Task Assignment in Multiprocessor Environment using Grouping Genetic Algorithm.

Gathala Sudha Anil Kumar
MTech Dual Degree.
Department of Computer Science and Engineering.
IIT Kharagpur.
sudhaanilkumar@yahoo.co.in

Abstract – **Task Assignment is an important scheduling problem in multiprocessor and distributed systems. The Task Assignment problem is concerned with how to assign tasks to the processors. If the system is static, as most current hard real time systems are, the task assignment is done offline. Many well known offline heuristic algorithms exist for the Task Assignment problem. In this paper we present a grouping genetic algorithm (GGA) to solve the Task Assignment problem for a static system and compare the GGA's performance with other well known heuristics.**

1. INTRODUCTION

The abstract goal of the static task allocation problem is as follows:

Given a collection of tasks and a set of processors on which these tasks will be executed, that mapping of tasks to processors should be found which does not overload any processor.

To achieve good performance in a multiprocessor system, it is essential to maintain a balanced load among all the processors. Therefore, a good solution to the Task Assignment problem loads all processors as equally as possible. In this paper, we compare the performance of different well known heuristic task assignment algorithms with that of GGA based task assignment. The rest of the paper is organized as follows: The second section presents the formal definition of the Task Assignment problem. The third section discusses the four well known heuristic algorithms. The fourth section presents the Grouping Genetic Algorithm. The fifth section presents some experimental results and the conclusions are presented in the sixth section.

2. THE TASK ASSIGNMENT PROBLEM

The following is the formal definition of the Task Assignment problem with 'N' tasks and 'M' processors, where we make use of Stinson's terminology for combinatorial optimization problems [1].

Problem Instance:

Tasks:	$T_1, T_2, \ldots, T_N$.
Task Utilizations:	$TU_1, TU_2, \ldots, TU_N$.
Processors:	$P_1, P_2, \ldots, P_N$.
Processor Utilizations:	$PU_1, PU_2, \ldots, PU_N$.

Where,

TU_i is the utilization of task T_i.

$PU_i = \sum (TU_k)$, for all T_K assigned to P_i.

Feasible Solution:

A feasible solution to the Task Assignment problem is the assignment of each task to one and only one of 'M' processors, such that no processor's utilization exceeds one. More formally, a feasible solution is the vector:

$a = (a_1, a_2, \ldots, a_N)$.

Where,

$a_i = k$, implies that the task T_i is assigned to P_k, and,

$PU_K \leq 1$, for all k belonging to [1, M].

Objective Function:

The objective function is:

$f_{obj}(a) = \sum (|PU_{mean} - PU_i|)$, for all i belonging to [1, M].

Where,

$PU_{mean} = (PU_{sum}) \div M$, with

$PU_{sum} = \sum PU_i$, for all i belonging to [1, M].

A Good Solution:

A good solution yields a low object function value.

An example:

Consider a system consisting of five processors on which eleven tasks should be scheduled. The individual task utilizations are shown in Table 1.

Table 1

Task	Task Utilization
T1	0.235
T2	0.265
T3	0.259
T4	0.198
T5	0.005
T6	0.597
T7	0.725
T8	0.654
T9	0.692
T10	0.719
T11	0.095

A solution to the above problem is:

a = (5,2,4,1,1,1,3,4,5,2,1).

The above solution is feasible since:

$PU_1 = TU_4 + TU_5 + TU_6 + TU_{11} = 0.895 \leq 1$.

$PU_2 = TU_{10} + TU_2 = 0.984 \leq 1$.

$PU_3 = TU_7 = 0.725 \leq 1$.

$PU_4 = TU_8 + TU_3 = 0.913 \leq 1$.

$PU_5 = TU_9 + TU_1 = 0.927 \leq 1$.

3. STATIC TASK ASSIGNMENT HEURISTICS

The problem of optimally scheduling tasks on a uniprocessor or multiprocessor system is NP-complete [2]. Therefore, different heuristics have been proposed to schedule tasks. In this section we discuss four well known heuristic algorithms for the Task Assignment problem.

A Round Robin Task Assignment

The Round Robin task assignment heuristic assigns tasks in sequential order to the processors coming back to the first when all processors have been given a task [3].

The Procedure

The step by step procedure of the Round Robin Task Assignment algorithm used in our experiments is as follows:

1. Set i = 1.
2. Set p = number of processors in the system.
3. If all the tasks are assigned go to step 9, otherwise go to step 4.
4. Obtain the ith task, the next to be assigned in sequential order, call it T_i.
5. Set t = i mod p.
6. Assign T_i to P_t.
7. Set i = i + 1
8. Go to step 3.
9. End Procedure.

An example:

We run the Round Robin Task Assignment algorithm on the example stated in the 2nd section.
The above procedure yields the following solution vector:
a = (1,2,3,4,5,1,2,3,4,5,1).
The Processor utilizations are:
$PU_1 = TU_1 + TU_6 + TU_{11} = 0.927$.
$PU_2 = TU_2 + TU_7 = 0.990$.
$PU_3 = TU_3 + TU_8 = 0.913$.
$PU_4 = TU_4 + TU_9 = 0.890$.
$PU_5 = TU_5 + TU_{10} = 0.724$.
The objective function value, $f_{obj}(a) = 0.329$.

B. Increasing Utilization Balancing

According to the Increasing utilization balancing heuristic, the task with minimum utilization among the unassigned tasks is obtained. The obtained task is assigned to the then minimum utilized processor.

The Procedure

The step by step procedure of the Increasing Utilization Balancing algorithm used in our experiments is as follows:

1. If all the tasks are assigned go to step 6, otherwise go to step 2.
2. Obtain the task with minimum utilization among the unassigned tasks and call it T_{min}.
3. Obtain the processor with minimum utilization, call it $P_{min.}$
4. Assign T_{min} to P_{min}.
5. Go to step 1.
6. End Procedure.

An example:

We run the Increasing Utilization Balancing algorithm on the example stated in the 2nd section.
The above procedure yields the following solution vector:
a = (4,1,5,3,1,2,1,3,4,5,2).
The Processor utilizations are:
$PU_1 = TU_5 + TU_2 + TU_7 = 0.995$.
$PU_2 = TU_{11} + TU_6 = 0.692$.
$PU_3 = TU_4 + TU_8 = 0.852$.
$PU_4 = TU_1 + TU_9 = 0.927$.
$PU_5 = TU_3 + TU_{10} = 0.978$.

The objective function value, $f_{obj}(a) = 0.467$.

C. Arbitrary Utilization Balancing

According to the Arbitrary Utilization Balancing heuristic, the tasks are assigned one by one in turn in an arbitrary order. Each task is assigned to the then minimum utilized processor.

The Procedure

The step by step procedure of the Arbitrary Utilization Balancing algorithm used in our experiments is as follows:

1. .If all the tasks are assigned go to step 6, otherwise go to step 2.
2. Obtain the next task to be assigned in arbitrary order and call it T_{next}.
3. Obtain the processor with minimum utilization, call it $P_{min.}$
4. Assign T_{next} to P_{min}.
5. Go to step 1.
6. End Procedure.

An example:

We run the Increasing Utilization Balancing algorithm on the example stated in the 2nd section.
The above procedure yields the following solution vector:
a = (1,2,3,4,5,5,4,1,3,2,5).
The Processor utilizations are:
$PU_1 = TU_1 + TU_8 = 0.889$.
$PU_2 = TU_2 + TU_{10} = 0.984$.
$PU_3 = TU_3 + TU_9 = 0.951$.
$PU_4 = TU_4 + TU_7 = 0.923$.
$PU_5 = TU_5 + TU_6 + TU_{11} = 0.697$.

The objective function value, $f_{obj}(a) = 0.383$.

D. Decreasing Utilization Balancing

According to the Decreasing utilization balancing heuristic, the task with maximum utilization among the unassigned tasks is obtained. The obtained task is assigned to the then minimum utilized processor.

The Procedure

We present the step by step procedure of the Decreasing Utilization Balancing algorithm used in our experiments:

1. If all the tasks are assigned go to step 6, otherwise go to step 2.
2. Obtain the task with maximum utilization among the unassigned tasks and call it T_{max}.
3. Obtain the processor with minimum utilization, call it $P_{min.}$
4. Assign T_{max} to P_{min}.
5. Go to step 1.
6. End Procedure.

An example:
We run the Decreasing Utilization Balancing algorithm on the example stated in the 2nd section.
The above procedure yields the following solution vector:
a = (3,5,4,2,1,5,1,4,3,2,1).
The Processor utilizations are:

$PU_1 = TU_7 + TU_{11} + TU_5 = 0.825.$
$PU_2 = TU_{10} + TU_4 = 0.917.$
$PU_3 = TU_9 + TU_1 = 0.927.$
$PU_4 = TU_8 + TU_3 = 0.913.$
$PU_5 = TU_6 + TU_2 = 0.862.$

The objective function value, $f_{obj}(a) = 0.181$.

4. THE GROUPING GENETIC ALGORITHM

The potential of Genetic Algorithms to yield good solutions, even for hard optimization problems, has been demonstrated by various applications. A recently developed area in the GA research is the Grouping Genetic Algorithm (GGA). The GGA is a GA heavily modified to suit the structure of grouping problems. Those are the problems where the aim is to find a good partition of a set or to group together the members of the set. The GGA differs from the classic GA in the two important ways. Firstly, a specific encoding scheme is used so that the relevant structures of grouping problems become the genes of the chromosomes. Secondly, special genetic operators are used to suit the new encoding scheme. Both of these aspects avoid the weakness of the classic GAs applied to grouping problems [4].

In the Task Assignment problem, the aim is to find a good partition of the task set, so that each partition can be assigned to a unique processor. Here the number of partitions is limited by the number of processors in the system and the sum utilization of each partition should be less than one. We recognize the Task Assignment problem as a grouping problem and hence apply GGA to solve the Task Assignment problem.

A. The Encoding scheme

Neither the standard nor the ordering genetic operators are suitable for the grouping problems [5]. The simple chromosomes (in classic GAs) are item oriented, instead of being group oriented. Since the cost function of a grouping problem is defined in terms of the groups, the group entity should be captured in the chromosome. Therefore a two chromosome encoding was suggested in [4]. The first chromosome (item chromosome) will be the simple chromosome (group chromosome) is made up of groups. Now the genetic operators are applied on the group chromosome and the item chromosome and the item chromosome is updated accordingly, after each operation. For the Task Assignment problem the item chromosome is the task chromosome, each gene of the task chromosome represents the processor to which the particular corresponding task is assigned to. The length of the task chromosome is governed by the number of tasks in the system. Each gene in the group chromosome represents a processor. The length of the group chromosome is limited by the number of processors in the system.

B The Crossover Operator

A crossover operator should produce an offspring out of two parents in such a way that the resulting offspring inherits the meaningful information from both the parents to the maximum possible extent. In a GGA the crossover is applied on the group chromosome.
The GGA crossover proceeds as follows:

1. Select at random two crossing sites, delimiting the crossing section, in each of the two parents.
2. Inject the contents of the crossing section of the first parent at the first crossing site of the second parent.
3. Eliminate all items now occurring twice from the groups they were members of in the second parent.
4. If necessary, adapt the resulting groups, according to the hard constraints of the problem and the cost function to optimize. At this stage, local problem dependent heuristics can be applied.
5. Apply the points 2 through 4 to the two parents with their roles interchanged in order to generate the second child.

A more complete explanation of the crossover operator may be found in [5].

C. The Mutation Operator

The mutation operator is operated over the group chromosome, where a randomly selected group is eliminated and a few items are removed from each of the remaining groups and a new group is then formed using any local problem dependent heuristic. For the Task assignment problem we use the Decreasing Utilization Balancing heuristic as the local problem dependent heuristic.

D. The Inversion Operator

A segment in the group chromosome is selected at random and the order of genes in that is reversed that is inverted.

E. The GGA Procedure

We present the GGA procedure used in our experiments.

1. Generate at random an initial population of POPSIZE individuals.
2. Evaluate each individual according to the objective function to optimize. If the current generation number is equal to MAXGEN terminate.
3. Use tournament selection to select first N_c individuals and perform crossover over them,

replace the last N_c individuals with the new off springs.
4. Mutate N_M individuals selected at random from the current population.
5. Apply the inversion operator to N_I randomly selected individuals from the current population. At this point, one generation has been completed and a new population has been created.
6. Go to step 2.

Where the POPSIZE stands for the population size in each generation and MAXGEN stands for the maximum number of generators for each GA run.

F. The Cost Function

Two conditions need to be met for a successful GA, namely,

1. A GA should propagate those individuals (groups) that define promising regions of the search space throughout the population. This will ensure an increased rate of sampling of the promising regions, according to the Schema Theorem [6].
2. A GA should ensure that the individuals (groups) representing points of the search space that are in promising regions are also recognized as such, i.e. serve as parents in the crossover. The crossover can then propagate the high-quality individuals (groups) they contain.

For the Task Assignment problem, according to the objective function, the processors with load equal (or nearly equal) to the average define the promising regions of the search space.
Where average is given by,

$$\text{Avg} = \sum (TU_i) \div M.$$

The first condition is thus met by casting groups (i.e. processors) as genes and applying crossover over the group chromosomes, hence transmitting groups rather than items(i.e. tasks). In order to meet the second condition, our cost function of the GA should be able to distinguish those individuals (i.e. processors) which have load equal (or nearly equal) to that of the average from others, since the former are the promising individuals. Consider two different solutions of the TA problem with two processors:
Solution 1:

PU_1 = Avg. + 0.1
PU_2 = Avg. + 0.1

Solution 2:

PU_1 = Avg. + 0.2
PU_2 = Avg.

Both the solutions score the same in terms of the objective function of the TA problem. However, the second solution has a promising individual (PU_2). Hence we need to have the second solution participate in the crossover more often than the former that is it should score more than the former, in terms of the cost function.
Consequently, we adapt the following the cost function:
$f_{cost}(a) = \sum \sqrt{(|PU_{mean} - PU_i|)}$, for all i belonging to [1, M].
Where,

$PU_{mean} = (PU_{sum}) \div M$, with
$PU_{sum} = \sum PU_i$, for all i belonging to [1, M].

The cost function, clearly, yields a lower value for the 2nd solution. Hence, the 2nd solution is preferred more than the 1st solution.

5. EXPERIMENTS AND RESULTS

A. Experimental Setup

We performed three sets of experiments. In the first set (Under load condition set) the sum of the utilizations of all tasks is less than the number of processors. In the second set (Just load condition set) the sum of the utilizations of all tasks is exactly equal to the number of processors. In the third set (Over load condition set) the sum of the utilizations of all tasks is slightly more than the number of processors. Processors may be slightly overloaded for the latter two sets; we designed these two experimental sets to observe how equally each heuristic would overload the processors (if overloading is allowed) in these conditions. Hence for the latter two sets, we dropped the hard constraint that each processor load should be less than one. In each of the three sets we varied the number of processors from 3 to 15. Each processor case was simulated for 10 different sets of 100 tasks each and the objective function values obtained are averaged to obtain the final objective function value. Each set of 100 tasks was obtained by generating 100 random floating point numbers each corresponding to the utilization of a task. The performance of the four heuristics discussed in the 2nd section and that of the GGA was compared in terms of the objective function value of the TA problem.
We performed the GGA experiments using a population size of 50 and tournament of size two as the selection strategy. The number of crossovers made in the formation is about 24. The number of mutations is about 20 and the number of inversions is about 5. The total number of objective function evaluations performed before obtaining the objective function value for each processor case (for each of 3 to 15) is (MAXGEN * POPSIZE * NUMBER_OF_TASK_SETS) 100*50*10, which is 50,000.

B. Experimental Results

We plotted the objective function value versus the number of processors for the four heuristics and the GGA. Figure 1, shows the plot for Under-Load case. Figure 2 and 3 show the plots for Just-Load and Over-Load cases respectively.

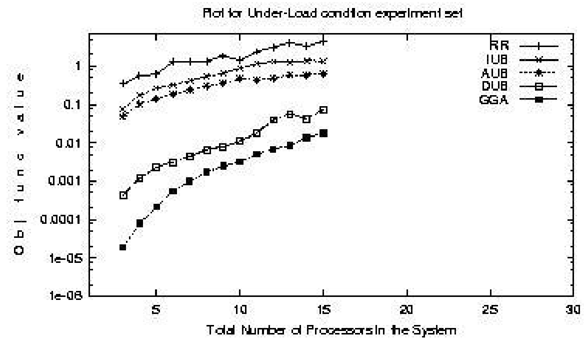

Figure 1: f_{obj} versus Number of processors in the system, with Under-Load conditions.

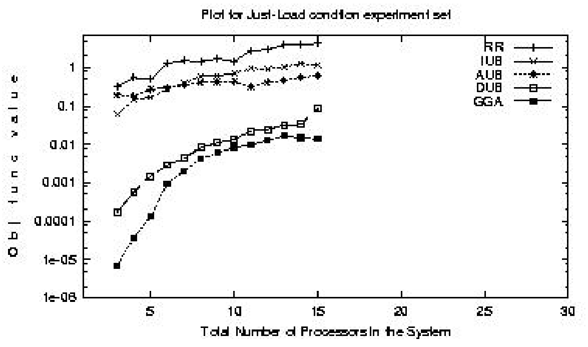

Figure 2: f_{obj} versus Number of processors in the system, with Just-Load conditions.

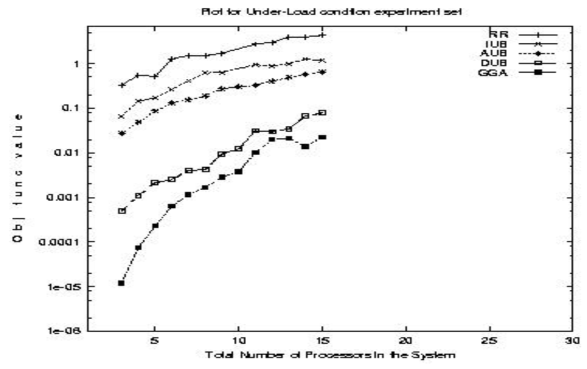

Figure 3: f_{obj} versus Number of processors in the system, with Over-Load conditions.

As the plots depict, the GGA performs better than the two heuristics.

6. CONCLUDING REMARKS AND FUTURE WORK

In this paper we have applied the popular GGA paradigm to solve the Task Assignment for a hard real time system. We have compared the performance of the GGA with four well known task assignment heuristics through simulations. We found that the GGA performs better than the four heuristic algorithms.

In the task assignment problem addressed in this paper, we ignored the failures in the system. However, failures do occur in hard real time systems. Fault-tolerance is an important issue in hard real time systems due to the critical nature of the supported tasks. The next step in this research is to study the Fault-Tolerant scheduling problem in hard real time systems.

7. REFERENCES

[1] D. Stinson, ***An Introduction to the Design and Analysis of Algorithms***, The Charles Babbage research center, Canada, 2nd Edition, 1987.

[2] M. R. Garey and D. S. Johnson, ***Computers and Intractability: a Guide to the Theory of NP-Completeness.*** W. H. Freeman, Nov. 1979.

[3] B. Wilkinson and M. Allen, ***Parallel Programming: Techniques and Applications using Networked and Parallel Computers***, Prentice Hall, 1999.

[4] E. Falkenauer, ***Genetic Algorithms and Grouping Problems***, John Wiley and Sons Inc., 1st Edition, 1998.

[5] E. Falkenauer, "A hybrid Genetic Algorithm for Bin Packing", ***Journal of Heuristics***, 1996, pp 2: 5-30.

[6] D. E. Goldberg, ***Genetic Algorithms in search, optimization and Machine Learning***, Pearson Education Asia, 2000.

[7] J. W. S. Liu, ***Real Time Systems***, Pearson Education Asia, 2000.