Scheduling for Energy Efficiency with Hadoop

Sebagenzi Jason
Faculty of Information, Adventist University of Central Africa (AUCA),
Senior Lecturer in Computer Science, Dean in the Faculty of Technology, Rwanda,

ABSTRACT
The main topic of this chapter is Hadoop energy efficiency scheduling, which includes an overview of the system, scheduling techniques, and the creation and application of a Hadoop energy control system. Additionally, testing, analysis, and introduction are provided for energy-efficient scheduling for multiple users.

Keywords
Hadoop; Map Reduce; Map Reduce Slots; scheduling algorithm; energy-efficient scheduling; dynamic management

1. INTRODUCTION
The Apache Foundation created Hadoop, a distributed infrastructure framework that enables users to create distributed systems without having to first grasp the underlying details. High-speed computer clusters and storage can be completely utilized by users. The Hadoop Distributed File System (HDFS), which is implemented by Hadoop, is a distributed file system with high fault tolerance capabilities and is intended to be used with inexpensive hardware.

Because of Hadoop's fast data transfer rate, it can be used with applications that need a lot of data to run. Data can be accessible as streams thanks to HDFS's relaxation of the POSIX file system's constraints. Hadoop is a well-known open-source project with a distributed computing focus that has drawn more and more interest. Utilized by numerous major corporations including Amazon, Facebook, Yahoo!, and IBM, it is extensively employed in numerous domains like log analysis, web search, advertising computing, and data mining.

Hadoop was not designed with dynamic node management because it is a large-scale system. In a conventional Hadoop cluster, resource utilization and energy efficiency are low since Hadoop chooses the number of nodes after the system is up and running. As a result, studying Hadoop dynamic management can help it operate more effectively and efficiently. For larger Hadoop applications and better energy conservation, this enhancement would be really important.

2. RESEARCH BACKGROUND
A distributed process software platform for massive data sets is called Hadoop. It is scalable, dependable, and effective. Hadoop contains many functioning copies of data so that it can redistribute in the event of a node failure. This makes Hadoop dependable because it is predicated on the loss of compute storage and facilities.

Its parallel architecture and increased processing speed through the use of parallel computing make it efficient. And lastly, Hadoop is scalable, able to manage data at the petabyte (PB) level. Furthermore, Hadoop is reasonably inexpensive because it runs on a standard server. Of all the modules of Hadoop, the Map Reduce and HDFS modules are the most crucial. All files are stored on the storage node of the Hadoop cluster using HDFS, which is the lowest level system in the system. Task Trackers and Job Trackers make up Map Reduce, the upper layer engine on HDFS.

MapReduce's core concepts are parallel task decomposition and result fusion. HDFS is the core storage support system for distributed computing. The Map Reduce programming technique can be used to compute massive data sets in parallel, typically bigger than 1 terabyte (TB). The program's main goal is to simplify and map; it makes references to other functional programming languages and borrows some traits from vector programming languages. Map denotes mapping, while Reduce denotes simplification.

With this architecture, program function on a distributed system is facilitated, and the programmer needs little expertise in distributed programming. To guarantee that all key mappings share the same key group, the current software implementation maps a group of key-value mappings into a new set of key-value pairs by specifying a mapping function. Figure 9.1 depicts a streamlined procedure for executing Map Reduce tasks.

Slave nodes (Task Trackers) and a single master node (Job Tracker) make up the Map Reduce framework. All of a job's tasks are scheduled by the master node, which also assigns them to slave nodes and keeps track of how each task is carried out, including retrying unsuccessful attempts. However, slave nodes are only in charge of carrying out the tasks that the master node assigns them. The input data set is typically divided into many blocks by a Map Reduce job so that the Map task can process them concurrently.

Prior to moving them to the Reduce job, the Framework will sort the Map's output. The HDFS contains the input and output data. unsuccessful attempts. However, slave nodes are only in charge of carrying out the tasks that the master node assigns them. The input data set is typically divided into many blocks by a Map Reduce job so that the Map task can process them concurrently.

Prior to moving them to the Reduce job, the Framework will sort the Map's output. The HDFS contains the input and output data

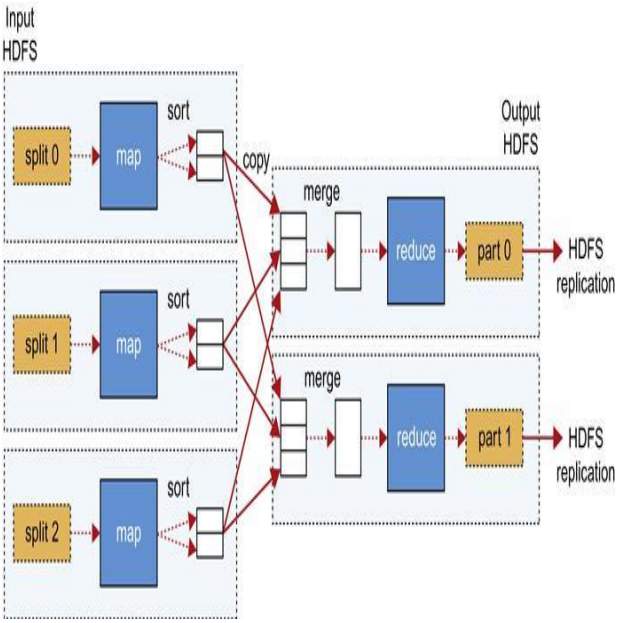

Fig.1. Map Reduce flowchart.

Highly fault tolerant, HDFS is made to run on inexpensive hardware. Large data volumes can be used with it, and it offers great throughput for accessing application data. With hundreds of server nodes on average, HDFS can automatically detect faults and recover quickly because any one of them could fail. HDFS is good to write and offers very high bandwidth data since its typical file sizes vary from gigabytes (GB) to terabytes (TB). The majority of HDFS data is read numerous times and written once.

A file doesn't need to be modified once it's written, closed, and created. Data consistency and high throughput data access issues are made easier as a result. The HDFS cluster typically consists of a Name Node and several Data Nodes, as seen in Figure 9.2's Master/Slave design. Name Node is the core server; it answers to client requests and maintains the file system namespace. The management of the data kept on a certain node is the responsibility of the Data Node. With namespace, users can store any kind of data on the HDFS file system.

A file is divided into one or more blocks for internal storage. The Name Node opens, closes, and renames files and directories that are stored in the file system namespace on a Data Node. It also establishes how the block is mapped to a particular Data Node. On HDFS, every data block is duplicated. One can adjust the quantity and dimensions of the duplicated blocks. HDFS files have a single writer and are only written once. The Data Node block status report from heartbeat signals is sent to the Name Node, which oversees the data copy operation. An inventory of every data block on a Data Node can be found in the block status report.

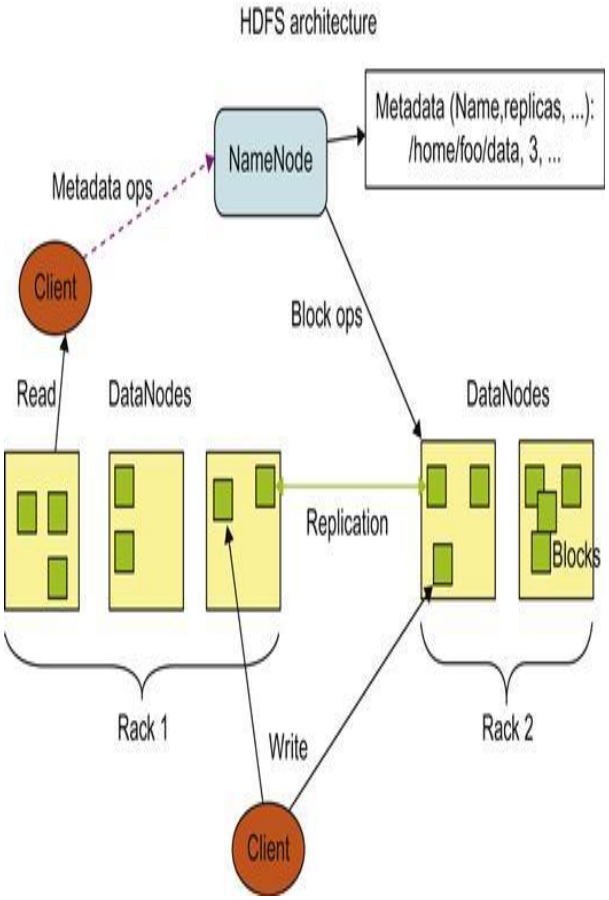

Fig.2. HDFS architecture.

3. RELATED RESEARCH WORK

The concept of "divide and conquer" underpins the entire Hadoop operating system. The "divide" phase is represented by the Map process, and the "conquer" stage by the Reduce process. Figure 9.3 depicts the full procedure.

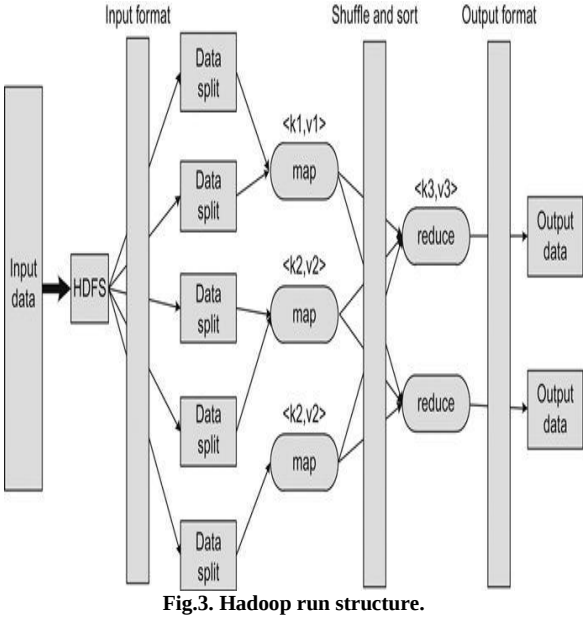

Fig.3. Hadoop run structure.

The following actions are part of the Map process: (1) read from the disk, (2) execute the Map task, then (3) write the outcomes to the disk in order of precedence.

The stages listed below are part of the Reduce process: First, sort and shuffle; next, perform the Reduce operation; and last, write the results to disk.

The output of the Map job will be sent to the output file by the practitioner class during the third stage of the Map step. The Mapper output key will be delayed in memory until it reaches a predetermined quantity of data if Combiner is supplied, rather than being written to the output right away. This portion of the data will next be combined in the Combiner and moved to the Practitioner.

Although it reduces performance, data is written to disk at this point to increase system durability. The Map key will be passed from the Hadoop framework to the Reducer during the initial Reduce phase. The HTTP protocol is used in this stage to transmit data remotely. Based on the idea that Hadoop's fault tolerance, which boosts task concurrency and speeds up response times, the Hadoop Online Prototype allows data from various tasks to interact through a pipeline in the third Map phase.

Scheduling algorithms

Dynamic management of Hadoop clusters

Already, research teams have devoted a lot of time to studying Hadoop. A Stanford team believes that there is still plenty to be done to save energy in Hadoop. For the placement of node data, they recommend utilizing a new approach [1]. U.C. Berkeley developed a model based on node, working time, and power that they claimed produced good results [2,3]. Moreover, they think that maximizing performance and energy efficiency are equally important [4]. To lower Hadoop's energy usage, Swiss scientists made modifications to the block allocation algorithm [5].

But none of these study examples could be applied in a dynamic setting because they were all focused on the allocation of static data blocks. Optimizing the heterogeneous Hadoop cluster was the main focus of Oban University researchers [6]. The Hadoop performance management and Map Reduce scheduling were also examined by researchers at the Technical University of Catalonia [7]. Using the adaptive Map Reduce scheduler to satisfy user-defined high-level performance goals while transparently and effectively utilizing hybrid systems' capabilities was the main emphasis of their research.

They discussed the modifications made to the adaptive scheduler to maximize the use of the underlying hybrid systems by enabling it to co-schedule jobs that are accelerable and nonaccelerable on the same heterogeneous Map Reduce cluster. Though energy consumption is not taken into account in the cluster, their research is more in line with the tight integration of the hardware and scheduling efficiency. Furthermore, researchers frequently employ dynamic voltage regulation to lower energy usage in [8,9]. The drawback is that it needs specialized hardware environments.

According to Intel study, there is a positive association between average usage and energy consumption inside the cluster. It is evident from the figure that energy usage rises with average utilization. In order to lower the average utilization and hence lower energy consumption, this architecture offers the ability to dynamically suspend and resume the nodes. Software is used to accomplish this; no specialized hardware is needed (Figure 9.4).

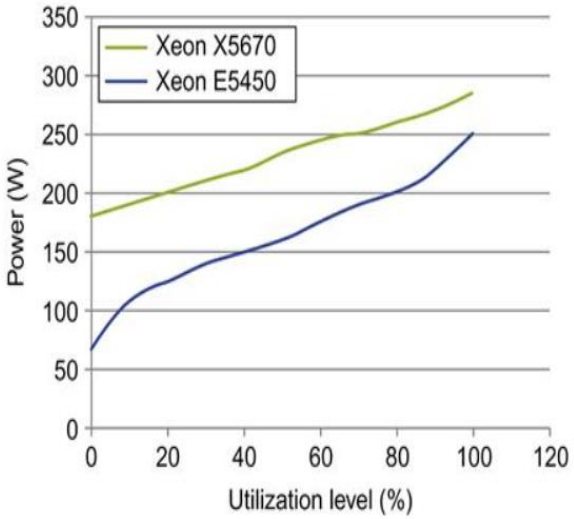

Fig.4. Power versus utilization.

This section presents Dynamic Adjusting and Negative Feedback (DANF), a novel Hadoop dynamic load balancing technique with the following features:

• Restarting and suspending the nodes in accordance with the cluster load lowers average utilization, node running time, and energy consumption.

• Increasing the stability of the cluster through feedback.

• Promote load balance, decrease load variance, and prevent jitter by using the jitter coefficient.

• It is simple to expand and use this design.

Load modeling

We need to construct a model in order to assess a node's load. According to research, a system's processor uses the most amount of energy—roughly 40% of all energy used in the system [10]. As a result, we take it into account while building our model. Another system module that uses energy is memory.

Load information

The majority of load models in use today solely account for CPU usage. We employ a vector in two dimensions with a coefficient of p. Let a node j have a load vector $L = < L_{\text{cpu}}, L_{\text{mem}} >$, We figure out its modulus, so the load is

$$L = \sqrt{p \times L_{\text{cpu}}^2 + (1-p) \times L_{\text{mem}}^2} \qquad (9.1)$$

Let's say that there are n nodes in a cluster, and the average load in it is

$$L_{\text{avg}} = \sum_{i=1}^{n} \frac{L_i}{n} \qquad (9.2)$$

Therefore, the range is (0,1), and p can be set based on various jobs; if the task requires a lot of CPU power, for example, p can be greater, around 0.8. On the other hand, we typically specify a smaller p when a task demands a high memory usage.

Period

Since this value is variable, we must choose an observation period in order to compute it. We won't be able to get the most

recent data if the timeframe is excessively long. But, an excessively short interval will cause the result to become contaminated by too many queries. Experiments indicate that the duration depends on the task. An intense computational program should therefore use a longer length of time and a data-intensive application should use a shorter period of time in order to waste less energy.

Negative feedback mechanism

A control system has numerous safeguards against abuse. Negative feedback in a mathematical model denotes a negative feedback coefficient. A fraction of the input is added as negative feedback to counteract changes in the output:

$$x' = -k \times \Delta y \tag{9.3}$$

Within the field of automatic control theory, the feedback method of root—which is based on Newton's method—is frequently employed in systems that automatically adjust, including those that automatically adjust for combustion, steam temperature, transportation, and bypass. The following is the primary formula:

$$X^{n+1} = X^n + \sqrt[3]{\frac{X^{n+1}}{A} - X^n} \tag{9.4}$$

where A has a close assignment to Xn. Assuming that we already know the beginning value of Xn in this case, we can apply Eq. (9.4) to get the DANF load in ti-1 and ti.

$$L_{\text{DANF}}^{t_i} = L_{\text{avg}}^{t_i} + \sqrt[3]{L_{\text{avg}}^{t_{i-1}} - L_{\text{avg}}^{t_i}} \tag{9.5}$$

4. PROPOSED APPROACH

Scheduling conditions

The load from Eq. (9.1) can use the default or user-defined threshold W_l and W_h. Here, W_l is the lower threshold and W_h is the upper threshold. When $W_l < L_{\text{DANF}} < W_h$, the system is in ideal status and no steps are required. When $L_{\text{DANF}} < W_l$, the system has a low load and we should suspend nodes one by one until they system is in ideal status. When $L_{\text{DANF}} > W_h$, the system has a high load and we must restart the suspended nodes.

Choosing a node to suspend

In a dynamic management system, one of the primary functions of DANF is node selection. As of the now, multiple [11]:

a. **Random**
This algorithm is straightforward and simple to comprehend. When the system hits the threshold, it chooses a node at random.

b. **Round-Robin**
Each node is assigned by this algorithm in a circular sequence, and it suspends in accordance with this order.

c. **Minimum load**
The minimum load node is chosen for suspension by this algorithm, which orders all node loads.

Upon examining these techniques, we see that if a system undergoes a major change and the suspend and restart operations take place on the same node, this could result in a considerable increase in I/O operations and negatively impact performance. We include a jitter coefficient while choosing the node in the DANF algorithm to lessen jitter.

Using Eq. (9.1) to calculate the load on each node, we add a jitter coefficient k to improve system stability:

$$k = \left| \frac{L^{t_i} - L^{t_{i-1}}}{\Delta t} \right| \tag{9.6}$$

Here, Δt is a unit time period $\Delta t = t_i - t_{i-1}$ and k has a range greater than 0; considering k, the node load is

$$L_{\text{node}}^{t_i} = k \times L^{t_i} \tag{9.7}$$

Currently, the coefficient would prevent the small load node with a substantial change from being chosen, significantly enhancing stability. The approach works well to stop repeated scheduling on the same system node.

Choosing a node to restart

We need to restart one or more nodes when the system reaches. As opposed to suspension, we eliminate the node from the queue according to rising load order by employing the principle of "first in, first out" (FIFO).

Pseudocode

DANF algorithm is provided in Algorithm 1.

```
Input: CPU and memory information of each node
Output: name and operation of a node
Initialization: allocating a name to each node
DO
calculate current load of Hadoop cluster, denote as c_l
 IF w_l< c_l<w_h
   continues;
 ELSE IF c_l<w_l
  FOR each node in active
    calculate load value of all current nodes and sort in increasing order
    sleep one or more of the lowest load nodes until w_l<c_l<w_h and
    put them in waiting queue
  END FOR
 ELSE
  open one or more nodes in the waiting queue until w_l< c_l<w_h
 ENDIF
ENDDO
```

ALGORITHM 1. DANF algorithm.

Energy control

System architecture

We used Java to implement this DANF technique. Node control modules, remote control, and resource gathering are all part of the system.

Detailed design

Resource collection

The implementation of the resource collection module involves the reading of the Linux file system procfs. In operating systems similar to UNIX, a unique file system called procfs (also known as the proc file system) displays details about

processes and other system data. As a result, we can use it to get memory and CPU information. a. Memory information

Total: The first line in /proc/meminfo;

Available: The second line in /proc/stat;

Mem=1 - Available/Total.

b. **CPU**

Total: The first line in /proc/stat;

Each CPU: The second line /proc/stat;from CPU0-CPUn;

user, nice, sys, idle: The following four column numbers;

We read these data twice, we present with "user_1 or user_2", user+sys is the used CPU.

CPU=(int)rintf(((float)((user_2+sys_2+nice_2)-(user_1+sys_1+nice_1))/(float)(total_2−total_1))*100).**Remote control**

Security SHell (SSH) is used in the implementation of the remote control module. In this case, an SSH connection can be established in Java thanks to the third-party lib Ganymed SSH-2 for Java.

a. **Create a connection using an IP**
Connection conn=new Connection(hostname);

b. **Using username and password to log in**
booleanisAuthenticated=conn.authenticateWithPassword(username,password);

c. **Begin a session and run the Linux shell**
Session sess=conn.openSession();
sess.execCommand("last");

d. **Receive the response from the console**
InputStreamstdout=new StreamGobbler(sess.getStdout());
BufferedReaderbr=new BufferedReader(new InputStreamReader(stdout));

e. **Get the status flag "0" success; "not 0" Failed**
System.out.println("ExitCode: "+sess.getExitStatus());

f. **Close cession and connection**
sess.close();
conn.close();

Node control

A shell and the Hadoop configuration file are used to implement node control. Nodes can be modified by the master to initiate and stop them. Add node:

./hadoop-daemon.sh start datanode;

./hadoop-daemon.sh start tasktracker.

Delete node;

a. **Add the following in core-site.xml in master node**

<property>

<name>dfs.hosts.exclude</name>

<value>/data/hadoop-0.20.2/conf/excludes</value>

</property>

dfs.hosts.exclude: node to be deleted

/data/hadoop-0.21.0/conf/excludes: The file and directory to be deleted.

b. **Using Java to write the node to be deleted in /data/hadoop-0.20.2/conf/excludes.**

c. **Refresh Name Node**

Hadoopdfsadmin –refreshNodes

The command can dynamic refresh dfs. hosts and dfs. hosts. Exclude without restart NameNode.

d. **Using remote SSH**

Stop datanode

./hadoop-daemon.sh stop datanode

Stop Tasktracker

./hadoop-daemon.sh stop tasktracker

Energy-efficient scheduling for multiple users

Problem formulation

[12–14] introduces a Map Reduce performance model. The model predicts how long the Map and Reduce stages will take to complete depending on the size of the input data set and the resources allotted.

Definition 1

Slots for Map Reduce. Every node in a Hadoop cluster can work on P Map and P Reduce tasks at the same time, depending on how the cluster is configured. Thus, this Hadoop cluster has $P \times P$ Map Reduce slots.

Definition 2

Waves of execution. Task assignment occurs in many rounds, referred to as execution waves, if the number of Map Reduce tasks exceeds the number of Map Reduce slots in the cluster.

Figure 9.5 shows an example executed in two waves of 20×20 Map Reduce slots.

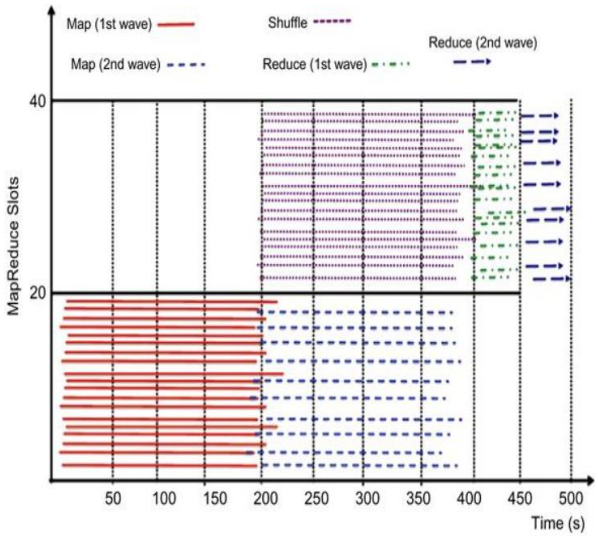

Fig.5. Execution example of TeraSort [15] in a 20×20 Map Reduce slot.

Consider a job represented as a set of *n* tasks processed by *P*×*P* Map Reduce slots (workers) in Hadoop environments. Each Map Reduce job consists of a specified number of Map and Reduce tasks. The job execution time and the specifics of the execution depend on the amount of resources (Map and Reduce slots) allocated for the job. A simple abstraction is adopted [12], where each Map Reduce job, J_i, is defined by the durations of its Map and Reduce stages, m_i and r_i, i.e., $J_i=(m_i, r_i)$. Let us consider the execution of two independent Map Reduce jobs, J_1 and J_2, in a Hadoop cluster with a FIFO scheduler.

There are no data dependencies between these jobs. Therefore, once the first job completes its Map stage and begins processing its Reduce stage, the next job can start its Map stage execution with the released Map resources in a pipelined fashion. There may be "overlap" in the executions of the Map stage of the next job and the Reduce stage of the previous one. We further consider the following problem. Let $J = \{J_1, J_2, \ldots, J_n\}$ be a set of *n* Map Reduce jobs with no data dependencies between them. Here, J_i requests $R_i \times R_i$ Map Reduce slots and has Map and Reduce phase durations (m_i, r_i), respectively.

The system scheduler can change a job's Map Reduce slots allocation depending on available resources. Let *T* be the makespan of all *n* jobs. We aim to determine an order (a schedule) of execution of jobs $J_i \in J$ such that the makespan of all jobs is minimized. Let us set the end-time of the Map stage and start-time of the Reduce stage of job J_i as , respectively. Thus, the actually allocated Map Reduce slots for job J_i are $P_i \times P_i$, the max available Map Reduce slots in the Hadoop cluster is *P*×*P*. Formally, the problem of minimizing the makespan, *T*, can therefore be formulated as

$$\text{Min } T \tag{9.8}$$

subject to

1.

$$\forall J_i, P_i \le P \tag{9.9}$$

2.

$$\forall J_i, t_r^i \le t_m^i \tag{9.10}$$

where the available capacity restriction is expressed as Equation (9.9), meaning that the number of Map Reduce slots that are actually allotted to a task (Pi) cannot exceed the total number of Map Reduce slots in the system (P). The time no overlapping restriction for the Map and Reduce stages for a single work is expressed in equation (9.10), which states that the end-time of the Map stage for a given job cannot be smaller than the start-time of the Reduce stage.

We suggest a new method to reduce the makespan of a set of given Map Reduce jobs based on the phrasing of the problem.

Revised Johnson's algorithm and HScheduler

Let's first review the traditional Johnson's method [16] and see if it can be directly applied to Map Reduce scheduling before introducing the new one.

Johnson's algorithm revisited

"There are n items which must go through one production stage or machine and then a second one," according to the original Johnson's algorithm [16]. Every stage has a single machine. A machine can only have one object on it at once. In order to modify the Map Reduce paradigm, we first describe the resources as Map Reduce slots and then use Johnson's method by treating the Map and Reduce stage resources as a single unit (similar to a single machine).

Using a similar notation to the one found in [12], let us consider a collection of *n* jobs, where each job, J_i, is represented by the pair, m_i, r_i, of Map and Reduce stage durations, respectively. Each job $J_i=(m_i, r_i)$ with an attribute D_i is defined as follows:

$$D_i = \begin{cases} (m_i, m), \text{if } \min(m_i, r_i) = m_i; \\ (r_i, r), \text{otherwise} \end{cases} \tag{9.11}$$

The first argument in D_i is called the stage duration and denoted as D_i^1. The second argument is called the stage type (Map or Reduce) and denoted as D_i^2. Notice that when r_i=0, Johnson's algorithm reduces to the shortest process time first algorithm, which is known to be optimal for minimizing total finish (flow) time of all jobs. Algorithm 2 presents the pseudocode of the Revised Johnson's algorithm for Map Reduce. First, it sorts all *n* jobs from the original set *J* in the ordered list *L* in such a way that job J_i precedes job J_{i+1} if and only if $\min(m_i, r_i) \le \min(m_{i+1}, r_{i+1})$. It finds the smallest value among all durations, if the stage type in D_i is *m* (i.e., it represents the Map stage), then the job J_i is placed at the head of the schedule. Otherwise, J_i is placed at the tail. Then, the allocated job is removed and other jobs are considered in the same fashion. The complexity of Johnson's algorithm is dominated by the sorting operation and thus is $O(n \log n)$.

Theorem 1

For a two-stage production system, Johnson's algorithm determines the theoretical bottom bound of total elapsed time (makespan) when every work moves through the identical two stages and uses every resource available.

Input: All Jobs' Map and Reduce durations, number of machines for the Hadoop cluster
Output: Scheduled jobs, makespan
1 List the Map and Reduce durations in two vertical columns (implemented in a list);
2 **for** *all entries* **do**
3 Find the shortest one among all durations;
4 In case of ties, for the sake of simplicity order the item with the smallest subscript first. In case of a tie between Map and Reduce, order the item according to the Map;
5 IF it is for the Map, place the corresponding item at the first place;
6 ELSE it is for the Reduce, place the corresponding item at the last place ;
7 Remove both time durations for that task;
8 Repeat the steps on the remaining set of items;
9 **end**
10 Compute makespan;

ALGORITHM 2 Revised Johnson's algorithm for Map Reduce.

Johnson [16] offers a thorough proof of Theorem 1. The optimal makespan, or theoretical lower bound, can be found using Johnson's technique in the following ways:
1. Taking into account that the Map and Reduce stages have n tasks. For a given Hadoop cluster of P machines (slots), let denote the work time of the ith task for the Map phase and denote the corresponding time for the Reduce phase.

2. Then, the optimal total elapsed time (makespan) is as follows:

$$T = \sum_{i=1}^{n} r_i + \max_{u=1}^{n} K_u \quad (9.12)$$

$$K_u = \sum_{i=1}^{u} m_i - \sum_{i=1}^{u-1} r_i \quad (9.13)$$

Observation 1

The assumptions of the traditional Johnson's algorithm for a two-stage production system and Map Reduce job processing match exactly if each job uses all Map or all Reduce slots during processing. Next, the theoretical bottom bound of lowering the makespan of all Map Reduce jobs can be found by utilizing Johnson's technique.

Example 1

We replicate the five Map Reduce jobs provided in [12] in Figures 9.6(a) and (b), correspondingly. Figure 9.6A displays the lengths of the Map and Reduce stages for every job, while Figure 9.6B use Johnson's technique to construct an ordered list of the five jobs. The ideal sequence, as per Johnson's approach, is δ=(2, 5, 1, 3, 4). Eq. (9.13) may be used to get the overall delay time for this sequence, which equals 4 units. Using Eqs. (9.12) and (9.13), the total elapsed time (makespan) is 47 units. The worst-case outcome, or 78 units, can be achieved as the top bound if the jobs are rearranged in sequence.

(A)

	m_i	r_i	D_i
J_1	1	4	(1, *m*)
J_2	2	3	(2, *m*)
J_3	4	5	(4, *m*)
J_4	30	4	(4, *r*)
J_5	6	30	(6, *m*)

(B)

	m_i	r_i	D_i
J_2	1	4	(1, *m*)
J_5	2	3	(2, *m*)
J_1	4	5	(4, *m*)
J_3	30	4	(4, *r*)
J_4	6	30	(6, *m*)

Fig.6. Five Map Reduce jobs examples by one cluster. (A) Before applying the algorithm; (B) After applying the algorithm.

Observation 2

In the case of numerous Map Reduce tasks, the actual makespan should additionally account for extra process timings, such as job setting up, migration, and dispatch, in addition to the Map and Reduce stages. Based on Eqs. (9.12) and (9.13), the actual makespan is adjusted to $\hat{T} = (1 + c_0)T$, where c_0 is a weight factor that depends on the job types.

We will validate Observation 2 in the performance evaluation section.

Observation 3

The length of the job stage is directly correlated with the quantity of resources allotted (Map and Reduce slots). The look of the jobs can be altered if the system scheduler allots more or fewer Map Reduce slots than are necessary.

Example 2

Examine Scenario 2 in [12]: Using Example 1, let jobs J1, J2, and J5 consist of 30 Map and 30 Reduce tasks, and jobs J3 and J4 consist of 20 Map and 20 Reduce tasks, with all other configurations remaining unchanged from Example 1. We replicate the findings in Figure 9.7, which shows how these five Map Reduce jobs are executed based on the created Johnson's schedule, Θ=(J2, J5, J1, J4, J3). Even though the system has 30×30 Map Reduce slots available, [12]

We develop a new method called HScheduler to effectively plan MapReduce processes to reduce makespan, based on Observations 1-3 and Claim 1. The MapReduce HScheduler algorithm's pseudocode is shown in Algorithm 3. First, by recalculating the jobs' real durations based on available slots, it assigns all available MapReduce slots to a specified set of jobs. By executing execution waves in varying

amounts depending on available slots, this modifies their Map and Reduce durations. It then scheduled all of the revised jobs

using the Revised Johnson's method. All available MapReduce slots to a specified set of jobs. By executing execution waves in varying amounts depending on available slots, this modifies Already, research teams have devoted a lot of time to studying Hadoop. A Stanford team believes that there is still plenty to be done to save energy in Hadoop. For the placement of node data, they recommend utilizing a new approach [1]. U.C. Berkeley developed a model based on node, working time, and power that they claimed produced good results [2,3]. Moreover, they think that maximizing performance and energy efficiency are equally important [4]. To lower Hadoop's energy usage, Swiss scientists made modifications to the block allocation algorithm [5].

But none of these study examples could be applied in a dynamic setting because they were all focused on the allocation of static data blocks. Optimizing the heterogeneous Hadoop cluster was the main focus of Oban University researchers [6]. The Hadoop performance management and Map Reduce scheduling were also examined by researchers at the Technical University of Catalonia [7]. Using the adaptive Map Reduce scheduler to satisfy user-defined high-level performance goals while transparently and effectively utilizing hybrid systems' capabilities was the main emphasis of their research.

They discussed the modifications made to the adaptive scheduler to maximize the use of the underlying hybrid systems by enabling it to co-schedule jobs that are accelerable and nonaccelerable on the same heterogeneous Map Reduce cluster. Though energy consumption is not taken into account in the cluster, their research is more in line with the tight integration of the hardware and scheduling efficiency. Furthermore, researchers frequently employ dynamic voltage regulation to lower energy usage in [8,9]. The drawback is that it needs specialized hardware environments.

According to Intel study, there is a positive association between average usage and energy consumption inside the cluster. It is evident from the figure that energy usage rises with average utilization. In order to lower the average utilization and hence lower energy consumption, this architecture offers the ability to dynamically suspend and resume the nodes. Software is used to accomplish this; no specialized hardware is needed (Figure 9.4).

Their Map and Reduce durations. It then scheduled all of the revised jobs using the Revised Johnson's method.
expects that jobs J3 and J4 only need 20×20 Map Reduce slots.

amounts depending on available slots, this modifies their Map and Reduce durations. It then scheduled all of the revised jobs using the Revised Johnson's method. All available MapReduce slots to a specified set of jobs. By executing execution waves in varying amounts depending on available slots, this modifies their Map and Reduce durations. It then scheduled all of the revised jobs using the Revised Johnson's method.
expects that jobs J3 and J4 only need 20×20 Map Reduce slots.

Nevertheless, the outcome differs significantly from [12] if we permit any task to utilize all of the system's Map Reduce slots during execution. This may be done simply in Hadoop (e.g., by dividing the big input files according to the number of Map Reduce slots that are available). In Scenario 2 [12], jobs J3 and J4 are now able to utilize every 30x30 Map Reduce slot that is available for the same example. In this case, J3 and J4 will have map and reduce durations of and, respectively, and (4, 20), and, respectivelyCompared to using only 20x20 Map Reduce slots, both are shorter. Thus, the makespan will be, as Figure 9.8 illustrates, where X1=1. Compared to the outcome of the two pools technique, this result is 12% less [12]. The fundamental tactic for our outcomes is thus the following idea.

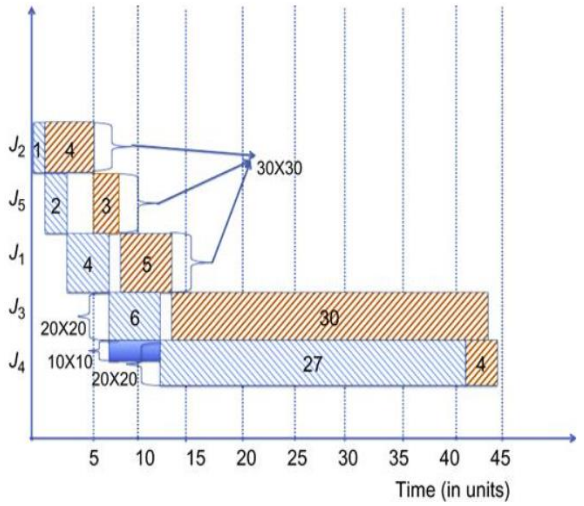

Fig.7. Five Map Reduce jobs executions in one cluster.

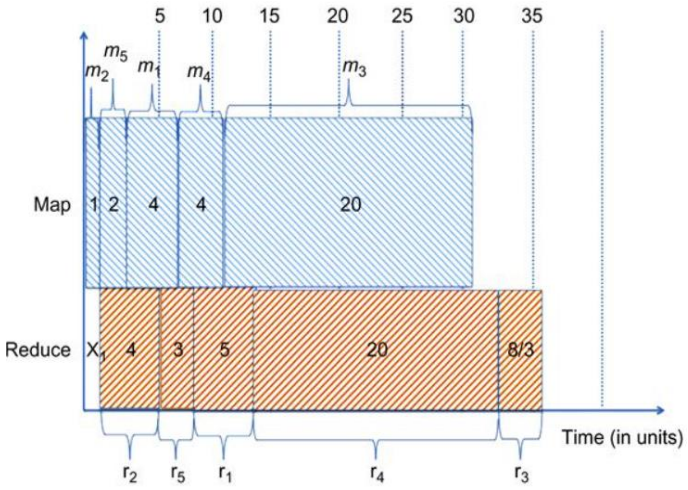

Fig.8. New result of five MapReduce jobs execution.

Claim 1

If a job requests more or less Map Reduce slots than the system can accommodate, the system scheduler may decide to allocate a higher or smaller number of Map Reduce slots for the work.

Assuming that there are $P \times P$ MapReduce slots in the given Hadoop cluster, then there are two Jobs *A* and *B* and each has requested MapReduce slots, *R*, and time duration, T_A, T_B, respectively. Note their theoretical makespan, T_1, can be easily computed using Eqs. (9.12) and (9.13) directly. Then, the actual makespan of job *A* and *B* using all available slots (*P*) is (T_A):

$$T_A = \frac{R}{P} T_1 \qquad (9.14)$$

We develop a new method called HScheduler to effectively plan MapReduce processes to reduce makespan, based on Observations 1-3 and Claim 1. The MapReduce HScheduler

algorithm's pseudocode is shown in Algorithm 3. First, by recalculating the jobs' real durations based on available slots, it assigns all available MapReduce slots to a specified set of jobs. By executing execution waves in varying amounts depending on available slots, this modifies their Map and Reduce durations. It then scheduled all of the revised jobs using the Revised Johnson's method.

The complexity of HScheduler is dominated by Johnson's algorithm and thus is $O(n \log n)$.

Input: All Jobs' Map and Reduce durations, number of machines for the Hadoop cluster
Output: Scheduled jobs, makespan
1 Compute the Map and Reduce durations of all jobs by their required slots and Equations (11–14);
2 **for** all jobs **do**
3 | IF a job's required slots $R_i \geq P$ (total available slots), THEN allocates all available slots to it and adds more execution waves based on tasks' splitting;
4 | ELSE IF $R_i < P$, allocates all available slots to it and records actual execution waves;
5 | End;
6 **end**
7 Call Revised Johnson's Algorithm;
8 Compute makespan;

ALGORITHM 3 HScheduler.

Performance evaluation

Evaluation platform

We construct a 16-node Hadoop cluster with 512 M RAM and a dual-core Pentium CPU on each node. Installed on every node was CentOS 6.3 and Hadoop 0.21.

5. REPORT OF THE PAPER

a. **Energy control system**

We use TeraSort as an example of an intensive calculation task and Hadoop WordCount as an example of a huge memory task. While TeraSort data are produced by TeraGen, WordCount data are sourced from Wikipedia. The data sizes for these two projects are 1G, 2G, 4G, and 8G in addition to 500 megabytes (M). We choose a threshold of 0.2 for the lower threshold and 0.8 for the higher threshold, and we set the time period to 10 s. Whereas p is set to 0.8 in TeraSort, it is set to 0.5 in WordCount. We note the mean result after five repetitions of each test.

b. **Energy-efficient scheduling**

For our trials, we employ workloads that are comparable to those in [12]: According to research done on the Yahoo! M45 cluster, this workload reflects a variety of MapReduce tasks [12]. The lengths of the Map and Reduce phases are derived from actual data from WordCount [17] and TeraSort [15], whereas the number of Map and Reduce tasks is produced using a normal distribution.

• Unimodal: the workload is measured using a single scale factor for the total workload, meaning that each job's scale factor is uniformly drawn from [1,10] and a Normal distribution with parameters round(N(154, 558)0.1) for the number of Map tasks and round(N(19,145)0.1) for the number of Reduce tasks. This is tested using a set of 50 WordCount [17] (with mean Map duration 65 s and mean Reduce duration 57 s, uniformly distributed) and 50 TeraSort jobs [15] (with mean Map duration 73 s and Reduce duration 58 s, uniformly distributed).

• Bimodal: 20 TeraSort tasks from [15] (with mean Map length 287 s and Reduce duration 306 s, uniformly distributed) and a subset of 20 WordCount from [17] (with mean Map duration 448 s and mean Reduce duration 413 s, uniformly distributed) are situated, respectively. In this instance, the remaining jobs (20%) are scaled using [8,10] and a Normal distribution with parameter round(N(154, 558)_0.3) for the number of Map tasks and round(N(19,145)_0.3) for the number of Reduce tasks. Eighty percent of the jobs are scaled using a factor evenly distributed between [1,2]. This simulates workloads with a high percentage of short jobs and a low percentage of long jobs.

All results are obtained by the average of six runs.

6. RESULTS ANALYSIS

Energy control system

a. **Load balance test**

We can compare the load balance in order to assess the selected node algorithm. The variance is computed using the following formula in this case.

$$\sigma^2 = (L_{avg} - L)^2 \qquad (9.15)$$

Figures 9.9 and 9.10 show the WordCount and TeraSort results, respectively, for the Random, Round-Robin, Minimum load, and DANF tests.

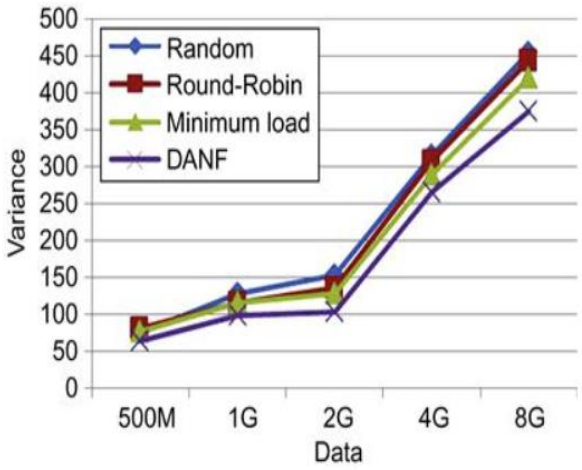

Fig.9. WordCount variance comparisons.

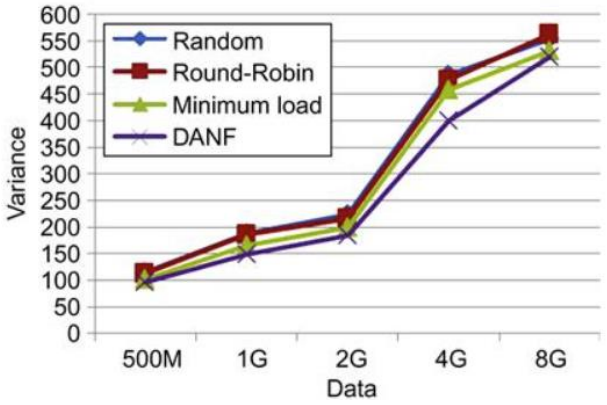

Fig.10.TeraSort variance comparisons.

It is evident from the result that TeraSort has a higher variance than WordCount. Our technique is more efficient and has a better load balance than other algorithms since the DANF has a smaller variance for both results when compared to others [18].

b. **Energy test**

$$E = P(u) \times T_{all} = [P_{min} + (P_{max} - P_{min}) \times u] \times T_{all} \quad (9.16)$$

Here, P_{min} is the system's idle energy consumption, P_{max} is the system's full load energy consumption, u is the average utilization in T_{all}, and T_{all} is the system's boot time. Further, T_s is the node working time and T_r is the node idle time. For $T_{all} = T_r + T_s$, the total node idle time is the sum of the idle time for all nodes (Tables 9.1–9.3).

Table 9.1 WordCount total working time

	500 MB	1G	2G	4G	8G
Without dynamic Management systems	111	153	250	430	836

Table.2 WordCount idle nodes

	500 MB	1G	2G	4G	8G
Idle nodes	1	1	2	3	4

Table.3 WordCount total idle time

	500 MB	1 G	2 G	4 G	8G
Without DANF dynamic management system (s)	32	96	208	448	928
With DANF dynamic management system (s)	40	120	250	570	1140

Fig.11. is the comparison (Tables 9.4–9.6).

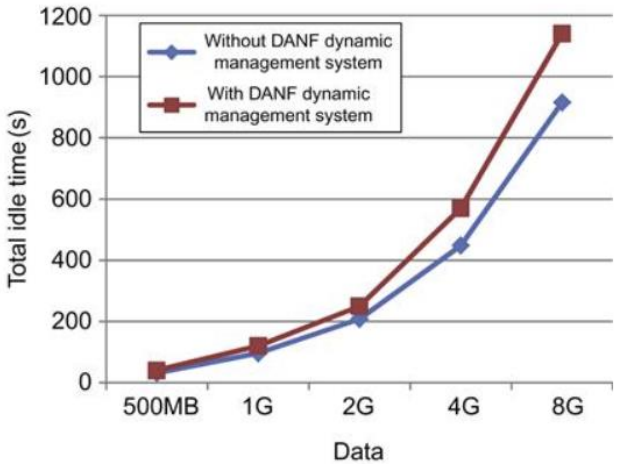

Fig.12. WordCount total idle time comparison.

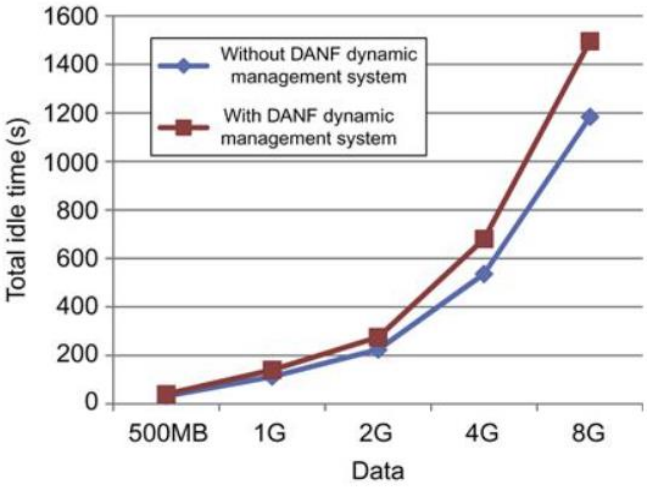

Fig.13. TeraSort total idle time comparison.

We can see that DANF lengthens the idle time in both experiments from Figures 9.11 and 9.12. Tests in TeraSort and WordCount clearly demonstrate the benefit of DANF.

In Eq. (9.14), u is the average utilization of system boot time, we use Ganglia to record: (Figures 9.13 and 9.14).

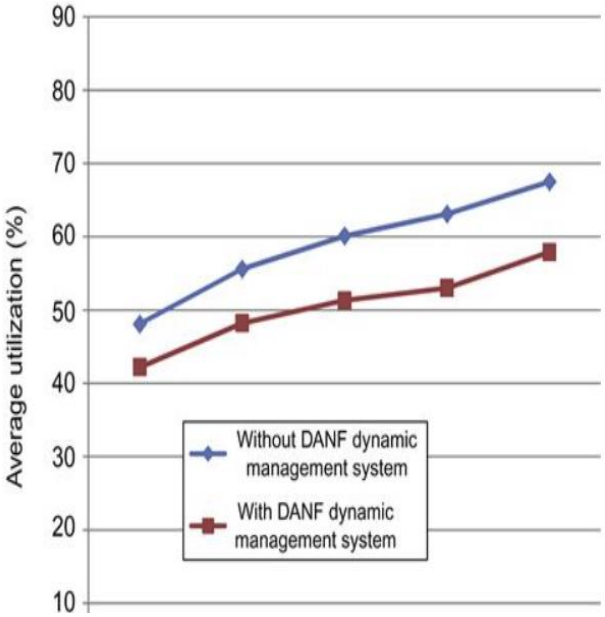

Fig.15. WordCount average utilization comparison.

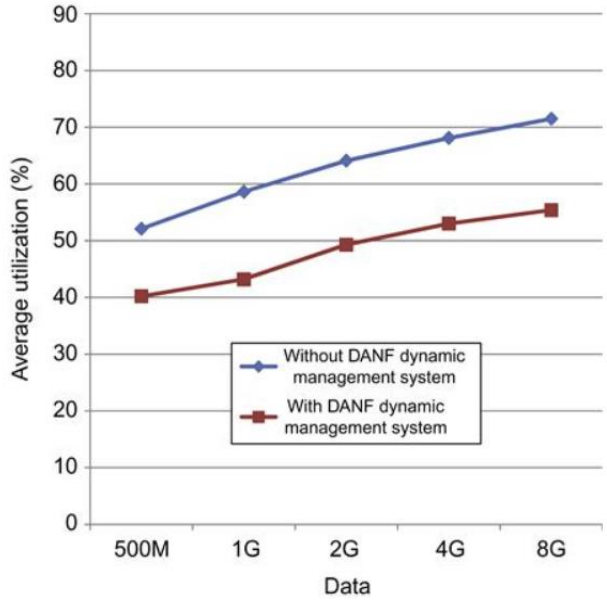

Fig.14. TeraSort average utilization comparison.

Since Tall is the same in Eq. (9.14), Pmin and Pmax are constants. As a result, there is a proportionality between average use and energy consumption. Pmin = 50 W and Pmax = 300 W were measured in our experimental setup. The data that compare energy use are data 9.15 and 9.16.

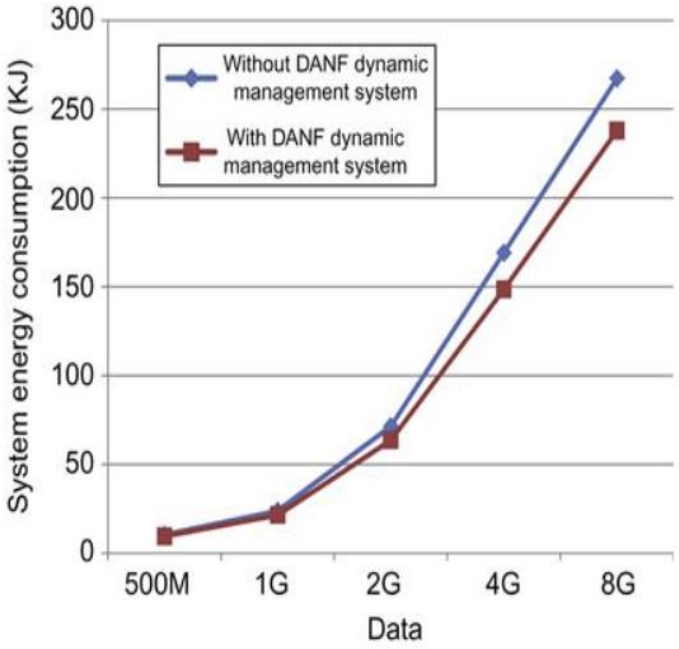

Fig.16. WordCount system energy consumption comparisons.

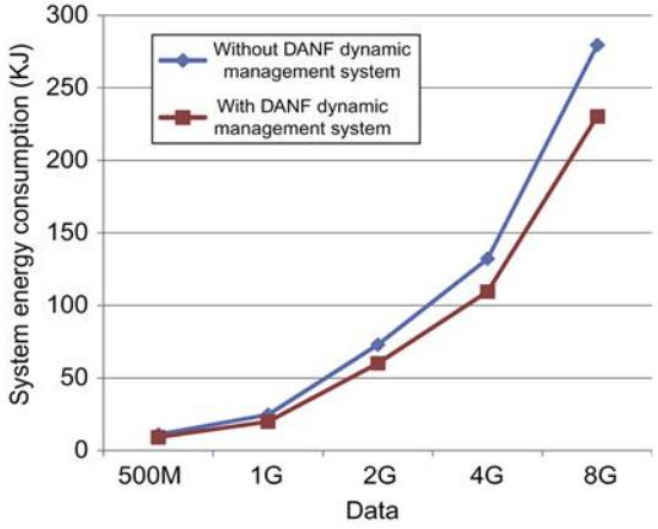

Fig.17. TeraSort system energy consumption comparisons.

As can be seen from the figure, the system's energy usage was lower with DANF than it was with the previous setup. The average energy decrease for all DANF test cases is 14%.

Energy-efficient scheduling

We compare the following algorithms:

• Random Order (Rand): using the work IDs as a guide, this algorithm schedules each job in a different order.

• Johnson's Algorithm in Reversed Order (R_Johnson): this algorithm schedules all jobs in the Revised Johnson's algorithm's reverse order. It is demonstrated in [16] to be the worst case scenario in terms of makespan of all tasks.

• HScheduler: this is our proposed algorithm.

• The traditional Johnson's algorithm, Johnson T, only takes into account the extra process time resulting from job setup, dispatch, migration, and other factors in a real Hadoop cluster. As such, it only functions as a theoretical lower bound.

• An additional strategy to reduce makespan was suggested in [12]: balanced pools (BP). To reduce the makespan, it divides the Hadoop cluster into two balanced pools and then assigns each job to the best pool.

For the BalancedPools algorithm, two pools with 12 and 24 MapReduce slots each are set, and 18 data nodes with two MapReduce slots each are set for all tests.

An analysis of four algorithms' makespan comparison is shown in Figures 9.17 and 9.18. The results of Johnson's algorithm represent the theoretical lower bounds, whereas R_Johnson represents the worst case scenario and serves as the upper limitation of makespan. Compared to HScheduler, Rand and R_Johnson have larger makespans. On average, HScheduler has a makespan that is 8–10% shorter than BP. In Unimodel and Bimodel, the average difference between HScheduler and the theoretical lower bound is 15% and 13%, respectively.

This is due to the fact that in an actual Hadoop context, HScheduler has extra process times, such as job setup, dispatch, and migration. We tested 50 TeraSort data (mean Map length 73 s and Reduce duration 58 s, uniformly distributed) without taking Bimodel or Unimodel into account, as shown in Figure 9.19. Similar results from comprehensive genuine experiments are observed; but, due to page limits, they are not included.

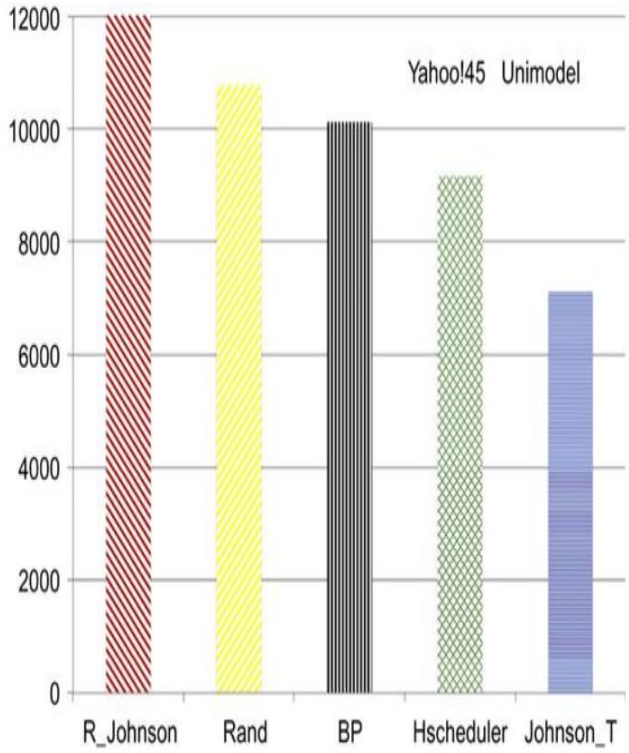

Fig.18. Comparison of makespan in Unimodel (in seconds).

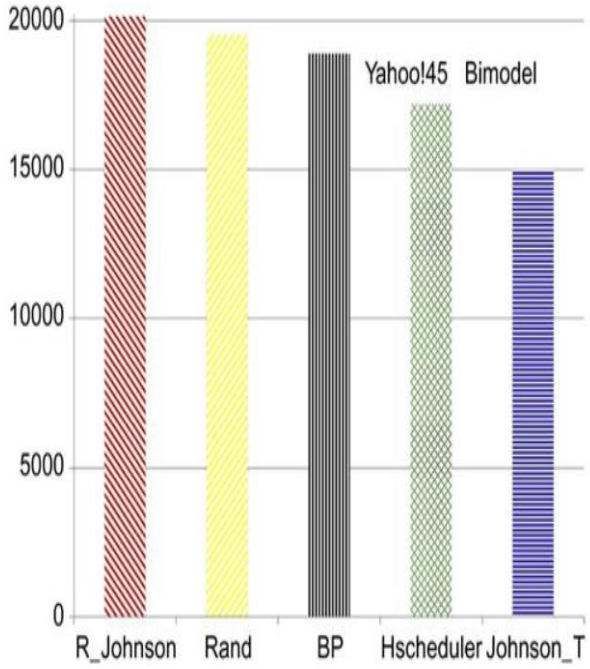

Fig.19. Comparison of makespan in Bimodel (in seconds).

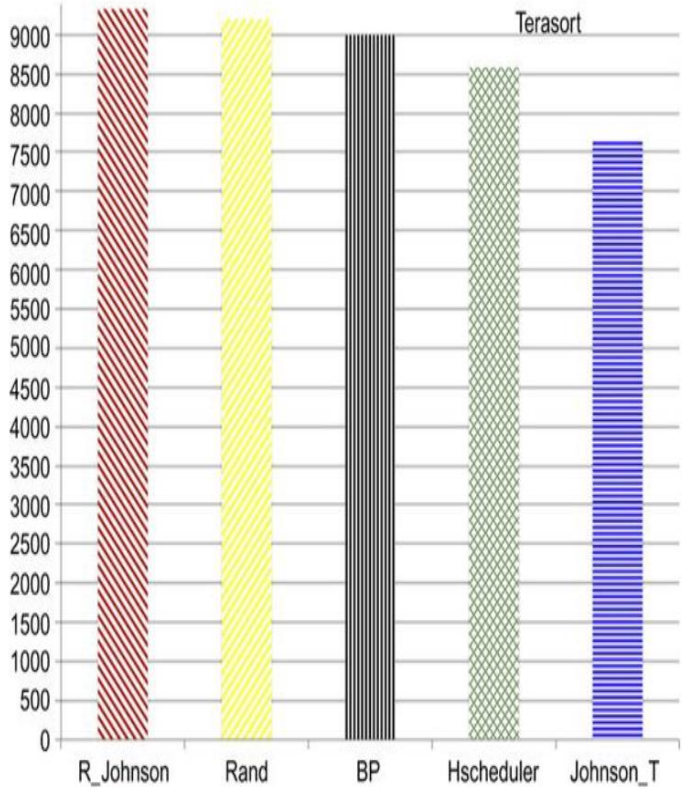

Fig.20. Comparison of makespan (in seconds) of 50 TeraSort jobs.

7. CONCLUSION

This chapter presented prior Hadoop research and covered the fundamental ideas and design of Hadoop, the MapReduce process, and the HDFS file system. The fundamental strategy of DANF, which combines Hadoop's energy efficiency and load balancing dynamically, was put into practice and tested. In

addition, an energy-efficient scheduler was presented and contrasted with a number of current methods. It was discovered that these two techniques lower energy usage in addition to increasing Hadoop cluster efficiency.

8. ACKNWLEDGEMENT

This paper is focused on how to manage energy efficiency by using Hadoop in Big data Analytics. My gratitude to the researchers who contributed to this paper.

9. REFERENCES

[1] Leverich J, Kozyrakis C. On the energy (in)efficiency of Hadoop clusters. *SIGOPS Oper Syst Rev*. 2010;44(1):61–65.

[2] Chen Y, Ganapathi AS, Fox A, Katz RH, Patterson DA. *Statistical workloads for energy efficient MapReduce : Technical Report* Berkeley: UCB/EECS; 2010.

[3] Chen Y, Keys L, Katz. RH. *Towards energy efficient MapReduce: Technical Report* Berkeley: UCB/EECS; 2009.

[4] Nedevschi S, Popa L, Iannaccone G, et al. *Reducing network energy consumption via rate-adaption and sleeping: Technical Report* Berkeley: UCB/EECS; 2007.

[5] Polo J, Carrera D, Becerra Y, Beltran V, Torres J, Ayguad E. Performance management of accelerated MapReduce workloads in heterogeneous clusters. ICPP2010 2010:653–62.

[6] Xie J, Yin S, Ruan X, Ding Z, Tian Y, Majors J, et al. Improving MapReduce performance through data placement in heterogeneous Hadoop clusters. IPDPSW 2010:1–9.

[7] Polo J, Carrera D, Becerra Y, Beltran V, Torres J, Ayguadé E. *Performance management of accelerated MapReduce workloads in heterogeneous clusters*. *Proceedings of the ICPP* San Diego, CA: IEEE Press; 2010; p. 653–62.

[8] Kim KH, Buyya R, Kim J. Power aware scheduling of bag-of-tasks applications with deadline constraints on DVS-enabled clusters. *CCGRID*. 2007;85(10):541–548.

[9] Lee YC, Zomaya AY. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. *CCGRID*. 2009;9:92–99.

[10] Cooper BF, Sillberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB, SoCC'10 2010;10:143–54.

[11] Bryhni H, Klovning E, Kurc O. A comparison of load balancing techniques for scalable web server. *IEEE Newt*. 2000;7/8:58–63.

[12] Verma A, Cherkasova L, Campbell RH. Orchestrating an ensemble of MapReduce jobs for minimizing their makespan. *IEEE Trans Dependable Sec Comput*. 2013; April [online version].

[13] Verma A, Cherkasova L, Campbell RH. *Two sides of a coin: optimizing the schedule of MapReduce jobs to minimize their makespan and improve cluster performance*. p. 11–18 *MASCOTS* Washington, DC: IEEE Computer Society; 2012.

[14] Verma A, Cherkasova L, Campbell RH. ARIA: automatic resource inference and allocation for MapReduce environments. In: Proc. of ICAC; 2011.

[15] <http://sortbenchmark.org/YahooHadoop.pdf>.

[16] Johnson S. Optimal two-and three-stage production schedules with setup times included. *Naval Res Log Quart*. 1954;1(1):61–68.

[17] 17.WordCount,<http://www.cs.cornell.edu/home/llee/data/simple/>.

[18] Beloglazov A, Abawajy J, Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Comput Syst*. 2012;28(5):755–7