Adaptive Decentralized Load Sharing Algorithms with Multiple Job Transfers In Distributed Computing Environments

Sunita Bansal¹, Divya Gupta¹, and Chittaranjan Hota²

¹ Birla Institute of Technology & Science
Computer Science & Information Systems Group
Research & Consultancy Division
Pilani, Rajasthan, 333031, INDIA
sunita_bansal@bits-pilani.ac.in

² Birla Institute of Technology & Science, Pilani
Computer Science & Information Systems Group
Information Processing Centre Unit
Hyderabad Campus, Hyderabad, AP, INDIA
hota@bits-hyderabad.ac.in

Abstract - We present a new enhanced version for Adaptive Stable Sender-Symmetric Algorithm which efficiently allocates jobs in the network and possesses multiple job transfer capabilities. Till now, all the algorithms schedule only one job at a time to other nodes in a distributed system. We here present a mechanism that is able to schedule multiple jobs coming to a node to multiple nodes in the network at any instance of the time, with minimum time. Parameters used to find the appropriate node to transfer the job do not solely depend on queue length but also CPU utilization, memory usage and available bandwidth of node. The proposed algorithm works well with different types of topologies. The extensive experimental results demonstrate that our new version of the algorithm outperforms the existing algorithms. The main goal of the algorithm is to minimize flow time and execution time for jobs. The result depends on the number of nodes in the network and the variability of the load on the nodes.

Index Terms - Distributed Scheduling, multiple Job Transfer, Resource Management

I. INTRODUCTION

As distributed systems offer a great computing capacity, there has been tremendous growth in it is usage. It is now required to make full advantage of it, by having a good resource allocation scheme. Distributed Scheduling is used in distributed systems, which redistributes the load of the system among different computers, so that overall performance of the system is maximized. However, in redistributing the load overall communication overhead is considerably increased in getting the load information at all the computers in the network.

Now days, load sharing has become a very crucial aspect in the distributed systems. A distributed system has a collection of autonomous computers, connected to each other in a network with some topology, with each node having it is own processor speed, and memory utilization. The bandwidth requirements between the nodes can also vary. To minimize response time and execution time of jobs, load sharing schemes played a vital role as it had a great affect on the performance of the whole system.

There are different types of load balancing algorithms, as sender-initiated, receiver-initiated, and symmetrically initiated [1] and adaptive algorithms [2]. We will discuss on Adaptive Algorithms. The proposed work is basically on the Stable Sender-Initiated Algorithm which is a kind of adaptive algorithm.

The major contributions of the paper are:
1. This algorithm works for decentralized networks that can follow any topology. Each node in the network can communicate with neighboring nodes. Node takes best decision, based on the available information.
2. Load sharing decision depends on CPU utilization and memory usage at receive node, and available bandwidth between sender and receiver nodes.
3. Whenever any node changes its state from sender to receiver or vice-versa, the new state information is broadcasted to all the other nodes that are senders. This reduces the overall communication overhead in exchanging the state information of the nodes.
4. Sender is now able to schedule multiple jobs coming to it in some span of time, to different or same receivers depending upon the requirements and conditions at that particular time.
5. The results from the simulations achieve the desired goal of minimizing the response time and execution time for each job in the distributed system.

The rest of the paper has been structured as follows. In Section II, we describe the related work. Section III presents the proposed algorithm and its features. In Section IV, we show the performance of the algorithm obtained from the results of the simulation runs. Finally, we conclude the paper.
II. RELATED WORK

In this section, more emphasis is on the distributed load sharing algorithms which are already there in existence. The drawbacks of each of the algorithms have motivated us to design a new enhanced version of one of the best algorithms known till now. Much of the work has already been done in this field [3, 4, 5]. All the load distribution algorithms, mainly work on their basis of transfer policy and location policy as major components. In Sender-Initiated Algorithms [3, 4] the load distribution activity is started by an overloaded node (sender) which attempts to send a task to an under loaded node (receiver). The algorithm becomes highly unstable at high system loads, as it doesn’t find any lightly loaded node in the network. In Receiver-Initiated Algorithms [1, 3, 4, 6] the under loaded receivers take the initiative to find the overloaded nodes (senders). Major drawback of this is that most transfers are preemptive and therefore expensive. Sender-Initiated algorithm makes greater use of the non-preemptive task transfers. In Symmetrically-Initiated Algorithms [1, 3, 4, 6], both senders and receivers search for receivers and senders, respectively, for task transfers. It uses the features of the above discussed algorithms and shows better results in case of high system loads and low system loads. In this, a little overhead is imposed without much exchange of messages. All these algorithms use polling to find the node to which or from which, the task can be transferred.

The Adaptive Algorithms that were proposed eliminate the system instability problem due to load sharing. The Stable Symmetrically Initiated Algorithm [5] uses the state information at each node during polling rather than discarding them and identifies the nodes as sender, receiver or OK in the system.

Another kind of Adaptive Algorithm is Stable Sender-Initiated Algorithm [6]. This Algorithm has two features, first, it doesn’t cause instability and second, load sharing is due to non-preemptive transfers only. In this algorithm each node maintains state vectors to keep track of its own state at other nodes and other nodes state. The Sender-initiated component and Receiver-initiated component use receiver and sender state vector to find the appropriate sender or receiver and that the state vectors are updated whenever any of the node changes its state. In this algorithm message overhead can be high if nodes change their state very often.

Javier and Xhafa [8] proposed genetic algorithm to minimize the make span time and flow time. In their work, they assume previous knowledge of the system, with regard to the processing requirements of the tasks as well as the processing capacity of the machines in the grid. On the basis of the knowledge of these parameters, an ETC (Effective Time to Compute) has been formulated. The ETC matrix entries give us the time to compute task n on processor m. Two different kinds of encodings have been used in this paper for the scheduling. In direct representation, an array schedule is used with size equal to the number of jobs. Each entry indicates the machine to schedule on. Other one which is permutation based, uses standard technique of a string of size n+m-1 and delimiters to separate the tasks on the various machines of the system. Andrew and Thomas [9] proposed a framework for task scheduling in heterogeneous computing. They introduced a new way to predict the execution time of the jobs on different machines. Jobs are considered to be heterogeneous and having different processing requirements. They use a smoothing formula for the purpose of approximating various parameters used during the process of scheduling and calculating the time during the processing on the grid. They also use fitness function incorporating the use of the communication costs. The goal of our study is to enhance the capabilities of Stable Sender-Initiated Algorithm and to include multiple job transfers. This results in overall high system throughput, low response time and minimum execution time for the jobs. The performance of the network on the whole is improved.

FAST (Fast Agent’s System Timer) is a software tool, that gets an accurate forecast of communication, computation times and memory use in a heterogeneous environment. There are two modules namely static and dynamic data acquisition. Static data module is use to predicate time and space of jobs, for that it uses highly sequential and optimized routines for the processor, as the matrix-matrix product routine dgemm() from BLAS library or Pdgemm for parallel matrix-matrix product. Dynamic data module predicates network bandwidth, latency, workload and memory usages. Fast developed four functions in the API. First, the fast_comm_time (data_desc, source, dest) function gives the communication time to transfer the data described by data_desc from the host source to the host dest, taking the current network characteristics into account. Then, the fast_comp_time (host,problem,data_desc) function gives an estimation of the time needed to execute the routine problem on the given host for the parameters data_desc, taking into account both the theoretical performance of this host for this problem, and the actual load of the host. The fast_comp_size (host, problem, data_desc) function gives, for the same arguments, the memory space needed by the routine. Finally, the fast_get_time (host,problem,data_desc,localizations) function aggregates the results of others functions and forecast the time needed to execute the given problem on host for the data described by data_desc, taking into account the prior localizations of data and the time to get them on host. This function returns -1, if the computation is impossible, for example if there is not enough memory available.

III. THE PROPOSED ENHANCED STABLE SENDER-INITIATED SYMMETRIC ALGORITHM

A. System Model

Here, we assume the distributed system as a collection of N nodes; nodes are connected to each other through a
communication channel. Nodes which are directly connected to a node constitute its buddy set. We also assume that each node has knowledge about its buddy set and communication latency. Load sharing is carried out within buddy sets. Two nodes may have few processors common to each set. The algorithm has been designed for decentralized distributed networks, forming any topology except central node dependencies. As nowadays, decentralized distributed systems are preferred because of their major drawbacks [4, 6]. Each node has its own computing speed, memory and network link. Whenever a job is transferred from a heavily loaded node to a lightly loaded node, the algorithm computes transfer and execution time. With the objective that transfer should not lead state change of the executing node and no further transfer takes place from the executing node. If no such node can be found, then job is executed by the receiving node. One such network example is shown in the Figure 1.

![Diagram](image1.png)

**Figure 1: Example of decentralized distributed network**

**B. Job Model**

For any node \( n_i \in N \), the incoming jobs, \( J_h \in J \), be the list of \( k \) incoming jobs at \( n_i \) node. Jobs are considered to be mutually exclusive and executable at any site except locally dependent jobs. Each job \( j_i \) arrived at a node \( n_i \) has mainly one parameter: job size, that is, the size which is calculated by the compiler. As soon as any node becomes a sender node, the sender node finds the appropriate and exactly one receiver for its processing. Once the job gets executed at the receiver node, the receiver node sends back the result to the originating node of the task.

**C. Policies Followed By the Algorithm**

All the distributed load sharing algorithms have four components. Transfer policy determines the state of the node, whether it should be a sender or a receiver. Selection policy, determines which task should be transferred to the receiving node for execution. Location policy determines the node to which the job should be transferred and information policy keeps the record of the states of each node and is responsible for collecting the updated knowledge about the nodes in the network.

The policies on which the proposed algorithm works has been discussed below:

**Transfer Policy:** The predefined threshold transfer policy considers a node as sender or receiver on the basis of available CPU utilization and non-paged memory. Predefined threshold can be set according to the nature of the job like whether it is CPU bound or I/O bound or of hybrid type. We consider a parameter \( \alpha \), where \( 0 \leq \alpha \leq 1 \), which will be based on weight age of memory and the CPU. Thus, in the above relationship, by tuning the parameter \( \alpha \) \( (0 \leq \alpha \leq 1) \), one can vary the estimate. A value of 0.5 for \( \alpha \) would mean that an equal weight age to memory and CPU. Based on \( \alpha \), we now calculate the state determining parameter \( \Psi \) where,

\[
\Psi = \alpha \ast (\text{CPU utilization}) + (1-\alpha) \ast (\text{non-paged memory})
\]

\( \Psi \) is compared with the threshold value to decide upon the state of the node as sender or receiver.

\( \Psi \geq \) node is a sender

\( \Psi < \) node is receiver

**Selection Policy:** All the newly coming tasks to the sender can be selected to be transferred to another node.

**Location Policy:** Location Policy finds the suitable node to share the load. Location policy makes sure that, overhead incurred in the transfer of the job and execution time to other node should be minimal. The sender node has the information about all the neighboring receiver nodes. It knows about the execution time and flow time of the receiver nodes. Execution time basically depicts the CPU utilization and memory usage. Bandwidth shows the flow time/transfer time of the job from sender node to receiver node. It considers the time taken by the job to be transferred from sender to receiver (BandwidthAvailable/JobSize) and the time taken by the result to flow back to sender from receiver (BandwidthAvailable/ResultSize).

**FlowTime** = \[
\frac{\text{BandwidthAvailable}}{\text{JobSize}} + \frac{\text{BandwidthAvailable}}{\text{ResultSize}}
\]

Depending on the above parameters, the sender node finds the best suitable receiver for the job that needs to be transferred. The sender does not poll a receiver every time it gets some job to be transferred. Rather it finds all its probable directly connected neighboring receivers in the network and then calculates the following:

\[
(j) = (\text{FlowTime} (j) + \text{ExecutionTime} (j))
\]

\[
(\text{Min}) = \text{Min} (j, (j+1).................. (N))
\]

\[\text{N} = \text{No Of Neighboring Receivers}\]

\( (\text{Min}) \) (total job execution time), gives the minimum value required to transfer and execute a job. The node with least value is transferred with the job. Whenever any receiver changes its state, it broadcasts the updated information to all its buddy set.

**Information Policy:** It follows state change driven policy for keeping the updated information with all the nodes. Whenever a node changes its state form sender to receiver or vice-versa, the message about this is broadcasted to sender nodes from its buddy set. So this saves the network from being flooded with extra message passing. This helps in preventing the extra overhead due to message passing as state change has been restricted to sender-receiver state change only. For all other changes happening, the messages are not communicated every time.

Till now, at any particular instance of time, only one job is transferred from sender to receiver. With the enhanced algorithm, more than one job can be transferred to receiver nodes at any point of time. As the sender has all the information about its neighboring receivers. So whenever,
During some span of time, the sender node is flooded with jobs, it can send all those jobs to appropriate receiver nodes for execution in less time.

D. The Algorithm

Most algorithms till now have considered queue length as the basic parameter for calculating the suitable receiver for sharing the load. However, it is not a reliable load indicator in a heterogeneous environment. It ignores the variations in computing power. Thus on the basis of this fact, we consider the total job execution time, for all the probable receiving nodes. The node which has minimum gets the job for execution.

Initially, each node multicasts message (Status CPU utilization, memory usages) to its buddy set. So there will be no message overhead to collect the right information of nodes. Each node will keep this information into respective list.

Each node periodically checks its job queue and threshold. The period can be set on the basis of network size. If node threshold is greater than the predefined threshold then it checks its status. If the states change then it will multicast message to its all buddy set. It counts number of jobs which makes a node as a sender. For each job it computes total execution time. The node which has minimum execution time, transfer the job to it. Update the receiver and sender nodes. Each node will periodically check its status. Figure 2 and 3 gives the detailed adaptive task with multiple job scheduling algorithms.

![Figure 2: Algorithm for nodes in Network](image)

```plaintext
1. For each ni ∈ Node
2. Time = 0
3. //Multicast message to buddy node
4. SendMessage(Status, CpuUtilization, Memory)
5. Maintains Sender, Receiver and Status list
6. //β is period length
7. While(time >= β)
8. {
9. // Compute Threshold
10. Ψ = α *(CPU utilization) + (1- α) *(non-paged memory)
11. // Compare Threshold with Predefined threshold
12. If Ψ >
13. NStatus = Sender
14. Else
15. Nstatus = Receiver
16. Call MulticastMessage(StatusList,NStatus)
17. End if
18. Copy all that job into SendList
19. Sort jobs according size in descending order
20. Count = Number of jobs SendList
21. // Create multiple thread = count
22. For I = 0 to Count
23. K = 0
24. For each Receiver at ni
26. Get Execution time using Fast Tool
27. Compute TotalTime[I][K] = FlowTime + Executing time
28. K++
29. End for receivers
30. Sort total time in Ascending order
31. Allocate job to first receiver from TotalTime
32. Remove job from SendList
33. Count++
34. End for Count
35. // Threshold is less than to predefined threshold
36. Else
37. Nstatus = Receiver
38. If (Nstatus != OStatus)
39. Call Multicast Message(Status,NStatus)
40. Execute jobs locally
41. End if
42. End While
```

Figure 3: Subroutine for message transfer

E. Discussion

At initial stage, when node joins the network, it checks its status, based on that it will transfer message to its buddy set. At higher load, there is very less communication between nodes because each sender has the list of receiver which is likely to empty, so less number of jobs will be transferred. In lower load all the receiver will wait to get the job from other side because sender list is likely to empty. In the case of moderate load each node has receiver list, based on requirement it will choose best receiver. Only message communication will take place where there is state change and job transfer.

IV. SIMULATIONS

In the simulation, our algorithm EASSSA (Enhanced Adaptive stable symmetrically Sender Initiated Algorithm) is compared with the algorithm ASSA (Adaptive stable symmetrically Sender Initiated Algorithm). We have developed fully java based thread simulation. We use FAST (Fast Agent’s System Timer) software to get an accurate forecast of communication and computation times and memory usage in a distributed heterogeneous environment. We also use DAJ library (Toolkit for Distributed algorithm) to create network and multicast or broadcast message between nodes. We have tested the algorithm with different number of nodes and jobs and their size to get the best results.

A. Effect of system size

In Figure 4, we show the average response time during low load and high load. The graph is for varying no. of nodes.

![Graph showing average response time](image)
To verify the effect of nodes, number of tasks has been changed. We have taken number of tasks equal to 20 for demonstrating the response time in case of low load for varying number of nodes. In high load number of tasks equals to 100 for varying number of nodes. Here result is better than the old algorithm. Here, overlapping is due to a certain reason. Number of senders and receivers are unknown in network and depend on the CPU utilization and memory usage at that point of time.

In Figure 5, both the older and enhanced version of the ASSSA Algorithm is compared, while keeping the no. of nodes and no. of incoming jobs constant for few execution attempts of the simulation the algorithm has been executed a number of times. And each time, the response time has been analyzed. For this, we have taken no. of jobs as 20 and no. of nodes as 10. From the graphs we can say that the response has improved to some extent. The same reason holds true for the variation in the response time in Figure 5. The algorithm takes lesser time, if the no. of senders is approximately equal to no. of receivers. But takes a little longer time, when the senders and receivers are unequal by quite a large number.

B. Effect of system utilization

In this paper, we have developed enhanced version of adaptive stable sender-symmetric algorithm and successfully showed a reduction in response time. This drop down in the response time is due to the multiple job transfers from sender node to receiver nodes. The algorithm considers CPU utilization, memory usage, bandwidth available and execution time to find the appropriate receiver and not just the queue length. Through simulation we demonstrated the decrease in response time depending on various parameters. The estimated job execution time has been calculated using FAST tool which has further helped in reducing the total response time for execution of tasks in the network. Message overhead is still a problem, which is a major challenge for future work. Although, our algorithm, takes care of handling message overhead till some extent, but still a lot more can be done in this area.

REFERENCES