 550
	l	1026	138 297 600	134 239 350
Gauss	m	782	254 241 900	248 501 900
	l	1227	345 623 200	345 699 650
Iteratif	m	262	23 961 200	33 722 500
	l	938	65 576 200	79 642 500
ms-gauss	m	768	4 605 112 750	4 605 112 750
	l	1482	2 567 263 050	2 572 987 200
prolog	m	214	83 841 600	66 086 450
	l	1313	380 674 800	261 475 800
qcd	m	326	1 176 047 050	1 176 047 050
	L	1026	2 041 522 500	2 041 522 500
elbow	m	103	6734	6708

stanford	m	90	703	686
ssc	5	288	169	144
	6	392	196	146
	7	512	194	149
	8	648	213	183
	9	200	264	213

As seen from the above table, number of tasks isn't playing any significant role in the time span of both the algorithms except that the Bee Colony algorithm is mostly producing a bit good results as compared to Greedy when the number of tasks is less.

After calculating the running time of each algorithm, we have also compared the times in milliseconds of both the algorithms after scheduling all the tasks. The table below shows the results:

Table 7: Comparison b/w running time of algorithms

GRAPH		**GREEDY TIME** (in milliseconds)	**BEE COLONY TIME** (in milliseconds)
Bellford	m	0.015	0.078
	l	0.531	0.875
Diamond-1	m	0.016	0.031
	l	2.25	2.5
Diamond-2	m	0.094	0.172
	l	3.0	3.53
Diamond-3	m	0.25	0.375
	l	0.781	1.093
Diamond-4	m	0.344	0.344
	l	1.031	0.938
Divconq	m	0.031	0.047

	l	0.11	0.172
Fft	m	0	0.016
	l	0.344	0.516
Gauss	m	0.906	1.047
	l	3.484	3.98
Iteratif	m	0.016	0.016
	l	1.219	0.25
ms-gauss	m	0.26	0.678
	l	1.735	3.578
prolog	m	0	0.015
	l	0.562	0.718
qcd	m	0.047	0.063
	L	1.67	0.703
elbow	m	0	0
stanford	m	0	0
ssc	5	0	0
	6	0	0.016
	7	0	0.031
	8	0.016	0.063
	9	0.015	0.11

From the above table, we can see that the BCO is taking a bit longer time to schedule all tasks when compared to the Greedy one, which was obvious because BCO adds the time of re-ordering the tasks to complete. To find out the dependency of "Time" of both algorithms, we have done some of the statistical analysis shown below:

7.1 GREEDY TIME vs. NUMBER OF TASKS

For this analysis, following measures are used:

- Standard Deviation
- Variance
- Correlation Analysis

The following outputs are obtained using SPSS 15.0.

Descriptive Statistics:

Table 8:

	MEAN	STD. DEVIATION	N
Greedy Time	0.603613	0.9251031	31

Correlations:

Table 9:

		GREEDY TIME	**TASKS**
GREEDY TIME:	Pearson Correlation	1	0.736
	Sig. (2-tailed)		0.000
	N		31
TASKS:	Pearson Correlation	0.736	1
	Sig. (2-tailed)	0.000	

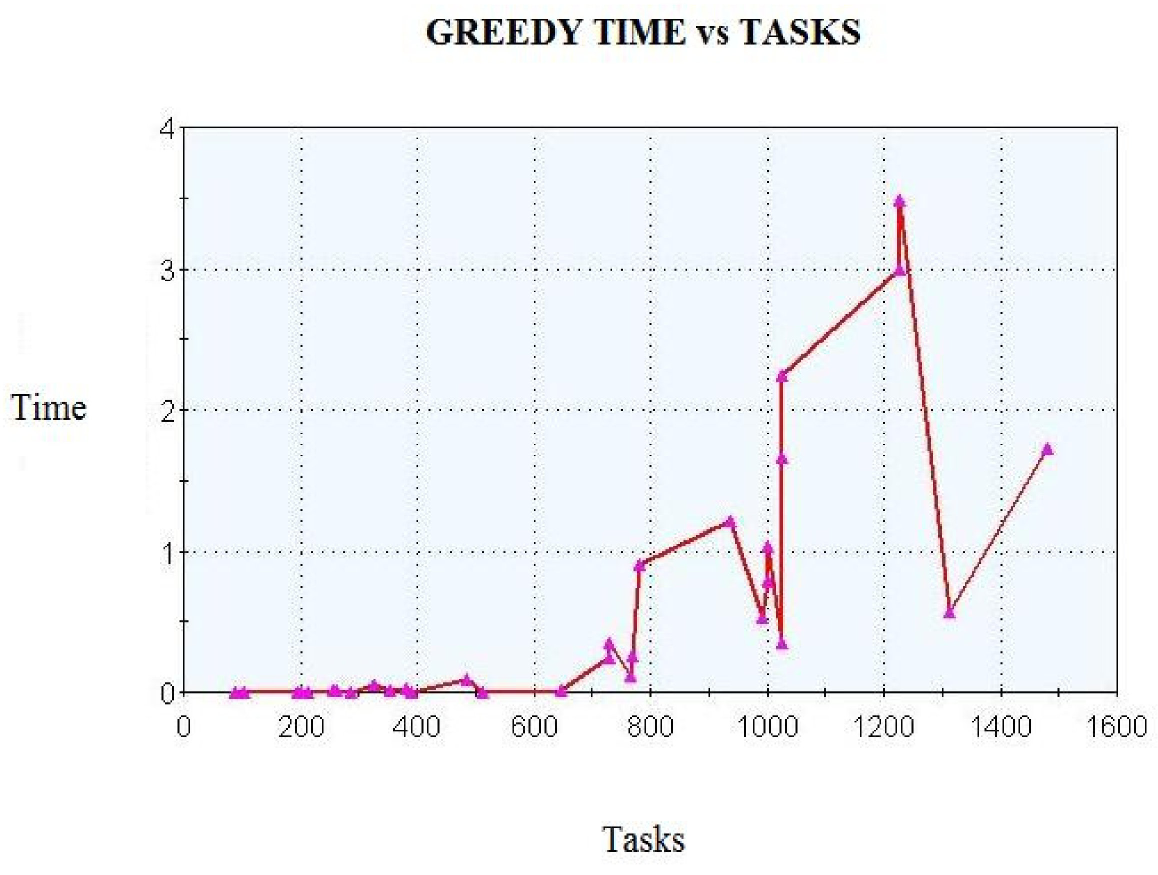


Fig. 2: Graph between Greedy Time vs. no. of Tasks

The above analysis is showing clear picture of relation between Greedy Time and number of tasks. The positive correlation (r = 0.736) is giving clear indication that there is precisely high relation between these two variables, here relation is obtained using time dependency on number of tasks, this value is showing that the shift rate is almost 73.6% with respect to the change in number of tasks. The graph above between "*Greedy Time vs. Tasks*" is giving almost the same picture i.e. the low number of tasks have no clear pattern, while on the other hand, when the number of tasks increased, time also increased.

The other factor of variation is the possible variation in data set. The following descriptive summary also proves the same result.

Descriptive Statistics:

Table 10:

	N	MEAN	STD. DEVIATION	VARIANCE
Data files	31	------	------	------
Greedy time	31	0.603613	0.9251031	0.856
Valid N (list wise)	31	------	------	------

It is clearly visible from the above table that high variation in independent variable (i.e. task) is very high; ultimately having impact on the results of correlation, but still results can be used just for the sake of analysis.

7.2 BCO TIME vs. NUMBER OF TASKS

Correlations:

Table 11:

		BCO TIME	TASKS
BCO TIME:	Pearson Correlation	1	0.744
	Sig. (2-tailed)		0.000
	N		31
TASKS:	Pearson Correlation	0.744	1
	Sig. (2-tailed)	0.000	

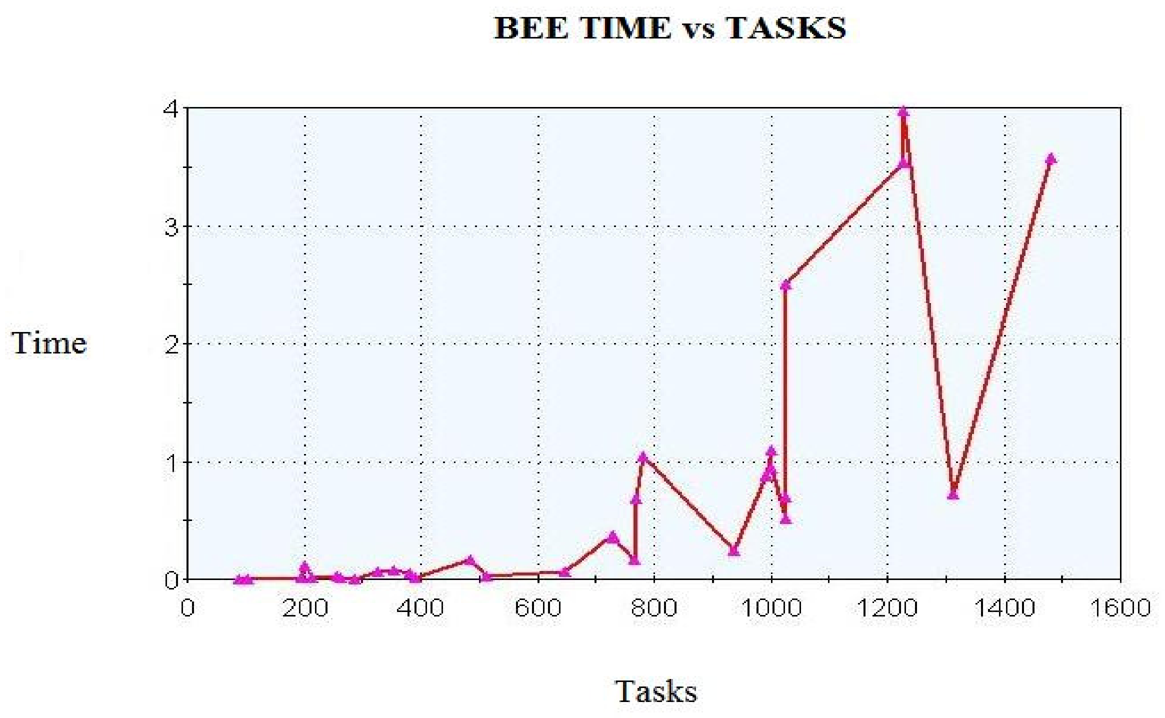


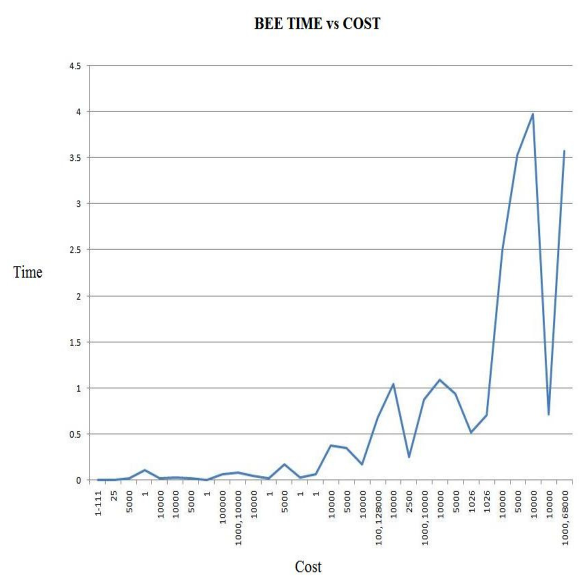
Fig. 3: Graph between Bee Time vs. no. of Tasks

Both the results and some parts of analysis of *"Greedy Time vs. Tasks"* and *"Bee Time vs. Tasks"* are same, but it gives very important information. On the basis of study of sample data (only), it can be concluded that there is no major difference between both types of approaches because there is only some minor differences in correlation coefficient, might because of number of tasks. It is quite possible that in future if some researcher comes up with the task having very low variation, then it might affect the results obtained in this work. But right now from the above analysis, it is easy to conclude that there is no significant difference in these two approaches.

This is an open ended question for further research to draw the line between these two approaches. It will help future research.

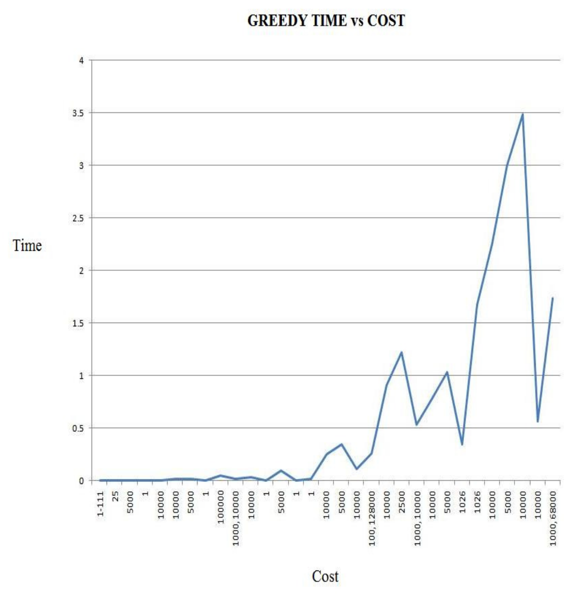
The results shows the same picture as got in greedy approach i.e. as number of task increases, it gives rise to the time but not all the times; rather 74% correlation shows their high positive relation but not 100%.

7.3 BCO TIME AND GREEDY TIME vs. TASK'S COST



BEE TIME vs. COST

Fig. 4: Graph between Bee Time vs. Cost

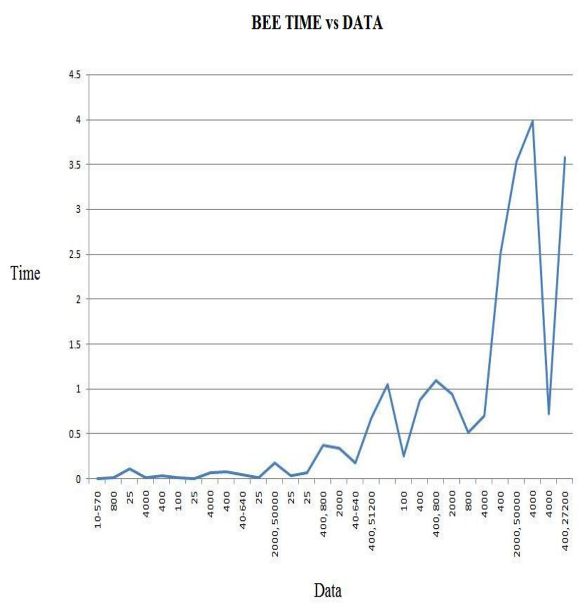


GREEDY TIME vs. COST

Fig. 5: Graph between Greedy Time vs. cost

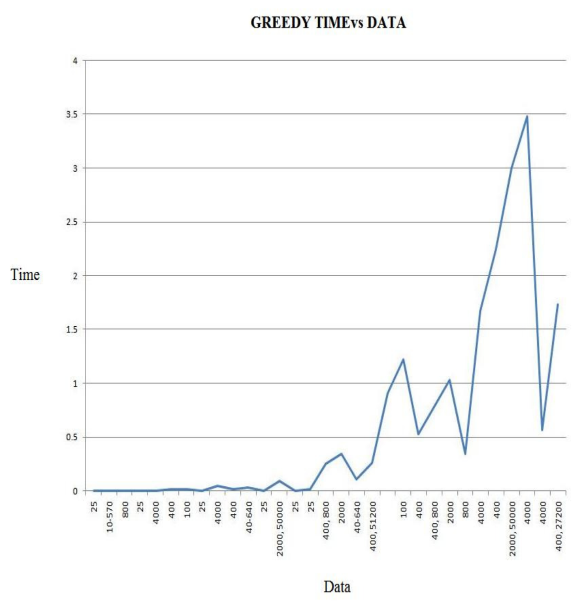
Above are the two graphs for analysis purpose, i.e. *”Bee Time vs. Cost”* and *”Greedy Time vs. Cost”*. The x-axis is containing the cost of task of each benchmark file and the y-axis is containing the respective algorithm’s time in milli seconds. According to the graphs obtained, we can say that the cost of each task isn’t making much difference on the running time of both algorithms because both graphs are almost same with some minor differences.

7.4 BCO TIME AND GREEDY TIME vs. TASK’S DATA



BEE TIME vs. DATA

Fig. 6: Graph between Bee Time vs. Data



GREEDY TIME vs. DATA

Fig. 7: Graph between Greedy Time vs. Data

The above two graphs are for the analysis of both algorithms with the data sent or received by the task. Again, y-axis is containing the running time of each algorithm in milli seconds and x-axis is containing the data of task in each benchmark file. The above two graphs are again showing almost the same result as it was in the previous graphs i.e. *”BCO TIME AND GREEDY TIME vs. TASK’S COST”*. Here, both graphs again are almost same and aren’t making any noticeable curves. From the above graphs, we might conclude that the data sent or received by any task during scheduling in both algorithms, don’t make any major difference with respect to the time.

All in all, there might be lot of possibilities of BCO not giving the promising results, some of them are mentioned below:

- Since it is impossible to try every possible combination of tasks to execute, therefore I had to rely on the results obtained which might not be the best, but at least optimized.

- The three basic parameters which I stated above for choosing a task to schedule might not be very efficient way to deal with this kind of problem. These three basic parameters were deduced by me after reading some articles on TSP solved by Bee Colony Optimization.

8.0 CONCLUSION AND FUTURE RESEARCH

I have implemented a BCO algorithm based on the bee's collective foraging behavior for solving multi-task scheduling problem. Since not many articles have published on BCO especially, it was very hard for me to gather exact information and help I needed to support the way I implemented the algorithm, therefore I believe there is much room for improvement.

Although Greedy algorithm's results have outperformed some of the best known results, we cannot say for sure that Greedy is the best solution to these kinds of problems because in most of the cases Greedy use to stick in local minima.

For future enhancements, I think if I keep on trying solving the same problem with other swarm intelligence algorithms, I might be able to get some promising results. Since there are a lot of ways to implement a Bee Colony algorithm, one has to try every possible way to implement it in order to get good results but due to the lack of time, I couldn't do it. Also, in order to select the task to schedule, one has to go through the articles written on waggle dance of bees because waggle dance is the most important factor by which bees decide what to do next.

REFERENCES

[1] "A Comparison of List Schedules for Parallel Processing Systems" by G.Bell, D.Siewiorek, and S.H.Fuller.

[2] "A Genetic Algorithm For Multi-Layer Multiprocessor Task Scheduling" by Ceyda Oguz and M.Fikret Ecran

[3] "A performance study of multiprocessor task scheduling" by Shiyuan Jin, Guy Schiavone, and Damla Turgut.

[4] "An Incremental Genetic Algorithm Approach to Multiprocessor Scheduling" by Annie S.Wu, Han Yu, Shiyuan Jin, Kuo-Chi Lin, and Guy Schiavone, Member, IEEE.

[5] " DSC - Scheduling Parallel Tasks on an Unbounded Number of Processors" by Tao Yang and Apostolos Gerasoulis.

[6] "General Multiprocessor Task Scheduling" by Jianer Chen and Chung-Yee Lee.

[7] "Multiprocessor Task Scheduling" by Mumin Kurtulus.

[8] "Scheduling multiprocessor tasks on dedicated processors" by Andreas Krämer.

[9] "Scheduling Multiprocessor Tasks with Genetic Algorithms" by Ricardo C. CorreÃ a, Member, IEEE, Afonso Ferreira, Member, IEEE, and Pascal Rebreyend.

[10] "Task Scheduling For Multiprocessor Systems Using Memetic Algorithms" by Prof. Sanjay Sutar, Jyoti P.Sawant, and Jyoti R.Jadhav.

[11] "A Bee Colony Optimization Algorithm for TSP" by Li-Pei Wong, Malcolm Yoke Hean Low, and Chin Soon Chong.

[12] "Decoding the language of the bee" by K. von Frisch, Decoding the language of the bee, Science, vol. 185, no. 4152, pp. 663-668, 1974.

[13] "A Bee colony optimization algorithm to job shop scheduling" by Chin Soon Chong, and Malcolm Yoke Hean Low.

[14] "Using a Bee Colony Algorithm for neighborhood search in Job Shop Scheduling problems" by Chin Soon Chong, and Malcolm Yoke Hean Low.

[15] "Bee colony optimization with local search for TSP" by Li-Pei Wong, Malcolm Yoke Hean Low, and Chin Soon Chong.

[16] "The Bees Algorithm – A Novel Tool for Complex Optimisation Problems" D.T. Pham, A. Ghanbarzadeh, E. Koç, S. Otri , S. Rahim , M. Zaidi