

PROJECT REPORT ON
**A HYBRID APPROACH ON HEURISTIC TASK SCHEDULING
ALGORITHM IN CLOUD COMPUTING**

Submitted in partial fulfillment of the requirements for the award of the degree of

**BACHELOR OF TECHNOLOGY IN
COMPUTER SCIENCE AND ENGINEERING
OF SASTRA UNIVERSITY**

By

K. SAIRAM [113003142]



**SHANMUGHA ARTS, SCIENCE, TECHNOLOGY & RESEARCH ACADEMY
(A University Under Section 3 of UGC Act, 1956)
TIRUMALAISAMUDRAM, THANJAVUR-613401,
TAMILNADU, INDIA.
April 2013**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SASTRAUNIVERSITY
(A University Under Section 3 of UGC Act, 1956)
(Shanmugha Arts, Science, Technology & Research Academy)
TIRUMALAISAMUDRAM, THANJAVUR-613401,
TAMILNADU, INDIA.



Bonafide Certificate

Certified that this project work entitled “**A Hybrid Approach on Heuristic Task Scheduling Algorithm in Cloud Computing.**” submitted to the Shanmugha Arts, Science, Technology & Research Academy, SASTRA University, Tirumalaisamudram – 613401 by **K. Sairam [113003142]** in partial fulfillment of the requirement for Degree of **BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING** is the original and Independent work carried out under the guidance, during the period December 2012– April 2013.

INTERNAL GUIDE

Dr. A. Umamakeswari

SCHOOL OF COMPUTING

ASSOCIATE DEAN-CSE

Dr .A. Umamakeswari

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SASTRA UNIVERSITY
(A University Under Section 3 of UGC Act, 1956)
(Shanmugha Arts, Science, Technology & Research Academy)
TIRUMALAISAMUDRAM, THANJAVUR-613401,
TAMILNADU, INDIA.



DECLARATION

We submit this project work entitled “ **A HYBRID APPROACH ON HEURISTIC TASK SCHEDULING ALGORITHM IN CLOUD COMPUTING** ” to Shanmugha Arts, Science, Technology & Research Academy (SASTRA University), Thirumalaisamudram – 613401, in partial fulfillment of the requirement for the award of the degree of **BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE & ENGINEERING** and declare that it is our original and independent work carried out under the guidance of Dr. A. Umamakeswari, School Of Computing, SASTRA.

Date:

Place: SASTRA UNIVERSITY

Sign:

Name: K. Sairam

Reg. No: 113003142

ACKNOWLEDGEMENT

First and Foremost, I take pride in thanking the Almighty who gave me strength for the successful completion of this project.

I have immense pleasure in expressing my heartfelt thanks to our Honorable Vice Chancellor **Prof.R.Sethuraman** for the benevolent advice and guidance during my tenure in the college.

I would like to express my gratitude to **Dr.G.Bhalachandran, Registrar, SASTRA UNIVERSITY**, for his benevolent guidance through-out my college life.

I wish to express my thanks to **Dr. S. Vaidhyasubramaniam, Dean of Planning and Development** for his encouragement and providing me with all the needed amenities.

I would like thank **Dr.S. Swaminathan, Dean of Sponsored Research, Director CenTAB**, for helping us achieve greater heights in life by providing me with wonderful college environment.

I wish to express my sincere thanks to **Dr.P.Swaminathan, Dean, School of Computing** for his extensive interest in encouraging student during project work.

I would like to express my gratitude and Thank our internal project guide **Dr. A. Umamakeswari, Associate Dean, CSE** for her advice and guidance, which helped me to complete the project.

I express my sincere thanks to all **Staff** in the School of Computing, for their fabulous help throughout the project.

I give all glory and honor to my Parents who gave me life, support and continued encouragement in all my achievements.

TABLE OF CONTENTS

CHAPTER NO.	CONTENT	PAGE NO.
1	Introduction	1
2	Time Frame Chart	2
3	Problem Statement	3
4	Objectives of the study	4
5	Literature Review and Survey	5
	5.1 Apache Hadoop	5
	5.2 Hadoop Schedulers	7
6	Software Requirement Specification	11
	6.1 Technology used.	11
	6.2 Functional Requirements	11
	6.3 Non Functional Requirements	12
	6.4 Software & Hardware Requirements	12
	6.4.1 Software Requirements	12
	6.4.2 Hardware Requirements	13
7	Conceptual Modeling	14
	7.1 Use Case Diagram	14
	7.2 Class Diagram	15
	7.3 Activity Diagram	16
8	Component / Interface Scenario	18
	8.1 Sequence Diagram	18

Chapter no.	Contents	Page no.
	8.2 Collaboration Diagram	19
9	Methodology and Approach	20
	9.1 Balance Reduce Algorithm	20
	9.2 Module Description	29
	9.3 Implementation	31
	9.4 Programming Codes	34
10	Output/Results	44
	10.1 Testing	44
	10.2 Output	47
11	Performance Analysis	51
12	Conclusion	53
13	References	54

1. INTRODUCTION

Cloud computing is the next generation in data processing. Computation systems are bombarded with large amounts of data for processing and there is a need for analysing the information based on the data. An efficient scheduler model should be developed in managing the valuable data in the cloud. As petabytes amount of data are generated every day , the main concern shifts towards the locality of the data distributed within the cloud computing systems. Data's are processed in a distributed environment in cloud involves many servers connected to a network. Client always needs a job to be processed quickly to avail the services out of it. Thus, the client requests the master machine to process the job, expecting faster computations in providing a solution for the requested job.

Whenever a job is given as input to the master machine, scheduler takes the prime responsibility in the further actions to take place that ultimately reflects on the job completion time. Scheduler takes the requested job as input and breaks it into individual smaller tasks. The goal is to find an efficient method to schedule the tasks in minimum time, so that the data within the tasks are processed quickly. The solutions to the individual tasks are later combined to provide a overall solution to the main job. In the cloud architecture, data's are replicated among selective servers in the cloud to achieve fault tolerance. Selection of tasks for mapping to the data locale server is a major challenge in the data replication environment. Each server is presented with the initial server load, that represents the server booting time and the time required to place the task on the server. The cost of the task represents the execution time of the task in the server.

Data replication strategy also portrays the idea of preferred server concept. During the scheduling process, every time the tasks looks for data location required for its computation across the servers. As in the case of data replication, data required for that task may be located in two or more servers in the cloud systems. The Balance-Reduce scheduler keeps track of the replication factor, Load of the server, cost of the tasks and the allocation takes place by mapping tasks to the server, whose load is minimum. A major advancement in the scheduling

strategy is achieved by adjusting the task data locality. The utility factor of the server is a major concern in the computational time for data processing. In a distributed scheduling environment, the scheduler should be designed in such a way that no server is under- utilized until the whole job is completed. The Balance-Reduce scheduler is designed in such a way that it utilizes the computational power of all servers until a job is completed.

2. Time Frame Chart

Tasks Carried Out	Time Period
Initial Requirement Gathering	Dec 13 - Jan 15
Obtaining the complete requirement set and brainstorming.	Jan 16 - Jan 31
Learning and Literature Survey (Understanding the algorithm)	Feb 1 - Feb 15
Analysis of algorithm	Feb 16 - Feb 28
Prototyping	Mar 1 - Mar 10
Implementing Core Modules	Mar 11 - Mar 20
Testing Phases	Mar 21 - Mar 29
Enhancements and Bug Fixes	Mar 30 - April 5

3. Problem Statement

Cloud computing refers to the use of shared computing resources to deliver computing as an utility, and serves as an alternate method to handle local server computations. Scheduling of jobs in cloud is of prime importance which reflects on job completion time. The algorithms behind the schedulers, which were implemented in the cloud systems are not efficient as it does not take into account, the server load, remote data and locality of the data. Hence, there is a delay in job finishing time as minimum load servers are underused.

In this fast moving world, people always want a requested job to be processed by the computation systems quickly. In a highly scalable cloud systems, there is high affinity for data locality and the load of the server. The heuristic scheduling approach takes into account these factors in processing the data dynamically, which are distributed across different clusters in the cloud. Hence, improving the scheduling algorithms behind the scheduler provides a faster way for achieving the results of time critical applications in cloud.

Scheduling the jobs via the Balance-Reduce Scheduler provides a better scope in reducing the job completion time. The modified Balance-Reduce scheduler implemented by using the Edmond-Karp algorithm improves the performance of the scheduler by a noticeable extent. The main advantage of the scheduler lies in its innate nature to use minimum load servers thus, improving the overall job completion time.

4. Objectives of the Study

The main aim is to improve the Balance-Reduce scheduler by upgrading the balance phase of the algorithm and providing a better scope in job scheduling in cloud.

The advancement involves the implementation of Edmond-Karps method to the balance phase and thereby decreasing the makespan value.

The reduce phase has also been modified to a certain extent such that the data needed is not recreated from the scratch and improves its execution.

Thus, the modifications mentioned above helps in reducing the overall execution time and improves upon the job completion time.

5. Literature Survey

5.1 Apache Hadoop

Apache Hadoop provides a distributed framework for managing the tasks across servers based on a job request. The Balance-Reduce scheduler is implemented as a pluggable scheduler in the Apache Hadoop environment. The data storage system across the network is managed by the distributed file system. Hadoop contains an distributed file system named Hadoop Distributed File System(HDFS) capable of storing large files of the order of terabytes, petabytes etc. The HDFS abstraction is necessary to distinguish itself from other storage systems like local file system, Amazon EC2 etc. HDFS is designed in such a way that it supports streaming data access and commodity hardware. The data processing pattern is of the form read many times, write once pattern. The analysis for read/write operations are considered for a larger dataset rather than a small record for data processing. HDFS can run on a single node or multi node clusters of commodity hardware and does not need high end hardware requirements for data processing.

Apache Hadoop works with ease in the Linux (ubuntu) environment. After the installation, the terminal provides the best way for accessing HDFS through Linux commands. In the latest version of Hadoop, job scheduler can be refactored out of Jobtracker to plug in additional schedulers other than the default FIFO scheduler. By default Hadoop uses the default FIFO scheduler which is not very efficient as the overall waiting time to process the job is increased. Therefore there is a need to shift to Balance-Reduce scheduler in scheduling the jobs quickly. The terms that are needed to be understood in Apache Hadoop environment are as follows.

5.1.1 Blocks

In HDFS, data's are stored in the form of blocks. Unlike the normal disk system whose block size is limited to few kilobytes or bytes, HDFS uses a larger block size format of 64 MB by default. The main reason for having such a block size is to minimize the seek time.

Furthermore Blocks provides fault tolerance and availability with the replication mechanism needed for the scheduler.

5.1.2 Namenode

Namenode manages the HDFS namespace. It maintains the metadata for all the files and directories in the tree. Without the namenode, the file system cannot be accessed. The namenode also gives information about the datanode on which the file blocks are located. It does not permanently store the block location of the files since the location information is reconstructed by the datanodes when the HDFS system starts. The only way to delete the files from the HDFS is to format the namenode before starting the distributed file system. Namenode is present only at the master machine. The metadata information is stored persistently on the local disk namely the edit logs and namespace image.

5.1.3 Datanodes

Data's inside HDFS are stored in datanodes for processing. They retrieve or store blocks based on the request by the client or namenode. It periodically reports to the namenode about the list of blocks, datanodes are currently holding.

5.1.4 Secondary Namenode

The main function of secondary namenode is to take the place of primary namenode, if the primary namenode fails. It periodically merges the edit log with namespace image to prevent the edit log becoming too large. The secondary namenode usually runs in a separate CPU as it requires much memory as the primary namenode to perform the merge operation. The copy of merged namespace can be used by the secondary namenode in case of the failure of primary name node.

5.1.5 Jobtracker

The Jobtracker is the service in Hadoop that schedules the tasks to specific servers(nodes) in the cluster, ideally the servers that have the data, based on the scheduler implemented inside Hadoop. JobTracker continuously monitors the job status in every node in the cluster. The data locality concept is made possible when the Jobtracker communicates with

the Namenode in finding the location of the data. The Jobtracker periodically locates the TaskTracker nodes near the location of data and submits the work to the TaskTracker nodes based on the data availability, load and cost factor.

5.1.6 TaskTracker

The TaskTracker is a node in the cluster that accepts tasks from the Jobtracker. TaskTracker gives a notification message to the Jobtracker when the tasks fails. The Jobtracker, then decides to resubmit the job to another TaskTracker nodes in the preferred server list. A heartbeat message is periodically sent to the Jobtracker to assure that it is still alive.

5.1.7 Map-Reduce

Hadoop Map-Reduce, which is the open source implementation of Google's mapreduce contains two major phases namely the map phase and the reduce phase. In Mapping phase a task is mapped to server based on the scheduler in Hadoop. Mapping consists of creating key/value pair for every map of tasks to server. The mapped server performs the operation and generates an output. Based on the key/value of each map the results are combined.

In Apache Hadoop mapper class, takes care of the internal mapping representation of input records key/value pairs to intermediate key/value pairs. The combiner class combines the intermediate key/value pairs, which helps to cut down the amount of data transferred from mapper class to reducer class by providing the result of mapper to reducer. In Reduce phase the output of all maps is reduced to produce a final combined output. The reducer class reduces a set of intermediate values which shares a key to a smaller set of values, thus providing the result of the job. Thus map-reduce runs each task by mapping them to servers and aggregating the result of map to produce a single reduced output for each job.

5.2 Hadoop Schedulers:

Hadoop framework provides a wide array of schedulers which varies based on working environment and are as follows.

5.2.1 Default FIFO Scheduler

In Hadoop, by default FIFO scheduler without priorities is used to schedule Jobs. In FIFO each job is spilt into many tasks. Each task is assigned to a particular free node by the JobTracker Node. By default, priorities are not assigned to the tasks. Even though this method is useful in assigning tasks on the servers, yet the overall waiting time is increased and is not efficient.

5.2.2 Fair Scheduler

Fair Scheduler, developed by Facebook, is an advancement of the FIFO scheduler which minimises the waiting time of job. It supports pre-emption. In this scheduling method, jobs are assigned to pools with minimum guaranteed mapping slots. If a pool does not receive its fair share, scheduler kills the jobs running with a capacity above threshold in order to give the mapping slots to low capacity pools. Therefore, jobs are scheduled in an interleaved manner. Task tracker tracks the wastage in time during the allocation wherein the longer waited jobs are assigned to the next corresponding free slot and shorter jobs are given resources to finish quickly.

5.2.3 Capacity Scheduler

Capacity Scheduler was developed by Yahoo. It takes into account the amount of users for job allocation in configuring Map-Reduce slots. Job queues are given their configured capacity. Jobs whose capacity is free is shared among other queues. Within a queue, with specific user limits, scheduler operates on a modified priority queue based on the time of submission of job and the priority setting is based on the user and the nature of the job. Based on the priority of the jobs, scheduling is done which ensures cluster capacity sharing among users rather than jobs.

5.2.4 Deadline Constrained Scheduler

Increasing the utilisation of system resources is the main focus of the Deadline Constrained Scheduler. A constrained model focuses on the deadline factor for the resources needed to be scheduled in the cloud systems. Different jobs have different deadlines associated with it, as a result of which, the scheduler provides a method to assign the variable quantity of tasks to the task tracker that meets the deadline specifications. Scheduling tests are done periodically on the tasks to determine the finishing time of the job against the deadline constraint necessary for mapping and reducing operations.

5.2.5 Resource Aware Scheduling

Availability of resources is a major concern in the scheduling process. The capacity and fair based schedulers mainly concentrate on the fairness and capacity of the allocation of resources without taking into consideration the resource availability. Resource availability involves CPU utilization, input/output operations from disks and a number of page faults in the memory subsystem. Therefore monitoring these resources in advance by the task tracker node provides a method for improving the load balancing in Hadoop clusters. Thus, this scheduler creates a dynamic way of computing available free slots in a node using the information about resource metrics from each node in a cluster which helps to improve the job response time in a machine having assured of no resource bottlenecks. Advertising the task tracking slots is mainly decided according to the resource availability. So, tasks in the slot with higher resource availability is scheduled first, instead of taking the next available free slot. Therefore at any point of time, there will only be resource rich machine available for scheduling the tasks.// intro

5.2.6 Dynamic Priority Scheduler

Dynamic Priority scheduler provides a dynamic time slot distribution among the processes based on their priorities. Priorities of the executing tasks dynamically change at execution. Based on the new priorities of the processes, rescheduling is done. This approach

allows users and processes to share the Map reduce proportionally. These time slots are called allocation intervals. In this scheduler, processes with shorter jobs are preferred. Pre-emption of tasks is done to avoid starvation and queue blocking. Unused slots which are given to a process are retrieved from it and provided to other processes. Starvation of low priority jobs can be tackled by giving a limit to the time each job is allowed to run. This allows for setting up different rates for various users and allows them to decide on pre-emption of running tasks. If queues are not used this is equivalent to FIFO scheduler and if all queue have the same rate this behaves as fair-share scheduler.

5.2.7 Delay Scheduling

The problems in Fair Scheduling based on data locality are solved in Delay Scheduling. The problems are Head of line scheduling and Sticky slots. Head of line scheduling is the unlikely nature of having data assigned local to the nodes for small jobs where jobs are allocated based on free slots irrespective of the locality of the nodes. The Sticky slots is the Tendency of assigning jobs to the same slot repeatedly. In Delay Scheduling, jobs wait till there is a opportunity to schedule with a local data, thus improving fairness based on locality.

6. Software Requirement Specification

6.1 Technologies Used

- Java
- Apache Hadoop

6.2 Functional Requirements

6.2.1 Job details and Resource information

The system should be able to obtain the needed data from the xml files or from hadoop API and provide those details to the algorithm. The local data details should also be presented to the system for scheduling operation. The number of tasks created for the job is also provided as the initial data to the system.

6.2.2 Balance Phase Computation

The balance phase computes the balance allocation for the tasks based on the locality of data and server load. Each allocation is done by creating a augmented path from the flow graph which is updated if any new path is obtained. The augmented path calculation is done using Edmond-Karp method by creating a BFS and traversing the graph to find the augmented path. The augmented path helps in allocating tasks to server based on load and locality. Balanced phase produces an initial allocation based on the server load.

6.2.3 Reduce Phase Computation

The reduce phase does the final allocation on the result of balance phase. It reduces the load of maximum load server by allocating its tasks to remote server with lower load. The allocation is done by creating remote data for the lower load server and placing tasks in it. The heuristic approach of the reduce phase checks at each step to find a better allocation compared with the previous step's allocation of tasks and aims to reduce the overall makespan.

6.3 Non Functional Requirements

6.3.1 Maintaining Log Record of all processes

Log records are maintained for each step of the process which helps in understanding the system and finding faults, which may happen at some point, for example in finding the augmented path. The log record provides a concise description for identifying the actual cause of problem and helps in their rectification.

6.3.2 Error Message and proper system recovery

The system should have proper mechanism to provide error message to the users so that user can know is the exact cause of system failure and system should not abruptly shut down in any case.

6.4 Software and Hardware Requirements

6.4.1 Software Requirements

- Netbeans IDE
- Java 1.7
- Apache Hadoop Core System
- Ubuntu 12.04 LTS(Linux)
- Rational Rose

6.4.2 Hardware Requirements

Hardware	Minimum	Maximum
CPU	2 GHz	3.2GHz Workstation
Memory	4 GB	16GB
Disk	5400 rpm IDE	7200rpm SATA
Storage Space	500 GB	>2 TB
Networking	100Mbps	1Gbps

7. Conceptual Modeling

UML, or Unified Modeling Language, is a specification language that is used in the software engineering field. It can be defined as a general purpose language that uses a graphical designation which can create an abstract model. This abstract model can then be used in a system. This system is called the UML model. The Object Management group is responsible for defining UML, and they do this via the UML Meta model.

7.1 Use Case Diagram

The use case diagram below depicts the various types of use-cases that are involved in the interaction between the program and the nodes.

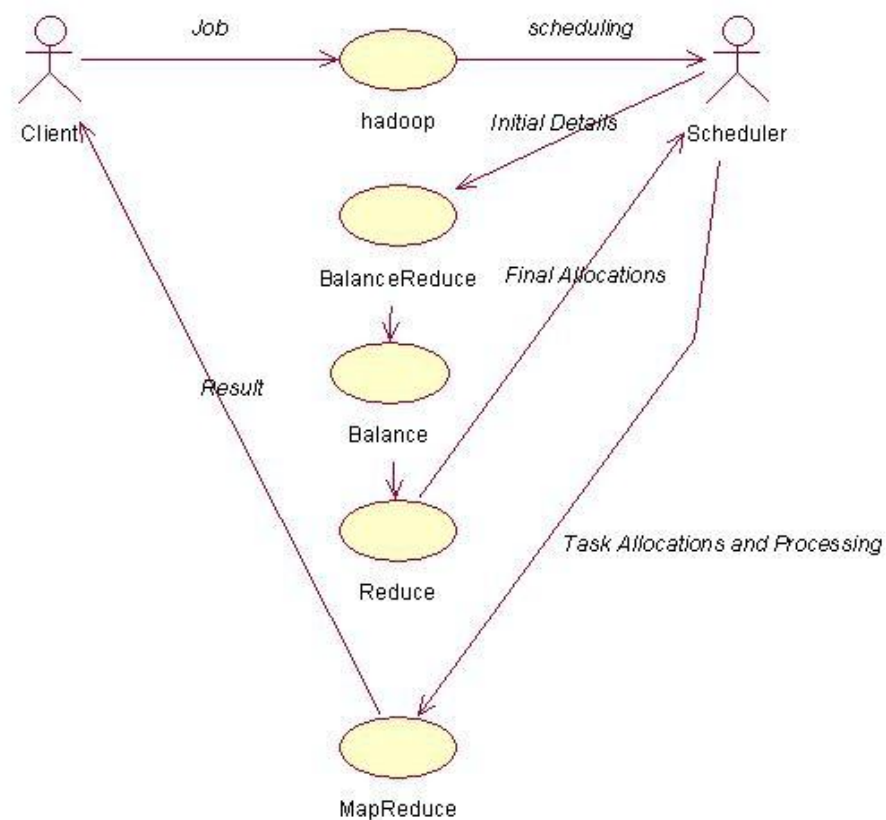


Fig.7.1 Use Case Diagram

7.2 Class Diagram

The class diagram depicts the member variables and member functions of the classes that are interacting. Each actor is considered a class. Their type of interaction is also mentioned, i.e., 1:1 or 1: n, etc.

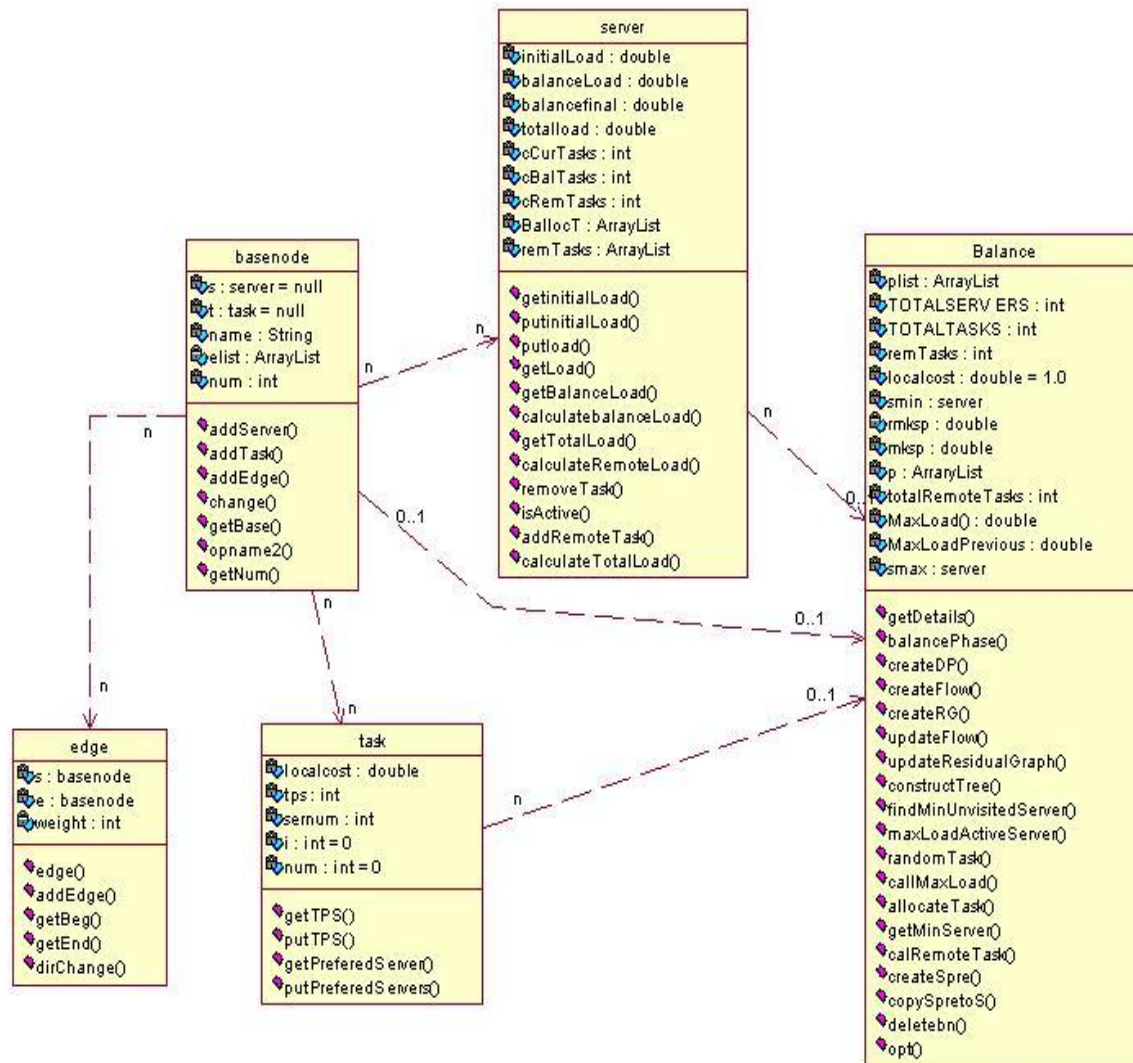


Fig. 7.2 Class diagram

7.3 Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modelling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system.

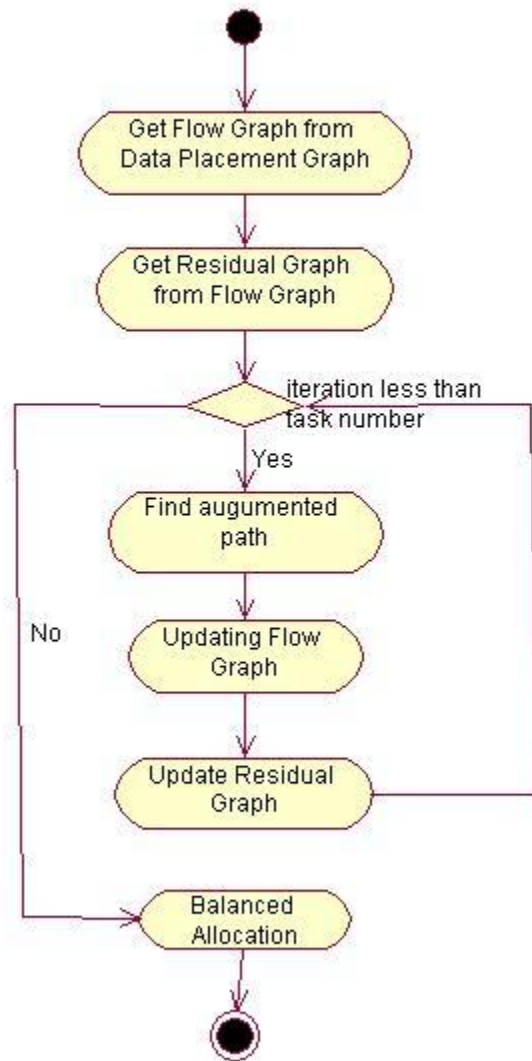


Fig. 7.3.1 Activity Diagram for Balance Phase

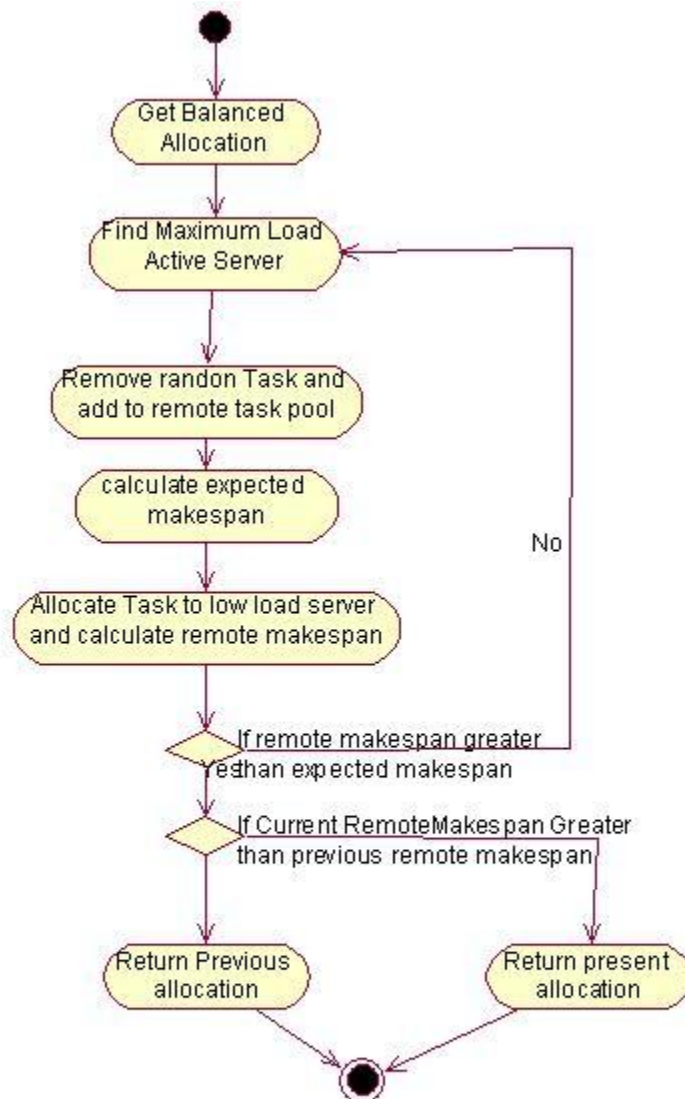


Fig 7.3.2 Activity Diagram for Reduce Phase

8. INTERACTION SCENARIO

8.1 Sequence Diagram

A sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a Message Sequence Chart. A sequence diagram shows object interactions arranged in time sequence.

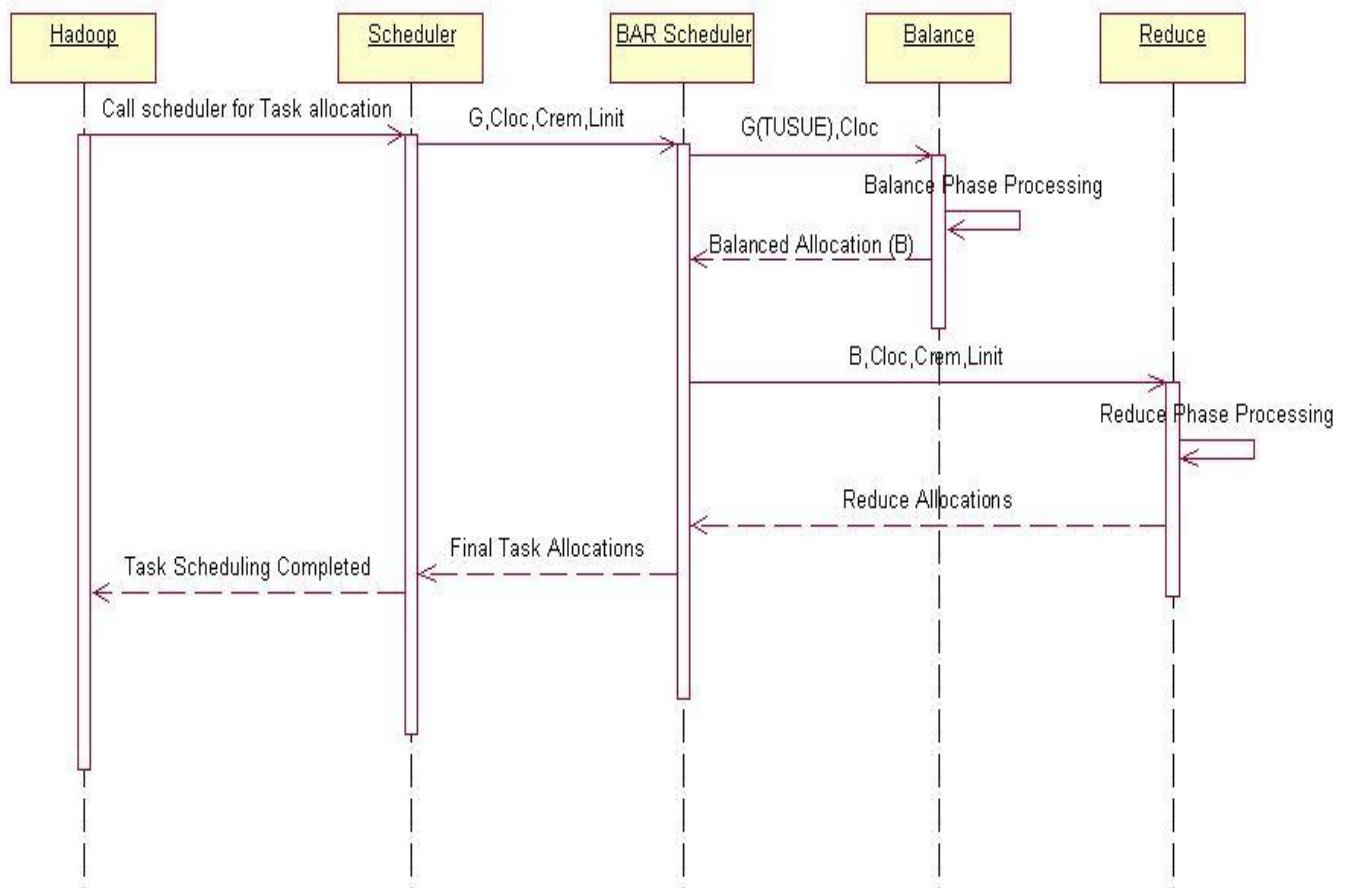


Fig. 8.1 Sequence diagram for Balance-Reduce algorithm

8.2 Collaboration Diagram

A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.

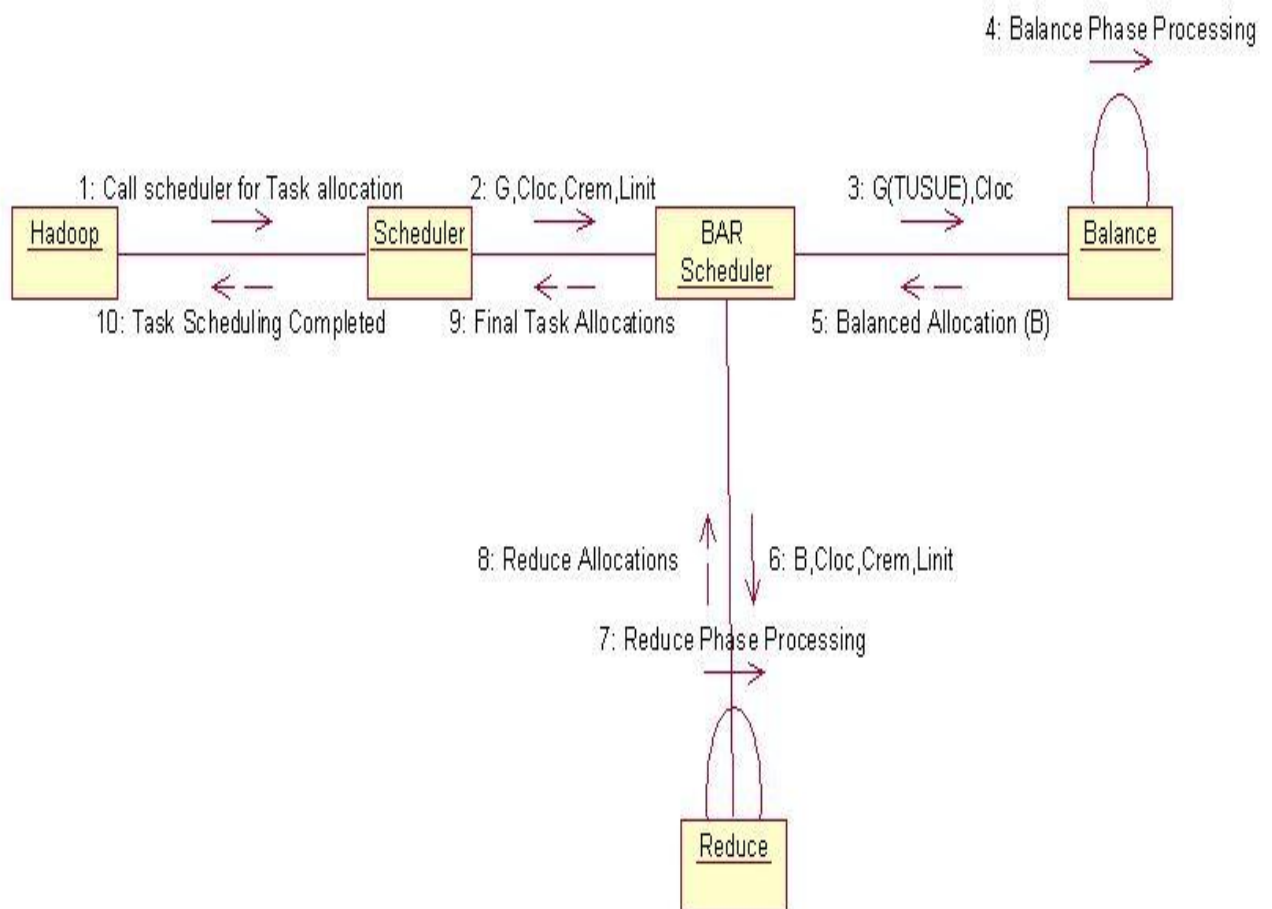


Fig 8.2 Collaboration Diagram for Balance-Reduce algorithm

9. Methodology and Approach

9.1 Balance Reduce Algorithm

The Algorithm implemented in the Balance-Reduce scheduler is divided into two phases. First phase is the **Balance phase** followed by the **Reduce phase**. The Balance phase balances the load across all servers evenly. The reduce phase reduces the load in the balanced allocation by transferring the load from maximum load server to the minimum load server.

9.1.1 Balance Phase

In the Balance phase, all tasks are allocated to the preferred servers based on the initial load. Preferred server indicates that the data needed by the tasks are local to that server. The selective replication strategy helps in replicating the blocks of data to certain servers. This approach helps in saving a considerable amount of time, as there is no need to search for all the servers in finding the data required for a particular task. The task checks for the preferred server list in which the required blocks of data are replicated and neglect the other servers not needed for its computation. Whenever a task prefers a server a mapping is done between them. Mapping can be easily represented in a Graph data structure. The nodes contain a set of all servers and tasks. As soon as the task prefers the server an edge is used to connect them. This graph is named as data placement graph which represents only the initial mapping of tasks to the preferred servers. The data placement graph is the base graph for finding the balanced allocation at the later stage of this phase.

The next step is to map the task to only one server in the preferred server list whose load is minimum. For this purpose the data flow graph is created from the data placement graph, which maps all the tasks from the data placement graph to a single target node. It represents the directed flow from source node to the target node. For each task, Balance phase Allocation periodically calculates the Minimum load unvisited server. If a task prefers that server, a mapping is done from the server to the target node through the task. An equivalent amount of reverse flow takes place from target node to

the server. It means that the capacity with which the data is sent to the target node can be equally received to the source node in the reverse direction. Therefore a residual graph is created from the data flow graph, which represents the reverse flow of data from the sink to the source node at each step of the iteration. Residual graph gives an idea about the multiple path ways through which the source and target node are connected. This graph will be particularly useful in a highly scalable cluster systems, in which there will be numerous pathways connecting the source and target node and there will be an necessity in choosing the best path by creating an augmented path from the residual graph.

An augmenting path will be created from the residual graph for each server by finding the best available path from the directed residual graph. The augmenting path will be found by using the Edmond-Karps algorithm, which runs in time complexity of $O(VE^2)$, where V represents the vertexes and E represents the edges. Its inclusion provides a major boost to the algorithm efficiency as the augmenting path will be found quickly under various dynamic changing environments. Ford-Fulkerson which was used in the Balance-Reduce algorithm does not guarantee termination for special cases involving irrational capacities (load). Therefore, a breadth first search is introduced in the residual graph by finding the augmenting path in minimum time. The main strategy is to provide a tighter bound to the complexity of the algorithm so that the algorithm will guarantee successful termination under various load conditions. Each iteration, adds a new augmenting path and is updated with the flow graph and residual graph accordingly at the end of the iteration. At the final iteration of this phase, the augmenting paths from the source node to the target node through which the data flows will be clearly visible and the edges that contributes to this path gives way for the balanced allocation across preferred servers. The Balance phase returns the balanced allocation for the set of tasks and servers and stops the iteration beyond which no more augmenting paths can be found. The minimum cut edges found from the augmenting path, represents the Balanced allocations. Min-cut edges represents that maximum flow is reached from the source to sink and no more flow is possible in all direction. Min-cut edges gives the minimum number of edges that must be removed from the graph to disconnect the source and target node.

The following terms need to be understood in the working of Balance Phase Algorithm.

1. Data Placement: The data placement is presented by a bipartite graph $G = (T \cup S, E)$, let $m=|T|$ and $n=|S|$, where T is the set of tasks, S is the set of servers and E is the set of edges between T and S . The edge $e(t,s)$ implies that the input data of task t which belongs to T is placed as server s which belongs to S . It also indicates that task t prefers server s . It is assumed that all the nodes in T have degree at least 1.

2. Allocation Strategy: An allocation strategy is a function $f : T \rightarrow S$ that allocates a task t to a server $f(t)$. An allocation is total if for each task t belongs to T , $f(t)$ is defined. Otherwise it is a partial allocation. Let α be a total allocation, so task t is allocated to server $\alpha(t)$. Under α , task t is local if $\alpha(t)$ is defined and there is an edge $e(t, \alpha(t))$ in the data placement graph G . Otherwise, task t is remote. Let l^α and r^α be the number of local tasks and the number of remote tasks respectively.

3. Cost of Task: All the servers and tasks are homogeneous, so that each task consumes identical execution cost on every server. Furthermore, the cost of task is defined as the sum of execution time and the input data transferring time. $C(t, \alpha)$ denotes the cost of task t which is performed on the server $\alpha(t)$. It is defined by **$C(t, \alpha) = \{ C_{loc}, \text{ if } t \text{ is local in } \alpha \text{ (or) } C_{rem}^\alpha, \text{ otherwise } \}$** . For all allocation, the local cost is constant. However, since remote tasks compete for network resources, the remote cost grows with the total number of remote tasks. Let **$C_{rem}^\alpha = C_{rem}(r^\alpha)$** , where $C_{rem}(\cdot)$ is a monotonically increasing function and r^α the remote task number.

4. Load of the Server: Under allocation α , some servers are assigned several tasks, The load of the server s denotes the time when s finished its work, Under α , the load of the server s is defined as **$L_s(\alpha) = L_s^{init} + L_s^{task}(\alpha)$** , where L_s^{init} is the initial load of the server s at the start time, **$L^{init} = \{ L_s^{init} \mid s \text{ belongs to } S \}$** be the set of all initial loads. **$L_s^{task}(\alpha) = \sum_{t: \alpha(t)=s} C(t, \alpha)$** is the total time to process tasks on server s . If the

allocations is total, $L_s^{\text{fin}}(\alpha) = L_s(\alpha)$ denotes the final load of server s . Let $S_{\text{ass}}^\alpha = \{s \mid \text{For all } t \text{ belongs to } T \text{ that } \alpha(t) = s\}$ denotes the server which are assigned tasks. For each server s , if s is in S_{ass}^α then s is active. The job completion time (which is also called makespan) under total allocation β is defined as $\text{makespan } \beta = \max L_s^{\text{fin}}(\beta)$, where s belongs to S_{ass}^β .

5. Local Allocation: Let L be an allocation. Under L , if all defined tasks are local, then L is a local allocation.

6. Balanced Allocation: Let $G(T \cup S, E)$ be a dataplacement, C_{loc} be a local cost, and β be a local allocation, β be a balanced allocation when for all t belongs to T , $L_{\beta(t)}(\beta) - C_{\text{loc}} \leq \min L_s(\beta)$ where s belongs to the preferred server for the task t . This equation is called Balance Policy. For finding the balanced allocation this equation must be satisfied.

Let us look into detail the Algorithm of Balance Allocation Phase in Balance-Reduce Scheduler.

Algorithm: Balance Allocation

Step 1: procedure BALANCE ($G(T \cup S, E)$, C_{loc} , L^{init})

Step 2: Define G_f is a flow network. N is a set of nodes in G_f . G_r is the residual graph. τ is the iteration number. P^τ is a augmenting path at iteration τ . Tree^τ is the search tree at iteration τ . f^τ is the flow on G_f after iteration τ . β is a total allocation.

Step 3: $G_f \longleftarrow \text{GetFlowNetwork}(G)$

Step 4: $G_r \longleftarrow \text{GetResidualGraph}(G_f)$

Step 5: $N \longleftarrow T \cup S \cup \{n^T\}$

Step 6: For all s belongs to S , $s.\text{num} \longleftarrow 0$

Step 7: $\tau \longleftarrow 1$

Step 8: While $\tau \leq m$ do

Step 9: For all n belongs to N , $n.visited = \text{false}$

Step 10: While $P = \text{null}$ do

Step 11: $s_0^\tau \longleftarrow \text{MinLoadUnvisitedServer}(S, C_{\text{loc}}, L^{\text{init}})$

Step 12: $(P^\tau, \text{Tree}^\tau) \longleftarrow \text{Augment}(s_0^\tau, n^T, G_r)$

Step 13: For all n belongs to Tree^τ $n.visited = \text{True}$

Step 14: Clear(Tree^τ)

Step 15: if P not equal to Null then

Step 16: $s_0^\tau.\text{num} \longleftarrow s_0^\tau.\text{num} + 1$

Step 17: end if

Step 18: end While

Step 19: $f^\tau \longleftarrow \text{UpdateFlow}(f^{\tau-1}, P^\tau)$

Step 20: $G_r \longleftarrow \text{UpdateGraph}(G_f, f^\tau)$

Step 21: $\tau \longleftarrow \tau + 1$

Step 22: end While

Step 23: $\beta \longleftarrow \text{FlowToAllocation}(f^m)$

Step 24: Return β

Step 25: end procedure

The Data placement graph $G(T \cup S, E)$ is converted to a flow network $G_f(T \cup S \cup \{n^T\}, A)$, where n^T is a target node, and A is a set of arcs with a positive capacity. An arc $a(n_i, n_j)$ represents a directed edge from node n_i to n_j . For all tasks t belongs to T and all servers s belongs to S , if there is an edge $e(s, t)$ in G , arc $a(s, t)$ exists in A . In G_f , for each t belongs to T , there exists an arc $a(t, n^T)$. Each arc has a capacity of one.

Allocate unassigned tasks to the minimum load servers iteratively, until all tasks have been allocated. At iteration τ ($1 \leq \tau \leq m$), Firstly, mark all nodes as unvisited. Secondly, select an unvisited server node whose load is minimal. Since all tasks are local, the load of server s can be computed as follows, $L_s = L_s^{\text{init}} + \text{Num}(s) * C_{\text{loc}}$ where $\text{Num}(s)$ represents the number of task on server s . Thirdly, find an augmenting path from the selected server node s_0^τ to n^T in the residential graph by growing a search tree Tree^τ which traverses the unvisited nodes. If there exists an Augmenting path, then stop the iteration, assign a unit of flow through the path $s_0^\tau \longrightarrow n^T$ and update the residential graph. Otherwise, mark all nodes in Tree^τ as visited, then turn to the second step of iteration. If the amount of flow equals to total task number m , the iterative process stops and returns a flow f^m with m units.

Convert the flow f^m to an allocation β . Since there are m units of flow in f^m , all tasks are allocated. Hence β is the total allocation.

9.1.2 Reduce Algorithm

In the reduce phase, using the result of balanced allocation from the Balance phase, Maximum load active server is calculated at each and every iteration. A Server is active if it has some allocated tasks. From the active server, a random task is taken and added to the remote task pool. The remote task pool contains a list of all remote tasks to be added to the minimum load server. Tasks are removed from the local balanced allocation, and the expected makespan for the local allocation is calculated at every iteration. The remote cost is calculated through a predefined remote cost function which is an increasing function, while allocating tasks to the minimum load server. The remote cost function will be used as a heuristic function in comparing the makespan values. In a particular case when all the tasks are removed from the server, it becomes inactive and is not used in calculating makespan values. Makespan calculation involves only active servers.

The expected makespan and the makespan after allocating remote tasks is calculated from the remote tasks allocations, if the former is greater or equal to the later, the total remote allocation is assigned to the previous total remote allocation. Otherwise if the makespan after adding remote cost in the present allocation is greater than equal to the makespan after adding remote cost in the previous allocation, the previous reduced allocation is returned, else the present reduced allocation is returned. This marks the end of reduce phase, which represents the overall balanced and reduced mapping of tasks across the servers based on load and locality.

Let us look into detail the algorithm of Reduce phase in Balance-Reduce scheduler

Algorithm: Reduce

Step 1: procedure REDUCE (C_{loc} , C_{rem} , β , L^{init})

Step 2: define P is a remote task pool, L_p is a local partial allocation, R and R_{pre} are total allocations, M_{exp} is an expected makespan.

Step 3: $P \leftarrow \text{Null}$

Step 4: $L_p \leftarrow \beta$

Step 5: $R_{\text{pre}} \leftarrow \beta$

Step 6: While true do

Step 7: $s_{\text{max}} \leftarrow \text{MaxLoadActiveServer}(L_p)$

Step 8: $t_l \leftarrow \text{RandomTask}(s_{\text{max}}, L_p)$

Step 9: $P \leftarrow P \cup \{t_l\}$

Step 10: $L_p \leftarrow \text{RemoveTask}(L_p, t_l)$

Step 11: $M_{\text{exp}} \leftarrow \text{MaxLoad}(L_p)$

Step 12: $C_{\text{rem}}^R \leftarrow C_{\text{rem}}(|P|)$

Step 13: $R \leftarrow \text{AllocateTasks}(P, L_p, C_{\text{rem}}^R, M_{\text{exp}}, L^{\text{init}})$

Step 14: if $\text{makespan}_R > M_{\text{exp}}$ then

Step 15: if $\text{makespan}_R \geq \text{makespan}_{R_{\text{pre}}}$ then

Step 16: return R_{pre}

Step 17: else

Step 18: return R

Step 19: end if

Step 20: end if

Step 21: $R_{\text{pre}} \leftarrow R$

Step 22: end while

Step 23: end procedure

In this Algorithm, A remote task pool P is defined which stores the tasks from the maximum load active servers. Tasks in P are allocated to low load servers, and these tasks are having remote data. L_p is a local partial allocation where for each task t in P , $L_p(t)$ is not defined. P is initialized to be empty,, while L_p is initialized to be the balanced total allocation which is produced by the balance phase. At each loop, update P and L_p , then generate a total allocation R . R_{pre} is a total allocation which is computed in the previous loop.

Select the maximum load server s_{max} under L_p , s_{max} is also the maximum load server under R_{pre} . To reduce the makespan, task is removed from s_{max} . In the next step this task is allocated to the lower load server.

Let t_i be the chosen task. t_i is added to the remote task pool P and update L_p by marking $L_p(t_i)$ undefined.

Calculate the expected makespan of R , The expected makespan M_{exp} equals to the maximum load of servers under L_p .

Calculate the remote cost. As task in L_p are data-local and tasks in P are data-remote, the remote task number should be $|P|$. Hence, $C_{rem}^R = C_{rem}(|P|)$.

, then allocate t_{rem} to the server with minimum load. Then the load of the server is updated.

If $makespan_R$ is larger than the expected makespan M_{exp} , it is impossible to reduce the makespan in the next loops. So a better allocation is selected between R and R_{pre} , then it is returned.

9.2 Module Description

The system has 4 modules which are given below:

9.2.1 Obtaining Initial Details Module

The module obtains the needed data from the xml files. The xml files are parsed using an xml parser which extracts the data from the value tags inside the property tag. Xml parsing is done using an open source simple xml parser which efficiently extracts the data. The xml files contains the data on servers and the data on tasks which are the input to system.

9.2.2 Balance phase Module

The balance phase module consists of 5 parts

9.2.2.1. GetFlowNetwork

A flow network graph is created from the data placement graph which is a flow graph from the server to the n^T node.

9.2.2.2. GetResidualGraph

A residual graph is created from the flow graph which denotes the reverse flow of all forward paths.

9.2.2.3. Augment path

An augmented path is created from the flow graph implemented using Edmond-Karp algorithm using BFS traversal technique.

9.2.2.4. Updateflow

The flow graph is updated after finding an valid augmented path from the server to the n^T and updating the flow in the graph

9.2.2.5. UpdateResidualGraph

The residual graph created from the flow graph is updated of the details on the augmented path to show the reverse flow from n^T to the servers.

9.2.3 Reduce Phase Module

The Reduce phase consists of 4 parts:

9.2.3.1 MaxLoadActiveServer

The server with maximum load among the servers is returned from the local allocation, provided the server is active.

9.2.3.2 RandomTask

Any task can be taken at random from the maxloadactiveserver and added to the task pool.

9.2.3.3 Makespan

The makespan value (M_{exp}) of the local allocation is returned which is the maximum load of the local allocation.

9.2.3.4 Allocate Tasks

Tasks from the task pool are allocated to servers with minimum load and a remote allocation is updated after each step of allocation.

The reduce also compares the value for M_{exp} and $makespan_R$ to heuristically decide on the final reduced allocation.

9.2.4 Output Module

The output module consist of 2 parts

9.2.4.1 Balance Allocation and Reduce allocation window

The balanced allocations with balance load and the list of active servers are displayed and reduced allocations with load values, tasks allocations and active servers are shown.

9.2.4.2 Line Chart of Balance-Reduce Scheduler. A Line Chart of all servers against the load is drawn depicting the change in server loads at each phase.

9.3 Implementation

9.3.1 Implementation in Hadoop

The Balance Reduce algorithm is implemented in Apache Hadoop for working in large clusters and distributed environment. A pluggable scheduler is created for the algorithm such that it can be plugged in and used by the scheduler. The algorithm needs more information on the type of the task and the file block details, various information which are needed can be obtained from internal core classes of Hadoop JobTracker, TaskTracker TaskScheduler, and NameNode.

JobTracker provides the details of the newly created tasks for the particular job submitted. It contains the details such as Task id, size of the task, type of the task and preferred servers for the task. Different type of tasks include MappingTasks, ReducingTasks, MainTasks etc., Balance Reduce algorithm takes only the main tasks which has the working part of the job. Task id represents the unique id for each task which uniquely identifies each task. Preferred servers provide the details of the servers which has the local data for a particular task.

TaskTracker provides the details of the active server nodes which are used for processing the tasks. The server nodes can be passive due to various reasons like unavailability etc.. so the algorithm uses only the active servers which are up currently. Each of the running server may have the load of the previous scheduled tasks if it is running for quite sometime or may be new. This load is the initial load of the server and is used in scheduling the next task. Initial load can vary according to the tasks allocated to the server and number of tasks being processed in these servers at that instant.

TaskScheduler allows to plug-in a new scheduler. It is to the Balance Reduce Scheduler for using it and the data locality has to be set for the Rack such that transferring data as remote data is possible.

NameNode provides the information about the data blocks that has be placed in the various servers. This provides the preferred server detail to the JobTracker such that each task will know the servers which will be having the data blocks it requires for processing. NameNode in the server divides the data into blocks and it will be distributed across various servers. These data blocks contains the resources for the tasks which include can be databases, allocation structure or files that are needed by the tasks. These data blocks are distributed by hadoop internally. A function is created which will gather all the information about the data block distribution and it will be provided to the JobTracker. JobTracker in turn assigns these values to the main tasks as the preferred servers. These preferred servers contains the data blocks required for a particular task. Hence it is calculated as the local cost for that task.

Balance Reduce Algorithm is executed with the following input details and it provides output as the simple data structure which contains the Server Id associated with the task ids which has to be scheduled in that particular server. This list may contain local tasks and remote tasks. Remote tasks are allocated to the particular server if the preferred servers of that task is overloaded with other tasks and the remote cost for that data block is less than the local cost in preferred servers.

The output of the Balance Reduce Algorithm is provided to the TaskScheduler and JobTracker. They will allocate the tasks with the particular servers and the appropriate server load for the next scheduling is updated. JobTracker in turn provide the details for the allocation to the Mapping phase of the Map-Reduce which distributes the tasks to the different servers according to the list provided by the JobTracker. Map phase will distribute the tasks to the servers mentioned and computation is done in the particular servers. Reduce phase will gather all the completed task to the master node which combines the tasks to the job and provides the output.

If the server is down due to the various reasons like power failure, overload, network errors etc., a delay is provided to the server and all the task scheduled to the server will wait until the server resume. If the server is down for more than the threshold value specified then all the tasks are rescheduled by the JobTracker or it is provided to the next active server.

JobTracker provides the replication of the tasks created in factor of 0.05 to 0.15 to avoid any loss of the tasks during the processing in the master and the slave. This also provide the way to eliminate the possible delay in the completion time due to failure of task in any slave. If a particular task which is not replicated fails in the slave it would be found only during the reduce stage of Map-Reduce cycle . In that situation the job tracker recreates that particular task and does the process of mapping and reducing and obtains the overall accurate result. Exceptions indicated above are handled by the hadoop system itself.

9.3.2 Simulation using XML files

Real time deployment of hadoop contains hundreds of servers and thousands of tasks handled by the servers each minute. Since it is difficult to provide the large scale implementation and non availability of resources, a simulation environment is created for showing the implementation in hadoop. XML files are created such that it will provide the details that are provided by the hadoop environment. Several XML files are created for various test cases and it is tested in balance reduce algorithm showed in hadoop simulation environment.

All the XML files contains a set of property and value tags which contains the initial details needed by the algorithm. The files are parsed using a simple XML parser and essential details are extracted from the file and given as input to the algorithm.

By utilizing the XML file methodology are large simulation environment with 100s of servers and 1000s of tasks can be created and tested successfully.

9.4 Programming Code

9.4.1 Balance Phase

```
public void balancephase(double localcost){
    createDP();
    createflow();
    createRG();
    ArrayList<basenode> plist = new ArrayList<basenode>();
    plist=null;
    for(int i=0;i<TOTALSERVERS;i++) {
        s[i].cCurTasks=0;
        int tou=1;
        while(tou<=TOTALTASKS){
            for(int i=0;i<TOTALSERVERS+TOTALTASKS+1;i++){
                rg[i].visited=false;
            }
            while(plist==null){
                basenode smin=findMinUnvisitedServer(localcost);
                plist=constructTree(smin);
                System.out.println(smin.name);
                if(plist==null){
                    smin.visited=true;
                }
                else if(plist!=null){
                    smin.s.cCurTasks+=1;
                }
                else{ }
            }
            updateflow(plist);
            updateResidualGraph(plist);
        }
    }
}
```



```

        tou++;
        plist=null;
    }
}

```

9.4.2 Creating Data Placement Graph

```

public void createDP(){
    for(int i=0;i<TOTALTASKS;i++)
    {
        int count= t[i].tps;
        for(int j=0;j<count;j++)
        {
            int ser=t[i].putPreferedServers(j);
            bn[ser].addedge(bn[ser],bn[TOTALSERVERS+i]);
        }
        bn[TOTALSERVERS+i].addedge(bn[TOTALSERVERS+i],
bn[TOTALSERVERS+TOTALTASKS]);
    }
}

public basenode findMinUnvisitedServer(double localcost){
    double minload=0;
    int sernum=-1;
    for(int i=0;i<TOTALSERVERS;i++){
        if(rg[i].visited==false){
            if(minload==0.0){
                minload=rg[i].s.getLoad(localcost);
                sernum=i;
            }
        }
    }
}

```

```

        else if(minload>s[i].getLoad(localcost)){
            minload=s[i].getLoad(localcost);
            sernum=i;
        }
    }
}
System.out.println(sernum);
return rg[sernum];
}

```

9.4.3 Creating Flow Graph

```

public void createflow(){
    flow=new basenode[TOTALSERVERS+TOTALTASKS+1];
    for(int i=0;i<TOTALSERVERS+TOTALTASKS+1;i++){
        flow[i]=new basenode();
        flow[i].cel=bn[i].cel;
        flow[i].indegree=0;
        flow[i].name=bn[i].name;
        flow[i].s=bn[i].s;
        flow[i].t=bn[i].t;
        flow[i].num=bn[i].num;
        flow[i].elist.clear();
        flow[i].visited=bn[i].visited;
        flow[i].intnum=bn[i].intnum;
    }
}

```

9.4.4 Creating Residual Graph

```

public void createRG(){
    rg=new basenode[TOTALSERVERS+TOTALTASKS+1];
}

```

```

for(int i=0;i<TOTALSERVERS+TOTALTASKS+1;i++){
    rg[i]=new basenode();
    rg[i].cel=bn[i].cel;
    rg[i].indegree=bn[i].indegree;
    rg[i].name=bn[i].name;
    rg[i].s=bn[i].s;
    rg[i].t=bn[i].t;
    rg[i].num=bn[i].num;
    rg[i].visited=bn[i].visited;
    rg[i].intnum=bn[i].intnum;
    if(!bn[i].elist.isEmpty()){
        int j=0;
        while(j<bn[i].elist.size()){
            edge tmp=bn[i].elist.get(j);
            rg[i].elist.add(new edge(tmp.s, tmp.e));
            j++;
        }
    }System.out.print(rg[i].name);
}
}

```

9.4.5 Update Flow Graph

```

public void updateflow(ArrayList<basenode> plist){
    basenode ser=null,e=null;
    for(int j=0;j<plist.size()-1;j++){
        for(int i=0;i<TOTALSERVERS+TOTALTASKS+1;i++){

            if(plist.get(j).intnum==flow[i].intnum){
                ser=flow[i];
            }
        }
    }
}

```

```

    }
    if(plist.get(j+1).intnum==flow[i].intnum){
        e=flow[i];
    }
}
if(e.t!=null){
    if(e.t.fs==null){
        e.t.fs=ser.s;
    }
    else if(e.t.fs!=null){
        for(int i=0;i<TOTALSERVERS;i++){
            if(e.t.fs==flow[i].s){
                for(int k=0;k<flow[i].elist.size();k++){
                    if( flow[i].elist.get(k).e.name.equals(e.name)){
                        System.out.println("Removed edge:"+flow[i].elist.get(k));
                        flow[i].elist.remove(k);
                        e.indegree-=1;
                        e.t.fs=ser.s;
                    }}}
            }
        }
    }
}

```

9.4.6 Update Residual Graph

```

private void updateResidualGraph(ArrayList<basenode> plist) {
    rg=null;
    createRG();
    for(int i=0;i<TOTALSERVERS+TOTALTASKS+1;i++){

        for(int j=0;j<flow[i].elist.size();j++){for(int z=0;z<bn[i].elist.size();z++){
            if((bn[i].elist.get(z).e.intnum==flow[i].elist.get(j).e.intnum)){

```

```

System.out.println(flow[i].elist.get(j).toString());
String tmp=flow[i].elist.get(j).e.name;
for(int k=0;k<TOTALSERVERS+TOTALTASKS+1;k++){
    if(tmp.equals(rg[k].name))
    { rg[i].changedir(rg[i],rg[k]);
      System.out.println("after"+rg[i].elist + "      "+ rg[k].elist);break;
    }
}
}
}}
}

```

9.4.7 Reduce phase

```

public server[] reduce(){

    int totalrmtasks=0;
    System.out.println("in reduce phase");
    ArrayList<task> P = new ArrayList<task>();
    createspre();
    while(true){
        server smax=MaxLoadActiveServer();
        task ta= RandomTask(smax);

        P.add(ta);
        smax.removeTask(ta);
        Double Maxload= calMaxload(1);
        System.out.println("Mexp:"+Maxload);
        allocateTask(ta,P);
        Double pMaxload= calMaxload(2);
        System.out.println("pMaxload > Maxload: "+pMaxload+">"+Maxload);
    }
}

```

```

        if(pMaxload>Maxload)
        {
            if(pMaxload>=calMaxload(3)){
                return spre;
            }
            else {
                return s;
            }
        }
        for(int i=0;i<TOTALSERVERS;i++){
            System.out.println(s[i]+"\\n\\tbalance
taskss"+s[i].BallocT+" "+"\\n\\treduce tasks"+s[i].remTasks+ s[i].balanceload + "
"+s[i].getTotalload());
        }
        copyStoSpre();
    }
}

ser.addedge(ser, e);

System.out.println(ser.elist.size()+ser.elist.toString()+e.indegree+e.toString());
    }
}
}

```

9.4.8 Finding Maximum Load Active Server

```

private balance.server MaxLoadActiveServer() {
    double tmp=0;
    int cc=0;

```

```

for(int i=0;i<TOTALSERVERS;i++){
    if(tmp==0.0&& s[i].isActive()){
        tmp=s[i].getTotalload();
        cc=i;
    }
    if(s[i].isActive()&& tmp<s[i].getTotalload()){
        tmp=s[i].getTotalload();
        cc=i;
    }
}
System.out.println("smax :"+s[cc]);
return s[cc];
}

```

9.4.9 Finding Random task

```

private task RandomTask(balance.server smax) {
    System.out.println("Random task:"+smax.BallocT.get(smax.BallocT.size()-
1));
    return smax.BallocT.get(smax.BallocT.size()-1);
}

```

9.4.10 Allocating remote tasks

```

private void allocateTask(task t, ArrayList<task> P) {
    server smin=getminServer();
    smin.addRemoteTask(t);
    calculateRemoteTasks(P);
    System.out.println("t"+t+"\t"+"and p is "+P);
}

```

9.4.11 Calculating Remote Tasks

```
private void calculateRemoteTasks(ArrayList<task> P) {
    rmtasks=0;
    for(int i=0;i<TOTALSERVERS;i++){
        rmtasks+=s[i].remTasks.size();
    }
    if(P.size()==rmtasks)
        System.out.println("rmtask==P.size"+P.size());
    else
        System.out.println("rmtask="+rmtasks+"P.size()" +P.size());
}

private void copyStoSpre() {
    spre=null;
    spre=new server[TOTALSERVERS];
    for(int i=0;i<TOTALSERVERS;i++){
        spre[i]= new server();
    }
    for(int i=0;i<TOTALSERVERS;i++){
        int k=0;
        for(int j=0;j<s[i].BallocT.size();j++){
            spre[i].BallocT.add(s[i].BallocT.get(j));
        }
        for(int j=0;j<s[i].remTasks.size();j++){
            spre[i].remTasks.add(s[i].remTasks.get(j));
        }
        if(s[i].BallocT.isEmpty())
            spre[i].BallocT.clear();
        if(s[i].remTasks.isEmpty())
            spre[i].remTasks.clear();
    }
}
```



```
spre[i].initialload=s[i].initialload;
spre[i].balanceload=s[i].balanceload;
spre[i].cBalTasks=s[i].cBalTasks;
spre[i].cRemTasks=s[i].cRemTasks;
spre[i].cCurTasks=s[i].cCurTasks;
spre[i].balancefinal=s[i].balancefinal;
spre[i].num=s[i].num;
spre[i].totalload=s[i].totalload;
System.out.println("copied s to spre "+i+" "+ s[i]);
    }
}
```

10. Output/Results

10.1 Testing

10.1.1 Functionality Testing

10.1.1.1 Testing For Invalid data in servers and tasks:

Each server and task should be provided with a valid value otherwise an error message would be shown as below

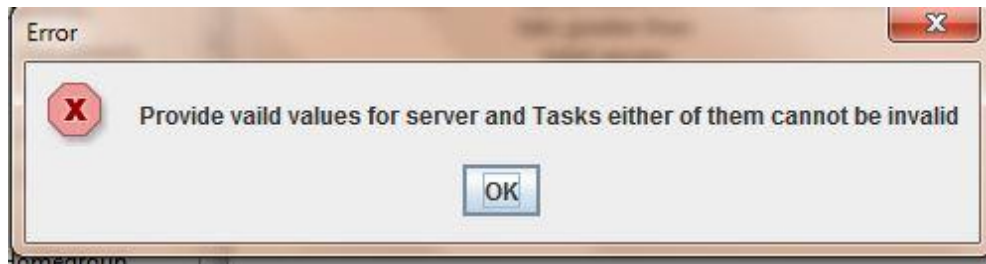


Fig 10.1.1.1 Error message for invalid data

10.1.1.2 Testing for Invalid data in Server Load details:

Each server should be provided with a valid server initial load details else an error message would be shown as below

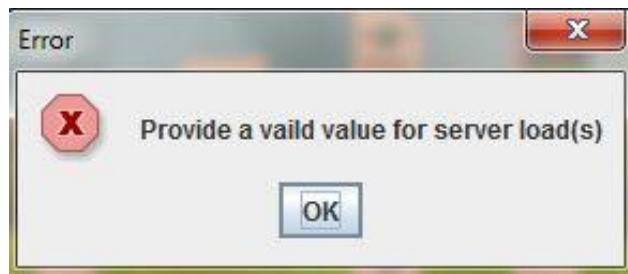


Fig 10.1.1.2 Error message for invalid server load details

10.1.1.3 Testing for invalid server count details for Tasks

Each task can have the total preferred servers within the range of total available servers value otherwise an error message would be shown as shown below



Fig 10.1.3 Error for invalid server count details

10.1.1.4 Testing for Invalid preferred servers details

Each task can have preferred servers within the range of Total available servers value if not an error message will be shown as below

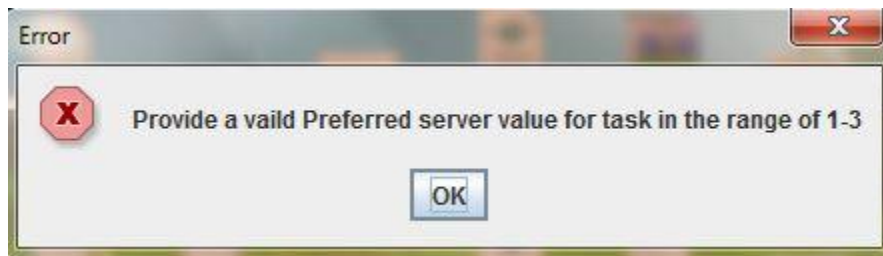


Fig 10.1.1.4 Error message for Invalid preferred servers

10.1.2 Integration Testing

The Whole project is divided into 4 modules which includes

1. Input Module
2. Balance Phase Module
3. Reduce Phase Module
4. Output Module

The input module is done by parsing an xml file using a simpleXML parser open source API and the module provides the correct details as needed by the other modules

The balance phase module is done using the Edmond-Karp algorithm which finds the augmented path and is added to the flow graph. The module is tested using various graph input and solid output is obtained.

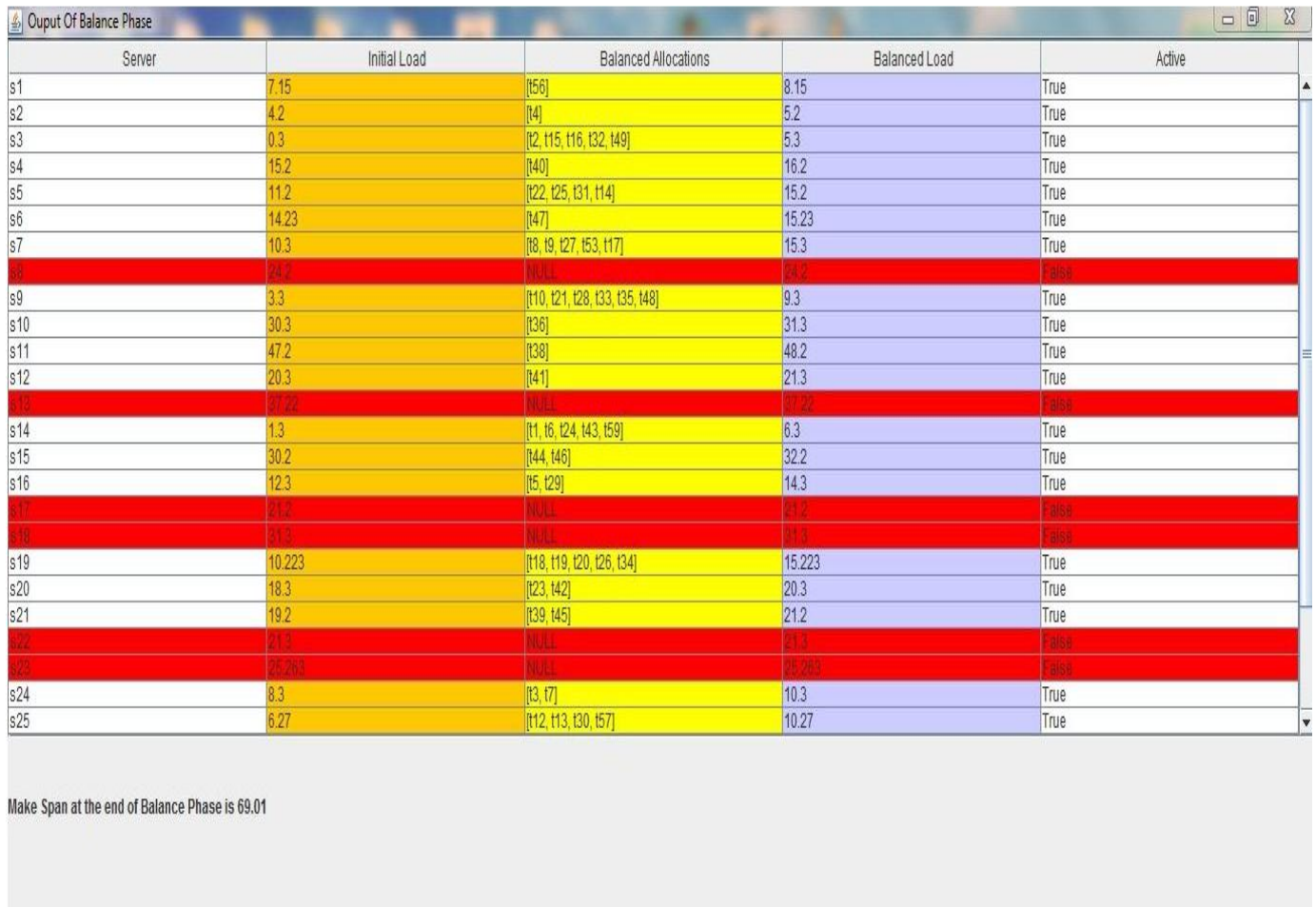
The Reduce Phase is done using the output of balance phase and the module has a heuristic function which gives the expected results as final allocations.

All the modules are tested individually and then integrated them to form the whole project. All modules work together and produce the result without any errors.

10.2 Output and Results

10.2.1 Output of Balance Phase

The below figure shows the output of balance phase which allocates tasks based on a straight forward approach based on the next available server.



Server	Initial Load	Balanced Allocations	Balanced Load	Active
s1	7.15	[t56]	8.15	True
s2	4.2	[t4]	5.2	True
s3	0.3	[t2, t15, t16, t32, t49]	5.3	True
s4	15.2	[t40]	16.2	True
s5	11.2	[t22, t25, t31, t14]	15.2	True
s6	14.23	[t47]	15.23	True
s7	10.3	[t8, t9, t27, t53, t17]	15.3	True
s8	24.2	NULL	24.2	False
s9	3.3	[t10, t21, t28, t33, t35, t48]	9.3	True
s10	30.3	[t36]	31.3	True
s11	47.2	[t38]	48.2	True
s12	20.3	[t41]	21.3	True
s13	27.22	NULL	27.22	False
s14	1.3	[t1, t6, t24, t43, t59]	6.3	True
s15	30.2	[t44, t46]	32.2	True
s16	12.3	[t5, t29]	14.3	True
s17	21.2	NULL	21.2	False
s18	31.3	NULL	31.3	False
s19	10.223	[t18, t19, t20, t26, t34]	15.223	True
s20	18.3	[t23, t42]	20.3	True
s21	19.2	[t39, t45]	21.2	True
s22	21.3	NULL	21.3	False
s23	25.263	NULL	25.263	False
s24	8.3	[t3, t7]	10.3	True
s25	6.27	[t12, t13, t30, t57]	10.27	True

Make Span at the end of Balance Phase is 69.01

Fig10.2.1 Output of Balance Phase

10.2.2 Output of Reduce Phase

The result of reduce phase is shown below figure which represents a total final reduced allocation based on data availability and server load details.

Server	Balanced Allocations	Remote Allocations	Initial Load	Final Load	Active
s1	[t56]	[t39, t53]	7.15	13.75	True
s2	[t4]	[t50, t36, t23]	4.2	13.6	True
s3	[t2, t15, t16, t32, t49]	[t38, t44, t17]	0.3	13.7	True
s4	NULL	NULL	15.2	15.2	False
s5	[t22, t25, t31]	NULL	11.2	14.2	True
s6	NULL	NULL	14.23	14.23	False
s7	[t8, t9, t27]	NULL	10.3	13.3	True
s8	NULL	NULL	24.2	24.2	False
s9	[t10, t21, t28, t33, t35, t48]	[t58]	3.3	12.1	True
s10	NULL	NULL	30.3	30.3	False
s11	NULL	NULL	47.2	47.2	False
s12	NULL	NULL	20.3	20.3	False
s13	NULL	NULL	37.22	37.22	False
s14	[t1, t6, t24, t43, t59]	[t46, t45]	1.3	11.9	True
s15	NULL	NULL	30.2	30.2	False
s16	[t5, t29]	NULL	12.3	14.3	True
s17	NULL	NULL	21.2	21.2	False
s18	NULL	NULL	31.3	31.3	False
s19	[t18, t19, t20, t26]	NULL	10.223	14.223	True
s20	NULL	NULL	18.3	18.3	False
s21	NULL	NULL	19.2	19.2	False
s22	NULL	NULL	21.3	21.3	False
s23	NULL	NULL	25.283	25.283	False
s24	[t3, t7]	[t34]	8.3	13.1	True
s25	[t12, t13, t30, t57]	[t47]	6.27	13.07	True

Make Span at the end of Reduce Phase is 14.3

Fig10.2.2 Reduce phase

10.2.3 Balance Reduce Line Chart with Makespan

The Line chart shows a pictorial representation of the load of servers at the end of each phase and makespan values at the end of each phase is also represented in the chart. As seen from the figure, the makespan value at the end of balance phase is very much greater than the reduce phase thus showing clearly the efficiency of the algorithm.

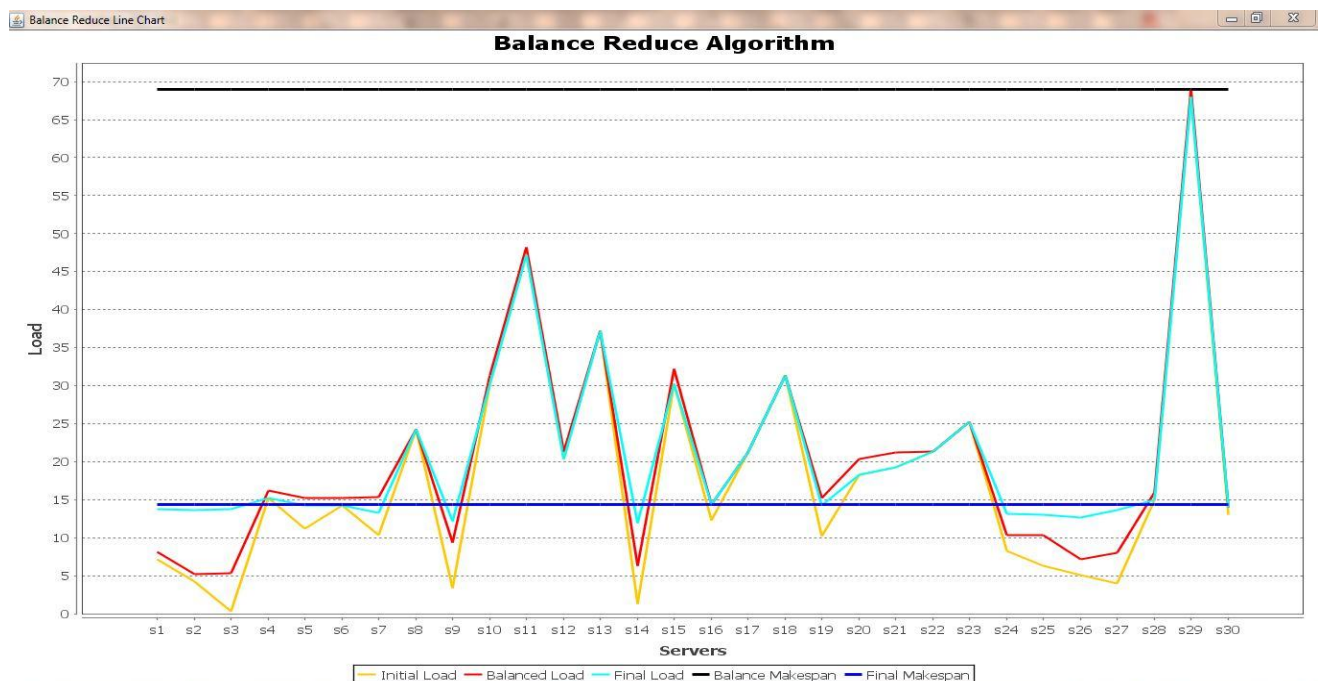
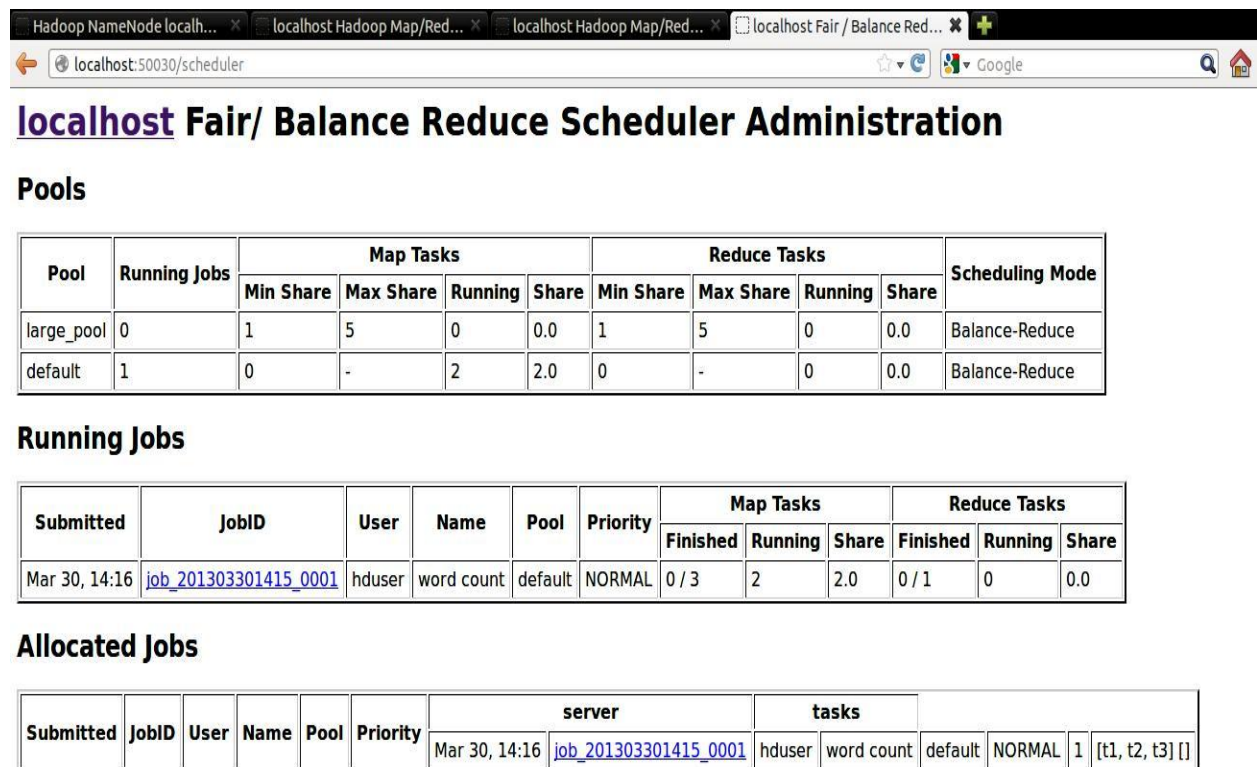


Fig 10.2.3 Balance Reduce Line Chart

10.3 Hadoop Implementation

Hadoop scheduler implementation can be obtained by accessing the scheduler parameter of hadoop via browser using the url <http://localhost:54311/scheduler>. The scheduled jobs can be viewed as shown below



The screenshot shows the Hadoop Fair Scheduler Administration interface in a web browser. The browser tabs include 'Hadoop NameNode localh...', 'localhost Hadoop Map/Red...', and 'localhost Fair / Balance Red...'. The address bar shows 'localhost:50030/scheduler'. The page title is 'localhost Fair/ Balance Reduce Scheduler Administration'.

Pools

Pool	Running Jobs	Map Tasks				Reduce Tasks				Scheduling Mode
		Min Share	Max Share	Running	Share	Min Share	Max Share	Running	Share	
large_pool	0	1	5	0	0.0	1	5	0	0.0	Balance-Reduce
default	1	0	-	2	2.0	0	-	0	0.0	Balance-Reduce

Running Jobs

Submitted	JobID	User	Name	Pool	Priority	Map Tasks			Reduce Tasks		
						Finished	Running	Share	Finished	Running	Share
Mar 30, 14:16	job_201303301415_0001	hduser	word count	default	NORMAL	0 / 3	2	2.0	0 / 1	0	0.0

Allocated Jobs

Submitted	JobID	User	Name	Pool	Priority	server		tasks				
Mar 30, 14:16	job_201303301415_0001	hduser	word count	default	NORMAL	1						

Fig 10.3 Scheduler output

11. Performance Analysis

In this section, several simulations are presented in order to investigate the effectiveness of the algorithm. For comparison, four related task scheduling algorithms are listed as follows:

11.1 MaxCover-BalAssign (MB)

This algorithm works iteratively to produce a sequence of total allocations, and then outputs the best one. Each iteration consists of two phase *maxcover* and *balassign*. Since the remote cost is unknown, it calculates the *virtual cost* which is a prediction of the remote cost. Then it computes a total allocation by taking advantage of the virtual cost.

11.2 Hadoop Default Scheduler (HDS)

When a server is idle, the scheduler chooses a data-local task, then allocates the task to the server. If there is no feasible task, then the scheduler will select a random task.

11.3 Delay Scheduling (DS)

It sets a delay threshold. If a server is idle and there is no task prefers the server, the scheduler skips the server and increases the delay counter by one. Hver, if the delay counter exceeds the delay threshold, the scheduler allocates a remote task to the server and sets the delay counter to be zero.

11.4 Good Cache Compute (GCC)

This policy is similar to DS. It sets a utility threshold which is the upper bound of the number of idle servers. The scheduler can skip servers when the idle server number is below the utility threshold. In our simulations, the utility threshold is set to $\text{TotalServerNumber} \times 90\%$.

Since HDS, GCC and DS are run-time scheduling algorithms, it is implemented them in compile-time scheme. Firstly, place all servers into a min-heap; secondly, pop a minimum load server, invoke a real-time scheduling algorithm to allocate a task, then update load of servers. The remote cost is renewed when a remote task is allocated. The second step is

repeated until all tasks are allocated. All algorithms are implemented carefully to reduce the redundant work.

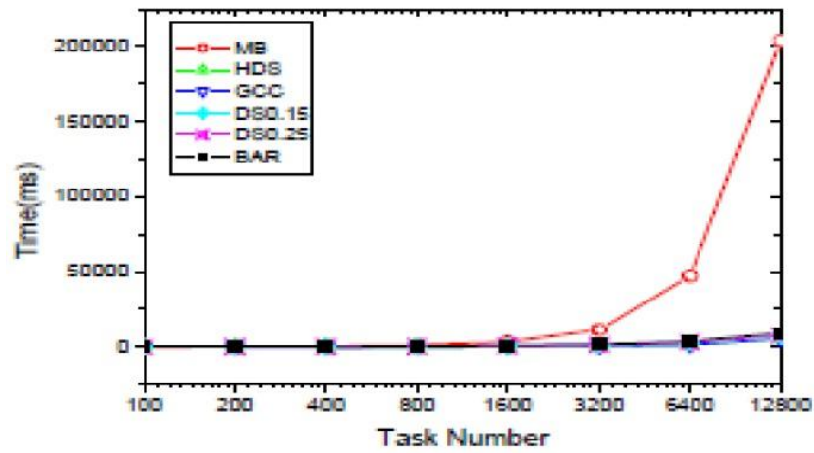


Fig 11.1 Performance analysis

12.Conclusion

The proposed algorithm yields better results under various working environments. The main domain for improving the performance of the cloud based scheduling system, namely the task scheduling algorithm should be designed in an effective way to locate the data inside the system for scheduling the job. By carefully analyzing the data locality systems inside the clusters,

- Edmond-Karps incorporated with the Balance-Reduce algorithm provided a method for balanced mapping of tasks with the servers inside a cluster.
- When large amount of servers in a cluster is idle, though the network state is good, high locality of data inside a cluster will have a negative consequence on the completion time of the job.
- The proposed modifications in the algorithm helps to adjust the data locality inside the cloud system based on the load and idle time of the servers in processing the data.

The Balanced mapping done using Edmond-Karps makes the mapping process quicker such that for scheduling the same job, it produces a better output in terms of job completion time in comparison to the existing Ford-Fulkerson algorithm. For every reduce operations in the Hadoop cluster, mapping operations provides the base. To solve a particular task inside a job, the output of mapping process is combined by the reducer, ultimately producing the needed output for the complete problem. The proposed strategy provides a way to solve various tasks of a particular job in a faster way by effectively utilizing the mapping slots inside the Hadoop cluster.

The proposed modification to the scheduler improves the overall efficiency in scheduling the tasks across the servers such that the job completion time is reduce considerably. Simultaneous reduction of tasks takes place as soon as the tasks are mapped, thereby producing a faster result under varying workloads.

13. References

- Apache Hadoop .The Apache Software Foundation. <http://hadoop.apache.org/>
- Jiahui Jin, Junzhou Luo, Aibo Song, Fang Don, Runqun Xiong, “BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing”, *11th IEEE / ACM International Symposium on Cluster, Cloud and Grid Computing*, 2011.
- B. Rao and D. L. Reddy, “Survey on improved scheduling in hadoop mapreduce in cloud environments,” *International Journal of Computer Applications*, vol. 34(9), pp. 29, 2011.
- Ford-Fulkerson Algorithm, Wikipedia the free encyclopedia.
- Edmond-Karps Algorithm, Wikipedia the free encyclopedia
- T. White, *Hadoop: The Definitive Guide*, 3rd Edition, O'Reilly Media, Inc., Jan 2012