An Adaptive Genetic Algorithm for Multiprocessor Task Assignment Problem with Limited Memory

Abbas Mehrabi , Saeed Mehrabi, and Ali D. Mehrabi

Abstract—**Multiprocessor and distributed systems both play a vital role in high performance computing. One of the most important issues in this area is to assign a set of tasks on a set of processors with limited memory to provide load balancing. Recently genetic algorithms, which are in the class of stochastic search algorithms, have been used for most combinatorial optimization problems. In this paper we solve the task assignment problem with considering load balancing by new method, based on the genetic (GAs) algorithms. Our GA employs a repair function to guarantee valid assignments during the process of algorithm. According to the effectiveness of this algorithm in comparison with heuristic algorithms including branch and bound and graph cuts, it can be used for task assignment problem in most parallel processing environments.**

Index Terms— **Genetic Algorithms; Combinatorial Optimization; Multiprocessor Systems; Load Balancing; Parallel Processing Environments.**

I. Introduction

Parallel processing systems, are used in most applications, such as information processing, weather modeling, database systems, real-time simulation of dynamic systems and image processing today. The maximum efficiency of these systems can be obtained, when the task assignment and partitioning methods are applied effectively. This problem has been proved to be *NP*-Complete [18]. Several approaches to the task allocation model have been identified. They are basically graph theoretical, integer programming and heuristic method.

In graph based approach, each running task in the system and the cost induced by the communication delay between them residing in separate processors are represented by a node and a weighted edge, respectively [17]. It minimizes the total interprocessor communication cost by performing a partitioning algorithm on the graph such that, each partition includes a set of tasks, which are assigned to a specified single processor. The limitation of its generalization is the high computing time complexity when more than two processor are used in the allocation model. Furthermore, it is difficult to incorporate various constraints into such a model. An instance of this task graph is shown in figure 1. The graph has eight nodes corresponding to problem tasks and some edges between them indicating the communication delay as the edges label.

The integer programming method [6], [7] is based on the implicit enumeration subject to the additional constraints. It allows constraints to be easily incorporated into the allocation model to meet various application requirements. This approach is limited by the amounts of time and memory needed to obtain an optimal solution since they grow as exponential functions of the problem order.

The heuristic method is to provide fast and effective algorithm for a suboptimum solution [9]. This technique requires less computation time than integer programming methods. They are useful on applications where an optimum solution is not obtainable within a critical time limit. They are also applicable to larger dimensional problems.

This paper is organized in 5 sections: in section 2 the multiprocessor task assignment problem is formulized. The section 3 is related to surveying the proposed genetic algorithm. We analyze our algorithm in section 4 by simulation. Finally we conclude this paper in section 5.

II. Problem Formulation

The task assignment problem in multiprocessor systems is defined as the assignment of n tasks $T_1, T_2, \ldots, T_n$ to m processors, $P_1, P_2, \ldots, P_m$ to achieve the following goals:

1. allow specification of total CPU load and memory constraints to facilitate a variety of engineering application requirements,
2. balance the utilization of individual processors in the multiprocessor computing system, and
3. minimize interprocessor communication cost.

The design of a mathematical model for task allocation for a multiprocessor computing system involves the following steps:

1. formulate the cost function to measure the processor communication (IPC) cost and processing cost, 2. formulate the problem constraints to meet the diverse requirements, and 3. drive an iterative algorithm to obtain a minimum total cost solution.

Manuscript received July 3, 2009; Revised July 29, 2009.

Abbas Mehrabi is now M.S.C student at Computer Engineering Department, Islamic Azad University, South Tehran Branch, Tehran, Iran. (email: abbas.mehrabi8141@gmail.com).

Saeed Mehrabi is B.S student at Department of Computer Science, Shahid Bahonar University of Kerman, Kerman, Iran. (email: mehrabi235@gmail.com).

Ali D. Mehrabi is with the Department of Mathematical and Computer Science, Yazd University, Yazd, Iran. (email: mehrabi@yazduni.ac.ir).

A. Problem Statement

The cost function is formulated as the sum of the IPC cost and the processing cost. If two different running tasks are assigned to a same processor, the communication delay between them is dissembled, since data transmission is done effectively. In contrast, if these two tasks are assigned to two different processors the cost due to the communication delay between them takes account in the total system cost.

According to assignment literature there are some parameters for problem formulation. We first introduce these parameters and return them anywhere needed. Our multiprocessor computing system has the interconnections in the form of heterogeneous in which the connected processors contributed to the system have not the same processing ability. Each processor has its local limited memory. In any static specified time interval, the set of the assigned tasks to any processor are located in its local memory for sequential execution. We denote m_i as the amount of memory needed for task i and c_{in} as the processing requirements of task i for execution on processor n. In other hands for processors, we have M_n as memory capacity of processor n, P_n processing capacity of processor n, c_{ijnl} communication cost between task i and j if i is assign to processor n and j is assigned to processor l, and

$$x_{in} = \begin{cases} 1 & \text{if task } i \text{ is assigned to processor } n, \\ 0 & \text{otherwise} \end{cases}$$

The total cost for processing the tasks is stated as [5]

$$\min\{\sum_i \sum_n c_{in} x_{in} + \sum_{(i,j)\in E} \sum_n \sum_l c_{ijnl} x_{in} x_{jl}\} \quad (1)$$

Subject to

$$\sum_n x_{in} = 1 \qquad \forall i, \quad (2)$$

$$\sum_i m_i x_{in} \le M_n. \quad (3)$$

as the problem constraints. Eq. (2) reveals this fact that some task is assigned to one and only one processor during the program execution. Eq. (3) states that the amount of memory required for all tasks assigned to a processor must not exceed the processor memory capacity.

B. Related Works

Richard and *et al.* [16] have used an algorithm based on branch and bound (BB) method. To employ the BB technique the allocation problem is represented by a search tree. The allocation decision represents a branching at the node corresponding to the given task. Consider a problem of allocating m tasks among n processors. Starting with task 1, each task is allocated to one of the n processor subject to the constraints imposed on the relations on tasks and processors. The number of tree levels m corresponds to m tasks. A feasible sequence of successive branches is called a *path*. A path from the root node to the last node corresponds to a *complete allocation*; otherwise, it is a *partial allocation*. The cost of a path is computed according to the cost equation.

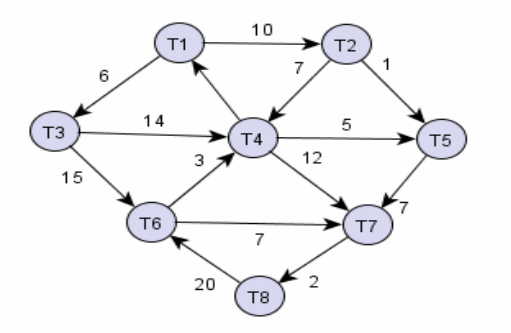

Fig. 1: A communication task graph.

Although this approach finds the optimum solution in most cases, it needs a lot of backtracks for large problem instances resulting in exponential time and space complexity.

Stone [17] investigates the underlying problem with the aid of network flow algorithms and graph cut approaches. This paper shows how the well-known Ford-Fulkerson algorithm for finding the maximum-flow can find an optimal partition of a modular program that runs on a two-processor system and its generalization to systems with three or more processors. To minimize the total running time, Stone modifies the module interconnection graph so that each *cutset* in the modified graph corresponds in a one-to-one fashion to a module assignment and the weight of the cutset is the total cost for that assignment. With this modification, we can solve a maximum flow problem on the modified graph. The minimum weight cutset obtained from this solution determines the module assignments, and this module assignment is optimal in terms of total cost. In the case of two or more processors, Stone distinguishes the processors in two categories: *source* and *sink* types in the interconnection graph and computes the maximum flow between these nodes by creating a cutset. The weight of the cutset corresponds to the total running time. The cutset with minimum weight creates an optimal assignment.

The maximum flow approach is more efficient than backtracking because worst-case performance of backtracking has a much greater complexity than maximum flow algorithm complexity. However the problem of finding the maximum flow in an interconnection graph is still a time consuming problem. In addition the problem extension to three or more processors using this approach is itself an intractable problem.

In next section we introduce our genetic algorithm which solve the underlying problem far better than previous algorithms in both time and space complexity according to experimental results.

III. The Genetic Algorithm

Genetic Algorithms (GAs) [8] which are famous meta-heuristics based on the mechanism of natural selection and natural genetics were first developed in 1960 by John Holland [11] at the Michigan State University. Recently, GAs have frequently been used for solving many search and optimization problems (see e. g. [14], [15]). The basic concept of GA is designed to simulate the processes in natural system necessary for evolution, specifically for those that follow the principle of survival of the fittest, first laid down by Charles Darwin.

The general outline of a genetic algorithm can be stated as follows. GA starts by generating a random population of candidate solutions. At each iteration of the algorithm a population of promising solutions is first selected. Various operations are then applied to this population in order to produce new candidate solutions. Two common and may most important of these operations are *crossover* and *mutation*. The crossover operator is applied to exchange partial solutions between pairs of solutions. There are many random methods for doing crossover in literature.

The first step in designing a genetic algorithm for a particular problem is to develop a suitable representation scheme, that's, a way to represent individuals in the GA population. Our algorithm uses an appropriate representation for chromosomes, with considering the task interconnection graph, the number of processors and memory constraints. As an example, consider the task graph shown in Fig. 1. A task assignment chromosome for this task graph is shown in Fig. 2.

Setting the i^{th} cell (called *gene* in GAs literature) of the chromosome array to j means the assigning of i^{th} task to processor j. This example uses 3 processors, P_1, P_2 and P_3.

A correct task assignment chromosome representation is one in which, the total amount of memory, which is needed for executing the tasks assigned to some processor does not violate the memory capacity of that processor. The total cost for a given assignment is the total execution time of the tasks assigned to all processors plus the sum of the communication delay time between tasks on separate processors.

A. The initial population generation

The first population of GAs plays an important role in reproductivity of individuals in the next generations. To create the initial population of chromosomes, we must assign tasks to processors in chromosome representation such that the total amount of memory, which is needed for executing the tasks assigned to each processor, does not violate the memory capacity of corresponding processor. We designed the procedure 1 for initial population generation, which is shown in figure 3. We mean the $M[P_i]$ by the amount of memory, which is needed for executing task P_i on some processor. *CPUMemory[j]* is related to the memory capacity of processor j.

Task i:	1	2	3	4	5	6	7	8
Processor j:	1	2	1	2	3	2	1	1

Fig. 2: Chromosome representation of task assignment.

B. Crossover and mutation operators

This proposed algorithm uses the two point crossover method, which has been used in most GAs for solving problems related to multiprocessor environments [14]. After some populations conducted, the characteristics of almost children like to be the same. In fact, the future individuals would like to inherit the all of their own attributes from the parents. This means that the individuals in the population can only go through a special part of search space and this cause to miss some best solutions. For searching almost the search space we use the mutation operator. Once a child solution has been generated through crossover, a mutation procedure is performed that mutates few randomly selected genes in the child solution. The rate of mutation is very important in GAs and is generally set to be a small value. The crossover operator procedure is shown in figure 4.

C. Repair operator

After the crossover and mutation operators, it is possible that some chromosomes to be invalid due to violation of processor memory capacity. For this reason we have used a new operator, which is called the *repair operator*. This operator is responsible for the conversion of the invalid child chromosomes to valid chromosomes.

Initial Population Generation Procedure 1

```
Begin
  Input: n: the number of tasks.
  Output: chrom: a new chromosome.
  for i:=1 to n do
    Begin
      int CPU[];
      int index=0;
      for k:=1 to CPU_number do
        if CPUmemory[k]-M[Pi]>=0
            CPU[++index]=k;
      int x=CPU[random(1,index)];
      CPUmemory[x]=CPUmemory[x]-M[Pi];
      chrom[j]=x;
    End
  Output: chrom.
End.
```

Fig. 3: The initial population generation procedure.

```
Crossover Operator Procedure 2

Begin
  Input: firstparent, secondparent: two parent
         chromosomes.
         n: the number of tasks.
  Output: firstchild, secondchild: two child
          chromosomes.
    int i=random(1,n-1);
    int j=random(i+1,n);
    firstchild=firstparent(1,i-1)//secondparent(i,j)
    //firstparent(j+1,n);
    secondchild=secondparent(1,i-1)//firstparent(i,j)
    //secondparent(j+1,n);

  Output: firstchild, secondchild.
End.
```

Fig. 4: The crossover operator procedure.

It starts with traversing the chromosome array and upon arriving to an invalid task assignment then searches another first processor such that its remaining memory is greater than or equal to the amount of memory which is needed for executing the corresponding task. The underlying task will be reassignment to the processor which is founded. The repair operator procedure is shown in Fig. 5.

D. Evaluation function and Selection operator

To compute the total system cost, we consider the execution time of the assigned tasks to all processors plus the communication delay time between them on the separate processors. Thus our final objective is to find a chromosome representation such that first it does not violate the memory capacity of processors and second the total system cost according to the chromosome representation to be optimal.

To select the chromosomes for the next generation of the algorithm, a number of chromosomes with small values for their evaluation function are moved to the next generation without any change. Among the remaining chromosomes in order, we apply the crossover operator to a number of them according to the crossover operator probability (P_c). Finally we apply the mutation operator to the remaining chromosomes with the large value for their evaluation function. The new generated offsprings are moved to the next generation to complete it.

IV. Experimental Results

In this section, we will present the experimental results and analyze the computational performance. The platform for conducting the experiments is a PC with a 2.4 GHz CPU and 512 MB RAM. All programs are coded in Java programming language in NetBeans IDE 6.5.1. We also have used the *yED Graph Editor* software from *yWorks* company for drawing the task graph. To simulate our genetic algorithm for interconnection task graph in figure 1, first we have used the data sets shown in table 1 and table 2 respectively. The data in table 1 stands for memory capacity of each processor in the heterogeneous multiprocessor system. Table 2 includes the execution time for executing some task, say T_i, on some processor, say P_j, plus the amount of memory for execution.

The stopping criterion in most GAs is the number of generations such that no improvement is obtained in the value of evaluation function. Parameter setting is an important component of a standard genetic algorithm which determines the convergence rate of the algorithm. According to our practice experiments, we find the following settings best fitted for our implementation. In all execution instances, we have set the algorithm parameters to probability of crossover operator P_c=0.7 and the probability of mutation operator P_m=0.3. In the case of task graph figure 1 after the execution of the algorithm with the given parameters we got the minimum cost 54 that is obtained from the task assignment given in table 3 after 100 times successive generations of the algorithm.

We have tested our algorithm on a variety of large task graph instances with random data sets to verify the correctness of our algorithm. Results have shown in Table 4. For each graph instance, the optimum solutions, which obtained after some generations, along the number of corresponding running processors for assignment are given in table. The results obtained not only are almost consistent with the two previous algorithms described in subsection 2.2, but importantly also they have a lower time complexity for finding the optimum solution.

```
Repair Operator Procedure 3

Begin
  Input: chrom: invalid chromosome.
         n: the number of tasks.
  Output: newchrom: valid chromosome.
  for i:=1 to n do
  Begin
    int y:=chrom[i];
    if CPUmemory[y]-M[Pi]<0 do
    Begin
      int CPU[];
      int index:=0;
      for k:=1 to CPU_number
        if CPUmemory[k]-M[Pi]>=0
          CPU[++index]:=k;
      int z:=CPU[random(1,index)];
      CPUmemory[z]:=CPUmemory[z]-M[Pi];
      newchrom[i]:=z;
    End
    else
      CPUmemory[y]:=CPUmemory[y]-M[Pi];
  End
  Output: newchrom.
End.
```

Fig. 5: The repair operator procedure.

Table 1: The memory capacity for each processor.

CPU Number	Memory Capacity
P_1	20 MB
P_2	15 MB
P_3	10 MB

TABLE 2: The execution time and amount of memory for each task.

Task Number	Execution time on P_1	Execution time on P_2	Execution time on P_3	Amount of memory
T_1	10 ns	7 ns	15 ns	3 MB
T_2	12 ns	5 ns	8 ns	7 MB
T_3	8 ns	12 ns	10 ns	4 MB
T_4	15 ns	14 ns	12 ns	2 MB
T_5	20 ns	16 ns	17 ns	5 MB
T_6	14 ns	17 ns	15 ns	2 MB
T_7	13 ns	10 ns	8 ns	6 MB
T_8	5 ns	2 ns	7 ns	4 MB

TABLE 3: The optimal task assignment.

CPU Number	The set of assigned tasks
P_1	$\{T_1, T_2, T_3\}$
P_2	$\{T_5, T_7\}$
P_3	$\{T_4, T_6, T_8\}$

TABLE 4: The results obtained by running our algorithm on "**tgins.***" task graph instances. The last column indicates the optimum solutions which are as same as our algorithm results.

Task graph instance (tgins)	Number of tasks	Number of communication edges	Number of Processors for assignment	Number of generations	Optimum solution
tgins 01	20	15	3	150	285
tgins 02	40	25	4	170	326
tgins 03	50	30	4	170	354
tgins 04	80	40	8	210	1025
tgins 05	100	60	15	220	1763
tgins 06	150	75	18	220	1952
tgins 07	200	120	25	310	2375
tgins 08	300	230	25	350	4578
tgins 09	500	350	80	380	6756
tgins 10	700	560	85	410	9398

V. Conclusion

One of the most important problems in multiprocessor systems is to assign a set of tasks on a set of processors with limited memory to provide load balancing. Some heuristic methods including branch and bound (BB) and graph cuts algorithms are developed for task assignment problem with some constraints. Although these approaches find the optimal task assignment, they have high time complexity. Also in the case of cut graph approach, problem extension to three or more processors is very difficult and time consuming.

In this paper we have presented a new method based on genetic algorithms for solving the task assignment problem in heterogeneous multiprocessor platforms with respect to load balancing. To prevent of creating invalid task assignments, our algorithm employs a repair operator. According to the experimental results, our algorithm finds the optimum solution in large graph instances in lower time complexity than the heuristic methods.

Acknowledgment

The authors would like to thank anonymous referee for suggestions which led to a substantial improvement of this paper. The authors are also privileged to thank their parents who remember them there are many other things in life behind writing paper!

References

[1] A. Billionnet, M. C. Costa, A. Sutter, "*An efficient algorithm for the task allocation problem",* J. ACM 39 (3) (1992) 502-518.

[2] U. Bodenhofer, "*Genetic Algorithms: Theory and Applications*". Lecture Notes. 3rd edition, 2003.

[3] S. Bokhari. *"Assignment Problems in Parallel and Distributed Computing"*, Kluwer, Boston, 1987.

[4] S. H. Bokhari. *"A shortest tree algorithm for optimal assignments across space and time in a distributed processor system",* IEEE Trans. Software Engrg. SE-7 (6) (1981).

[5] W. W. Chu, L. M. Lan, *"Task allocation and precedence relation for distributed real-time systems",* IEEE Transactions on Computers 36 (1987) 667-669.

[6] W. W. Chu, "*Optimal file allocation in multiple computing system*", IEEE Trans. Comput., vol. C-18, pp. 885-889, Oct. 1969.

[7] O. I. El-Dessouki, and W. H. Huan, "*Distributed enumeration on network computers*," IEEE Trans. Comput., vol. C-29, pp. 1068-1079, Dec. 1980.

[8] D. E. Goldberg, "*Genetic Algorithms in Search, Optimization and Machine Learning"*. Massachusetts: Addison Wesley, 1989.

[9] V. B. Gylys and J. A. Edward, "*Optimal partitioning of workload for distributed systems*". In Dig. COMPCON, Fall 1976, 1976, pp. 353-357.

[10] Y. Hamam, Kh. S. Hindi, "*Assignment of program modules to processors: A simulated annealing approach"*, European Journal of Operational Research, 122 pp. 509-513, 2000.

[11] J. H. Holland, *"adaptation in natural and artificial systems".* MA: MIT Press, 1992.

[12] V. M. Lo, *"Heuristic algorithms for task assignment in distributed systems",* IEEE Transactions on Computers, 1988, pp. 1384- 1397.

[13] V. F. Magirou, J. Z. Milis, *"An algorithm for the multiprocessor assignment problem",* Oper. Res. Lett. 8 (1989) 351-356.

[14] A. Mehrabi, S. Mehrabi, "*A New Genetic Algorithm for Multiprocessor Scheduling Problem*", (in Persian). In Proc. of the National Conference on Software Engineering (RSEC), Tehran, Iran, 2009.

[15] S. Mehrabi, A. Mehrabi and A. D. Mehrabi, "*A New Hybrid Genetic Algorithm for Maximum Independent Set Problem*", In Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT'09), Sofia, Bulgaria, 2009.

[16] P. Richard, E. S. Lee, and M. Tsuchiya, "*A task allocation model for distributed computing systems*", IEEE Trans. on Computers, vol. C-31, pp. 41-47, No. 1, Jan. 1982.

[17] H. S. Stone. *"Multiprocessor scheduling with the aid of network flow algorithms",* IEEE Trans. Software Engrg. SE-3 (1) (1977) 85-93.

[18] J. D. Ullman, " *NP-Complete Scheduling Problems*", JCSS,10(1975),384-93.