ISSN 1392–124X INFORMATION TECHNOLOGY AND CONTROL, 2015 Vol. 41, No. 1, 1–0

Performance and Energy Optimization of the Terasort Algorithm by Task Self-Resizing

Jie Song[1], **Shu Xu**[1], **Li Zhang**[1], **Claus Pahl**[3], **Ge Yu**[2]

Software College[1]*, School of Information Science and Engineering*[2]*, Northeastern University,*
No.11, Lane 3, WenHua Road, HePing District, Shenyang, Liaoning, China
Irish Centre for Cloud Computing and Commerce IC4, Dublin City University, Dublin 9, Ireland[3]
e-mail: songjie@mail.neu.edu.cn

Abstract. In applications of MapReduce, Terasort is one of the most successful ones, which has helped Hadoop to win the Sort Benchmark three times. While Terasort is known for its sorting speed on big data, its performance and energy consumption still can be optimized. We have analyzed the characteristics of Terasort and have identified the existence of idle nodes, which does not only waste energy but also loses performance. Therefore, we optimize Terasort through a single-task distributed algorithm and a task self-resizing algorithm to save time and reduce the energy that is consumed by map nodes and reduce nodes, which is caused by task schedule and waiting for input data. The algorithm proposed in this paper has proved to be effective in optimizing performance and energy consumption through a series of experiments. It can also be adapted to other applications in the MapReduce environment.

Keywords: Big Data; Energy Consumption; MapReduce; Terasort; Task Resizing.

1. Introduction

Two important indicators of algorithm quality are time complexity and space complexity. Nowadays, energy consumption has become a third important indicator because of the non-negligible energy consumed by IT equipment [1]. In 2010, energy consumption of data centers has accounted for 1.3% of the world's total electricity consumption, and is expected to increase in the next five years by 56% [2]. Consequently, the storage, time and energy consumption of algorithms should be properly analyzed and estimated.

Sorting is an algorithm that puts elements of a list in a certain order [3]. Sorting is the most widely used type of algorithm across various applications. For example, the web correlation sorting algorithm of search engines can decide whether users could find the desired information in the page [4]. Meanwhile, sorting is a common task for program designers, and the choice of the sorting algorithm will directly affect the time, storage and energy consumption of applications.

In this paper, we study on the performance and energy optimization of Terasort [5]. Terasort as a sorting algorithm is an important application of MapReduce and has become a contributor to big data applications. In 1998, Jim Gray created the Sort Benchmark, which defines a large data set with 100 byte data records. Sorting algorithms are evaluated on the time they use to sort the data and write to disk completely [6]. In 2008, Terasort won the first prize of the 1TB sort benchmark, taking 209 seconds, nearly 90 seconds faster than the previous year's record holder. In 2009, with a cluster of 1460 nodes, Terasort won again by sorting 1TB data in 62 seconds [7]. In 2013, Terasort won again with a cluster of 2100 nodes, sorting 1.42TB of data in one minute [6].

Terasort is successful in sorting big data, but we still believe that its performance and energy consumption can be optimized because in Terasort the performance is decreased and energy is wasted while some nodes are idle. In MapReduce, there are two main reasons for idleness. Firstly, in MapReduce, there are two kinds of tasks: map tasks (*mappers* for short) and reduce tasks (*reducers* for short). Although Terasort has optimized the data transfer between *mappers* and *reducers* to guarantee the parallelism of *reducers*, *reducers* do not start until all mappers are complete, which causes the idleness of nodes while *reducers* wait for *mappers*. Secondly, the mapper size could be reduced to ensure that all *mappers* are executed synchronously and *reducers* are started as simultaneously as possible. However, this will add more costs to task scheduling, which means sometimes nodes are idle because they are waiting for scheduled *mappers*.

In this paper, we propose that one and only mapper with a proper size is scheduled at a node to ensure that nodes will not wait for new tasks, and that a mapper can change the size of processing data dynamically to adjust its size by itself to ensure that all *mappers* complete simultaneously and, consequently, *reducers* do not wait on *mappers* any more. We focus on three aspects. Firstly, a mapper is allocated to a node

once with a proper initial size by the single-task distribution algorithm. Secondly, a mapper is given priority on processing; the first is local data, then in-rack data and remote data. Thirdly, a self-resizing algorithm adjusts the size of the *mappers*, which are represented by the number of data blocks that should be processed, to ensure the parallelism of *mappers*. There are some challenges to implement these approaches. Firstly, we need to define abstractions to model the task and its data, the parallelism and the execution cost. Secondly, there are a number of combinations of the initial task distribution. Thus, we need to provide a method to find the best one efficiently. Thirdly, we need to design an efficient self-resizing algorithm that is self-motivated and decentralized to ensure the parallelism among nodes and that aims to reduce the resizing cost. Finally, we need to ensure fault tolerance if there is only one mapper for a node, because fault tolerance cannot be guaranteed by task replication and recovery. Optimization algorithms such as Genetic Algorithms (GA), Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO) are suitable to solve this problem, but they are too expensive to be adopted in real-time task distribution and self-resizing.

The remainder of this paper is organized as follows. Section 2 introduces MapReduce and Terasort and analyzes the disadvantages of Terasort as our motivation. Section 3 proposes single-task distribution and self-resizing algorithms to improve the performance and decrease the energy consumption of Terasort, describes the task model and algorithms, and then gives a detailed example. In Section 4, the effectiveness of the proposed algorithms is proven by experiments. Section 5 introduces related work. In Section 6, conclusions and future work are summarized.

2. Background and Motivation

In this section, firstly, we introduce the structure and execution process of the MapReduce framework. Secondly, we describe the core of Terasort and emphasize its two technical difficulties, which make Terasort different from other sorting algorithms. Finally, we analyze the disadvantages of Terasort and put forward the basic ideas and steps of our proposed algorithms.

2.1. MapReduce

In MapReduce, the computation is moved closer to where the data are located. A MapReduce job is divided into tasks, map tasks (*mappers*) and reduce tasks (*reducers*). All *mappers* and *reducers* are performed on nodes in parallel. Both input and output of mappers and reducers are key-value pairs. $< Key_M^{In}, Value_M^{In} >$ and $< Key_M^{Out}, Value_M^{Out} >$ are the input and output format of *mappers*, respectively; $< Key_R^{In}, Value_R^{In} >$ and $< Key_R^{Out}, Value_R^{Out} >$ are the input and output format of *reducers*, respectively. Key_M^{Out} should be implicitly converted to Key_R^{In}, and $Value_R^{In}$ is the collection of $Value_M^{Out}$. *Reducers* would not start until the slowest *mapper* is complete, and the job is accomplished when the slowest *reducer* has completed. The parallelism is decreased if *mappers* (*reducers*) do not finish synchronously.

MapReduce splits the input data set into M subsets (where M is larger than the number of nodes), and each subset is the input of one *mapper*. The *mapper* reads each record of the subset, processes it and outputs $< Key_M^{Out}, Value_M^{Out} >$ pairs as the intermediate result. A "split" function partitions the intermediate records into R disjoint buckets by applying a hash function to Key_M^{Out} of each output record. Each bucket is written to the local disk where the *mapper* is executed. The input dataset is *mapped* to $M \times R$ intermediate files when the M *mappers* terminate. All $< Key_M^{Out}, Value_M^{Out} >$ pairs with the same value of Key_M^{Out}, let this be the j-th bucket, are stored in file F_{ij} ($1 \leqslant i \leqslant M$, $1 \leqslant j \leqslant R$).

The second phase of a MapReduce job executes R *reducers*, where R is typically the number of nodes. The input for each reducer R_j consists of the files F_{ij} ($1 \leqslant i \leqslant M$). These files are transferred over the network from the *mappers*' local disks. Note that again all output records from the maps with the same hash value are consumed by the same *reducer*, regardless of which *mapper* produced the data. Each *reducer* processes or combines the records assigned to it in some way, and then writes records to an output file (in the distributed file system), which forms part of the computation's final output.

2.2. Terasort

Terasort is an application of the MapReduce programming model based on the Hadoop platform. The core of Terasort is its map stage. As shown in Figure 1, each mapper divides the data into R data blocks, where all data in the i-th ($i > 0$) block are larger those that in the $(i+1)$-th block. At the reduce stage, the i-th *reducer* processes (sorts) all *mapper*'s i-th blocks. Thus, the results of the i-th *reducer* will be larger than those of the $(i+1)$-th one. Finally, it outputs all of the *reducers*' results sequentially, which are the sorting results [5]. As shown in Figure 1, we assume that there is a data set whose range of value is distributed from 1 to 100 on average. If there are 5 *reducers* in the MapReduce cluster, then the range of *Partition*0 will be [1, 20], and so on. After all data have been parti-

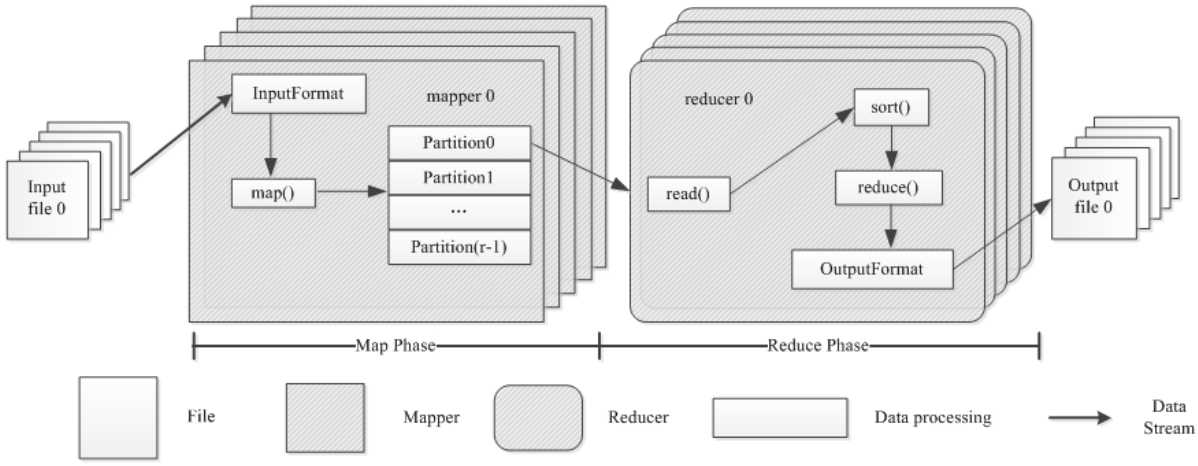

Figure 1. Structure of Terasort algorithm

tioned, the data in a *Partition* will be sent to a specific *reducer* whose number is the same as the corresponding *Partition*. Finally, the outputs of *reducers* are the results sequentially, i.e., the range of *output file* 0 is [1, 20], while the range of *output file* 4 is [80-100]. Note that some detailed features, which are not the emphasis of Terasort, are abbreviated in this paper.

Terasort has two technical difficulties. Firstly, it adopts a sampling technique to determine the ranges of R blocks. Data sampling is performed before *mappers* start. It samples data from the input data first, then sorts the data, and divides them into R blocks. After finding the upper and lower limits of each block (break-point), these break-points are stored in a distributed cache. Secondly, Terasort adopts tries to quickly determine which block a record belongs to. A 'trie' [8,9], also known as digital tree or radix tree or prefix tree, is an ordered tree data structure that is used to store a dynamic set or an associative array. Tries treat a record as a sequence of characters, and unlike a binary search tree, no node in a trie stores the key associated with that node. Instead, its position in the tree defines the key with which it is associated. All descendants of a node have a common prefix of the sequence associated with that node, and the root is associated with the empty sequence. In tries, the time of finding a sequence does not depend on the number of the tree's nodes, but rather the length of the sequence [5].

First, a Terasort *mapper* reads the break-points of each block from the distributed cache and builds a trie accordingly. Leaves of a trie stand for *reducers*. When data are processed, for each record, the *reducer* it belongs to can be determined by querying the trie. For example, if break-points are "abc", "dab" and "ddd" for 4 blocks (*reducers*), the trie in Terasort is built as

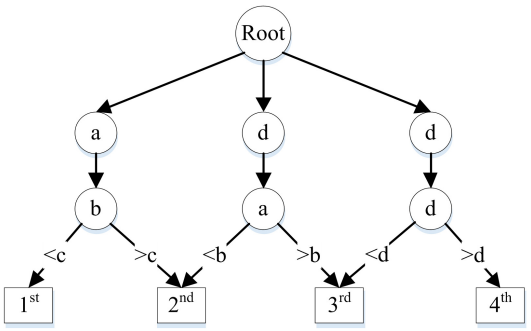

Figure 2. Sample trie in Terasort

shown in Figure 2. For a string "daz", z>b, according to the trie, it belongs to the 3rd block.

2.3. Motivation

In Terasort, only when all i-th blocks of the *mappers* have been processed, the i-th reducer can start sorting. Therefore, if there is one *mapper* executing more quickly than others, the *reducers* will still have to wait for the other *mappers*, which wastes energy and decreases performance. To address this problem, we can reduce the waiting time by decreasing the size of the split (*mapper*). Assuming that the split size is 64MB, in the worst case all *mappers* are waiting for the processing of 64MB data. In some special applications with simple maps or less data, small splits may improve parallelism, but will lead to more scheduling costs. We can take 1TB input data as an example. If the split size is 64MB, the total scheduling time is 16384, and suppose a single scheduling cost unit is 0.01 seconds, the total time waiting for the task is about 164 seconds. Assuming that the idle power consumption of a computer is 40 watt, then there is

an energy waste of 6560 joule, which should not be ignored.

We also noticed that the time complexity of a trie query is O(*M*) (where *M* is the length of the sequence), which means that the cost of processing each record in a *mapper* is a constant related to the performance of node. In Terasort, every record in the *mapper* should be processed, so the cost of a *mapper* is linear to its data size If a *mapper* can adjust the processing data size according to its current processing speed and the unprocessed data, it may guarantee the synchronization of *mappers* without additional centralized scheduling cost. Furthermore, data also no longer have to be divided into many splits (*mappers*), which can also save task scheduling time. Based on this idea, the task distribution and self-resizing algorithms in the paper are proposed as follows:

Step 1: Each node is assigned one and only *mapper*, and data are assigned to *mappers* fairly by single task distribution algorithm.

Step 2: All *mappers* process their assigned data until one mapper is complete.

Step 3: The fastest *mapper* resizes its assigned data block itself by snatching data from an unfinished *mapper*. The speed of processing local data is faster than that of processing remote data, which has to be considered as well.

Step 4: Step 3 will be repeated until all the mappers have completed.

The above approach avoids negative effects on performance and energy consumption introduced by scheduling, and ensures the parallelism among *mappers* and minimizes waiting time for *reducers*, i.e., this approach optimizes Terasort in both performance and energy consumption.

3. Schedule Model and Algorithm

In this paper, Terasort is optimized by minimizing the time of "nodes waiting for tasks" and "*reducer* waiting for inputs". The first issue is addressed by single-task distribution, while the second is addressed by task self-resizing. In a traditional MapReduce cluster, a job is submitted into the job queue first and next divided into a number of tasks, which are then distributed to nodes according to a certain algorithm. When a task is complete in a slave node, the master node will distribute a new task and repeat the cycle until the job is complete. Each task has a fixed size and the distribution algorithm can guarantee that the task is distributed to the node which is closest to the data, while the parallelism of the *mapper* is ensured by assigning a smaller task size (64MB as default). Fault-tolerance is supported through a replication mechanism. Single-task distribution can minimize the scheduling cost, but it is difficult to determine the task size and ensure task replication due to the unbalanced data distribution and heterogeneous nodes. The parallelism of *mappers* does not improve either if they only process the pre-assigned data. Thus, we adopt the task self-resizing mechanism to solve these problems. The variables used in this section are given in Table 1.

Table 1. Description of variables

Iterms	Description
p	Number of racks
q	Number of nodes in the cluster, used for simplifying the description. We assume that all racks have the same number of nodes
Δ	Parallelism
n_{ij}	Node j in rack i
ϵ_{ij}	Performance constant of node n_{ij}
m_{ij}	Map task on n_{ij}, each node executes only one task
t_{ij}	Execution time of map task m_{ij}
S^k	The k-th $p \times q$ task matrix, which is the mapping plan between data blocks and tasks
L_{ij}^k	Number of local data blocks of m_{ij} in the k-th task matrix
$L_r(i,j)$	Number of local data blocks of m_{ij} in the first task matrix, equal to $L_{i,j}^1$
G_{ij}^k	Number of in-rack data blocks of m_{ij} in the k-th task matrix
$G_r(i,j)$	Number of in-rack data blocks of m_{ij} in the first task matrix, equal to $G_{i,j}^1$
R_{ij}^k	Number of remote data blocks of m_{ij} in the k-th task matrix
$R_r(i,j)$	Number of remote data blocks of m_{ij} in the first task matrix, equal to $R_{i,j}^1$
V_i	The mean of block size in node of the i-rack originally
V	The mean of block size of nodes among racks
$\alpha : \beta : \gamma$	I/O speed ratio of accessing local, in-rack, and remote data

3.1. Model

Firstly, we define the task model and the targets of resizing. Then we define the algorithms for single-task distribution and task self-resizing. Note that in this section the task mentioned refers to the map task.

Definition 1. Data Block: A data block is a partition of the processed data in a map task. The size of a task is represented as the number of its blocks. Let m_{ij} be a map task on the j-th node of the i-th rack, and L_{ij}, G_{ij} and R_{ij} be the number of local, in-rack (regional) and remote data blocks it contains, respectively.

Definition 2. Task Matrix: The task matrix $S = [s_{ij}]_{p\times q}$ represents the distribution of a job's data block in the cluster. Each element in task matrix s_{ij} represents the data size of the task on the j-th ($1 \leqslant j < q$) node of the i-th ($1 \leqslant i < p$) rack. The data size is represented by the number of local, in-rack and remote data blocks, denoted by $s_{ij} =< L_{ij}, G_{ij}, R_{ij} >$. s_{ij} is null if there is no such node in that rack.

S^0 is the initial task matrix in which all $G_{ij} = R_{ij} = 0$ and tasks process their local data only. Take matrix (1) as an example. There are 4 racks and 13 nodes in the cluster and each task only processes its local data. The element s_{11} means that there are 5 data blocks on the first data node in the first rack. The fourth row means that there are 2 nodes in the fourth rack.

$$s^0 = \begin{bmatrix} <5,0,0> & <6,0,0> & <7,0,0> & <8,0,0> \\ <3,0,0> & <4,0,0> & <5,0,0> & null \\ <5,0,0> & <5,0,0> & <5,0,0> & <5,0,0> \\ <2,0,0> & <3,0,0> & null & null \end{bmatrix} \quad (1)$$

The task matrix should be implemented as a jagged array because racks contain different numbers of nodes. To simplify the model and algorithm description, we assume that there are q racks and p nodes in each rack. However, our approach is also suitable for a jagged task matrix.

Definition 3. *Parallelism*: The parallelism Δ of the map phase is evaluated as the total difference between the maximum execution time of the task and that of the others, meaning how long the other tasks are waiting for the slowest one. This can also represent the wasted energy. Let m_{pq} be the slowest task, then parallelism is defined as in equation (2), with $\Delta \in (0, 1]$.

$$\Delta^{-1} = 1 + \sum_{i=1,j=1}^{p,q} (t_{pq} - t_{ij}) \quad (2)$$

Definition 4. *Single-Task Distribution*: Single-task distribution is a process of dividing a job into tasks and distributing tasks to the nodes after the job is submitted. One node has one and only one task, and all data blocks are assigned to tasks to ensure the maximum parallelism.

If we execute Terasort with S^0, the parallelism of the *mappers* is related to the data placement on nodes. A balanced data distribution across nodes will lead to a higher parallelism, while the worst situation is that the slowest node processes the maximum data and other nodes remain idle until it has completed. In big data environments, it is difficult to ensure parallelism through the pre-defined and balanced data placement. In single-task distribution we assume that the cluster is homogeneous. Thus, the performance of the map task is only related to the number of data blocks it processes and whether these blocks are local, in-rack or remote data blocks.

The single-task distribution algorithm transforms S^0 to S^1 on the assumption that the performance of each node is equal. It is easy to prove that if $\forall s_{ij}$ and $s_{ab} \in S, \alpha L_{ij}+\beta G_{ij}+\gamma R_{ij} = \alpha L_{ab}+\beta G_{ab}+\gamma R_{ab}$, then parallelism is maximal. However, the performance of each node varies and is unknown before it is executed. During the execution, to improve the parallelism, S^1 will be transformed to S^2, S^3 and S^k through the task self-resizing algorithm. The definition of self-resizing is given below.

Definition 5. Task self-resizing: Task self-resizing is an algorithm invoked by a task when it completes earlier than others, in order to adjust its size dynamically according to the current task matrix. The task size is presented as the size and location of the data that the task should process.

Task self-resizing is an effective algorithm to maximize the parallelism among *mappers*. Each time the task self-resizing algorithm is invoked, S^k will be transformed to S^{k+1} correspondingly ($k = 0, 1, 2, 3...$). In every transformation, $L^k_{ij}, G^k_{ij}, R^k_{ij}$ are adjusted, and based on S^k, S^{k+1} satisfies $\Delta^k < \Delta^{k+1}$ and $\sum_{i=1,j=1}^{p,q}(L^{k+1}_{ij}+G^{k+1}_{ij}+R^{k+1}_{ij}) = \sum_{i=1,j=1}^{p,q}(L^k_{ij}+G^k_{ij}+R^k_{ij})$.

The goal of self-resizing is to make the execution time of each task approximately equal. Therefore, we need to consider the regularity of the execution time of a task. It depends on the node's performance, the number of data blocks, the blocks' location and the task (algorithm) complexity. In addition, the node's performance is a constant and the data block size and location are adjusted by the self-resizing algorithm. The task complexity varies because of varying functionalities. The complexity of a *mapper* may change with the characteristics of input data, which results in more difficulty to define the execution-time function of a task. Fortunately, because of the specifics of Terasort, there is only one query of a trie in a map task, and the time complexity of this query depends only on the height of the tree, which is constant. Consequently, the time complexity O(*M*) is unchanged for any data, and we can define the execution time t_{ij} of task m_{ij} as

$$t_{ij} = \varepsilon_{ij} \times (\alpha L_{ij} + \beta G_{ij} + \gamma R_{ij}) \quad (3)$$

Equation (3) is an expression of the time consumption of task m_{ij} running on n_{ij}, which is the j-th node in the i-th rack. $\alpha : \beta : \gamma$ denotes the I/O speed ratio of accessing local, in-rack or remote data. ε_{ij} is a performance constant defined by multiplying node performance and task complexity. Equation (4) defines t_i as the execution time of tasks in the i-th rack and *t* as the execution time of all tasks.

$$t_i = \max_{j-1}^{q} t_{ij} \qquad t = \max_{i-1}^{p} t_i \tag{4}$$

In Terasort, whether it is multiple-task distribution or single-task distribution, the results of the *mappers* are unchanged. Similarly, the task self-resizing does not cause the output errors. According to equation (4), the time consumption of the map phase is equal to the time consumption of the slowest mapper, so we can accelerate the slowest task and optimize both time and energy consumption of Terasort in the map phase. Moreover, in a MapReduce cluster, due to hardware and network problems, a task failure is treated as a normal phenomenon. When a task fails, single-task distribution causes the loss of the corresponding results, requiring the task's re-distribution. If the size of new task and the aborted one are the same, then in the most cases, the new task would not catch up with the other tasks because of its delay. Therefore, there is no guarantee of fault tolerance because the execution time of the map phase depends on the execution time of the last aborted and restarted task. As this is time-consuming, task resizing needs to be adapted to task failures.

The resizing algorithm supports the fault tolerant mechanism in a low-cost way. In general, fault tolerance can be implemented by recovery or replication mechanisms. A traditional Terasort adopts the replication mechanism provided by MapReduce framework. It starts a larger number of redundant tasks and if there is one task completed, its replicates are abandoned. The replication mechanism is simple, but it wastes resources and affects performance. The MapReduce framework cannot adopt the recovery mechanism because the mechanism needs set up checkpoints for each task and replicate the necessary intermediate results as a backup. For the traditional Terasort, it is equally difficult. On the one hand, there are too many tasks to afford setting a check-point one by one; on the other hand, it is also costly to remotely backup the task's intermediate outputs which are stored locally. However, the single-task distribution approach makes the recovery mechanism possible. In Terasort, the *mapper* then adds to each data a tag with its domain and sends it to the corresponding reducer. The number of reducers is certain, so the *mapper* does not write intermediate results to the local disk, but to the remote disk of the reducer directly [5]. This technology has been presented in [10]. Based on this, we do not need to backup intermediate results, and what we need to record are only the data blocks which have been processed. When a task fails, some appropriate adjustments in task resizing are made to other running tasks, rather than starting a new task. The resizing algorithm is described in detail in the next section.

3.2. Algorithms

From the previous section we know that there is one and only one map task on each node. The size of a map task is the number of data blocks it should process, where the data blocks could be local, in-rack or remote. The goal of the single-task distribution and task self-resizing algorithms is maximizing the parallelism in the map phase by changing the task size itself before it is complete.

The single-task distribution algorithm calculates S^1 according to S^0 by re-distributing data blocks to each task fairly. The basic idea is as follows: firstly, we adjust the data blocks of tasks rack-by-rack to make sure that tasks in the same rack are completed at the same time, and then we carry out the same operation across the racks by treating racks as tasks (nodes) in the same rack.

A row of the task matrix stands for tasks running in the same rack. $\forall i \in [1, p]$, we consider tasks from m^0_{i1} to m^0_{iq}. They are sorted according to their size increasingly. Without loss of generality, let m^0_{i1} contain the fewest data blocks and m^0_{iq} contains the most data blocks, i.e., $L^0_{i1} = \min(L^0_{i1}, L^0_{i2}...L^0_{iq}), L^0_{iq} = \max(L^0_{i1}, L^0_{i2}...L^0_{iq})$ and $V_i = \frac{1}{q}\sum_{j=1}^{q} L^0_{ij}$ If there are k tasks whose blocks are fewer than those of V_i, then m^0_{i1} to m^0_{ik} can be treated as a virtual task m^0_{iw}, and the others can be treated as a virtual task m^0_{iv} which has L^0_{iw} and L^0_{iv} local data blocks respectively, i.e., $L^0_{iw} = \sum_{j=1}^{k} L^0_{ij}, L^0_{iv} = \sum_{j=k+1}^{q} L^0_{ij}$ Thus, the task distribution problem in the i-th rack can be changed to "how many data blocks of m^0_{iv} should be moved to m^0_{iw} to ensure two virtual tasks can be completed simultaneously". Let m^0_{iw} process x data blocks of m^0_{iv}. Then, we obtain equation (5):

$$\begin{cases} L^1_{iw} = L^0_{iw}, G^1_{iw} = x \\ L^1_{iv} = L^0_{iv} - x, G^1_{iv} = 0 \\ \alpha L^1_{iw} + \beta G^1_{iw} = \alpha L^1_{iv} + \beta G^1_{iv} \end{cases} \Rightarrow x = \frac{\alpha\left(L^0_{iv} - L^0_{iw}\right)}{\alpha + \beta}$$

$$\therefore \begin{cases} L^1_{iw} = L^0_{iw}, G^1_{iw} = \dfrac{\alpha\left(L^0_{iv} - L^0_{iw}\right)}{\alpha + \beta} \\ L^1_{iv} = L^0_{iv} - \dfrac{\alpha\left(L^0_{iv} - L^0_{iw}\right)}{\alpha + \beta}, G^1_{iv} = 0 \end{cases} \tag{5}$$

To ensure the parallelism of inner tasks in virtual tasks m^0_{iw} and m^0_{iv}, $\forall g \in [1,k]$ m^0_{ig} gets $\frac{V_i - L^0_{ig}}{\sum_{j=1}^{k}(V_i - L^0_{ij})} \times x$ from m^0_{iv} proportionally. $\forall u \in [k+1,q]$ m^0_{iu} provides $\frac{L^0_{iu} - V_i}{\sum_{j=k+1}^{q}(L^0_{ij} - V_i)} \times x$ data blocks to m^0_{iw}. Regularly, the task order for receiving blocks is from m^0_{i1} to m^0_{ik}, and the task order for providing blocks is from m^0_{iq} to m^0_{ik+1}. Thus, it is ensured that m^0_{i1} gets more blocks, m^0_{iq} gives more blocks, which is consistent with the actual situation. To summarize:

$$
\begin{aligned}
&L^1_{ig} = Lr(i,g)\\
&Lr(i,g) = \begin{cases} L^0_{ig} & (g \leqslant k) \\ L^0_{ig} - \dfrac{L^0_{ig} - V_i}{\sum\limits_{j=k+1}^{q}\left(L^0_{ij} - V_i\right)} \times x & (g > k) \end{cases}\\
&G^1_{ig} = Gr(i,g) = \begin{cases} \dfrac{V_i - L^0_{ig}}{\sum\limits_{j=1}^{k}\left(V_i - L^0_{ij}\right)} \times x & (g \leqslant k) \\ 0 & (g > k) \end{cases}\\
&R^1_{ig} = Rr(i,g) = 0
\end{aligned}
\tag{6}
$$

Through equation (6), the parallelism of tasks in each rack is ensured, but the workload among racks is still unbalanced. Here, we treat tasks in the same rack as a virtual task, and then the problem of "parallelism of racks" is changed to the problem of "parallelism of tasks in the same rack", so that the operations defined in equation (6) are also suitable for these virtual tasks.

The rack that has fewer local data blocks in S^0 is assigned some local data blocks from other racks as remote data blocks. Let $V = \frac{1}{p}\sum_{i=1,j=1}^{p,q} Lr(i,j)$ be the average number of data blocks in a rack. Here, we still assume that there are c racks whose total data blocks are less than V. Then, these racks can be treated as a virtual task m^0_w and the other racks be treated as a virtual task m^0_v. They have L^0_w and L^0_v local data blocks, respectively, $L^0_w = \sum_{k=1,j=1}^{c,q} Lr(k,j), L^0_v = \sum_{k=c+1,j=1}^{p,q} Lr(k,j)$. By the same approach, we can calculate how many data blocks are assigned to a rack as remote blocks, and how many data blocks a task in the rack provide proportionally, and how many remote data blocks a task in the rack receive proportionally. The deduction is abbreviated here, and for $\forall h \in [1,p], \forall g \in [1,q]$, S^1 is given as in equation (7):

$$
\begin{aligned}
&L^0_w = \sum_{k=1,j=1}^{c,q} Lr(k,j), L^0_v = \sum_{k=c+1,j=1}^{p,q} Lr(k,j)\\
&G^0_w = \sum_{k=1,j=1}^{c,q} Gr(k,j), G^0_v = \sum_{k=c+1,j=1}^{p,q} Gr(k,j)\\
&x = \frac{\alpha \times \left(L^0_v - L^0_w\right) + \beta \times \left(G^0_v - G^0_w\right)}{\alpha + \gamma}\\
&L^1_{hg} = \\
&\quad \begin{cases} Lr(h,g) & (h \leqslant c) \\ Lr(h,g) - \dfrac{(Lr(h,g) - V) \times x}{\sum\limits_{k=c+1}^{p}\left(\sum\limits_{j=1}^{q} Lr(k,j) - V\right)} & (h > c) \end{cases}\\
&G^1_{hg} = Gr(h,g)\\
&R^1_{hg} = \begin{cases} \dfrac{V - Lr(h,g)}{\sum\limits_{k=1}^{c}\left(V - \sum\limits_{j=1}^{q} Lr(k,j)\right)} \times x & (h \leqslant c) \\ 0 & (h > c) \end{cases}
\end{aligned}
\tag{7}
$$

Single-task distribution transforms S^0 to S^1, and then S^1 is distributed to each node and referred to by the self-resizing algorithm. After single-task distribution, all tasks finish at the same time if the nodes are homogeneous. However, the nodes are normally heterogeneous. To ensure parallelism, we need a self-resizing algorithm to adjust the data blocks of the faster tasks. A self-resizing algorithm is invoked by the task itself when all its data blocks in S^1 have been processed, and it transforms S^1 to S^2 by snatching one more data block from other tasks, and then S^2 to S^3 analogously. After k times self-resizing, all tasks are completed at S^k. The basic ideas of the self-resizing algorithm are the following: each task processes local data blocks preferentially; the faster tasks process as many remote data blocks as possible, or process the in-rack data blocks if there is not a remote one available. The slower tasks process as many local data blocks as possible. With this approach, we minimize the possibility that all the other tasks wait for the slowest task to process the last unprocessed remote data block.

In Terasort, the speed of a map task is unrelated to the data features it processes, so theoretically all tasks complete simultaneously, otherwise a task completes earlier if it is running on a higher-performance node, at which time this task should snatch a data block from a slower task. The rules of snatching are as follows:

(1) If task i has some local data blocks which are assigned to other tasks in S^0, then these data blocks should be snatched first and processed locally;

(2) Task i randomly snatches a remote data block from another in-rack task j if possible. Remote data blocks of task j are also remote data blocks of task i because tasks i and j are in the same rack.

(3) If there is no further remote data block in the in-rack tasks, task i randomly snatches a remote data block from another remote task j if possible. Here, the remote data block of j is from one of the tasks that are in the same rack as task i, i.e., it is an in-rack data block for task i.

(4) If there is no further in-rack data block that has been assigned to out-rack tasks, task i randomly snatches a remote data block from another remote task j if possible. This data block is also a remote data block of task i.

(5) If there is no task containing a remote data block, task i randomly snatches an in-rack data block from any other in-rack task.

(6) If there is no in-rack task containing an in-rack data block, task i randomly snatches an in-rack data block from any other remote task.

(7) If there is no remote task containing an in-rack data block, task i randomly snatches a local data block of any other in-rack task.

(8) If there is no in-rack task containing any unprocessed data block, task i randomly snatches a local data block from any other remote task, until all tasks are complete.

Figure 3 shows how the self-resizing algorithm works. For simplicity, we assume that there are two racks in the cluster, each rack has two nodes, and the I/O speed ratio of local data, in-rack data and remote data is 1:1:2. After the single-task distribution, the initial task matrix is S^1, and four self-resizing steps are shown in Figure 3. In Figure 3, each row represents a task matrix of *mappers* when the proposed algorithms are executed. After the single-task distribution, S^0 has changed to S^1, block a_3 was moved to node d, while b_6 and b_7 were distributed to node a, and b_5, d_4 and d_5 were distributed to node c. Then, after task a completed, the node a found block a_3 and snatched it according to the first rule of the self-resizing algorithm. Before all the tasks were complete, each step satisfied the self-resizing algorithm. Noticing that tasks a and b were faster than tasks c and d, nodes a and b contain more data blocks than nodes c and d.

The self-resizing algorithm ensures that the possibility of processing remote data blocks by faster asks is higher than that for slower tasks. It improves both the performance and parallelism of Terasort.

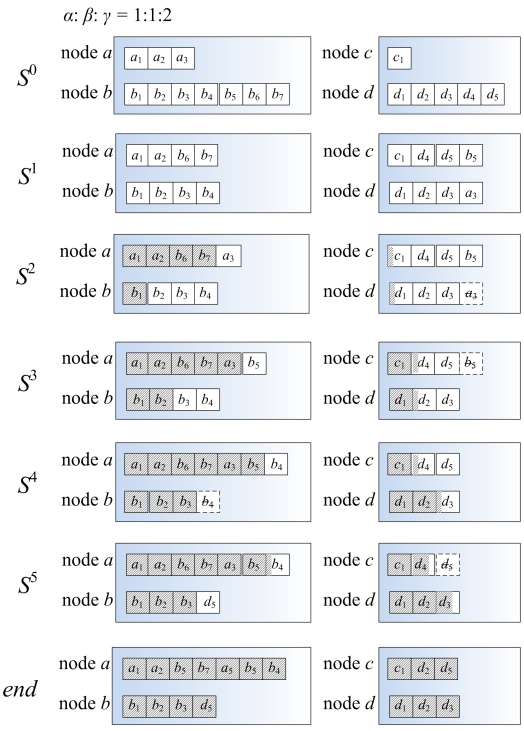

Figure 3. An example of the data processing steps in the map phase. There are one step of single-task distribution and four steps of self-resizing. The small boxes refer to data blocks. It can be distinguished whether a data block is local, in-rack or remote from its id. A shadow boxed (or part of box) means that the data block has been processed.

4. Experiments

We built a Hadoop MapReduce cluster with 12 computers and compared the performance and energy consumption between the original Terasort and the optimized Terasort. Details of the testbed are shown in Table 2.

As shown in Figure 4, the total time consumption of sorting the same data size by the original and optimized Terasort is different. The optimization effects are also different across the different cases. In case *A*, the optimized efficiency (the ratio between reduced value and original value) of time consumption and that of energy consumption are 6.32% and 9.42%, respectively. In case *B* they rise to 8.62% and 13.27%, respectively. In case *C* they continuously rise to 11.96% and 15.90%, respectively. The optimized Terasort sorts faster than the original one, and also consumes less energy. The original Terasort also considers the different processing capacities of different nodes, but does not consider the consumption of time and energy during scheduling and node idleness. Consequently, the optimized Terasort gains bet-

Table 2. Description of the testbed

Iterms	Description
Node	1 master node and 11 slave nodes. Homogeneous computers. Intel Core i5-2300 2.80GHz, 8GB memory, 1TB hard disk, onboard video, audio and network card. 1000Mb network.
Operating System	CentOS 5.6, Linux 2.6.18 Kernel
MapReduce Platform	Hadoop 1.0.4
Energy Consumption Measuring approach	PowerBay power-meter (http://www.northmeter.com /index-en.html), power precision ±0.01 0.1W, maximum 2200W, measurement frequency 1.5-3 second. Obey specification GB/T17215-2003. In addition, to avoid accidental error, the experiments are performed 10 times, and the results shown below are the mean values.
System Information Collecting Tool	SAR
Test cases	Test cases with different data size run under the original Terasort [11] and the optimized Terasort, and data are randomly distributed on the nodes. Case A: 3 GB per node Case B: 5 GB per node Case C: 10 GB per node

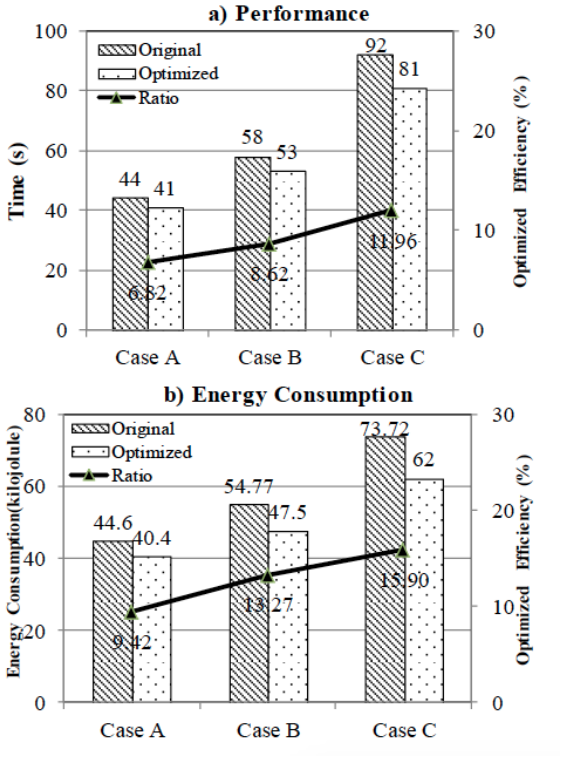

Figure 4. Comparison between time consumption and energy consumption of the original and optimized TeraSort by sorting different data sizes, and the optimized efficiency, which is the ratio between reduced value and original value.

ter performance and energy efficiency. The optimization is more obvious when more data are involved.

From the optimized efficiency (black line) in Figure 4, we know that there is a positive linear correlation between the superiority of the optimized Terasort and the data size. This means that when sorting really big data sets, the optimization effect of time and energy consumption is more obvious. Our testbed is a homogeneous cluster. However, the optimization effect of the proposed approach is more significant in a heterogeneous cluster because in a heterogeneous cluster, the self-resizing algorithm adjusts the number of data blocks, while the original MapReduce adjusts the number of tasks (splits). Obviously, the former has a finer adjusting granularity.

From Figure 4, we can see that time consumption is positively related to energy consumption. It can be concluded that in a MapReduce Terasort, performance improvement is in accordance with the energy consumption optimization. There are two main opinions about the relationship between performance and energy consumption of a distributed application. On the one hand, it is generally believed that when more nodes are involved, jobs are executed faster while more energy is consumed. On the other hand, the waiting time is reduced so that both performance and energy consumption are optimized. For example, nodes are idle while waiting for the scheduled tasks, or the CPU is idle while waiting for I/O operations. These negatively affect both performance and energy efficiency. Our experiment proves the latter in Terasort. Moreover, the experimental results prove that the mean of the improved time consumption is 9%, while the mean of the improved energy consumption is 12%. Generally, multiplying time and power is energy consumption. Even if we assume power to be almost constant, the energy consumption is still optimized as a consequence of the reduction in time consumption (optimization by 9%).

To prove that minimizing the scheduling time and waiting time of a reducer is the main reason of op-

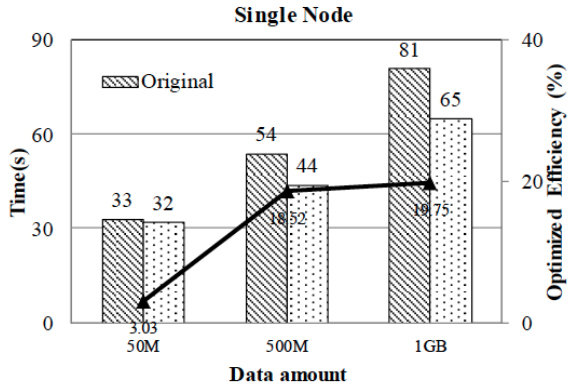

Figure 5. Time consumption of the original and the optimized Terasort, and the optimized efficiency. The environment in the experiment is a single node 'cluster', which means that there is only one *mapper* and one *reducer*, $\Delta = 1$, i.e., the *reducer* will not wait for the *mapper*.

timization, we designed two additional experiments. The first one logs the execution time of each *mapper* and calculates Δ (see Definition 3) of the original Terasort and the optimized Terasort. The parallelism Δ of the optimized Terasort is 4.31 times larger than that of the original one in case *C*, proving that the optimized Terasort has better parallelism. In the second experiment, we executed the original Terasort and the optimized Terasort on a single node. In that situation, there is no *reducer* waiting for *mappers*, so the optimization is the result of saving the task scheduling time.

The results shown in Figure 5 are in accordance with those shown in Figure 4. The total time taken by sorting the same data size under two kinds of Terasort is different. The optimization effects are also different among the different cases. In case *A*, due to the small dataset, the optimization is not significant, but for case *B*, the optimized efficiency is as significant as 18.52% and for case *C* the optimized efficiency is stable at 19.75%. Moreover, the optimized efficiency is better than that in Figure 4 because in the original Terasort, the node is both *JobTracker* and *TaskTracker*, so the scheduling cost is much higher than that of the optimized Terasort.

In conclusion, we have proven that the optimized Terasort reduces the time and energy consumption caused by the scheduling and node waiting. That is to say, compared with the original one, the single-task distribution and self-resizing algorithms improve the performance and reduce the energy consumption of Terasort.

5. Applications

The proposed approach can be applied not only in Terasort, but also in many other applications whose task complexity is only related to the data size, but not data distribution or other features. Terasort has two important operations, one is sampling and building the trie tree, the other one is marking records. Their performance only depends on the data mount. Besides Terasort, there are also many MapReduce applications with these characteristics. Taking Join and PageRank algorithms as examples, the *mappers* read and parse all inputs and send them to *reducers*. In general MapReduce applications, the data size is main effect of performance, but not data distribution, especially in the big data environment. Thus, the proposed approach has a wide applicability.

The proposed approach could be applied in any master-slave based distributed environment, in which computations are moved to the data side. For example, in Yarn, although it divides the *JobTracker* into *ApplicationMaster* and *ResourceMaster* to decrease the burden of the *JobTracker*, the scheduling algorithm also causes the non-ignorable lack of performance and waste of energy. Improving parallelism is a general requirement of parallel computing. The proposed algorithm also contributes to the parallelism guarantee of other parallel computing systems.

6. Related Work

Most of the work on sorting algorithm optimization does not focus on MapReduce. For example, Bunse et al. [12] use standard sorting algorithms to improve the energy efficiency of data sorting. Beckmann et al. [13] propose an approach using solid state disks to advance the energy efficiency of sorting algorithms. Thus, our approach is different from the traditional sorting optimization. We focus on the MapReduce framework, and optimize both time consumption and energy consumption. We believe that node waiting in Terasort is a weakness, and there is limited work that optimizes Terasort or other MapReduce applications from this perspective.

Nowadays, improving performance and reducing energy consumption are the two trends of MapReduce optimization in big data environments. Generally, there are two research directions, the one is performance and energy consumption optimization from the job and task perspective, and the other one is data processing optimization from the scheduling perspective.

For example, He [14] , Zhou [15] and Babu [16] focus on performance optimization. He [14] divides a job into several small jobs, and when one job is

complete, all its related data are deleted in order to reduce the size of intermediate data saved on the local disk in a reasonable level. However, we would get a wrong order if the input is divided in the sort algorithm. Zhou [15] adopts a $< key_1, key_2, value >$ triple instead of a $< key, value >$ pair and adds a key-value routing strategy to improve the efficiency of MapReduce. However, the *mapper* is required to send the $< key_1, key_2, value >$ triples to the *JobTracker*, which increases the burden of the master node. Babu [16] adjusts the MapReduce parameters to implement automatic optimization, but it is difficult to determine the highly-impact parameters with additional operations, which means that time and energy are wasted.

Meanwhile, many papers [17-20] focus on reducing the energy consumption. Wirtz and Ge[17] consider how MapReduce efficiency changes with two runtime configurations: resource allocation that changes the number of available concurrent tasks and DVFS (Dynamic Voltage and Frequency Scaling) that adjusts the processor frequency according to the workloads. Wirtz [18] proposes a centric data movement approach and present an analytical framework with methods and metrics for evaluating costly built-in data movements in MapReduce. Chen [19] considers that interactive jobs operate on a small fraction of the data and, thus, can be served by a small pool of dedicated machines. The less time-sensitive jobs can run on the rest of the cluster in a batch fashion. Lang and Patel [20] focus on developing a framework for systematically considering various MapReduce node power-down strategies and their impact on the overall energy consumption and workload response time. However, this research focuses on resource allocation and task scheduling (see next paragraph). In our approach, we adopt a task self-tuning strategy. We adjust the size of multiple tasks dynamically, which makes our approach different from existing work.

Other works adjust the task execution by optimizing the scheduling algorithm. These works follow two main ideas. One is to adjust the running order of multiple tasks. For example, FIFO is the default scheduling algorithm in Hadoop MapReduce, while Facebook and Yahoo engineers put forward new scheduling algorithms called Fair Scheduling [21] and Capacity Scheduling [22]. The two have been widely recognized and are adopted in practice. The other one is the optimization by task allocation, which determines where the task is distributed and how much resources should be allocated to the task. Wang [23] proposes a task scheduling model using an effective genetic algorithm with practical encoding and decoding methods and specially designed genetic operators. Yong [24] proposes a dynamic slot mechanism to save energy and improve performance. However, an experimental verification is lacking here. Zhou [25] proposes an energy-efficient scheduling policy, called green scheduling, which relaxes fairness slightly to create as many opportunities as possible for overlapping resource complementary tasks. Here, a quantitative analysis is also lacking. Our scheduling approach is different from these, as it places an emphasis on the task size and highlights the reduction of nodes waiting times, ensuring data locality and parallelism. Our approach is designed for Terasort and other applications whose tasks have fixed time- complexity and data-size-determined performance.

7. Conclusions and Future work

This paper proposes a single-task distribution and a task self-resizing algorithm to reduce execution time and energy consumption of Terasort. In contrast to other task scheduling approaches, we focus on minimizing the time of *mappers* waiting for a task and the time of *reducers* waiting for the *mappers*' outputs. In the experiments, we compared our improved Terasort and the original one. The results show that with our algorithms the execution time and energy consumption of Terasort are reduced.

Two conclusions can be drawn from our work. The first is that time and energy are wasted in M n MapReduce applications because of nodes waiting for inputs. This problem was solved by adopting coarsely granular tasks. The second is that the performance and energy consumption of MapReduce applications is closely related to the parallelism among nodes. Parallelism could have been ensured by adopting finely granular tasks, which is however inconsistent with the first issue. Thus, another, adopted option is adjusting the size of tasks dynamically to maximize parallelism.

Our future work will focus on how to adapt the single-task distribution and task self-resizing algorithms to other MapReduce applications. We will furthermore consider the parallelism of *reducers*.

Acknowledgements

This research was funded by a grant (No. 61202088, 61173028, 61433008) from the National Natural Science Foundation of China, and a grant (No. N110417002) from the Fundamental Research Funds for the Central Universities of China, and a grant (No.2013M540232) from the Research Funds of China Postdoc, and the a grant (No.201403314) from the the Science Foundation of Liaoning Provence.

References

[1] **C. Lei, L. Luo, W. Wu.** Cloud Computing Based Cluster Energy Monitoring and Energy Saving Method Study. *Computer Applications and Software*, 2011, 28, 239–242.

[2] **J. Koomey.** Growth in data center electricity use 2005 to 2010. *Technical report, Oakland, Analytics Press, CA*, 2011.

[3] **V. Estivill-Castro, D. Wood.** A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 1992, 24, 441–476.

[4] **M. Ajtai, J. Komlós, E. Szemerédi.** An O(n log n) sorting network. *In: Proceedings of the fifteenth annual ACM symposium on Theory of computing, New York, USA*, 1983, pp. 1–9.

[5] **Owen O'Malley.** Terabyte sort on apache hadoop. *Technical report, Yahoo*, 2008.

[6] **Sort benchmark.** http://sortbenchmark.org/

[7] **Owen O'Malley.** Winning a 60 Second Dash with a Yellow Elephant. *Technical report, Yahoo*, 2009.

[8] **Trie.** http://en.wikipedia.org/wiki/Trie

[9] **Black, E. Paul.** Trie, dictionary of Algorithms and Data Structures. *Technical report, National Institute of Standards and Technology, USA*, 2010.

[10] **W. Zhang, C. Jie, Y. Wang.** MapReduce programming framework operation method based on pipeline communication. *Patent, http://www.google.com/patents/CN101996079A ?cl=en*.

[11] **Hadoop Terasort Example.** http://hadoop.apache.org/docs/current/api/org/apache /hadoop/examples/terasort/package-summary.html

[12] **C. Bunse, H. Höpfner, S. Roychoudhury, E. Mansour.** Energy efficient data sorting using standard sorting algorithms. *Software and Data Technologies*, 2011, 247–260.

[13] **A. Beckmann, U. Meyer, P. Sanders, J. Singler.** Energy-efficient sorting using solid state disks. *Sustainable Computing: Informatics and Systems*, 2011, 1, 151–163.

[14] **R. He.** The performance Optimization and Improvement of MapReduce in Hadoop. *Unpublished.*.

[15] **F. Zhou.** An improved MapReduce parallel programming model. *Science & Technology Association Forum*, 2009, 2, 65–66.

[16] **S. Babu.** Towards automatic optimization of MapReduce programs. *In:Proceedings of the 1st ACM symposium on Cloud computing, New York, USA*, 2010, pp. 137–142.

[17] **T. Wirtz, R. Ge.** Improving MapReduce energy efficiency for computation intensive workloads. *In: Proceedings of Green Computing Conference and Workshops, Orlando, FL*, 2011, pp. 1–8.

[18] **T Wirtz, R Ge, Z Zong, Z Chen** Power and energy characteristics of MapReduce data movements. *In: Proceedings of the 1st ACM symposium on Cloud computing, Arlington, Virginia*, 2011, pp. 1–8.

[19] **Y P Chen, S Alspaugh, D Borthakur, R Katz** Energy efficiency for large-scale MapReduce workloads with significant interactive analysis. *In: Proceedings of the 7th ACM european conference on Computer Systems, Bern, Switzderland*, 2012, pp. 43–56.

[20] **W. Lang, J. M. Patel.** Energy management for MapReduce clusters. *In: Proceedings of the VLDB Endowmen*, 2010, 3, pp. 129–139.

[21] **M. Isard, V. Prabhakaran, J.Currey, U. Wieder, K. Talwar, A. Goldberg.** Quiney: Fair scheduling for distribured computing clusters. *In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Big Sky, Montana.*, 2009, pp. 261–276.

[22] **M. Zaharia, D. Borthakur, J.S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica.** Job Scheduling for Multi-User MapReduce Clusters. *EECS Department, University of California, Berkeley*, 2009.

[23] **X.Wang, Y. Wang, H. Zhu.** Energy-efficient task scheduling model based on MapReduce for cloud computing using genetic algorithm. *Journal of Computers*,2012, 7, 2962–2970.

[24] **M. Yong, N. Garegrat, S. Mohan.** Towards a resource aware scheduler in hadoop. *In: Proceedings of the 2009 IEEE International Conference on Web Services, Los Angeles, CA, USA*,2009, pp. 102–109.

[25] **T. Zhu, C. Shu, H. Yu.** Green scheduling: A scheduling policy for improving the energy efficiency of fair scheduler. *In: Proceedings of the 2011 IEEE International Conference on Parallel and Distributed Computing, Applications and Technologies, Tainan, Taiwan*, 2011, pp. 319–326.