2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing

Jiahui Jin, Junzhou Luo, Aibo Song, Fang Dong and Runqun Xiong
School of Computer Science and Engineering,
Southeast University,
Nanjing, P.R. China
{jhjin,jluo,absong,fdong,xrq918}@seu.edu.cn

***Abstract*—Large scale data processing is increasingly common in cloud computing systems like MapReduce, Hadoop, and Dryad in recent years. In these systems, files are split into many small blocks and all blocks are replicated over several servers. To process files efficiently, each job is divided into many tasks and each task is allocated to a server to deals with a file block. Because network bandwidth is a scarce resource in these systems, enhancing task data locality(placing tasks on servers that contain their input blocks) is crucial for the job completion time. Although there have been many approaches on improving data locality, most of them either are greedy and ignore global optimization, or suffer from high computation complexity. To address these problems, we propose a heuristic task scheduling algorithm called BAlance-Reduce(BAR) , in which an initial task allocation will be produced at first, then the job completion time can be reduced gradually by tuning the initial task allocation. By taking a global view, BAR can adjust data locality dynamically according to network state and cluster workload. The simulation results show that BAR is able to deal with large problem instances in a few seconds and outperforms previous related algorithms in term of the job completion time.**

***Index Terms*—Cloud Computing, Task Scheduling, Data Locality, Hadoop, Dryad**

I. Introduction

In recent years, large scale data processing has emerged as an important part of state-of-the-art internet applications such as search engines, online map services and social networkings. These applications not only handle vast amount of data, but also generate a large quantity of data everyday. Cloud computing systems like MapReduce[1], Hadoop[2] and Dryad[3] which are based on simplified parallel programming models, have been designed for data-intensive applications. For example, Facebook's Hadoop data warehouse stores more than 15PB of data(2.5PB after compression); on a single day, more than 10,000 jobs are submitted to process a large amount of data, meanwhile more than 60TB of new data(10TB after compression) are loaded[4].

The general architecture of a cloud computing system [1, 3, 5, 6] is illustrated in Fig. 1. In this architecture, a file is split into fixed-size *blocks* which are stored on servers. For fault tolerance, all blocks are replicated and spread over the cluster. To process the file, the scheduler divides a job into small tasks, each of which is allocated to an idle server to deal with a file block concurrently. For example, in Fig. 1, a file with size 896MB is partitioned into seven 128MB blocks(each

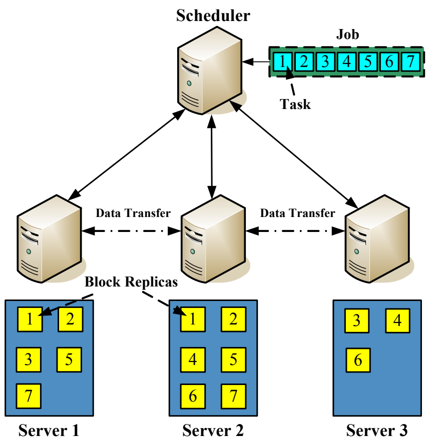


Fig. 1. Architecture of Cloud Computing System

one has two replicas); to process the file, the scheduler divide a job into seven tasks, each of which processes a file block. Under this mode, all tasks can work in parallel to speed up execution of the job. The job can not be finished until all tasks are done. The time span between job's start time and job's finish time, is called *job completion time*. In addition, if a task reads its block from server's local disk, it is called *data-local*; otherwise, the task is called *data-remote* if it retrieves a copy of its block from a remote server. Since most cloud computing systems are implemented on commodity or virtual hardware[5–7], the data transfer cost gives a great impact on the system performance[1, 8].

To handle this issue, there has been much work on enhancing task *data locality* by scheduling tasks close to their data[2, 6, 9–11]. In Hadoop system, for an idle server, the scheduler greedily searches for a data-local task and allocates it to the server[2, 6]. As this policy is quite simple, it leads to limited data locality. To improve data locality, some approaches have attempted to delay schedule the job until an appropriate server is arrived[9, 10]. A limitation of these policies is that servers are not always become idle quickly enough as assumed. If the cluster is *overloaded*, preserving high data locality wastes a large amount of time waiting. To assign tasks efficiently, Fischer et al.[11] propose a flow-based algorithm which employs maximum flow algorithm and increasing threshold techniques. However, this algorithm suffers from high computation complexity.

To address these problems, a task scheduling problem, which takes account of data locality, network state and cluster workload, is proposed. A two phase task scheduling algorithm is introduced to solve this problem. In this algorithm, 1) an initial task allocation is produced, where all tasks are data-local; 2) the job completion time is reduced gradually by adjusting the initial task allocation. The simulation results show that our algorithm outperforms existing algorithms in term of job completion time.

The rest of this paper is organized as follows. In the next section, we list the related work. In Section III, the scheduling problem is formalized. In Section IV, we propose a two phase heuristic algorithm. Section V reports the results of our simulation experiments. Finally, we conclude this paper in Section VI.

II. Related Work

A. *Data-aware scheduling on distributed systems*

Over the past decade, data-intensive applications are emerged as an important part of distributed computing. Meanwhile considerable work has been done on data-aware scheduling on distributed systems. Stork[12] is a specialized scheduler for data placement and data movement in Grid. The main idea of Stork is to map data close to computational resources. Though Stork can be coupled with a computational task scheduler, no attempt is made to use data locality to reduce data transfer cost. The Gfarm[13] architecture is designed for petascale data-intensive computing. Their model specifically targets applications where the data primarily consist of a set of records or objects which are analysed independently. In Gfarm, several greedy scheduling algorithms are implemented to improve data locality. However these algorithms do not take account of the global optimization of all tasks. Raicu et al.[9] have implemented task diffusion on Falkon[14]. Data diffusion acquires compute and storage resources dynamically, replicates data in response to demand, and schedules computations close to data. Its task scheduling policy sets a threshold on the minimum processor utilization to adjust data locality and resource utilization. However, the simple policy can not improve system performance significantly.

B. *Scheduling on cloud computing systems*

Scheduling on cloud computing systems has been studied extensively in early literature. The default Hadoop scheduler schedules jobs by FIFO where jobs are scheduled sequentially. To achieve data locality, for each idle server, the scheduler greedily searches for a data-local task in the head-of-line job and allocates it to the server[2]. However the simple policy leads to limited data locality; meanwhile the completion time of small jobs is increased. To enhance both fairness and data locality of jobs in a shared cluster, Zaharia *et al.*[10] propose delay scheduling which improves max-min fairness[15]: when the job that should be scheduled next according to fairness cannot launch a data-local task, it waits for a small amount of time, letting other jobs launch task instead. As servers are assumed to become idle quickly enough, it is worth waiting for a local task. However, this assumption is too strict, so delay scheduling does not work well when servers free up slowly. A close work to delay scheduling is Quincy[16]. Quincy maps the scheduling problem to a graph data structure according to a global cost model, and solves the problem by a well-known min-cost flow algorithm. Quincy can achieve better fairness, but it has a negligible effect on improving data locality. Hadoop on Demand(HOD) is a management system for provisioning virtual Hadoop clusters over a large physical cluster[17]. It is inefficient that map tasks need read input splits across two virtual clusters frequently. To reduce the data transferring overhead in HOD, Seo *et al.*[18] designs a prefetching scheme and a preshuffling scheme. However, these methods occupy much network bandwidth, so system performance may be decreased. To optimize the performance of multiple MapReduce workflows, Sandholm *et al.*[19] develop a dynamic prioritization algorithm, but data locality is not enhanced in this algorithm. To discover task straggler, Zaharia *et al.*[7] propose a system called LATE that makes better estimates of tasks' rest execution time. It is shown that LATE executes speculative tasks more efficiently than the Hadoop's current scheduler in heterogeneous environments where the performance of servers are uncontrollable.

To assign tasks efficiently in Hadoop, Fischer *et al.*[11] introduced an idealized Hadoop model called Hadoop Task Assignment problem. Given a placement of input blocks over servers, the objective of this problem is to find the assignment which minimized the job completion time. It is indicated that Hadoop Task Assignment problem is NP-complete. To solve the problem, a flow-based algorithm called MaxCover-BalAssign is provided. MaxCover-BalAssign works iteratively to produce a sequence of assignments and output the best one. It computes in time $O(m^2n)$, where m is task number and n is server number. The solution has been shown to be near optimal. However, it takes a long time to deal with a large problem instance.

III. Problem Formalization

In this section, the system model is formalized. We consider scheduling a set of independent tasks on a homogeneous platform. As shown in Fig. 1, there are $m(m = 7)$ tasks and $n(n = 3)$ servers, where each task processes an input block on a server. On one hand, as input blocks are fixed-size, we assume that data-local tasks take identical constant local cost. On the other hand, as a larger remote task number will cause a higher network contention, remote cost is increased when the remote task number become larger. A job is not completed until all tasks are finished. In addition, we take account of cluster workload: at the start time, if most servers are idle, the cluster is underloaded; in an overloaded cluster, many servers can not be idle in a short time. Base on these assumptions, our goal is to find an allocation strategy that minimizes the job completion time. This problem has been shown to be NP-complete in a restrict case(all servers are idle at the start time)[11]. The following definitions are used throughout the paper.

Definition 1 (Data Placement)**.** The data placement is presented by a bipartite graph $G = (T \cup S, E)$, let $m = |T|$ and $n = |S|$, where T is the set of tasks, S is the set of servers and $E \subseteq T \times S$ is the set of edges between T and S. An edge $e(t, s)$ implies that the input data of task $t \in T$ is placed at server $s \in S$, it also indicates indicated that task t *prefers* server s. And $S^G_{pre}(t)$ is the set of task t's preferred servers in G. We assume that $|S^G_{pre}(t)| \geq 1$. It means that all of the nodes in T have degree at least 1.

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

Definition 3 (Cost of Task)**.** Assume that all servers are homogeneous, as well as tasks[1]; so each task consumes identical execution cost on every server. Furthermore, we define the cost of a task as the sum of the execution time and the input data transferring time. $C(t, \alpha)$ denotes the cost of task t which is performed on the server $\alpha(t)$. It is defined by

$$C(t, \alpha) = \begin{cases} C_{loc}, & \text{if } t \text{ is local in } \alpha \\ C^\alpha_{rem}, & \text{otherwise.} \end{cases} \tag{1}$$

We call C_{loc} and C^α_{rem} the local cost and the remote cost, respectively. Since the time of reading input data from local disk can be ignored, the local cost indicates the execution time; while the remote cost is the sum of the execution time and the data transferring time. For the sake of simplicity, we assume that all the local tasks spend identical execution time, as well as the remote tasks. For all allocations, the local cost is constant. However, since remote tasks compete for network resources, the remote cost grows with the total number of remote tasks. Let

$$C^\alpha_{rem} = C_{rem}(r^\alpha), \tag{2}$$

where $C_{rem}(\cdot)$ is a monotone increasing function, and r^α is the remote task number.

Definition 4 (Load of Server)**.** Under allocation α, some servers are assigned several tasks. We assume that tasks are performed sequentially on a server. The *load* of a server s denotes the time when s finishes its work. Under α, the load of server s is defined as

$$L_s(\alpha) = L^{init}_s + L^{task}_s(\alpha), \tag{3}$$

[1]In most cloud computing systems, the cluster consist of homogeneous servers. Meanwhile the tasks which come from the same job are homogeneous, and they also process the same amount of input data.[1, 5, 6]

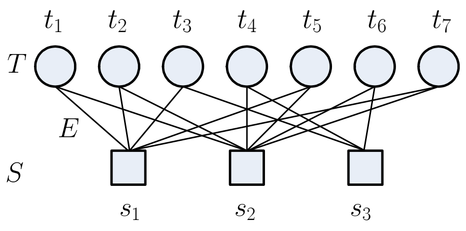


Fig. 2. A data placement $G_p(T \cup S, E)$

TABLE I
THE INITIAL LOAD OF SERVERS

Servers	s_1	s_2	s_3
Initial load	7.1	4.2	0.3

TABLE II
ALLOCATION β

Tasks	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Servers	s_2	s_2	s_3	s_3	s_3	s_2	s_3

where L^{init}_s is the *initial load* of server s at the start time, let

$$L^{init} = \{L^{init}_s | s \in S\} \tag{4}$$

be the set of all initial loads;

$$L^{task}_s(\alpha) = \sum_{t:\alpha(t)=s} C(t, \alpha) \tag{5}$$

is the total time to process tasks on server s. If α is total,

$$L^{fin}_s(\alpha) = L_s(\alpha) \tag{6}$$

denotes the finial load of server s. Let

$$S^\alpha_{ass} = \{s \,|\, \exists t \in T \text{ that } \alpha(t) = s\} \tag{7}$$

denote the servers which are assigned tasks. For each server s, if s is in S^α_{ass} then s is *active*. The job completion time (which is also called $makespan$) under total allocation β is defined as

$$makespan_\beta = \max_{s \in S^\beta_{ass}} L^{fin}_s(\beta). \tag{8}$$

Based on these definitions, the scheduling problem can be defined as :

Problem. Given a data placement $G = (T \cup S, E)$, a local cost value C_{loc}, a remote cost function $C_{rem}(\cdot)$ and a initial load set L^{init}, the objective of the problem is to find a total allocation that minimize the makespan.

The following example gives to explain the problem clearly.

Example 1. In Fig. 2, there are seven tasks and three servers. Each task has two input data replicas. The initial load of each server is shown in Table I. And allocation β is shown in Table II. Under β, task t_5 and task t_7 are remote tasks. Let local cost C_{loc} be 1 and remote cost function $C_{rem}(r) = 1 + 0.1 \cdot r$. Since there are two remote tasks, the remote cost is $C^\beta_{rem} = 1 + 0.1 * 2 = 1.2$. By Equation (3), it is easy to obtain the final load of each servers that $L_{s_1}^{fin}(\beta) = 7.1$, $L_{s_2}^{fin}(\beta) = 4.2 + 3 * 1 = 7.2$ and $L_{s_3}^{fin}(\beta) = 0.3 + 2 * 1 + 2 * 1.2 = 4.7$. Because server s_1 dose not process any task, S_{ass}^{β} is $\{s_2, s_3\}$. By Equation (8), $makespan_{\beta}$ is 7.2.

IV. Balance-Reduce Algorithm

In this section, we introduce a data locality driven task scheduling algorithm, called BAlance-Reduce(BAR), which finds a good solution in time $O(\max\{m+n, n\log n\} \cdot m)$. On finding a feasible solution, a critical obstacle is that the remote cost can not be calculated before the remote task number is known. Moreover, it is hard to obtain a near-optimal solution when the remote cost changes frequently. For example, when we allocate a remote task, the remote task number increase by one, so the remote cost may also increase. Furthermore, the load of the servers which have been allocated remote tasks must be updated.

In order to make sure the remote cost, BAR is split into two phases, *balance* and *reduce*:

- *Balance*: Given a data placement graph G, a initial load set L^{init} and a local cost C_{loc}, the balance phase returns a total allocation $\mathcal{B}$. Under $\mathcal{B}$, all tasks are allocated to their preferred servers evenly.
- *Reduce*: Given a local cost C_{loc}, a remote cost function $C_{rem}(\cdot)$, a total allocation $\mathcal{B}$ computed by the balance phase, and an initial load set L^{init}, the reduce phase works iteratively to produce a sequence of total allocations and returns the best one. By taking advantage of $\mathcal{B}$, the remote cost can be computed at the beginning of each iteration.

A. Balance phase

In this section, we describe the balance phase in detail. In this phase, a balanced total allocation is produced where all tasks are allocated to their preferred servers evenly. To introduce balanced total allocation, we present following definitions.

Definition 5 (Local Allocation)**.** Let $\mathcal{L}$ be an allocation. Under $\mathcal{L}$, if all defined tasks are local, then $\mathcal{L}$ is a *local allocation*.

Definition 6 (Balanced Allocation)**.** Let $G(T \cup S, E)$ be a data placement, C_{loc} be a local cost, and $\mathcal{B}$ be a local allocation. $\mathcal{B}$ is a balanced allocation when

$$\forall t \in T \quad L_{\mathcal{B}(t)}(\mathcal{B}) - C_{loc} \leq \min_{s \in S_{pre}^{G}(t)} L_s(\mathcal{B}), \tag{9}$$

where $L_s(\mathcal{B})$ and $L_{\mathcal{B}(t)}(\mathcal{B})$ is computed by Equation (3). Equation (9) is called *balance policy*.

To explain the balance policy, we formalize residual load below.

Definition 7 (Residual Load)**.** Under local allocation $\mathcal{L}$, for each task t, *t-residual-load* of server s is defined by

$$L_s^{res}(\mathcal{L}, t) = \begin{cases} L_s(\mathcal{L}) - C_{loc}, & \text{if } s = \mathcal{L}(t) \\ L_s(\mathcal{L}), & \text{otherwise.} \end{cases} \tag{10}$$

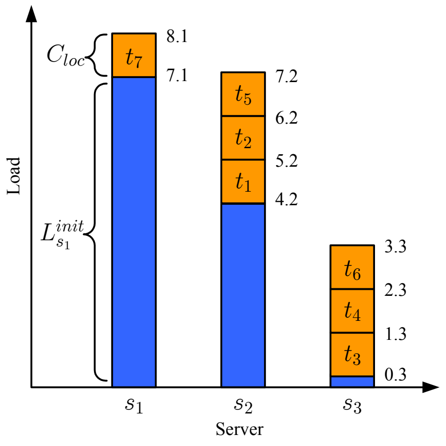


Fig. 3. A balanced total allocation $\mathcal{B}$

If t is allocated to s, $L_s^{res}(\alpha, t)$ denotes the residual load of s apart from t's local cost. Otherwise, $L_s^{res}(\alpha, t)$ is the same as the load of server s.

By Definition 7, the balance policy is represented as follows:

$$\forall t \in T \quad L_{\mathcal{B}(t)}^{res}(\mathcal{B}, t) = \min_{s \in S_{pre}^{G}(t)} L_s^{res}(\mathcal{B}, t). \tag{11}$$

Therefore, under a balanced total allocation, each task t is allocated to one of its preferred servers whose t-residual-load is minimal.

Example 2. Let the data placement, the initial loads, and the local cost be the same as those in Example 1. Fig. 3 shows a balanced total allocation $\mathcal{B}$. The balanced total allocation is not unique. For example, we can swap t_7 and t_1 to generate another balanced total allocation. We note that all tasks are allocated to their preferred servers whose residual loads are minimal, therefore, the makspans of all balanced total allocations are equal to each other.

Theorem 1. *Let $\mathcal{B}$ be a balanced total allocation. $makespan_{\mathcal{B}}$ is minimal among all local total allocations' makespans.*

Proof. Assume, for the sake of contradiction, that there exits a local total allocation $\mathcal{L}^O$ which does not follow the balance policy, and $makespan_{\mathcal{L}^O}$ is less than the makespans of all balanced allocations. Without loss of generality, let task t^o be the task which disobeys the balance policy and $\mathcal{L}^O(t^o) = s^o$. Then, there exits a server s^i which is preferred by t^o and $L_{s^o}^{fin}(\mathcal{L}^O) - C_{loc} > L_{s^i}^{fin}(\mathcal{L}^O)$. Move t^o to s^i. This result in another allocation $\mathcal{L}^B$. Then, $L_{s^o}^{fin}(\mathcal{L}^B) = L_{s^o}^{fin}(\mathcal{L}^O) - C_{loc}$ and $L_{s^i}^{fin}(\mathcal{L}^B) = L_{s^i}^{fin}(\mathcal{L}^O) + C_{loc} < L_{s^o}^{fin}(\mathcal{L}^O)$.

Case 1: $\forall s \in S - \{s^o\}$, $L_{s^o}^{fin}(\mathcal{L}^O) > L_s^{fin}(\mathcal{L}^O)$, so that s^o is the only server whose finial load is maximal. In this case, $makespan_{\mathcal{L}^B} < makespan_{\mathcal{L}^O}$.

Case 2: $\exists s \in S - \{s^o\}$, $L_{s^o}^{fin}(\mathcal{L}^O) \leq L_s^{fin}(\mathcal{L}^O)$. In this case, $makespan_{\mathcal{L}^B} \leq makespan_{\mathcal{L}^O}$.

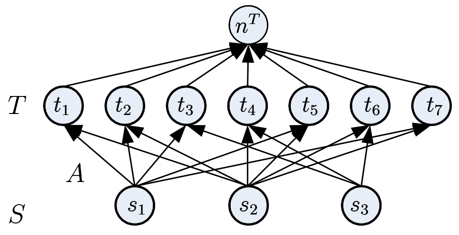


Fig. 4. A flow network G'_f

The motion continues until all tasks follows the balance policy. Finally, we get a balanced total allocation $\mathcal{B}$ that $makespan_{\mathcal{B}} \leq makespan_{\mathcal{L}^O}$. This contradicts that $makespan_{\mathcal{B}} > makespan_{\mathcal{L}^O}$. Thus, $makespan_{\mathcal{B}}$ is minimal among all local allocations' makespans. □

Theorem 2. *Let $\mathcal{B}$ be a balanced total allocation, and t be a task. For each server s, if $L_s^{fin}(\mathcal{B}) < L_{\mathcal{B}(t)}^{fin}(\mathcal{B}) - C_{loc}$, then t does not prefer s.*

Proof. Assume, for the sake of contradiction, that there exits a server s' that $L_{\mathcal{B}(t)}^{fin}(\mathcal{B}) - C_{loc} > L_{s'}^{fin}(\mathcal{B})$ and t prefers s'. Since $\mathcal{B}$ is a balanced total allocation, the assumption contradicts the balance policy. □

In the rest of this section, we introduce an algorithm called Balanced Allocation Algorithm(BAA) that finds a balanced total allocation in time $O(\max\{m+n, n\log n\} \cdot m)$. This algorithm consists of the following three steps.

1) Convert data placement graph $G(T \cup S, E)$ to a flow network $G_f(T \cup S \cup \{n^T\}, A)$, where n^T is a target node, and A is a set of arcs with a positive capacity. An arc $a(n_i, n_j)$ is a directed edge from node n_i to node n_j. For all tasks $t \in T$ and all servers $s \in S$, if there is an edge $e(s,t)$ in G, arc $a(s,t)$ exists in A. In G_f, for each $t \in T$, there exists an arc $a(t, n^T)$. Each arc has a capacity of one. Fig. 4 shows a flow network G'_f which is converted from the data placement graph G_p. G_p is shown in Fig. 2.
2) Allocate unassigned tasks to the minimum load servers iteratively, until all tasks have been allocated. At iteration $\tau (1 \leq \tau \leq m)$, firstly, we mark all nodes *unvisited*. Secondly, select an unvisited server node whose load is minimal. Since all tasks are local, the load of server s can be computed as follows:

$$L_s = L_s^{init} + Num(s) \cdot C_{loc}, \qquad (12)$$

where $Num(s)$ is the number of tasks on server s. Thirdly, find an augmenting path from the selected server node s_0^τ to n^T in the residential graph[20] by growing a search tree Tree$^\tau$[21, 22] which traverses unvisited nodes. If there exists an augmenting path, then we stop the iteration, assign a unit of flow through the path $s_0^\tau \rightarrow n^T$ and update the residential graph. Otherwise, we mark all nodes in Tree$^\tau$ visited, then turn to the second step at this iteration. If the amount of flow equals to total task number m, Step 2 stops, and returns a flow f^m with m units.
3) Convert flow f^m to an allocation $\mathcal{B}$ by Rule 1. Since there are m units of flows in f^m, all task are allocated in this step. Hence, $\mathcal{B}$ is a total allocation.

Rule 1. Given a flow f, for all $s \in S$, if there exist an arc $a(s,t)$ that $f(a(s,t)) = 1$, then $\mathcal{B}(t) = s$.

Algorithm 1 Balanced Allocation

```
1: procedure BALANCE($G(T \cup S, E), C_{loc}, L^{init}$)
2:     define: $G_f$ is a flow network. $N$ is the set of nodes
       in $G_f$. $G_r$ is the residual graph. $\tau$ is the iteration number.
       $P^\tau$ is a augmenting path at iteration $\tau$. Tree$^\tau$ is the search
       tree at iteration $\tau$. $f^\tau$ is the flow on $G_f$ after iteration $\tau$.
       $\mathcal{B}$ is a total allocation.
3:
4:     $G_f \leftarrow$ GetFlowNetwork($G$)
5:     $G_r \leftarrow$ GetResidualGraph($G_f$)
6:     $N \leftarrow T \cup S \cup \{n^T\}$
7:     $\forall s \in S\ s.num \leftarrow 0$
8:     $\tau \leftarrow 1$
9:     while $\tau \leq m$ do
10:        $\forall n \in N\ n.visited = false$
11:        while $P = null$ do
12:            $s_0^\tau \leftarrow$ MinLoadUnvisitedServer($S, C_{loc}, L^{init}$)
13:            $\langle P^\tau$, Tree$^\tau \rangle \leftarrow$ Augment($s_0^\tau$,$n^T$,$G_r$)
14:            $\forall n \in$ Tree$^\tau$ $n.visited = true$
15:            Clear(Tree$^\tau$)
16:            if $P \neq null$ then
17:                $s_0^\tau.num \leftarrow s_0^\tau.num + 1$
18:            end if
19:        end while
20:        $f^\tau \leftarrow$ UpdateFlow($f^{\tau-1}, P^\tau$)
21:        $G_r \leftarrow$ UpdateGraph($G_f, f^\tau$)
22:        $\tau \leftarrow \tau + 1$
23:    end while
24:    $\mathcal{B} \leftarrow$ FlowToAllocation($f^m$)
25:    return $\mathcal{B}$
26: end procedure
```

Theorem 3. *BAA finds a balanced total allocation in time $O(\max\{m+n, n\log n\} \cdot m)$.*

Proof. Let $\mathcal{B}$ be a total allocation obtained by BAA. Assume to the contrary that there exist a task t_f which dissatisfies balance policy. Without loss of generality, let s_k be $\mathcal{B}(t_f)$, s_l be one of t_f's preferred servers, and $L_{s_k}(\mathcal{B}) - C_{loc} > L_{s_l}(\mathcal{B})$. Suppose at iteration μ, $\mathcal{B}^\mu$ is a partial allocation which is converted from f^μ by Rule 1, $s_k = s_0^\mu$ is the start node in P^μ(the augmenting path at iteration μ) and $L_{s_k}(\mathcal{B}^{\mu-1}) + C_{loc} = L_{s_k}(\mathcal{B}^\mu) = L_{s_k}(\mathcal{B}^{\mu+1}) = \cdots = L_{s_k}(\mathcal{B}^{m-1}) = L_{s_k}^{fin}(\mathcal{B})$. As P^μ starts from s_k, for any server s' whose load is less than the load of s_k, there dose not exist a feasible augmenting

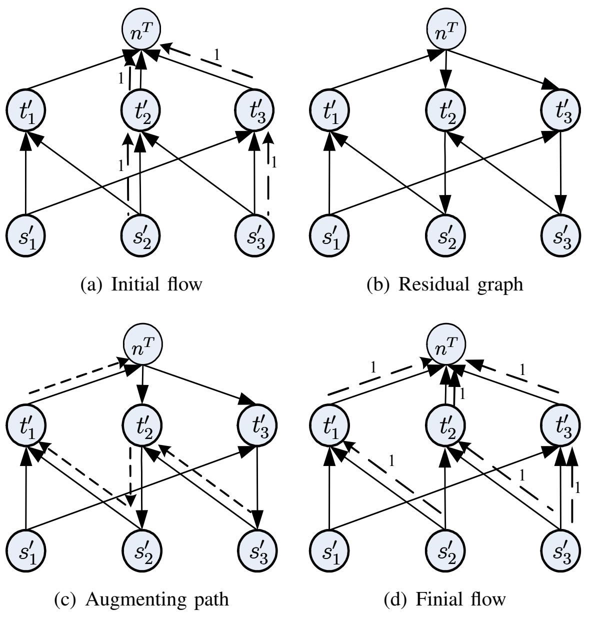


Fig. 5. On finding a balanced total allocation

path which starts from s'. But this conflicts that there exists a server s_l that $L_{s_l}(\mathcal{B}^\mu) \le L_{s_l}(\mathcal{B}^{fin}) < L_{s_k}(\mathcal{B}^{fin}) = L_{s_k}(\mathcal{B}^\mu)$ and a feasible augmenting path $P^\mu_{s_l}$ which starts from s_l:

Case 1: if P^μ: $s_k(s^\mu_0) \to t^\mu_0 \to s^\mu_1 \to t^\mu_1 \cdots \to s^\mu_q \to t^\mu_q \to n^T$, then $P^\mu_{s_l}$:$s_l \to t_f \to s_k(s^\mu_0) \to t^\mu_0 \cdots \to s^\mu_q \to t^\mu_q \to n^T$.

Case 2: if P^μ: $s_k(s^\mu_0) \to t^f \to s^\mu_1 \to t^\mu_1 \cdots \to s^\mu_q \to t^\mu_q \to n^T$, then $P^\mu_{s_l}$:$s_l \to t_f \to s^\mu_1 \to t^\mu_1 \cdots \to s^\mu_q \to t^\mu_q \to n^T$.

Thus, $\mathcal{B}$ is a balanced total allocation.

Let $e = |A|$. As $m = |T|$ and $n = |S|$, a data placement is converted to a flow network in $O(m+n+e)$. In Step 2, at each iteration, it takes time $O(m+n+e)$ to traverse nodes and arcs and costs time $O(\log n)$ to find a minimum load server. There are at most n servers, so each iteration takes time $O(\max\{m+n+e, n\log n\})$. Hence, the time complexity of Step 2 is $O\big(\max\{m+n+e, n\log n\} \cdot m)\big)$. Converting a residual graph to a total allocation takes time $O(e)$. Thus the running time of BAA is $O\big(\max\{m+n+e, n\log n\} \cdot m\big)$. Furthermore, in most cloud computing systems, the number of file replicas is a small constant(2 or 3), so that $e = O(m)$. The running time of BAA takes time $O(\max\{m+n, n\log n\} \cdot m)$. □

Example 3. In Fig. 5, we update flow f^2 on a flow network, and convert f^3 to a balanced total allocation $\mathcal{B}$. s'_1,s'_2,s'_3 are three servers with initial loads 3.1, 2.2, 1.9, respectively. Fig. 5(a) shows a initial flow which has been computed at previous iterations. According to flow f^2, task t'_1 and t'_2 are allocated to server s'_1 and s'_2, respectively, so that the load of servers $L_{s'_1}$, $L_{s'_2}$, $L_{s'_3}$ should be 3.1, 3.2, 2.9. Since the load of s'_3 is minimal, we select s'_3 as start node, and find a path: $s'_3 \to t'_2 \to s'_2 \to t'_1 \to n^T$ in the residual graph. Finally, f^3 is updated, and a balanced total allocation($\mathcal{B}(t'_1) = s'_2$, $\mathcal{B}(t'_2) = s'_3$, $\mathcal{B}(t'_3) = s'_3$) is computed.

B. Reduce phase

In this section, the reduce phase is described in detail. In this phase, we generate a sequence of total allocations, and reduce the makespan iteratively. Pseudocode for this algorithm is shown in Algorithm 2.

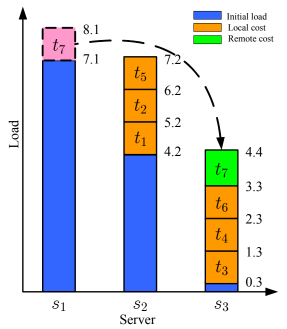
Algorithm 2 Reduce Makespan

```
 1: procedure REDUCE($C_{loc}, C_{rem}, \mathcal{B}, L^{init}$)
 2:     define: $P$ is a remote task pool, $\mathcal{L}_p$ is a local partial
        allocation, $\mathcal{R}$ and $\mathcal{R}_{pre}$ are total allocations, $M_{exp}$ is an
        expected makespan.
 3:
 4:     $P \leftarrow \phi$
 5:     $\mathcal{L}_p \leftarrow \mathcal{B}$
 6:     $\mathcal{R}_{pre} \leftarrow \mathcal{B}$
 7:     while $true$ do
 8:         $s_{max} \leftarrow$ MaxLoadActiveServer($\mathcal{L}_p$)
 9:         $t_l \leftarrow$ RandomTask($s_{max}, \mathcal{L}_p$)
10:         $P \leftarrow P \cup \{t_l\}$
11:         $\mathcal{L}_p \leftarrow$ RemoveTask($\mathcal{L}_p, t_l$)
12:         $M_{exp} \leftarrow$ MaxLoad($\mathcal{L}_p$)
13:         $C^{\mathcal{R}}_{rem} \leftarrow C_{rem}(|P|)$
14:         $\mathcal{R} \leftarrow$ AllocateTasks($P$,$\mathcal{L}_p$,$C^{\mathcal{R}}_{rem}$,$M_{exp}$,$L^{init}$)
15:         if $makespan_{\mathcal{R}} > M_{exp}$ then
16:             if $makespan_{\mathcal{R}} \ge makespan_{\mathcal{R}_{pre}}$ then
17:                 return $\mathcal{R}_{pre}$
18:             else
19:                 return $\mathcal{R}$
20:             end if
21:         end if
22:         $\mathcal{R}_{pre} \leftarrow \mathcal{R}$
23:     end while
24: end procedure
```

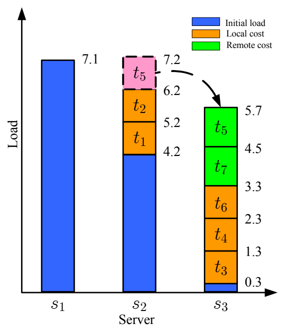
In this algorithm, we define a remote task pool P which stores the tasks from the maximum load servers[2]. Tasks in P are allocated to low load servers, and these tasks are data-remote. $\mathcal{L}_p$ is a local partial allocation where for each task t in P, $\mathcal{L}_p(t)$ is not defined. P is initialized to be empty, while $\mathcal{L}_p$ is initialized to be the balanced total allocation which is produced by balance phase. At each loop, we update P and $\mathcal{L}_p$, then generate a total allocation $\mathcal{R}$. $\mathcal{R}_{pre}$ is a total allocation which is computed by the previous loop. The reduce procedure will be described detailedly as follows:

1) Select the maximum load server s_{max} under $\mathcal{L}_p$. s_{max} is also the maximum load server under $\mathcal{R}_{pre}$. To reduce the makespan, we remove a task from s_{max}. In the next steps, this task will be allocated to a low load server.
2) Let t_l be the chosen task. By Theorem 2, t_l dose not prefer the servers whose load less than $L^{fin}_{s_{max}}(\mathcal{B}) - C_{loc}$. We add t_l to the remote task pool P and update $\mathcal{L}_p$ by marking $\mathcal{L}_p(t_l)$ undefined.

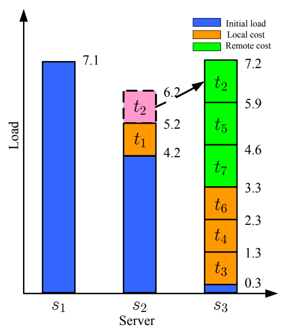
[2]All servers which are mentioned in this section are active.



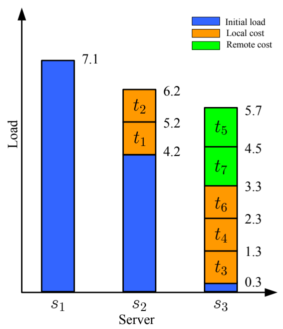
(a) Loop 1



(b) Loop 2



(c) Loop 3



(d) Result

Fig. 6. An example of the reduce phase

3) Calculate the expected makespan of $\mathcal{R}$. The expected makespan M_{exp} equals to the maximum load of servers under $\mathcal{L}_p$.
4) Calculate the remote cost. As task in $\mathcal{L}_p$ are data-local and tasks in P are data-remote, the remote task number should be $|P|$. Hence, $C^{\mathcal{R}}_{rem} = C_{rem}(|P|)$.
5) Allocate tasks. For all tasks $t_{loc} \notin P$, we allocate them to $\mathcal{L}_p(t_{loc})$, so $\mathcal{R}(t_{loc}) = \mathcal{L}_p(t_{loc})$. Then we allocate remote tasks one-by-one. For any task $t_{rem} \in P$, it is allocated to the server whose load is no more than $M_{exp} - C^{\mathcal{R}}_{rem}$. If there is no appropriate server, we allocate t_{rem} to the server with minimum load. Then we update the load of the server.
6) Return result. If $makespan_{\mathcal{R}}$ is larger than the expected makespan M_{exp}, it is impossible to reduce the makespan in the next loops. So a better allocation is selected between $\mathcal{R}$ and $\mathcal{R}_{pre}$, then it is returned.

Example 4. Fig. 6 shows an example of the reduce phase. The date placement, the local cost, the remote cost function and the initial load are the same as those in Example 1. The input balanced total allocation is shown in Example 2. At loop 1, s_1 is the active server with maximum load. The expected makespan is 7.2. We choose t_7 and allocate it to s_3. Since the reduced makespan is equal to the expected makespan, the algorithm enters the next loop. At loop 2, task t_5 is added to the remote task pool, so that there are two remote tasks and the remote cost is increased to 1.2. The reduced makespan is 6.2 which is the same as expected makespan. At loop 3, however, the reduced makespan increase to 7.2 and the expected makespan is 5.2, so we stop the reduce phase. Finally, we get a total allocation whose makespan is 6.2.

C. Balance-Reduce Algorithm

Combining the balance phase and the reduce phase, the pseudocode of BAlance-Reduce(BAR) is shown in Algorithm 3.

Algorithm 3 Balance-Reduce Algorithm

```
1: procedure BALANCE-REDUCE($G, C_{loc}, C_{rem}, L^{init}$)
2:     define: $\mathcal{B}$, $\mathcal{R}$ are total allocations.
3:     $\mathcal{B} \leftarrow$ Balance($G, C_{loc}, L^{init}$)
4:     $\mathcal{R} \leftarrow$ Reduce($C_{loc}, C_{rem}, \mathcal{B}, L^{init}$)
5:     return $\mathcal{R}$
6: end procedure
```

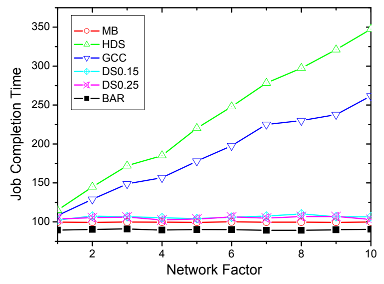
Finally, we analyse the time complexity of BAR. In the balance phase, it is shown that BAA is implemented in time $O(\max\{m + n, n \log n\} \cdot m)$. In reduce phase, most loops allocate tasks in time $O(m)$. However, since the last loop need allocate some tasks to the minimum load servers, it takes $O(m \log n)$ time. There are at most m loops, so reduce phase runs in time $O\big(m(m-1) + m \log n\big)$.

Combining the running time of the two phases, BAR gives time complexity $O(\max\{m + n, n \log n\} \cdot m)$.

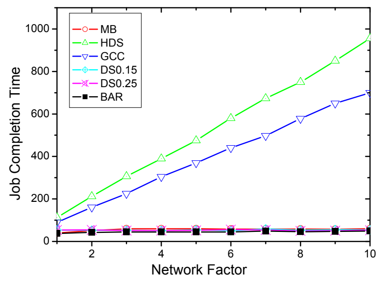
V. Performance Evaluation

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

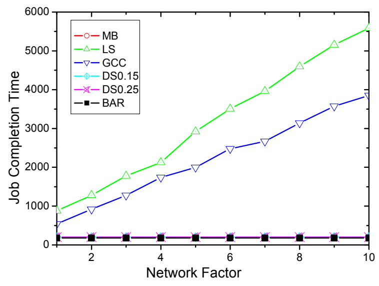
- MaxCover-BalAssign(MB)[11]. This algorithm works iteratively to produce a sequence of total allocations, and then outputs the best one. Each iteration consists of two phase *maxcover* and *balassign*. Since the remote cost is unknown, it calculates the *virtual cost* which is a prediction of the remote cost. Then it computes a total allocation by taking advantage of the virtual cost.
- Hadoop Default Scheduler(HDS)[6]. When a server is idle, the scheduler chooses a data-local task, then allocates the task to the server. If there is no feasible task, then the scheduler will select a random task.
- Delay Scheduling(DS)[10]. It sets a delay threshold. If a server is idle and there is no task prefers the server, the scheduler skips the server and increases the delay counter by one. However, if the delay counter exceeds the delay threshold, the scheduler allocates a remote task to the server and sets the delay counter to be zero. In our simulations, we set the delay threshold to be $\text{TotalServerNumber} \times 15\%$, and denote the algorithm by DS0.15. In addition, for comparison, DS0.25 is also implemented in the simulations.
- Good Cache Compute (GCC)[9]. This policy is similar to DS. It sets an utility threshold which is the upper bound of the number of idle servers. The scheduler can skip



(a) Job completion time on E1

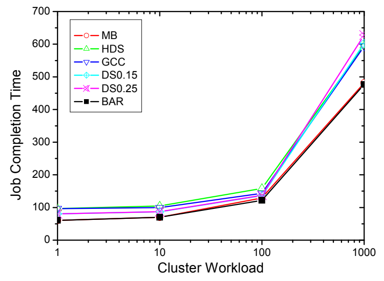


(b) Job completion time on E2

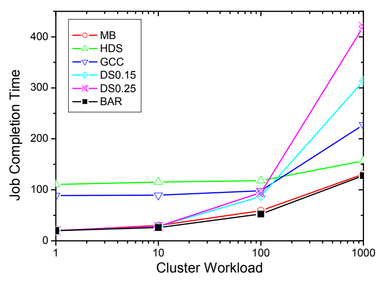


(c) Job completion time on E3

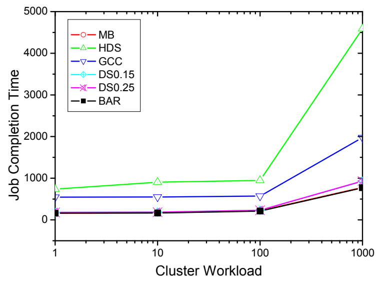
Fig. 7. Effect of network state



(a) Job completion time on E1



(b) Job completion time on E2



(c) Job completion time on E3

Fig. 8. Effect of cluster workload

servers when the the idle server number is below the utility threshold. In our simulations, the utility threshold is set to $\mathrm{TotalServerNumber} \times 90\%$.

In our simulations, the data placement is generated by Hadoop's Rack-aware Replica Placement[6]. We assume that all servers are in the same rack. The number of block replicas is set to 3 which is the default setting in Hadoop[6]. Because the median map task is 19s long in FaceBook[10], local cost C_{loc} is set to 20. The remote cost is defined as $C_{rem}(r) = C_{loc} + q \cdot r$, where r is remote task number and q is *network factor*. Larger value of q introduces scarcer network resources. $\frac{1}{q}$ describes the data transfer rate. Since transferring a 128MB file block takes about 1s in 10Gbit Ethernet and about 10s in 1Gbit Ethernet, we choose $q \in [1, 10]$ in our simulations. The initial load of a server can be estimated by Zaharia's approach[7] in practice. However, in our simulation, we set L_s^{init} to a random value in $[0, W]$, where W describes cluster workload. A small value of W indicates that most server will be idle soon. While a huge value of W implies that the cluster is overloaded that some servers may not be available in a short period of time.

In the rest of this section, BAR is evaluated for performance and computation time. Furthermore, we investigate the effectiveness of our algorithm under a wide range of platform configurations.

A. Performance

BAR is evaluated by comparing the job completion time and the data locality to that of MB, HDS, GCC, DS0.15, DS0.25. Table III lists three kinds of computational systems. E1 is a small scale system while E3 is large scale. E2 is special where task number is much smaller than server number. However, it is common in production cloud computing systems[10].

TABLE III
COMPUTATIONAL SYSTEMS

Name	Server Number	Task Number
E1	100	300
E2	2000	100
E3	2000	15000

1) Effect of network state: To investigate the effect of network congestion, we compare the algorithms by changing the network factor q from 1 to 10. And also, we set W to be 40, so that all servers will be available soon. Fig. 7 shows that BAR works well under various network state, while a poor network environment effects HDS and GCC very much. From Table IV(a), we see that BAR, MB and DS benefit from the high data locality. It is interesting that BAR outperforms DS0.25 and MB on E2, while the data locality of DS0.25 and MB is higher. This is because that the job is small, and many servers do not have input data; to achieve data locality, DS0.25 and MB skip a lot of remote servers, while BAR adjusts data locality to make the job completed early.

2) Effect of cluster workload: To evaluate our algorithm on different cluster workloads, we set W to 1, 10, 100 and 1000, while network factor q is fixed to 1. As shown in Fig. 8, BAR and MB exhibit a significant improvement. We observe that DS has a negligible effect on small jobs when the workload of cluster is high; since the time interval between two idle

TABLE IV
JOB COMPLETION TIME AND DATA LOCALITY

(a) $q = 10, W = 40$

System	BAR		MB		HDS		GCC		DS0.15		DS0.25	
	MR[a]	DR[b]	MR	DR	MR	DR	MR	DR	MR	DR	MR	DR
E1	1×	99.9%	1.1×	100%	3.84×	91.4%	2.89×	94.3%	1.17×	99.8%	1.13×	99.9%
E2	1×	97.3%	1.19×	100%	19.35×	6.7%	14.16×	32.6%	1.11×	99%	1.06×	100%
E3	1×	100%	1.03×	100%	31.05×	96.4%	21.39×	97.5%	1.12×	100%	1.11×	100%

(b) $q = 1, W = 1000$

System	BAR		MB		HDS		GCC		DS0.15		DS0.25	
	MR	DR	MR	DR	MR	DR	MR	DR	MR	DR	MR	DR
E1	1×	80.3%	1.01×	80.6%	1.24×	62.2%	1.24×	68.9%	1.26×	82.1%	1.31×	86.9%
E2	1×	27.3%	1.01×	26.8%	1.19×	5.8%	1.71×	28.6%	2.15×	48.7%	2.55×	63.5%
E3	1×	98.1%	1.01×	98.2%	5.95×	90.4%	2.56×	91.7%	1.20×	99.8%	1.21×	99.9%

(c) $q = 10, W = 1000$

System	BAR		MB		HDS		GCC		DS0.15		DS0.25	
	MR	DR	MR	DR	MR	DR	MR	DR	MR	DR	MR	DR
E1	1×	89.9%	1.01×	90%	2.33×	75.1%	2.02×	78.2%	1.67×	83.1%	1.52×	86.9%
E2	1×	64.1%	1×	64.1%	2.71×	5.3%	2.31×	28.3%	2.21×	42.3%	2.01×	64.3%
E3	1×	99.5%	1.01×	99.7%	30.11×	88.5%	14.64×	89.8%	1.14×	99.9%	1.05×	100%

[a] Makespan Ratio(MR)=$\frac{\text{makespan of Algorithm A}}{\text{makespan of BAR}}$ [b] Data locality Ratio(DR)=$\frac{\text{data-local task number}}{\text{total task number}}$

servers is long, it is expensive to skip a server. By taking advantage of the reduce phase, BAR schedules some data-remote tasks to decrease the job completion time. As shown in Table IV(b), on E2, although DS0.15 and DS0.25 achieve higher data locality than BAR, DS0.15 and DS0.25 is 2.15 times and 2.55 times larger than BAR in the job completion time, respectively. Because DS0.25 skips more servers than DS0.15, it performs worse. However, BAR schedules many remote tasks when the network state is good, so job can be completed in a short time. On E1 and E3, there are many input blocks on every server, so the performance gap between BAR and DS is decreased.

3) Effect of extreme condition: We consider some extreme conditions where the network environment is poor and the cluster is overloaded. To create these conditions, we set q to be 10 and W to be 1000. Table IV(c) shows that BAR still works well. On E2, BAR and MB decrease the job completion time to at least 50%. We note that BAR adjusts data locality dynamically. By comparing Table IV(b) and Table IV(c), BAR and MB schedules more local tasks when network state is worse, while HDS, GCC and DS are static. We observe that the performance of MB is close to which of BAR; however, since BAR exploits a fine-grain adjustment of data locality, it always performs better than MB does.

B. Computation Time

In this section, we evaluate the computation time of our algorithm. Since HDS, GCC and DS are run-time scheduling algorithms[23], we implement them in a compile-time scheme. Firstly, we place all servers into a min-heap; secondly, pop a

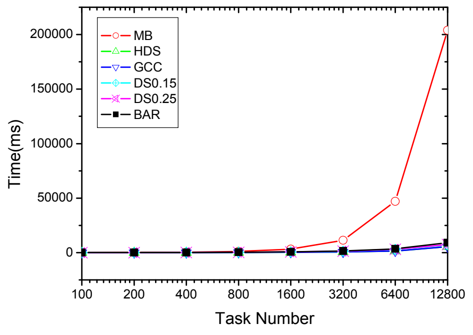


Fig. 9. Computation time

minimum load server, invoke a real-time scheduling algorithm to allocate a task, then update load of servers. The remote cost is renewed when a remote task is allocated. We do the second step repeatedly until all task are allocated. All algorithms are implemented carefully to reduce the redundant work.

The simulations are implemented in Java and runs on a HP PC with Intel(R) Core(TM) Quad CPU Q8200 and 4GB memory. The server number is set to 2000, and the task number ranges from 100 to 12800. Fig. 9 shows that when the job scale is small, all algorithms can be finished in one second; however, when the task number exceeds 800, the running time of MB increases significantly. We see that the running time of BAR is slightly longer than the greedy algorithms. When the task number is 12800, the running time of BAR, HDS, GCC, DS0.15 and DS0.25 are 9.1s, 5.5s, 5.4s, 7.1s and 7.3s, respectively, while MB takes 203 seconds. Thus, BAR can deal with a large problem instance in a few seconds.

VI. Conclusion and Future Work

As large scale data-intensive applications grow in popularity, many cloud computing systems like MapReduce, Hadoop and Dryad have emerged in recent years. In the general cloud computing architecture, network bandwidth is a scarce resource. To improve the system performance, a task scheduling algorithm must take into account task data locality. However, when most of the servers can not be idle soon and network state is good, enforcing high locality has a negative effect on job completion time.

In this paper, we present a data locality driven task scheduling algorithm called BAlance-Reduce(BAR). BAR schedules tasks by taking a global view and adjusts task data locality dynamically according to network state and cluster workload. In a poor network environment, BAR tries its best to enhance data locality. When cluster is overloaded, BAR decreases data locality to make tasks start early. We evaluate BAR by comparing it to other related algorithms. The simulation results show that BAR exhibit an improvement and can deal with a large problem instance in a few seconds.

As a future work, we plan to implement BAR into a production cloud computing system such as Hadoop. In a real-world platform, the network state and the cluster workload change frequently, so it is necessary to update the scheduling strategy by an efficient rescheduling algorithm. The rescheduling algorithm is expected to handle machine failure, task straggler, network anomaly. However, as the scheduler calls rescheduling frequently, the rescheduling algorithm should be low-complexity.

Acknowledgment

This work is supported by National Natural Science Foundation of China under Grants No. 61070161, No. 61070158, No. 61003257 and No. 90912002, National Key Basic Research program of China under Grants No. 2010CB328104, China National Key Technology R&D Program under Grants No. 2010BAI88B03, China Specialized Research Fund for the Doctoral Program of Higher Education under Grants No. 200802860031, Jiangsu Provincial Natural Science Foundation of China under Grants No. BK2008030, China National S&T Major Project under Grants No. 2009ZX03004-004-04, State Key Laboratory of Information Security, Jiangsu Provincial Key Laboratory of Network and Information Security under Grants No. BM2003201, and Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grants No. 93K-9.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] Hadoop. [Online]. Available: http://hadoop.apache.org/

[3] M. Isard *et al.*, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07*. New York, NY, USA: ACM, 2007, pp. 59–72.

[4] A. Thusoo *et al.*, "Data warehousing and analytics infrastructure at facebook," in *SIGMOD '10*. New York, NY, USA: ACM, 2010, pp. 1013–1020.

[5] S. Ghemawat *et al.*, "The google file system," in *SOSP '03*. New York, NY, USA: ACM, 2003, pp. 29–43.

[6] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[7] M. Zaharia *et al.*, "Improving mapreduce performance in heterogeneous environments," in *OSDI'08*. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.

[8] G. Wang and T. S. E. Ng, "The impact of virtualization on network performance of Amazon EC2 data center," ser. INFOCOM'10. Piscataway, NJ, USA: IEEE Press, 2010, pp. 1163–1171.

[9] I. Raicu *et al.*, "The quest for scalable support of data-intensive workloads in distributed systems," ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 207–216.

[10] M. Zaharia *et al.*, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys '10*. New York, NY, USA: ACM, 2010.

[11] M. J. Fischer *et al.*, "Assigning tasks for efficiency in hadoop: extended abstract," in *SPAA '10*. New York, NY, USA: ACM, 2010, pp. 30–39.

[12] T. Kosar and M. Livny, "Stork: Making data placement a first class citizen in the grid," ser. ICDCS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 342–349.

[13] O. Tatebe *et al.*, "Grid datafarm architecture for petascale data intensive computing," ser. CCGRID '02. Washington, DC, USA: IEEE Computer Society, 2002.

[14] I. Raicu *et al.*, "Falkon: a fast and light-weight task execution framework," ser. SC '07. New York, NY, USA: ACM, 2007, pp. 43:1–43:12.

[15] Max-min fairness. [Online]. Available: http://en.wikipedia.org/wiki/Max-min_fairness

[16] M. Isard *et al.*, "Quincy: fair scheduling for distributed computing clusters," ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 261–276.

[17] Hadoop on demand. [Online]. Available: http://hadoop.apache.org/common/docs/r0.18.3/hod.html

[18] S. Seo *et al.*, "HPMR: Prefetching and pre-shuffling in shared mapreduce computation environment," in *CLUSTER '09*. IEEE, 2009, pp. 1–8.

[19] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *SIGMETRICS '09*. New York, NY, USA: ACM, 2009, pp. 299–310.

[20] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[21] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[22] L. R. Ford and D. R. Fulkerson, "Maximal flow through a network," *Canadian J. Math.*, vol. 8, pp. 399–404, 1956.

[23] D. Jiang *et al.*, "The performance of mapreduce: An in-depth study," *PVLDB*, vol. 3, no. 1, pp. 472–483, 2010.