Efficient Parallel Iterated Greedy Algorithm for Solving Task Assignment Problem in Heterogeneous Systems

1Harish K, 2R Sibi, 3S Varun Kumar
Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli
Email: 1 karthikeyanharish@yahoo.co.in, 2rsibi93@gmail.com, 3kvarunsvk@gmail.com

Abstract— Efficient allocation or assignment of tasks has been a constant problem for research in the domain of Combinatorial Optimization. With Task Assignment being an NP-Hard Problem for more than 3 processors, considerable effort has gone into developing heuristic algorithms that provide an efficient and approximate solutions that match the expected values. Currently trending algorithms utilize Graph Matching and Graph Partitioning techniques to achieve the assignment of Tasks. We have presented an algorithm that is different from the Graph Based methodologies and instead uses Greedy Algorithm to achieve the effective task assignment. This is a Parallel Iterated Greedy Algorithm which minimizes the cost of computation and at the same time provides faster convergence to a value that is closer to the theoretically computed value. The algorithm was implemented using Message Passing Interface (MPI)

Index Terms—Load Balancing, Task Assignment, Iterated Greedy Algorithm, Message Passing Interface

I. INTRODUCTION

The notion of connected processors confined within a single system or a small geographical spread is outdated. With the advancement in technology, the idea of parallel computing has caught the fancy of researchers and users alike. Parallel Computing is a boon to mitigate the high levels of computation that real time systems often bequeath to the processors. Such applications that involve high quantity of computations are divided into manageable portions and run independently on the parallel processors which are often different from each other. This is the concept of Heterogeneous systems where processors differ from each other on several fronts including the computational power and inter-processor communication [8].

The caveat when it comes to Parallel Computing is the assignment of tasks to the processors. Though the number of processors are in abundance, it is the allocation of tasks to these processors that actually matters in our objective to minimize turnaround time and maximize processor utilization. This assignment can be done in a static fashion wherein the allocation of tasks is done before the execution of the said tasks. This increases the pre-execution time. A more robust approach to the same would be to go in for dynamic assignment of tasks. The distributed computing systems provides another dimension to the problem of task assignment-reliability. Task Assignment has to be done so as to guarantee reliability in systems [5].

A. Greedy Algorithm

Greedy Algorithm is an algorithm that follows the idea of making locally optimal choice in the hope that they yield globally optimal value. The algorithm need not always provide a global optimal value but these are used to provide an approximately optimal value in a reasonable amount of time. The algorithm when repeatedly iterated over, refines its value and yields a value much closer to the optimal value when compared to the result at the end of the first iteration [4]. Such an iterated greedy algorithm can be terminated after convergence, i.e. repetition of same or similar values for an extended duration of time. Other considerations apart from the duration of convergence can be used as validating criteria or objectives for proposed algorithms [3] [6]. Considerable research [7] has been done in providing reliability to the task assignment through iterated greedy algorithm.

B. Motivation for the work done

The algorithm proposed in [1] is a considerable foray into incorporating elements of parallel processing to an iterated greedy algorithm. We have adapted the suggestion for further research and produced an algorithm that provides parallelism in the destruction phase too. The addition of parallelism here does increase the communication overhead, but the benefits of a curtailed iteration procedure outweighs the overhead for larger number of tasks. This method has provided better results when compared to its predecessors.

The paper is organized as follows. The Mathematical Formulation of the problem statement is done in Section 2. Sections 3, 4 serve as a literature survey elaborating on the algorithms proposed by Kang et al and Mohan et al in [2], [1] respectively. Section 5 contains our proposed algorithm. Section 6 is on our Experimental Results and Observation while Section 7 deals with the Conclusion.

ISSN (Online): 2347 - 2812, Volume-2, Issue -5, 2014
Section 8 provides an insight into the scope for further research.

II. MATHEMATICAL FORMULATION

The task allocation problem is usually represented by a graph called the Task Interaction Graph (TIG): G (V,E). The tasks are represented as vertices (V). An edge (E) between two tasks denotes the communication required between them. We know that the system is heterogeneous, the execution times of different tasks differ with the processor. A matrix \{ec_{ij}\} stores the execution time of task i on processor j. Each edge has a weight \(w_{ij}\) associated with it. This represents the amount of data which will be moved between the tasks i and j. The communication cost between processors could differ as in any heterogeneous system; this is represented by dlk where k and l are any two processors. “dlk” depends on the distance between processors and the cost associated in transferring unit data between them. We consider this distance metric to be symmetric i.e. the communication cost from process i to j is equal to the communication cost from process j to i. Also, we assume that two tasks assigned to the same processor carries no communication overhead.

The objective of the task assignment problem is to try and minimize the total time involved in the process. The communication cost between any two process is found out by multiplying the distance between the two processors and the data to be transferred between them – \(wij \times dlk\) (assuming i is assigned to k and j is assigned to l)

Let \(\phi\) be a mapping which assigns tasks to processors, e.g. \(\phi[j] = 1\) (this implies that the \(j^{th}\) task was assigned to processor 1)

Let \(\Phi\) be the set of all possible mappings \(\phi\)

We consider two of the costs that are incurred

1. Processor execution cost

\[
PEC = \sum_{i=1}^{N} ec_{ij}
\]  

(1)

2. Inter-processor communication cost.

\[
IPCC = \sum_{i=1}^{N} \sum_{j=1}^{N} d_{\phi(i)\phi(j)} ec_{ij}
\]  

(2)

Therefore, the objective now is to minimize the total cost (sum of the two costs) = PEC+IPCC subject to the constraints:
- Resource Constraints:

\[
B_k = \sum_{j=1}^{N} b_j \land \phi(j) = k
\]  

(3)

- Processor is used if a task is allocated to it.
- All tasks are allocated just once.

III. ITERATED GREEDY ALGORITHM

PROPOSED BY Q. KONG ET AL IN [2]

The algorithm starts by finding an initial greedy solution to the problem. This is generated randomly which is then followed by a number of iterations that are performed to make the initial solution more optimal. As in algorithm [1], the following algorithm too has four steps – Construction, Destruction, Optimize and Acceptance.

In the destruction phase, \(d\) tasks are eliminated from the present solution and subsequently reassigned to the available processors in the construction phase. The acceptance criterion specifies whether the new arrangement can be accepted or not. This process is repeated until a stopping condition is met.

An outline of the procedure as given in [2],

Procedure IteratedGreedy ()

\{
    \(X_o=\text{GenerateInitialSolution};\)
    \(X = \text{LocalSearch}(X_o);\)
    \text{Repeat}
        \(X_p = \text{Destruction}(X);\)
        \(X = \text{Construction}(X_p);\)
        \(X = \text{LocalSearch}(X);\)
        \(X = \text{AcceptanceCriterion}(X, X^*);\)
    \text{Until termination condition met}
\}

A. Initial Solution

Random assignment of tasks to the processors conforming to the constraints defined earlier gives us the initial solution. This initial solution defines the number of iterations which will be required to find the optimal solution. The closer this solution is to the final optimal solution, the lower is the number of iterations required. Thus the random assignment is pivotal to the effective parallelization of the iterated greedy algorithm.

B. Destruction Phase

In the Destruction phase, a portion of the current selection (the incumbent solution) is deselected leaving a new partial solution. As stated in [2], the number of tasks to be removed is randomly chosen from 0 to \(N\). The complexity of the phase is \(O(d^N)\). As can be seen, the number \(d\) determines the totally running time of the algorithm.

C. Construction Phase

After the destruction phase, we are left with a partial assignment solution. The removed \(d\) tasks are re-assigned to the available processors such that it minimizes the cost according to the greedy constructive heuristic technique adopted from [8]. The complexity of the construction phase is \(O(d^N*K)\)
D. Acceptance Criteria

Once a candidate solution is constructed, the acceptance criterion specifies whether it can be accepted as the new incumbent or to discard it and retain the old value. But, as mentioned in [2], adopting a worse solution at this stage might help provide diversification and exploring more regions at the next iteration.

E. Termination Condition

Different criteria can be devised like the maximum number of iterations, time-limit and so on. We will use computation time-limit as our stopping criteria as the termination condition as we will be comparing our algorithm with the other heuristic algorithm mentioned in [1], [2].

IV. IMPROVED ITERATED GREEDY ALGORITHM PROPOSED BY MOHAN ET AL IN [1]

A. Algorithm Discussion

The algorithm proposed in [2], though efficient utilizes a weak acceptance criteria of “temperature”, [1] proposes instead a heuristic test of System Cost < 1.2*current as the acceptance criteria for the solution. Also a better destruction value was chosen in [1], rather than the random value generated in [2] - Maintain $d$ as 0.3N since it has been observed to yield better performance as seen from [2].

Parallel processing is incorporated into the algorithm proposed in [2] and is used in generating the initial random solution. Some ‘m’ processes are called and their own initial solution is generated by completing the three phases – destruction, construction and acceptance for some number of iterations. During these iterations, the processes constantly communicate with each other in order to find the solution which is optimum locally. After a specified number of iterations, a continue signal $s$, is sent to these processes which continue with their iterations. All other processes which have not received the continue signal are stopped after completion of the iterations. The process returns the final solution back to the main process.

B. Pseudocode for Improved Iterated Greedy Algorithm

```
ParallelIteratedGreedy()
{
    min_sol, sol_id;
    If (root process) //root processor/
    {
        For (i=1 to no_of_processors) //communication phase/
        Receive (par_sol, procs_id)
    }
    If par_sol<min_sol
    {
        min_sol=par_sol
        sol_id=procs_id
    }
    Send (continue, sol_id); //send back continue signal //
    Else //operations of other processes/
    {
        Xo= GenerateInitialSolution;
        X = LocalSearch (Xo);
        Repeat
        Xp = Destruction(X);
        XC = Construction (Xp);
        X = AcceptanceCriterion (X, X*);
        Until m iterations
    }
    Send (X, rootprocs_id); // Sending locally minimum value to root processes for acceptance //
    Receive (signal, rootprocs_id);
    If (signal=continue)
    {
        Repeat
        Xp = Destruction(X);
        XC = Construction (Xp);
        X = Optimize (XC);
        X = AcceptanceCriterion (X, X*);
        Until termination condition met
    }
    Send (x, rootprocs_id); // sending optimal solution back to main process //
}
```

V. OUR CONTRIBUTION

A. Proposed Changes

The Parallel Iterated Greedy Algorithm proposed in [1] uses a random methodology to remove tasks from the task set X in the destruction phase to generate Xp. This random approach increases the time of convergence to the final optimal value. The reason being that a task removed at random from the vector X could be an optimal assignment. This task is chosen with probability $1/N$ from X. For ‘d’ tasks this probability value becomes $d/N$. So when such an optimally assigned task is removed from the solution set for construction, it leads to extra iterations, thereby prolonging the time of convergence.

To overcome this challenge of random selection of tasks in the destruction phase, we propose parallelizing the destruction phase. The parallel processing in the destruction phase is in addition to the one incorporated in the Initial Solution in [1]. The Destruction module is separately defined in the next sub-section.

In this destruction module we create more processes that do the task in parallel. For each of the processes we pass X as input. These processes in turn generate a unique X’ which is a subset of the vector X with ‘d’ components removed. The value of d is the pre-determined destruction value, $d=0.3N$.

The new processes send their X’ value back to the calling process with id=process_pid. The calling process compares the cost of each of the X’ and proceeds to return
the smallest of them.

The check process_pid<=level_of_parallelisation in the Destruction module is to distinguish the process that has been created in the destruction module and the one created in the main module.

B. Pseudocode for the Effective Iterated Greedy Algorithm

```c
paralliteratedgreedy()
{
/*min_sol stores the minimum cost and sol_id is the pid of the process that contains the minimum solution*/
if(root_process)                                //root processor/
{
    min_sol_cost,min_sol_pid
/*Note that X,Xo are the vectors containing the processor assignment*/
    Xo= GenerateInitialSolution;
    X = LocalSearch(Xo);
    min_sol_cost=cost(X) //initial assignment of the min_cost
    for (i=1 to level_of_parallelisation)
    Send (X, i) //send the solution after the local search to each processor
    For (i=1 to level_of_parellisation)
    {
        Receive (X, i+10) //recieve the generated solution from each process
        If (cost(X) < min_dest_cost)
        {
            min_dest_cost = cost(X)
            min_dest_pid = i
            Xp = X
        }
    }
    Send (continue,min_sol_pid)
    Receive (final_optimised_sol,min_sol_pid)
}
else if(not(root_process))
{
    Receive (X,root_process_pid)
    Repeat
    Xp = Destruction(X);
    Xc = Construction(Xp);
    X* = LocalSearch(Xc);
    X = AcceptanceCriterion (X, X*);
    Until m iterations
    Send (X,root_process_pid)
    Receive (signal,root_process_pid)
    If (signal=continue)
    {
        Repeat
        Xp = Destruction(X);
        Xc = Construction(Xp);
        X* = LocalSearch(Xc);
        X = AcceptanceCriterion (X, X*);
        Until termination condition is met
        Send (X,root_process_pid)
    }
    Destruction (X)
    
    If (process_pid<=level_of_parallelisation)_
    {
        min_dest_cost, min_dest_pid, Xp
        min_dest_cost = MAX_INT
        For (i=1 to level_of_parallelisation)
        Send (X, i+10) //send the solution after the local search to each processor. Since 1 to level_of_parallelisation is taken by the main function.
        We generate id by an offset of 10 units
        For (i=1 to level_of_parellisation)
        {
            Receive (X', i+10) //recieve the generated solution from each process
            If (cost(X') < min_dest_cost)
            {
                min_dest_cost = cost(X')
                min_dest_pid = i
                Xp = X'
            }
        }
    }else
    {
        Receive(X, process_pid)
        Generate X' is the vector which is a subset of X with 'd' components removed
        Send (X',process_pid)
    }
    Return Xp
}
```

![Fig 1: A plot comparing the relationship between the Time of Convergence and Number of Tasks (N) for different processor count (n)](image)

VI. EXPERIMENTAL RESULTS AND OBSERVATIONS

The given pseudo code was translated into a C working program and implemented on Dev C in an Intel i3 Core Processor, the same system parameters used in [1]. This was to ensure correlation between the programs and the experimental outcomes. The implementation used Message Pass Interface (MPI) to implement “send”, “receive” functionalities in the code.

The time of convergence of values for different numbers of tasks (N) with different processor count (n) was noted and the graph was plotted. When there is a single
processor, i.e. \( n=1 \), we can surmise without any loss of data, the values were that of the algorithm proposed in [2]. When \( n=1 \), it implies the absence of parallelization in the algorithm similar to the one proposed in [2]. This is represented in Fig 1. Another set of data points were collected that compared the Time of Convergence for the algorithm proposed in [1] and our algorithm. This is graphically represented by Fig 2.

VII. CONCLUSIONS

The graphical representation of our results highlighted several key benefits of our proposed algorithm. The proposed parallelization in the destruction phase helps reduce the time of convergence. As the processor count increases, the time of convergence decreases. This can be seen as an effect of an increase in computation power.

Additionally, the communication overhead that resulted in slower convergence for larger number of tasks as seen in [1] does not appear here. It is reasonable to conclude that with the parallelization in destruction, the number of iterations is sharply reduced which offsets the communication overhead.

Comparison with the algorithm in [1] yields the conclusion that while for smaller number of tasks [1] perform better, primarily due to a larger number of processes being created, for larger number of tasks Efficient Iterated Greedy Algorithm works better for similar reason of the decrease in number of iterations leading to faster convergence.

![Fig 2: A plot comparing the time of convergence for n=5 between the two algorithms](image)

VIII. FURTHER WORK

The solution still offers considerable scope for research and these are the lines along which further work will have to proceed.

- The initial solution determines the number of iterations and thereby the time of convergence. A more nuanced approach to deriving an initial solution will yield better solution.
- Enhancing the reliability of the algorithm to enable utility in critical systems.
- Adapting environment or use-specific acceptability criterion and terminating condition to encourage the use of the algorithm in more diverse applications.

REFERENCES


---

ISSN (Online): 2347 - 2812, Volume-2, Issue -5, 2014