Load Balanced Allocation of Multiple Tasks in a Distributed Computing System

Biplab Kumer Sarker[1], Anil Kumar Tripathi,[2]
Deo Prakash Vidyarthi[3], Laurence Tianruo Yang[4] and Kuniaki Uehara[5]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
[2] Institute of Technology, Banaras Hindu University, Varanasi, India
[3] Jawaharal Nehru University, New Delhi, India
[4] Department of Computer Science, St. Francis Xavier University, Canada
[5] Graduate School of Science and Technology, Kobe University, Japan
sarker@unb.ca[1] anilkt@bhu.ac.in[2] dpv@mail.jnu.ac.in[3]
lyang@stfx.ca[4] uehara@kobe-u.ac.jp[5]

Abstract. A Distributed Computing Systems (DCS) calls for proper partitioning of tasks into modules and allocating them to various nodes so as to enable parallel execution of their modules by individual different processors of the system. A number of algorithms have been proposed for allocation of tasks in a Distributed Computing System. Most of the models proposed in literature consider modules of a single task for static allocation, for the purpose of allocation onto a DCS. Moreover, they did not consider the architectural capability of the processing nodes and the way of connectivity among them. This work considers allocation of disjoint multiple tasks with their corresponding modules and proposes a parallel algorithm for a realistic situation wherein multiple disjoint tasks with their modules compete for execution on an arbitrarily connected DCS based on well-known A* technique. The proposed algorithm also considers a load balanced allocation for the purpose. The paper justifies the effectiveness of the algorithm with the experimental results by comparing with previously reported works.

1 Introduction

In a distributed computing systems, processing nodes networked together, participate in various computational tasks to achieve minimum turn around time of the submitted tasks. The problem becomes more complex when the communicating modules in a task itself are assigned to different processing nodes to achieve the goal. Therefore, the inter module communication cost(IMC) needs to be minimized to obtain the minimum turn around time. On the other hand, the capacity of computational load for each processing node is needed to be considered for the whole system to maximize its throughput. This problem has been studied as task allocation or mapping problem in the literatures [2-6, 8-11]. Since the problem is NP-hard [2, 9] and thus many heuristic solutions are possible for this problem.

Most of the algorithms for Task Allocation (TA) problem proposed by the scientists and researchers [2, 6] so far, do make one or more assumptions. These consider a single task partitioned into corresponding modules for the execution and the repercussion of a single task allocation on a DCS. Whereas in reality, a DCS receives number of tasks from time to time for the execution. Factually, a DCS facilitates concurrent execution of modules belonging to various unrelated tasks [7, 12, 13]. The modules of any particular task, having IMC, do cooperatively execute and do not depend on the modules of the other tasks. This leads to the situation wherein, a processing node may be assigned modules belonging to different tasks. It is to mention that the real issue of task allocation must not ignore the possibility of multiple modules assignment of various tasks to the processing nodes in a dynamic fashion [7].

Considering these view and furthermore, taking into account the architectural capability of the processing nodes and the optimality of the solution guaranteed by A* based TA [2], in this paper we present a parallel algorithm for load balanced allocation in a DCS.

The paper is organized as follows. The next section discusses the load parameter for multiple tasks which is used in our case as a cost function to minimize turnaround time. This is the basis of effectiveness of the allocation. In section 3, the A* algorithm for task allocation is proposed. Three examples are also illustrated in this section. In the next section, the comparative observations with the existing algorithms are presented. Section 5 also justifies the effectiveness and scalability of our algorithm. Finally, the work is concluded indicating the future directions.

1.1 Assumptions

As the task allocation problem remains to be NP-hard, various heuristic solutions are proposed with one or more assumptions [2]. This work also makes certain assumptions that are as follows.

1. The processing nodes in the DCS are heterogeneous. The tasks are disjoint and have no inter-task communication. Only the modules within a task have interdependencies and communication requirements.

2. Execution and communication matrices for the task graphs are assumed to be given. These matrices are different for every task and calculated in units of time. While partitioning the task into modules, we assume that the memory requirements of the modules are also calculated.

3. The assumption of the availability of interconnection graph accommodates nonregular type of interconnection networks.

Here, in this paper, the word ‘processor’ and ‘processing node’, ‘assignment’ and ‘allocation’ have been used to refer the same.

2 Load

The tasks submitted into a DCS are partitioned into suitable modules and then these modules are to be allocated to the processing nodes. Each task can be

represented by a Task Graph $(TG) = (V_t, E_t)$, where (1) V_t is a set of vertices, each of which represents a module of the task $m_1, m_2, ..., m_n$ and (2) $E_t \subseteq V_t \times V_t$ is a set of edges each of which represents the Inter Module Communication (IMC) between the two modules at the end of the edge. We can also represent the network of processors $p_1, p_2, ..., p_n$ in a DCS as a Processor Graph $PG = (V_p, E_p)$; where vertices represent the processors and the edges represent the communication links between processors (see Fig.2). The goal of TA is to allocate the Task Graphs (TG) to a network of processors in a DCS (i.e. to PG) to achieve the minimum turn-around time of tasks [2].

A processor's load comprises of all the execution and communication costs associated with its assigned modules of the task [6]. The time required by the heaviest-loaded processor will determine the entire tasks' completion time. So, the TA problem must find a mapping of the set of m modules of l tasks to n processors so as to minimize tasks completion time. Our goal is to allocate the modules in such a way that does not cause any processing node to be overloaded because an overloaded node may affect adversely in the turn around time of the tasks in a heterogeneous DCS.

The load in a processing node p is calculated as follows

$$\sum_{l=1}^{k}\sum_{i=1}^{mi} X_{ilp}.M_{ilp} + \sum_{\substack{q=1\\q\neq p}}^{n}\sum_{l=1}^{k}\sum_{i=1}^{mi}\sum_{\substack{j=1\\j\neq i}}^{m}(C_{ijl} + CC_{pq}).M_{ilp}.M_{jlq} \qquad (1)$$

where, $CC_{pq} = C_{fi}.L^{i}_{pq}$

X_{ilp} = execution cost of ith module of task l on processing node p
C_{ijl} = Inter-Module Communication(IMC)Cost between i^{th} and j^{th} module of task l
M_{ilp} = assignment matrix of ith module of lth task on processing node p

$$M_{ilp} = \begin{cases} 1 & \text{if module } m_i \text{ of task } l \text{ is assigned to processor } p \\ 0 & \text{otherwise} \end{cases}$$

M_{jlq} = assignment matrix of jth module of lth task on any other processing node q

$$M_{jlq} = \begin{cases} 1 & \text{if module } m_j \text{ of task } l \text{ is assigned to processor } q \\ 0 & \text{otherwise} \end{cases}$$

L^{i}_{pq}= connection matrix of two processors p and q, describing the links (direct/ single indirect/ double indirect etc.) of connection paths among the processing nodes in Processor Graph (PG).
C_{fi}= coefficient matrix which has n entries describing the IPC (Inter Processor Communication) costs for the links of connection paths among the processing nodes. For example, C_{f1}=5 (for direct connection between the processors),

C_{f2}=10 (for processors which are indirectly connected by one link), C_{f3}= 20 (for processors which are indirectly connected by two links) etc.

The first part of the above equation 1 is the total execution cost of the modules of all the tasks allocated on a processing node p. The second part is the communication overhead on p with the modules of the tasks allocated on the other processing node such as q in the DCS. The ith entry of the coefficient matrix C_{fi} corresponds to communication between two processors via i links. If processors p and q are not directly connected, we find L^2, multiply it by C_{f2}, (2nd field of C_f), and check whether this comes out to be non-zero; if it does, we replace L^1 in calculation with L^2; if not, we find out $L^3$and multiply it with C_{f3} and check whether the product comes out to be non-zero. We continue like this until we find a non-zero value and then replace L^i in calculation with this (it is to be mentioned that we shall find a non-zero value within n multiplications, where n is the no. of processing nodes).

2.1 Global Table(GT)

To allocate the modules optimally so that no processor becomes overloaded, the load on each of the n processing nodes needs to be computed. By finding the processing node with heaviest load, the optimal assignment out of all possible assignments will allot the minimum load to the heaviest loaded processor. Thus it is necessary to consider realistic view that only a finite number of modules can be allocated to a processor depending on the architectural capability of the processing nodes in a DCS. Consequently, earlier algorithms [2, 5, 6] have continued to assume that all the modules will be eventually allocated no matter how large the memory requirements are, and/or how many modules a processor can accommodate and what is the current status of the system due to the existing allocation. These algorithms do not consider the requirement of allocation of modules of multiple tasks. In the proposed algorithm, we have shed off these unrealistic assumptions and make use of a data structure STATUS associated with every processor, which has two fields showing: the maximum number of modules that can be allocated to the processor and the memory capacity of the processor.

Whenever a module is chosen for allocation onto a processing node, the STATUS is checked and it is ascertained whether the processor can accommodate the module at hand. If not, another processor is chosen if available. The consequence might be that a certain module is not allocated at all. This data structure is implemented by constructing a Global table (GT) to maintain the track of maximum number of modules that can be allocated to a processing node depending upon its memory capacity. This is a dynamic table, which keeps the information of the remaining memory of nodes and the number of modules can be allocated on the nodes. Whenever a new task arrives, this GT is to be consulted and to be modified. Here, we present the basic structure of the GT. In this table, it is shown that there are 4 processing nodes in a DCS. The number of modules, the processing nodes can accommodate are 4, 3, 4, and 5, respectively,

and the memory capacity of the nodes are 10, 8, 9 and 12 respectively. After some modules are assigned the other column of the processing nodes will be filled up by the corresponding numbers which is shown in the illustrative examples in sec. 4.

Table 1. The structure of the GT.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	4	10			
p_2	3	8			
p_3	4	9			
p_4	5	12			

Here, P_{node}= Processing node of the DCS
M_{mod} =Maximum number of modules can be assigned
M_{cap} = Maximum memory capacity of a P_{node}
Mod_{assign} = Modules assigned of each task
R_{mod} = Remaining number of modules can be assigned
R_{mem} = Remaining available memory

3 Proposed Algorithm for TA

In the A* algorithm [1, 2], for a tree search, it starts from the root, usually is called the start node (usually a null solution of the problem). Intermediate tree nodes represent the partial solutions, and leaf nodes represent the complete solution or goal. A cost function f computes each node's associated cost. The value of f for a node n, which is the estimated cost of the cheapest solution through n, is computed as

$$f(n) = g(n) + h(n) \tag{2}$$

Where, $g(n)$ is the search-path cost from the start node to the current node and $h(n)$ is a lower-bound estimate of the path cost from current node to the goal node (solution), using any heuristic information available. To expand a node means to generate all of its successors or children and to compute the f value for each of them. The nodes are ordered for search according to the cost; that is, the algorithm first selects the node with the minimum expansion cost. The algorithm maintains a sorted list, called OPEN, of nodes (according to their f values) and always selects a node with the best expansion cost. Because the algorithm always selects the best-cost node, it guarantees an optimal solution [2].

To compute the cost function, $g(n)$ is the cost of a partial assignment at node n which is the load on the heaviest loaded processing node (p_i); this is done using

the equation 1. For the computation of $h(n)$, two sets A_p (the set of modules that are already assigned to the heaviest loaded p) and U (the set of modules that are unassigned at this stage of the search and have one or more communication links with any module in set A_p), are defined. Each module m_i in U will be assigned either to p or any other processor q that has a direct or indirect communication link with p. So, two kinds of costs with each m_i's assignment can be associated: either X_{ilp}(the execution cost of m_i of task l on p) or the sum of communication costs of all the modules in set A_p that has a link with m_i. This implies that to consider m_i's assignment, it is to be decided whether m_i should go to p or not (by taking the minimum of these two cases' cost).

To support the run-time allocation of tasks to processors, we construct a manager-worker style parallel algorithm whose pseudo-code is given in sec. 3.1. One processor called the manager is responsible for keeping track of the assigned and unassigned tasks using a Global Table (GT) which is consulted and updated during every allocation. It always consists of the information about the total memory of the processing nodes and the remaining memory after assignment, no. of assigned modules and the remaining no. of modules can be assigned.

3.1 The Algorithm

1. *As a 'Manager' node, processor P_0 maintains the status of the Global Table (GT) for each processing node($P_1, P_2, ..., P_n$) termed as 'worker' in terms of available memory (M) and the modules that are already assigned to it.*
2. *'Manager' node maintains a list S of unallocated tasks with all modules (all tasks are in S at the beginning) and a list OPEN, empty at the beginning. Another list V is maintained by taking one Task ta from S and put it in another list V and reset OPEN.*
3. *The 'workers' checks possible allocation of modules in V using the A*(2) algorithm and verifying STATUS of them by P_0; then allocate them; if not possible, deallocate the partially allocated modules of the task and move onto the next task, modifying the STATUS in between and update the Global Table (GT)by the Manager.)*
4. *If S is not empty yet, go to step 2.*
5. *Stop (end of allocation).*

4 Implementation Results

In this section, we present three small examples with various number of TGs and PGs to justify the proposed algorithm with respect to allocation and status of the global table.

Case 1

For case 1, we have considered a set of three tasks shown as TGs partitioned with their corresponding modules $T_1(m_{11}, m_{21}, m_{31}, m_{41})$, $T_2(m_{12}, m_{22}, m_{32})$, $T_3(m_{13}, m_{23}, m_{33})$ and a DCS as PG, consists of four processors (p_1, p_2, p_3, p_4)

interconnected as shown in Fig. 2. Here, the IMC costs shown as in the figure represent the communication costs between the modules of the tasks in time unit. For example, the communication cost between m_{11} (the first module of task T_1) with m_{21} (the second module of task T_1) is 10 unit. The adjacency matrix L^i_{pq} of processing nodes are assumed to be given which represents how the processing nodes are connected among each other. For example, the processing nodes p_2 and p_3 are not directly connected, so $L^1_{p2p3} = 0$. But they are connected with at least one indirect link (through p_1 or p_4). So, $L^2_{p2p3} = 1$.

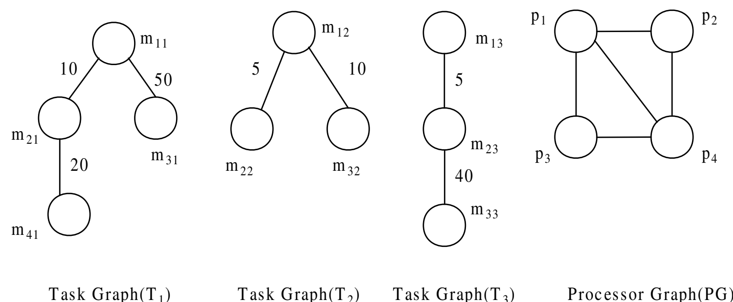

Fig. 1. Example of task graphs T_1, T_2 and T_3 with their modules and a DCS as processor graph.

The Results for case 1

Total cost (communication and execution) at all the processing nodes is 500 units. Time required by the algorithm was 0.06 seconds. Note that the results are conducted in a single processor machine.

Table 2. The final status of the GT using A* for case 1.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	4	10	$m_{21}m_{41}m_{22}$	1	1
p_2	3	8	$m_{12}m_{32}m_{23}$	0	2
p_3	4	9	$m_{13}m_{33}$	2	2
p_4	5	12	$m_{11}m_{31}$	3	5

Case 2

The algorithm is implemented with other two cases. In case 2, a DCS consists of five tasks partitioned with their corresponding modules $T_1(m_{11}, m_{21}, m_{31}, m_{41}, m_{51}), T_2(m_{12}, m_{22}, m_{32}, m_{42}), T_3(m_{13}, m_{23}, m_{33}, m_{43}), T_4(m_{14}, m_{24}, m_{34}, m_{44}, m_{54}, m_{64}, m_{74}), T_5(m_{15}, m_{25}, m_{35}, m_{45}, m_{55}, m_{65}, m_{75}, m_{85})$ and a set of five

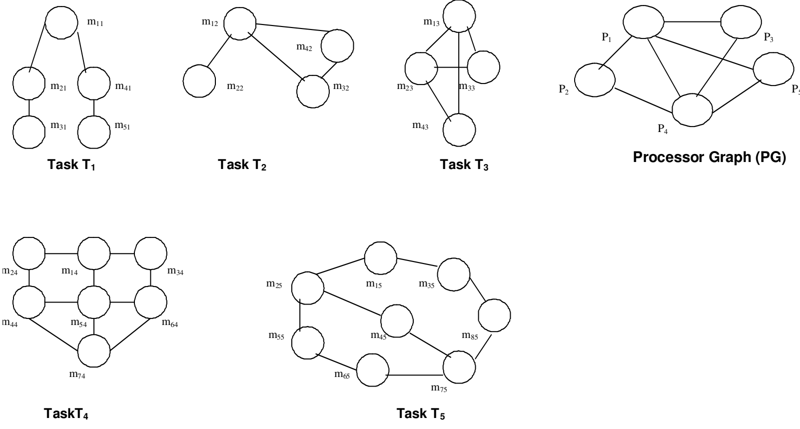

Fig. 2. Example of task graphs T_1, T_2, T_3, T_4 and T_5 with their modules and a DCS as processor graph.

processing nodes $(p_1, p_2, p_3, p_4, p_5)$ interconnected in some fashion (Fig. 2).

The Results using case 2

Total cost at all the processing nodes is 1585 unit. Time required by the algorithm was 0.17 seconds.

Table 3. The final status of the GT using A* for case 2.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	10	50	$m_{21}m_{51}m_{12}m_{42}$ $m_{33}m_{43}m_{14}m_{34}$ $m_{64}m_{74}$	0	19
p_2	9	40	$m_{41}m_{22}m_{13}m_{24}$ $m_{25}m_{85}$	3	21
p_3	7	35	$m_{32}m_{23}m_{44}$ $m_{45}m_{65}$	2	21
p_4	6	30	$m_{11}m_{31}m_{54}m_{15}$	2	14
p_5	4	10	$m_{35}m_{55}m_{75}$	1	2

Case 3

A set of 8(eight) tasks with their corresponding modules $T_1(m_{11}, m_{21}, m_{31}, m_{41})$, $T_2(m_{12}, m_{22}, m_{32}, m_{42}, m_{52}), T_3(m_{13}, m_{23}, m_{33}, m_{43}, m_{53}, m_{63}), T_4(m_{14}, m_{24}, m_{34}, m_{44}), T_5(m_{15}, m_{25}, m_{35}, m_{45}, m_{55}), T_6(m_{16}, m_{26}, m_{36}, m_{46}, m_{56}, m_{66}), T_7(m_{17}, m_{27}, m_{37}, m_{47}), T_8(m_{18}, m_{28}, m_{38}, m_{48}, m_{58})$ and a set of 6(six) processors $(p_1, p_2, p_3, p_4, p_5, p_6)$ have been considered. Due to space limitation, here, we present only the final status of allocation using GT.

The Results using case 3

Total cost at all the processing nodes is 1380 unit. Time required by the algorithm was 0.19 seconds.

Table 4. The final status of the GT using A* for case 3.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	10	50	$m_{21}m_{51}m_{12}m_{42}$ $m_{33}m_{43}m_{14}m_{34}$ $m_{64}m_{74}$	0	19
p_2	9	40	$m_{41}m_{22}m_{13}m_{24}$ $m_{25}m_{85}$	3	21
p_3	7	35	$m_{32}m_{23}m_{44}m_{45}$ m_{65}	2	21
p_4	6	30	$m_{11}m_{31}m_{54}m_{15}$	2	14
p_5	4	10	$m_{35}m_{55}m_{75}$	1	2

Table 5. The final status of the GT using EA* for case 1.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	4	10	$m_{21}m_{41}m_{22}m_{32}m_{23}m_{33}$	-2	1
p_2	3	8	$m_{12}m_{13}$	1	2
p_3	4	9		4	2
p_4	5	12	$m_{11}m_{31}$	3	5

Table 6. The final status of the GT using EA* for case 2.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	10	50	$m_{21}m_{51}m_{12}m_{42}$ $m_{23}m_{43}m_{14}m_{34}$ $m_{25}m_{85}m_{74}$	-1	19
p_2	9	40	$m_{41}m_{22}m_{13}$ $m_{24}m_{55}$	4	21
p_3	7	35	$m_{32}m_{33}m_{44}$ $m_{45}m_{65}$	2	21
p_4	6	30	$m_{11}m_{31}m_{54}$	2	14
p_5	4	10	$m_{15}m_{65}m_{75}$	1	2

5 Comparative Observations

The TA algorithms that consider only modules of one task do not consider the limitation of memory and the number of modules that can be assigned to a particular processor. This is so because these algorithms are not meant for assignment of modules belonging to the multiple disjoint tasks. Such a single task assignment problem is easier to solve because of this reason.

Table 7. The final status of the GT using EA* for case 3.

P_{node}	M_{mod}	M_{cap}	Mod_{assign}	R_{mod}	R_{mem}
p_1	10	70	$m_{11}m_{21}m_{41}m_{52}m_{13}m_{33}m_{63}m_{14}m_{34}m_{44}m_{15}$ $m_{55}m_{16}m_{46}m_{66}m_{17}m_{47}m_{18}m_{44}m_{18}m_{58}$	-10	35
p_2	8	50	$m_{31}m_{23}m_{24}m_{35}$ $m_{26}m_{27}m_{38}$	1	23
p_3	6	40	$m_{53}m_{36}m_{37}m_{38}$	2	20
p_4	7	35	$m_{43}m_{25}m_{45}m_{56}$	3	16
p_5	6	40	m_{32}	2	22
p_6	6	33	$m_{12}m_{22}m_{42}$	3	8

However, we can execute the Single Task Allocation (STA) algorithms [2, 5, 6] multiple time ones for each task using the GT data structure to record the status of allocation and the system as done in our proposed algorithm (sec. 4). Now we compare the status of allocation and the execution time requirements of the method used in EA* and our proposed allocation algorithm. The STA based on A* [2] referred to as EA* in the subsequent discussion has been executed multiple times and the run times have been obtained. So, in the experiment, we have executed the tasks one by one for the cases 1, 2 and 3 without considering the processor connectivity (how the processor are connected i.e. with direct connection/indirect connection etc., because it is not possible to do so in EA*) for the EA* as described in the algorithm of [2]. In the work [6], another modified version of EA* is proposed but still it was developed for single task allocation with their modules by using the same idea of [2]. In [5], an algorithm has been presented to reduce the search space using the idea of [2, 6]. Therefore, here we present the comparative results with the algorithm proposed in [2], as all the other STA algorithms are basically based on this.

If we look at the results shown in the Tables 5, 6 and 7 for allocation of tasks using EA* for the cases 1, 2 and 3, respectively, it is observed that balanced load allocation can not be achieved. In all the cases, presented in the tables, some processing nodes are overloaded (indicated by '-' sign) according to the Vth column of the GT considering their existing architectural capabilities. Thus, it is justified that the EA*, in the form, as reported in [2, 5, 6], can not be used for the allocation of multiple tasks. Here, A* is used to refer our A* based algorithm proposed in section 4. Furthermore, since we did not get good results using EA* in terms of allocation, hence we did not present here the running time and the total cost required by EA*.

5.1 Experimental Results

It is observed from results using A* in section 4 that the time taken by our algorithm is not very effective considering the small number of tasks and their corresponding modules. But the fact is that the experiments were conducted in a single machine to make comparisons with the earlier algorithms.

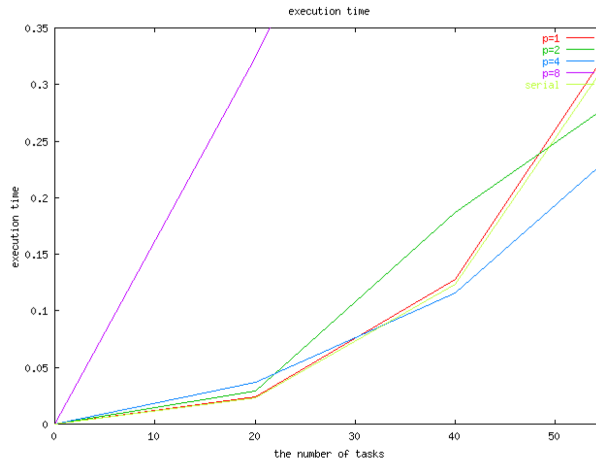

Fig. 3. Execution time using number of tasks = 60(approx.)

Therefore, to investigate the effectiveness and the scalability of our proposed algorithm we further experimented with large number of task graphs with corresponding modules. For simulation purpose we use Sun Fire 12K, 8 processors based Distributed Multiprocessor systems and Message Passing Interface (MPI) as programming environment. In the Fig. 3, 4 and 5, the x-axis represents the no. of tasks and y-axis represents the execution time in seconds. It is observed from the figures that for an amount of large number of tasks, our parallel algorithm performs better (Fig. 5) than the other two (Fig. 3 and 4) with respect to the execution time with an increasing number of processors. In Fig. 3, the 'pink' colored line represents the execution time for 60(approx.) tasks using 8 processing nodes and it is greater than the time required by all other processors. Fig. 4 represents the time required by our algorithm while number of tasks is equal to 100. Here also the 'pink' colored line indicates the number the running time taken by 8 processing nodes and it is still greater than the time required by the other processing nodes. Fig. 5 shows the time required while there is a large number of tasks is assigned i.e. the number of tasks is equal to 400 by the processing nodes. The results show that until the number of tasks is 200, the result is same using 8 processing nodes('pink' colored line) with the time required by other processors but as soon as the number of tasks increases(beyond 200), the timed required by 8 processors also decreases compared to the time required by other processors. Thus, we can say that for a large number of tasks our algorithm can perform well and scalable with the large number of increasing tasks. Note that the no. of tasks with their corresponding modules is generated randomly for these experiments.

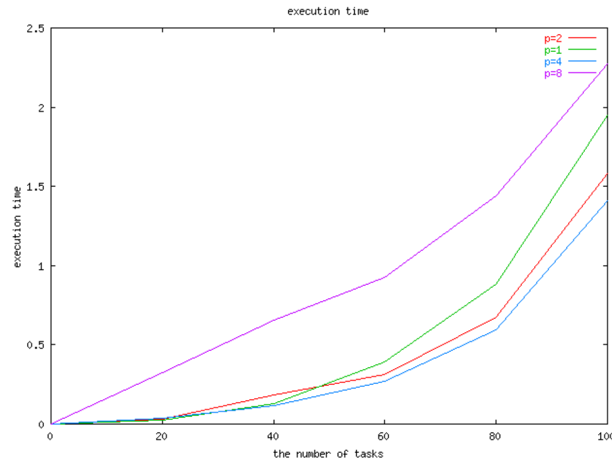

Fig. 4. Execution time using number of tasks = 100.

6 Conclusion and Future Work

Our proposed algorithm has attributed the efficiency in allocating multiple tasks by optimizing a good load balanced among the processing nodes in a heterogeneous DCS. By realizing the dynamic situation of a system we have introduced a data structure Global Table and an algorithm which allocates the multiple tasks with their modules in such a way so that no processor becomes overloaded due to the allocation by considering their status based on their architectural capability.

Furthermore, we have conducted experiments for a large number of tasks with the corresponding modules. Comparing the results, obtained using our algorithm, it is evident that our proposed algorithm can provide effective solution in terms of scalability for the TA problem for a large number of tasks coming onto a DCS. As for future work we will concentrate on implementation of our proposed algorithm for a real time DCS.

References

1. N.J. Nilson, *Problem Solving Methods in Artificial Intelligence*. McGraw Hill International Edition, 1971.
2. C.C. Shen and W.H. Tsai, “A Graph Matching Approach to Optimal Task Assignment in Distributed Computing System Using A Minimax Criterion”, *IEEE Transactions on Computers*, vol. C-34, no. 1, pp. 197-203, 1985.
3. A.K. Tripathi, D.P. Vidyarthi and A.N.Mantri, “A Genetic Task Allocation Algorithm for Distributed Computing System Incorporating Problem Specific Knowledge”, *International Journal of High Speed Computing*, vol. 8, no. 4, pp. 363-370, 1996.

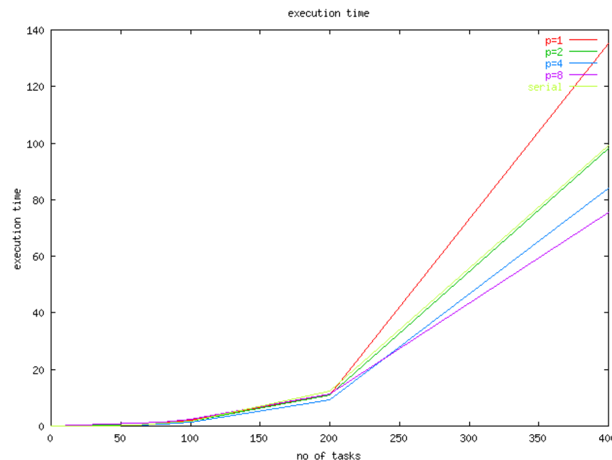

Fig. 5. Execution time using number of tasks = 400.

4. A.K. Tripathi, B.K. Sarker, N. Kumar and D.P. Vidyarthi, “A GA Based Multiple Task Allocation Considering Load”, *International Journal of High Speed Computing*, vol. 11, no. 4, pp. 203-214, 2000.
5. M. Kafil and I. Ahmed , “Optimal Task Assignment in Heterogeneous Distributed Computing System”, *IEEE Concurrency*, vol. 6, no. 3, pp. 42-51, 1998.
6. Ramakrishnan, H.Chao, and L.A.Dunning, “A Close Look at Task Assignment in Distributed Systems”, *Proceedings of IEEE Infocom-91*, pp. 806-812, 1991.
7. D.P.Vidyarthi, A.K.Tripathi and B.K.Sarker, “Allocation Aspects in Distributed Computing System”, *IETE Technical Review*, vol. 18, no. 6, pp. 279-285, 2001.
8. P.Y.R.Richard Ma, E.Y.S.Lee and J. Tsuchiya, “A Task Allocation Model for Distributed Computing Systems”, *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 41-47, 1982.
9. S.H.Bokhari, “On the Mapping Problem”, *IEEE Transactions on Computers*, vol. C-30, pp. 207-214, March, 1981.
10. Pradeep K. Sinha, Distributed Operating System, IEEE Press, Prentice Hall of India Ltd., 1998.
11. A.S.Tanenbaum, Distributed Operating Systems, Prentice-Hall, Englewood Cliffs, 1995.
12. A.K.Tripathi, B.K.Sarker, N.Kumar and D.P.Vidyarthi, “Multiple Task Allocation with Load Considerations”, *International Journal of Information and Computing Science (IJICS)*, vol.3, no.1, pp. 36-44, 2000.
13. D.P.Vidyarthi, A.K.Tripathi and B.K.Sarker, “Multiple Task Management in Distributed Computing System”, *Journal of the CSI*, vol. 31, no. 1, pp. 19-25, 2001.