Yale University
Department of Computer Science

P.O. Box 208205
New Haven, CT 06520–8285

Assigning Tasks for Efficiency in Hadoop

Michael J. Fischer
Yale University

Xueyuan Su
Yale University

Yitong Yin
Nanjing University, China

YALEU/DCS/TR-1423
March 31, 2010
(corrected to May 14, 2010)

Assigning Tasks for Efficiency in Hadoop*

Michael J. Fischer
Department of Computer Science
Yale University
michael.fischer@yale.edu

Xueyuan Su†
Department of Computer Science
Yale University
xueyuan.su@yale.edu

Yitong Yin‡
State Key Laboratory for Novel Software Technology
Nanjing University, China
yinyt@nju.edu.cn

Abstract

In recent years Google's MapReduce has emerged as a leading large-scale data processing architecture. Adopted by companies such as Amazon, Facebook, Google, IBM and Yahoo! in daily use, and more recently put in use by several universities, it allows parallel processing of huge volumes of data over cluster of machines. Hadoop is a free Java implementation of MapReduce. In Hadoop, files are split into blocks and replicated and spread over all servers in a network. Each job is also split into many small pieces called tasks. Several tasks are processed on a single server, and a job is not completed until all the assigned tasks are finished. A crucial factor that affects the completion time of a job is the particular assignment of tasks to servers. Given a placement of the input data over servers, one wishes to find the assignment that minimizes the total completion time. In this paper, an idealized Hadoop model is proposed to investigate the Hadoop task assignment problem. It is shown that there is no feasible algorithm to find the optimal Hadoop task assignment unless $\mathcal{P} = \mathcal{NP}$. Assignments that are computed by the round robin algorithm inspired by the current Hadoop scheduler are shown to deviate from optimum by a multiplicative factor in the worst case. A flow-based algorithm is presented that computes assignments that are optimal to within an additive constant.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*; F.1.3 [**Computation by Abstract Devices**]: Complexity Measures and Classes—*reducibility and completeness*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: task assignment, load balancing, NP-completeness, approximation algorithm, MapReduce, Hadoop

*To be presented at SPAA'10, June 13–15, 2010, Thira, Santorini, Greece.

†Supported by the Kempner Fellowship from the Department of Computer Science at Yale University.

‡Supported by the National Science Foundation of China under Grant No. 60721002. This work was done when Yitong Yin was at Yale University.

1 Introduction

1.1 Background

The cloud computing paradigm has recently received significant attention in the media. The cloud is a metaphor for the Internet, which is an abstraction for the complex infrastructure it conceals. Cloud computing refers to both the applications delivered as services over the Internet and the hardware and software that provide such services. It envisions shifting data storage and computing power away from local servers, across the network cloud, and into large clusters of machines hosted by companies such as Amazon, Google, IBM, Microsoft, Yahoo! and so on.

Google's MapReduce [8, 9, 16] parallel computing architecture, for example, splits workload over large clusters of commodity PCs and enables automatic parallelization. By exploiting parallel processing, it provides a software platform that lets one easily write and run applications that process vast amounts of data.

Apache Hadoop [4] is a free Java implementation of MapReduce in the open source software community. It is originally designed to efficiently process large volumes of data by parallel processing over commodity computers in local networks. In academia, researchers have adapted Hadoop to several different architectures. For example, Ranger et al. [18] evaluate MapReduce in multi-core and multi-processor systems, Kruijf et al. [7] implement MapReduce on the Cell B.E. processor architecture, and He et al. [14] propose a MapReduce framework on graphics processors. Many related applications using Hadoop have also been developed to solve various practical problems.

1.2 The MapReduce Framework

A Hadoop system runs on top of a distributed file system, called the Hadoop Distributed File System (HDFS). HDFS usually runs on networked commodity PCs, where data are replicated and locally stored on hard disks of each machine. To store and process huge volume of data sets, HDFS typically uses a block size of 64MB. Therefore, moving computation close to the data is a design goal in the MapReduce framework.

In the MapReduce framework, any application is specified by jobs. A MapReduce job splits the input data into independent blocks, which are processed by the *map* tasks in parallel. Each map task processes a single block[1] consisting of some number of records. Each record in turn consists of a key/value pair. A map task applies the user defined map function to each input key/value pair and produces intermediate key/value pairs. The framework then sorts the intermediate data, and forwards them to the *reduce* tasks via interconnected networks. After receiving all intermediate key/value pairs with the same key, a reduce task executes the user defined reduce function and produces the output data. Finally, these output data are written back to the HDFS.

In such a framework, there is a single server, called the *master*, that keeps track of all jobs in the whole distributed system. The master runs a special process, called the *jobtracker*, that is responsible for task assignment and scheduling for the whole system. For the rest of servers that are called the *slaves*, each of them runs a process called the *tasktracker*. The tasktracker schedules the several tasks assigned to the single server in a way similar to a normal operating system.

[1] Strictly speaking, a map task in Hadoop sometimes processes data that comes from two successive file blocks. This occurs because file blocks do not respect logical record boundaries, so the last logical record processed by a map task might lie partly in the current data block and partly in the succeeding block, requiring the map task to access the succeeding block in order to fetch the tail end of its last logical record.

The map task assignment is a vital part that affects the completion time of the whole job. First, each reduce task cannot begin until it receives the required intermediate data from all finished map tasks. Second, the assignment determines the location of intermediate data and the pattern of the communication traffic. Therefore, some algorithms should be in place to optimize the task assignment.

1.3 Related Work

Since Kuhn [15] proposed the first method for the classic assignment problem in 1955, variations of the assignment problem have been under extensive study in many areas [5]. In the classic assignment problem, there are identical number of jobs and persons. An assignment is a one-to-one mapping from tasks to persons. Each job introduces a cost when it is assigned to a person. Therefore, an optimal assignment minimizes the total cost over all persons.

In the area of parallel and distributed computing, when jobs are processed in parallel over several machines, one is interested in minimizing the maximum processing time of any machines. This problem is sometimes called the minimum makespan scheduling problem. This problem in general is known to be $\mathcal{NP}$-complete [11]. Under the identical-machine model, there are some well-known approximation algorithms. For example, Graham [12] proposed a $(2 - 1/n)$-approximation algorithm in 1966, where n is the total number of machines. Graham [13] proposed another $4/3$-approximation algorithm in 1969. However, under the unrelated-machine model, this problem is known to be APX-hard, both in terms of its offline [17] and online [1, 2] approximability.

As some researchers [3, 4] pointed out, the scheduling mechanisms and polices that assign tasks to servers within the MapReduce framework can have a profound effect on efficiency. An early version of Hadoop uses a simple heuristic algorithm that greedily exploits data locality. Zaharia, Konwinski and Joseph [19] proposed some heuristic refinements based on experimental results.

1.4 Our Contributions

We investigate task assignment in Hadoop. In Section 2, we propose an idealized Hadoop model to evaluate the cost of task assignments. Based on this model, we show in Section 3 that there is no feasible algorithm to find the optimal assignment unless $\mathcal{P} = \mathcal{NP}$. In Section 4, we show that task assignments computed by a simple greedy round-robin algorithm might deviate from the optimum by a multiplicative factor. In Section 5, we present an algorithm that employs maximum flow and increasing threshold techniques to compute task assignments that are optimal to within an additive constant.

2 Problem Formalization

Definition 1 *A* Map-Reduce schema (MR-schema) *is a pair* (T, S)*, where* T *is a set of* tasks *and* S *is a set of* servers*. Let* $m = |T|$ *and* $n = |S|$*. A* task assignment *is a function* $A\colon T \to S$ *that assigns each task* t *to a server* $A(t)$*.*[2] *Let* $\mathcal{A} = \{T \to S\}$ *be the set of all possible task assignments.*

[2]In an MR-schema, it is common that $|T| \geq |S|$. Therefore in this paper, unlike the classic assignment problem where an assignment refers to a *one-to-one* mapping or a *permutation* [5, 15], we instead use the notion of *many-to-one* mapping.

An MR-system *is a triple (T, S, w), where (T, S) is an MR-schema and $w : T \times \mathcal{A} \to \mathbb{Q}^+$ is a* cost function.

Intuitively, $w(t, A)$ is the time to perform task t on server $A(t)$ in the context of the complete assignment A. The motivation for this level of generality is that the time to execute a task t in Hadoop depends not only on the task and the server speed, but also on possible network congestion, which in turn is influenced by the other tasks running on the cluster.

Definition 2 *The* load *of server s under assignment A is defined as $L_s^A = \sum_{t:A(t)=s} w(t, A)$. The* maximum load *under assignment A is defined as $L^A = \max_s L_s^A$. The* total load *under assignment A is defined as $H^A = \sum_s L_s^A$.*

An MR-system models a cloud computer where all servers work in parallel. Tasks assigned to the same server are processed sequentially, whereas tasks assigned to different servers run in parallel. Thus, the total completion time of the cloud under task assignment A is given by the maximum load L^A.

Our notion of an MR-system is very general and admits arbitrary cost functions. To usefully model Hadoop as an MR-system, we need a realistic but simplified cost model.

In Hadoop, the cost of a map task depends frequently on the location of its data. If the data is on the server's local disk, then the cost (execution time) is considerably lower than if the data is located remotely and must be fetched across the network before being processed.

We make several simplifying assumptions. We assume that all tasks and all servers are identical, so that for any particular assignment of tasks to servers, all tasks whose data is locally available take the same amount of time w_{loc}, and all tasks whose data is remote take the same amount of time w_{rem}. However, we do not assume that w_{rem} is constant over all assignments. Rather, we let it grow with the total number of tasks whose data is remote. This reflects the increased data fetch time due to overall network congestion. Thus, $w_{\text{rem}}(r)$ is the cost of each remote task in every assignment with exactly r remote tasks. We assume that $w_{\text{rem}}(r) \geq w_{\text{loc}}$ for all r and that $w_{\text{rem}}(r)$ is (weakly) monotone increasing in r.

We formalize these concepts below. In each of the following, (T, S) is an MR-schema.

Definition 3 *A* data placement *is a relation $\rho \subseteq T \times S$ such that for every task $t \in T$, there exists at least one server $s \in S$ such that $\rho(t, s)$ holds.*

The placement relation describes where the input data blocks are placed. If $\rho(t, s)$ holds, then server s locally stores a replica of the data block that task t needs.

Definition 4 *We represent the placement relation ρ by an unweighted bipartite graph, called the* placement graph. *In the placement graph $G_\rho = ((T, S), E)$, T consists of m task nodes and S consists of n server nodes. There is an edge $(t, s) \in E$ iff $\rho(t, s)$ holds.*

Definition 5 *A* partial assignment *α is a partial function from T to S. We regard a partial assignment as a set of ordered pairs with pairwise distinct first elements, so for partial assignments β and α, $\beta \supseteq \alpha$ means β* extends *α. If $s \in S$, the* restriction of *α to s is the partial assignment $\alpha|_s = \alpha \cap (T \times \{s\})$. Thus, $\alpha|_s$ agrees with α for those tasks that α assigns to s, but all other tasks are unassigned in $\alpha|_s$.*

Definition 6 *Let ρ be a data placement and β be a partial assignment. A task $t \in T$ is* local in β *if $\beta(t)$ is defined and $\rho(t, \beta(t))$. A task $t \in T$ is* remote in α *if $\beta(t)$ is defined and $\neg\rho(t, \beta(t))$. Otherwise t is* unassigned in β. *Let ℓ^β, r^β and u^β be the number of local tasks, remote tasks, and unassigned tasks in β, respectively. For any $s \in S$, let ℓ_s^β be the number of local tasks assigned to s by β. Let $k^\beta = \max_{s \in S} \ell_s^\beta$.*

Definition 7 *Let ρ be a data placement, β be a partial assignment, $w_{\text{loc}} \in \mathbb{Q}^+$, and $w_{\text{rem}} : \mathbb{N} \to \mathbb{Q}^+$ such that $w_{\text{loc}} \le w_{\text{rem}}(0) \le w_{\text{rem}}(1) \le w_{\text{rem}}(2) \ldots$. Let $w_{\text{rem}}^\beta = w_{\text{rem}}(r^\beta + u^\beta)$. The* Hadoop cost function *with parameters ρ, w_{loc}, and $w_{\text{rem}}(\cdot)$ is the function w defined by*

$$w(t, \beta) = \begin{cases} w_{\text{loc}} & \text{if } t \text{ is local in } \beta, \\ w_{\text{rem}}^\beta & \text{otherwise.} \end{cases}$$

We call ρ the placement *of w, and w_{loc} and $w_{\text{rem}}(\cdot)$ the* local *and* remote *costs of w, respectively. Let $K^\beta = k^\beta \cdot w_{\text{loc}}$.*

The definition of remote cost under a partial assignment β is pessimistic. It assumes that tasks not assigned by β will eventually become remote, and each remote task will eventually have cost $w_{\text{rem}}(r^\beta + u^\beta)$. This definition agrees with the definition of remote cost under a complete assignment A, because $u^A = 0$ and thus $w_{\text{rem}}^A = w_{\text{rem}}(r^A + u^A) = w_{\text{rem}}(r^A)$.

Since ρ is encoded by mn bits, w_{loc} is encoded by one rational number, and $w_{\text{rem}}(\cdot)$ is encoded by $m + 1$ rational numbers, the Hadoop cost function $w(\rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$ is encoded by mn bits plus $m + 2$ rational numbers.

Definition 8 *A* Hadoop MR-system (HMR-system) *is the MR-system (T, S, w), where w is the Hadoop cost function with parameters ρ, w_{loc}, and $w_{\text{rem}}(\cdot)$. A HMR-system is defined by $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.*

Problem 1 Hadoop Task Assignment Problem (HTA)

---

1. **Instance:** An HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

Sometimes the cost of running a task on a server only depends on the placement relation and its data locality, but not on the assignment of other tasks.

Definition 9 *A Hadoop cost function w is called* uniform *if $w_{\text{rem}}(r) = c$ for some constant c and all $r \in \mathbb{N}$. A* uniform HMR-system (UHMR-system) *is an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where w is uniform.*

Problem 2 Uniform Hadoop Task Assignment Problem (UHTA)

---

1. **Instance:** A UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

The number of replicas of each data block may be bounded, often by a small number such as 2 or 3.

Definition 10 *Call a placement graph $G = ((T, S), E)$ j-*replica-bounded *if the degree of t is at most j for all $t \in T$. A j-*replica-bounded-UHMR-system *(j-*UHMR-system*) is a UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where G_ρ is j-replica-bounded.*

Problem 3 j-Uniform Hadoop Task Assignment Problem (j-UHTA)

1. **Instance:** A j-UHMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$.
2. **Objective:** Find an assignment A that minimizes L^A.

3 Hardness of Task Assignment

In this section, we analyze the hardness of the various HTA optimization problems by showing the corresponding decision problems to be $\mathcal{NP}$-complete.

3.1 Task Assignment Decision Problems

Definition 11 *Given a server* capacity *k, a task assignment A is k-*feasible *if $L^A \leq k$. An HMR-system is k-*admissible *if there exists a k-feasible task assignment.*

The decision problem corresponding to a class of HMR-systems and capacity k asks whether a given HMR-system in the class is k-admissible. Thus, the k-HTA problem asks about arbitrary HMR-systems, the k-UHTA problem asks about arbitrary UHMR-systems, and the k-j-UHTA problem (which we write (j, k)-UHTA) asks about arbitrary j-UHMR-systems.

3.2 $\mathcal{NP}$-completeness of (2,3)-UHTA

The (2,3)-UHTA problem is a very restricted subclass of the general k-admissibility problem for HMR-systems. In this section, we restrict even further by taking $w_{\text{loc}} = 1$ and $w_{\text{rem}} = 3$. This problem represents a simple scenario where the cost function assumes only the two possible values 1 and 3, each data block has at most 2 replicas, and each server has capacity 3. Despite its obvious simplicity, we show that (2,3)-UHTA is $\mathcal{NP}$-complete. It follows that all of the less restritive decision problems are also $\mathcal{NP}$-complete, and the correponding optimization problems do not have feasible solutions unless $\mathcal{P} = \mathcal{NP}$.

Theorem 3.1 $(2, 3)$-UHTA *with costs $w_{\text{loc}} = 1$ and $w_{\text{rem}} = 3$ is $\mathcal{NP}$-complete.*

The proof method is to construct a polynomial-time reduction from 3SAT to (2,3)-UHTA. Let $\mathcal{G}$ be the set of all 2-replica-bounded placement graphs. Given $G_\rho \in \mathcal{G}$, we define the HMR-system $\mathcal{M}_G = (T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$, where $w_{\text{loc}} = 1$ and $w_{\text{rem}}(r) = 3$ for all r. We say that G is 3-*admissible* if $\mathcal{M}_G$ is 3-admissible. We construct a polynomial-time computable mapping

$f : \text{3CNF} \rightarrow \mathcal{G}$, and show that a 3CNF formula ϕ is satisfiable iff $f(\phi)$ is 3-admissible. We shorten "3-admissible" to "admissible" in the following discussion.

We first describe the construction of f. Let $\phi = C_1 \wedge C_2 \cdots \wedge C_\alpha$ be a 3CNF formula, where each $C_u = (l_{u1} \vee l_{u2} \vee l_{u3})$ is a clause and each l_{uv} is a literal. Let $x_1, \cdots, x_\beta$ be the variables that appear in ϕ. Therefore, ϕ contains exactly 3α instances of literals, each of which is either x_i or $\neg x_i$, where $i \in [1, \beta]$.[3] Let ω be the maximum number of occurrences of any literal in ϕ. Table 1 summarizes the parameters of ϕ.

Table 1: Parameters of the 3CNF ϕ

clauses (C_u)	α	variables (v_i)	β
literals (l_{uv})	3α	max-occur of any literal	ω

For example, in $\phi = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_4 \vee \neg x_6)$, we have $\alpha = 3$, $\beta = 6$, and $\omega = 2$ since x_1 occurs twice.

Given ϕ, we construct the corresponding placement graph G which comprises several disjoint copies of the three types of gadget described below, connected together with additional edges.

The first type of gadget is called a *clause gadget*. Each clause gadget u contains a *clause server* C_u, three *literal tasks* l_{u1}, l_{u2}, l_{u3} and an *auxiliary task* a_u. There is an edge between each of these tasks and the clause server. Since ϕ contains α clauses, G contains α clause gadgets. Thus, G contains α clause servers, 3α literal tasks and α auxiliary tasks. Figure 1 describes the structure of the u-th clause gadget. We use circles and boxes to represent tasks and servers, respectively.

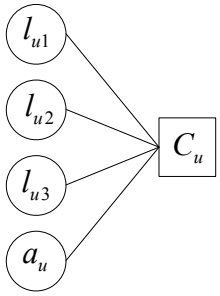

Figure 1: The structure of the u-th clause gadget.

The second type of gadget is called a *variable gadget*. Each variable gadget contains 2ω *ring servers* placed around a circle. Let $R_j^{(i)}$ denote the server at position $j \in [1, 2\omega]$ in ring i. Define the set $\mathcal{T}_i$ to be the servers in odd-numbered positions. Similarly, define the set $\mathcal{F}_i$ to be the servers in even-numbered positions. Between each pair of ring servers $R_j^{(i)}$ and $R_{j+1}^{(i)}$, we place a *ring task* $r_j^{(i)}$ connected to its two neighboring servers. To complete the circle, $r_{2\omega}^{(i)}$ is connected to $R_{2\omega}^{(i)}$ and $R_1^{(i)}$. There are also ω *variable tasks* $v_j^{(i)} : j \in [1, \omega]$ in ring i, but they do not connect to any ring server. Since ϕ contains β variables, G contains β variable gadgets. Thus, G contains $2\beta\omega$ ring servers, $2\beta\omega$ ring tasks and $\beta\omega$ variable tasks. Figure 2 describes the structure of the i-th variable gadget.

[3]The notation [a,b] in our discussion represents the set of integers $\{a, a+1, \cdots, b-1, b\}$.

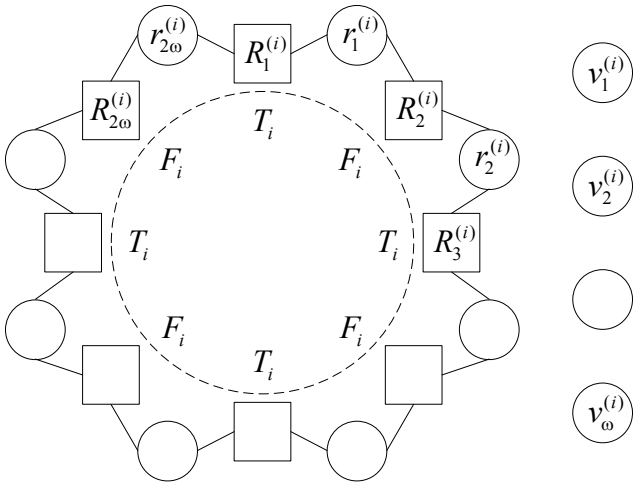

Figure 2: The structure of the i-th variable gadget.

The third type of gadget is called a *sink gadget*. The sink gadget contains a *sink server* P and three *sink tasks* p_1, p_2, p_3. Each sink task is connected to the sink server. G only contains one sink gadget. Figure 3 describes the structure of the sink gadget.

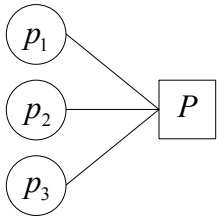

Figure 3: The structure of the sink gadget.

There are also some inter-gadget edges in G. We connect each variable task $v_j^{(i)}$ to the sink server P. We also connect each literal task l_{uv} to a unique ring server $R_j^{(i)}$. To be more precise, if literal l_{uv} is the j-th occurrence of x_i in ϕ, connect the literal task l_{uv} to ring server $R_{2j-1}^{(i)} \in \mathcal{T}_i$; if literal l_{uv} is the j-th occurrence of $\neg x_i$ in ϕ, connect the literal task l_{uv} to ring server $R_{2j}^{(i)} \in \mathcal{F}_i$. These inter-gadget edges complete the graph G. Table 2 summarizes the parameters of G.

Table 2: Parameters of the HMR-graph G

clause server C_u	α	literal task l_{uv}	3α
auxiliary task a_u	α	ring server $R_j^{(i)}$	$2\beta\omega$
ring task $r_j^{(i)}$	$2\beta\omega$	variable task $v_j^{(i)}$	$\beta\omega$
sink server P	1	sink task p_j	3

Lemma 3.2 *For any $\phi \in$ 3CNF, the graph $f(\phi)$ is 2-replica-bounded.*

Proof. We count the number of edges from each task node in $f(\phi)$. Each clause task has 2 edges, each auxiliary task has 1 edge, each ring task has 2 edges, each variable task has 1 edge, and each sink task has 1 edge. Therefore, $f(\phi)$ is 2-replica-bounded. □

The following lemma is immediate.

Lemma 3.3 *The mapping $f : 3\text{CNF} \to \mathcal{G}$ is polynomial-time computable.*

Lemma 3.4 *If ϕ is satisfiable, then $G = f(\phi)$ is admissible.*

Proof.

Let σ be a satisfying truth assignment for ϕ, and we construct a feasible assignment A in $G = f(\phi)$. First of all, assign each sink task to the sink server, i.e., let $A(p_i) = P$ for all $i \in [1, 3]$. Then assign each auxiliary task a_u to the clause server C_u, i.e., let $A(a_u) = C_u$ for all $u \in [1, \alpha]$. If $\sigma(x_i) = \mathit{true}$, then assign ring tasks $r_j^{(i)} : j \in [1, 2\omega]$ to ring servers in $\mathcal{T}_i$, variable tasks $v_j^{(i)} : j \in [1, \omega]$ to ring servers in $\mathcal{F}_i$. If $\sigma(x_i) = \mathit{false}$, then assign ring tasks $r_j^{(i)} : j \in [1, 2\omega]$ to ring servers in $\mathcal{F}_i$, variable tasks $v_j^{(i)} : j \in [1, \omega]$ to ring servers in $\mathcal{T}_i$. If literal $l_{uv} = x_i$ and $\sigma(x_i) = \mathit{true}$, then assign task l_{uv} to its local ring server in $\mathcal{T}_i$. If literal $l_{uv} = \neg x_i$ and $\sigma(x_i) = \mathit{false}$, then assign task l_{uv} to its local ring server in $\mathcal{F}_i$. Otherwise, assign task l_{uv} to its local clause server C_u.

We then check this task assignment is feasible. Each ring server is assigned either at most three local tasks (two ring tasks and one literal task), or one remote variable task. In either case, the load does not exceed the capacity 3. The number of tasks assigned to each clause server C_u is exactly the number of false literals in C_u under σ plus one (the auxiliary task), and each task is local to C_u. Thus, the load is at most 3. The sink server is assigned three local sink tasks and the load is exactly 3. Therefore, all constraints are satisfied and A is feasible. This completes the proof of Lemma 3.4. □

The proof of the converse of Lemma 3.4 is more involved. The method is given a feasible assignment A in $G = f(\phi)$, we first construct a feasible assignment B in G such that $B(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$. Then we remove the sink tasks and the sink server from further consideration and consider the resulting graph G'. After that, we partition G' into two subgraphs, and construct a feasible assignment B' such that no tasks from one partition are remotely assigned to servers in the other partition. This step involves a case analysis. Finally, a natural way of constructing the satisfying truth assignment for ϕ follows.

Lemma 3.5 *Let A be a feasible task assignment. Then there exists a feasible task assignment B such that $B(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$.*

Proof. When A satisfies that $A(t) \neq P$ for all $t \in T - \{p_1, p_2, p_3\}$, let $B = A$. Otherwise, assume there exists a task t' such that $A(t') = P$ and $t' \in T - \{p_1, p_2, p_3\}$. Since the capacity of P is 3, there is at least one sink task, say p_1, is not assigned to P. Let $A(p_1) = Q$. Since $\rho(p_1, Q)$ does not hold, Q has only been assigned p_1 and $L_Q^A = 3$. Let $B(p_1) = P$ and $B(t') = Q$. Repeat the same process for all tasks other than p_1, p_2, p_3 that are assigned to P in A. Then let $B(t) = A(t)$ for the remaining tasks $t \in T$. To see B is feasible, note that $L_s^B \leq L_s^A \leq 3$ for all servers $s \in S$. □

Let G' be the subgraph induced by $(T - \{p_1, p_2, p_3\}, S - \{P\}) = (T', S')$. We have the following lemma.

Lemma 3.6 *Let A be a feasible task assignment in G. Then there exists a feasible task assignment A' in G'.*

Proof. Given A, Lemma 3.5 tells us that there exists another feasible assignment B in G such that $B(t) \neq P$ for all $t \in T'$. Let $A'(t) = B(t)$ for all $t \in T'$. Then A' is an assignment in G' since $A'(t) \in S - \{P\}$ for all $t \in T'$. To see A' is feasible, note that $L_s^{A'} \leq L_s^B \leq 3$ for all servers $s \in S'$. □

We further partition G' into two subgraphs G_C and G_R. G_C is induced by nodes $\{C_u : u \in [1, \alpha]\} \cup \{a_u : u \in [1, \alpha]\} \cup \{l_{uv} : u \in [1, \alpha], v \in [1, 3]\}$ and G_R is induced by nodes $\{R_j^{(i)} : i \in [1, \beta], j \in [1, 2\omega]\} \cup \{r_j^{(i)} : i \in [1, \beta], j \in [1, 2\omega]\} \cup \{v_j^{(i)} : i \in [1, \beta], j \in [1, \omega]\}$. In other words, G_C consists of all clause gadgets while G_R consists of all variable gadgets.

If a task in one partition is remotely assigned to a server in the other partition, we call this task a *cross-boundary* task. Let n_c^A be the number of cross-boundary tasks that are in G_C and assigned to servers in G_R by A, n_r^A be the number of cross-boundary tasks that are in G_R and assigned to servers in G_C by A. We have the following lemmas.

Lemma 3.7 *Let A be a feasible assignment in G' such that $n_c^A > 0$ and $n_r^A > 0$. Then there exist a feasible assignment B in G' such that one of n_c^B and n_r^B equals $|n_c^A - n_r^A|$ and the other one equals 0.*

Proof. Assume $t_i \in G_C$, $s_i \in G_R$ and $A(t_i) = s_i$; $t_i' \in G_R$, $s_i' \in G_C$ and $A(t_i') = s_i'$. Then each of s_i and s_i' is assigned one remote task. Let $B(t_i) = s_i'$ and $B(t_i') = s_i$, and then $L_{s_i}^B \leq L_{s_i}^A = 3$ and $L_{s_i'}^B \leq L_{s_i'}^A = 3$. This process decreases n_c and n_r each by one, and the resulting assignment is also feasible. Repeat the same process until the smaller one of n_c and n_r becomes 0. Then let $B(t) = A(t)$ for all the remaining tasks $t \in T'$. It is obvious that B is feasible, and one of n_c^B and n_r^B equals $|n_c^A - n_r^A|$ and the other one equals 0. □

Lemma 3.8 *Let A be a feasible assignment in G' such that $n_c^A = 0$. Then $n_r^A = 0$.*

Proof. For the sake of contradiction, assume $t_i \in G_R$, $s_i \in G_C$ and $A(t_i) = s_i$. For each server $s_j \in G_C$, there is one auxiliary task $a_u : u \in [1, \alpha]$ such that $\rho(a_u, s_j)$ holds. Since $w_{\text{loc}} = 1$ and $w_{\text{rem}} = 3$, if A is feasible then $A(a_u) \neq A(a_v)$ for $u \neq v$. Since there are α auxiliary tasks and α servers in G_C, one server is assigned exactly one auxiliary task. Since $A(t_i) = s_i$, $L_{s_i}^A \geq 1 + 3 > 3$, contradicting the fact that A is feasible. Therefore, there is no $t_i \in G_R$ and $s_i \in G_C$ such that $A(t_i) = s_i$. Thus, $n_r^A = 0$. □

Lemma 3.9 *Let A be a feasible assignment in G' such that $n_r^A = 0$. Then $n_c^A = 0$.*

Proof. For the sake of contradiction, assume $t_i \in G_C$, $s_i \in G_R$ and $A(t_i) = s_i$. Let k_0, k_1, k_2, k_3 denote the number of ring servers filled to load $0, 1, 2, 3$, respectively. From the total number of servers in G_R, we have

$$k_0 + k_1 + k_2 + k_3 = 2\beta\omega \tag{1}$$

Similarly, from the total number of tasks in G_R, we have

$$0 \cdot k_0 + 1 \cdot k_1 + 2 \cdot k_2 + 1 \cdot k_3 = 3\beta\omega \tag{2}$$

Subtracting (1) from (2) gives $k_2 = \beta\omega + k_0$. Assigning both neighboring ring tasks to the same ring server fills it to load 2. Since there are only $2\beta\omega$ ring servers, we have $k_2 \leq \beta\omega$. Hence, $k_0 = 0$ and $k_2 = \beta\omega$. This implies that all ring tasks are assigned to ring servers in alternating positions in each ring.

There are $\beta\omega$ remaining ring servers and $\beta\omega$ variable tasks. Therefore, a variable task is remotely assigned to one of the remaining ring servers by A.

Now consider the server s_i that has been remotely assigned $t_i \in G_C$. If it is assigned two ring tasks, its load is $L^A_{s_i} = 2 + 3 > 3$. If it is assigned one variable task, its load is $L^A_{s_i} = 3 + 3 > 3$. A is not feasible in either case. Therefore, there is no $t_i \in G_C$ and $s_i \in G_R$ such that $A(t_i) = s_i$. Thus, $n^A_c = 0$. □

Now we prove the following Lemma.

Lemma 3.10 *If $G = f(\phi)$ is admissible, then ϕ is satisfiable.*

Proof. Given feasible task assignment A in $G = f(\phi)$, we construct the satisfying truth assignment σ for ϕ. From Lemmas 3.6, 3.7, 3.8 and 3.9, we construct a feasible assignment B in G', such that $n^B_c = n^B_r = 0$, and in each variable gadget i, either servers in $\mathcal{T}_i$ or servers in $\mathcal{F}_i$ are saturated by variable tasks. If ring servers in $\mathcal{F}_i$ are saturated by variable tasks, let $\sigma(x_i) = true$. If ring servers in $\mathcal{T}_i$ are saturated by variable tasks, let $\sigma(x_i) = false$.

To check that this truth assignment is a satisfying assignment, note that for the three literal tasks l_{u1}, l_{u2}, l_{u3}, at most two of them are assigned to the clause server C_u. There must be one literal task, say l_{uv}, that is locally assigned to a ring server. In this case, $\sigma(l_{uv}) = true$ and thus the clause $\sigma(C_u) = true$. This fact holds for all clauses and thus indicates that $\sigma(\phi) = \sigma(\bigwedge C_u) = true$. This completes the proof of Lemma 3.10. □

Finally we prove the main theorem.

Proof of Theorem 3.1: Lemmas 3.3, 3.4 and 3.10 establish that 3SAT $\leq_p$ (2,3)-UHTA via f. Therefore, (2,3)-UHTA is $\mathcal{NP}$-hard. It is easy to see that (2,3)-UHTA $\in \mathcal{NP}$ because in time $O(mn)$ a nondeterministic Turing machine could guess the assignment and accept iff the maximum load under the assignment does not exceed 3. Therefore, $(2,3)$-UHTA is $\mathcal{NP}$-complete. □

4 A Round Robin Algorithm

In this section, we analyze a simple round robin algorithm for the UHTA problem. Algorithm 1 is inspired by the Hadoop scheduler algorithm. It scans over each server in a round robin fashion. When assigning a new task to a server, Algorithm 1 tries heuristically to exploit data locality. Since we have not specified the order of assigned tasks, Algorithm 1 may produce many possible outputs (assignments).

Algorithm 1 is analogous to the Hadoop scheduler algorithm up to core version 0.19. There are three differences, though. First, the Hadoop algorithm assumes three kinds of placement: data-local,

Algorithm 1 The round robin algorithm exploring locality.

input: a set of unassigned tasks T, a list of servers $\{s_1, s_2, \cdots, s_n\}$, a placement relation ρ
define $i \leftarrow 1$ as an index variable
define A as an assignment
$A(t) = \perp$ (task t is unassigned) for all t
while exists unassigned task **do**
  if exists unassigned task t such that $\rho(t, s_i)$ holds **then**
    update A by assigning $A(t) = s_i$
  else
    pick any unassigned task t', update A by assigning $A(t') = s_i$
  end if
  $i \leftarrow (i \bmod n) + 1$
end while
output: assignment A

rack-local and rack-remote, whereas Algorithm 1 assumes only two: local and remote. Second, the Hadoop scheduler works incrementally rather than assigning all tasks initially. Last, the Hadoop algorithm is deterministic, whereas Algorithm 1 is nondeterministic.

Theorem 4.1 *If $w_{\text{rem}} > w_{\text{loc}}$, increasing the number of data block replicas may increase the maximum load of the assignment computed by Algorithm 1.*

Proof. The number of edges in the placement graph is equal to the number of data block replicas, and thus adding a new edge in the placement graph is equivalent to adding a new replica in the system. Consider the simple placement graph G where $m = n$, and there is an edge between task t_i and s_i for all $1 \leq i \leq n$. Running Algorithm 1 gives an assignment A in which task t_i is assigned to s_i for all $1 \leq i \leq n$, and thus $L^A = w_{\text{loc}}$. Now we add one edge between task t_n and server s_1. We run Algorithm 1 on this new placement graph G' to get assignment A'. It might assign task t_n to server s_1 in the first step. Following that, it assigns t_i to s_i for $2 \leq i \leq n-1$, and it finally assigns t_1 to s_n. Since t_1 is remote to s_n, this gives $L^{A'} = w_{\text{rem}}$. Therefore $L^{A'} > L^A$. □

Theorem 4.1 indicates that increasing the number of data block replicas is not always beneficial for Algorithm 1. In the remaining part of this section, we show that the assignments computed by Algorithm 1 might deviate from the optimum by a multiplicative factor. In the following, let O be an assignment that minimizes L^O.

Theorem 4.2 *Let A be an assignment computed by Algorithm 1. Then $L^A \leq (w_{\text{rem}}/w_{\text{loc}}) \cdot L^O$.*

Proof. On the one hand, pigeonhole principle says there is a server assigned at least $\lceil m/n \rceil$ tasks. Since the cost of each task is at least w_{loc}, the load of this server is at least $\lceil m/n \rceil \cdot w_{\text{loc}}$. Thus, $L^O \geq \lceil m/n \rceil \cdot w_{\text{loc}}$. On the other hand, Algorithm 1 runs in a round robin fashion where one task is assigned at a time. Therefore, the number of tasks assigned to each server is at most $\lceil m/n \rceil$. Since the cost of each task is at most w_{rem}, the load of a server is at most $\lceil m/n \rceil \cdot w_{\text{rem}}$. Thus, $L^A \leq \lceil m/n \rceil \cdot w_{\text{rem}}$. Combining the two, we have $L^A \leq (w_{\text{rem}}/w_{\text{loc}}) \cdot L^O$. □

Theorem 4.3 *Let T and S be such that $m \leq n(n-2)$. There exist a placement ρ and an assignment A such that A is a possible output of Algorithm 1, $L^A \geq \lfloor m/n \rfloor \cdot w_{\text{rem}}$, and $L^O = \lceil m/n \rceil \cdot w_{\text{loc}}$.*

Proof. We prove the theorem by constructing a placement graph G_ρ. Partition the set T of tasks into n disjoint subsets $T_i : 1 \leq i \leq n$, such that $\lceil m/n \rceil \geq |T_i| \geq |T_j| \geq \lfloor m/n \rfloor$ for all $1 \leq i \leq j \leq n$. Now in the placement graph G_ρ, connect tasks in T_i to server s_i, for all $1 \leq i \leq n$. These set of edges guarantee that $L^O = \lceil m/n \rceil \cdot w_{\text{loc}}$. We then connect each task in T_n to a different server in the subset $S' = \{s_1, s_2, \cdots, s_{n-1}\}$. Since $m \leq n(n-2)$, we have $\lceil m/n \rceil \leq m/n + 1 \leq n - 1$, which guarantees $|S'| \geq |T_n|$. This completes the placement graph G_ρ. Now run Algorithm 1 on G_ρ. There is a possible output A where tasks in T_n are assigned to servers in S'. In that case, all tasks that are local to server s_n are assigned elsewhere, and thus s_n is assigned remote tasks. Since s_n is assigned at least $\lfloor m/n \rfloor$ tasks, this gives $L^A \geq \lfloor m/n \rfloor \cdot w_{\text{rem}}$. □

When $n \mid m$, the lower bound in Theorem 4.3 matches the upper bound in Theorem 4.2.

5 A Flow-based Algorithm

Theorem 3.1 shows that the problem of computing an optimal task assignment for the HTA problem is $\mathcal{NP}$-complete. Nevertheless, it is feasible to find task assignments whose load is at most an additive constant greater than the optimal load. We present such an algorithm in this section.

For two partial assignments α and β such that $\beta \supseteq \alpha$, we define a new notation called *virtual load from* α below.

Definition 12 *For any task t and partial assignment β that extends α, let*

$$v^\alpha(t, \beta) = \begin{cases} w_{\text{loc}} & \text{if } t \text{ is local in } \beta, \\ w^\alpha_{\text{rem}} & \text{otherwise.} \end{cases}$$

The virtual load of server s under β from α *is* $V_s^{\beta,\alpha} = \sum_{t:\beta(t)=s} v^\alpha(t, \beta)$. *The* maximum virtual load under β from α *is* $V^{\beta,\alpha} = \max_{s \in S} V_s^{\beta,\alpha}$.

Thus, v assumes pessimistically that tasks not assigned by β will eventually become remote, and each remote task will eventully have cost w^α_{rem}. When α is clear from context, we omit α and write $v(t, \beta)$, V_s^β and V^β, respectively. Note that $v^\alpha(t, \alpha) = w(t, \alpha)$ as in Definition 7.

Algorithm 2 works iteratively to produce a sequence of assignments and then outputs the best one, i.e., the one of least maximum server load. The iteration is controlled by an integer variable τ which is initialized to 1 and incremented on each iteration. Each iteration consists of two phases, *max-cover* and *bal-assign*:

- *Max-cover*: Given as input a placement graph G_ρ, an integer value τ, and a partial assignment α, max-cover returns a partial assignment α' of a subset T' of tasks, such that α' assigns no server more than τ tasks, every task in T' is local in α', and $|T'|$ is maximized over all such assignments. Thus, α' makes as many tasks local as is possible without assigning more than τ tasks to any one server. The name "max-cover" follows the intuition that we are actually trying to "cover" as many tasks as possible by their local servers, subject to the constraint that no server is assigned more than τ tasks.

- *Bal-assign*: Given as input a set of tasks T, a set of servers S, a partial assignment α computed by max-cover, and a cost function w, bal-assign uses a simple greedy algorithm to extend

α to a complete assignment B by repeatedly choosing a server with minimal virtual load and assigning some unassigned task to it. This continues until all tasks are assigned. It thus generates a sequence of partial assignments $\alpha = \alpha_0 \subseteq \alpha_1 \subseteq \cdots \subseteq \alpha_u = B$, where $u = u^\alpha$. Every task t assigned in bal-assign contributes $v^\alpha(t, B) \leq w^\alpha_{\text{rem}}$ to the virtual load of the server that it is assigned to. At the end, $w^B_{\text{rem}} \leq w^\alpha_{\text{rem}}$, and equality holds only when $r^B = r^\alpha + u^\alpha$.

The astute reader might feel that it is intellectually attractive to use real server load as the criterion to choose servers in bal-assign because it embeds more accurate information. We do not know if this change ever results in a better assignment. We do know that it may require more computation. Whenever a local task is assigned, $r + u$ decreases by 1, so the remote cost $w_{\text{rem}}(r + u)$ may also decrease. If it does, the loads of all servers that have been assigned remote tasks must be recomputed. In the current version of the algorithm, we do not need to update virtual load when a local task is assigned because the virtual cost of remote tasks never changes in the course of bal-assign.

Algorithm 2 A flow-based algorithm for HTA.

1: **input:** an HMR-system $(T, S, \rho, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
2: **define** A, B as assignments
3: **define** α as a partial assignment
4: $\alpha(t) = \perp$ (task t is unassigned) for all t
5: **for** $\tau = 1$ to m **do**
6: $\quad \alpha \leftarrow$ max-cover(G_ρ, τ, α)
7: $\quad B \leftarrow$ bal-assign$(T, S, \alpha, w_{\text{loc}}, w_{\text{rem}}(\cdot))$
8: **end for**
9: **set** A equal to a B with least maximum load
10: **output:** assignment A

5.1 Algorithm Description

We describe Algorithm 2 in greater detail here.

5.1.1 Max-cover

Max-cover (line 6 of Algorithm 2) augments the partial assignment $\alpha^{\tau-1}$ computed by the previous iteration to produce α^τ. (We define α^0 to be the empty partial assignment.) Thus, $\alpha^\tau \supseteq \alpha^{\tau-1}$, and α^τ maximizes the total number of local tasks assigned subject to the constraint that no server is assigned more than τ tasks in all.

The core of the max-cover phase is an augmenting path algorithm by Ford and Fulkerson [10]. The Ford-Fulkerson algorithm takes as input a network with edge capacities and an existing network flow, and outputs a maximum flow that respects the capacity constraints. A fact about this algorithm is well-known [6, 10].

Fact 5.1 *Given a flow network with integral capacities and an initial integral s-t flow f, the Ford-Fulkerson algorithm computes an integral maximum s-t flow f' in time $O(|E| \cdot (|f'| - |f|))$, where $|E|$ is the number of edges in the network and $|f|$ is the value of the flow f, i.e., the amount of flow passing from the source to the sink.*

During the max-cover phase at iteration τ, the input placement graph G_ρ is first converted to a corresponding flow network G'_ρ. G'_ρ includes all nodes in G_ρ and an extra source u and an extra sink v. In G'_ρ, there is an edge (u,t) for all $t \in T$ and an edge (s,v) for all $s \in S$. All of the original edges (t,s) in G_ρ remain in G'_ρ. The edge capacity is defined as follows: edge (s,v) has capacity τ for all $s \in S$, while all the other edges have capacity 1. Therefore, for any pair of (t,s), if there is a flow through the path $u \to t \to s \to v$, the value of this flow is no greater than 1. Then the input partial assignment α is converted into a network flow f_α as follows: if task t is assigned to server s in the partial assignment α, assign one unit of flow through the path $u \to t \to s \to v$.

The Ford-Fulkerson algorithm is then run on graph G'_ρ with flow f_α to find a maximum flow f'_α. From Fact 5.1, we know that the Ford-Fulkerson algorithm takes time $O(|E| \cdot (|f'_\alpha| - |f_\alpha|))$ in this iteration. This output flow f'_α at iteration τ will act as the input flow to the Ford-Fulkerson algorithm at iteration $\tau+1$. The flow network at iteration $\tau+1$ is the same as the one at iteration τ except that each edge (s,v) has capacity $\tau+1$ for all $s \in S$. This incremental use of Ford-Fulkerson algorithm in successive iterations helps reduce the time complexity of the whole algorithm.

At the end of the max-cover phase, the augmented flow f'_α is converted back into a partial assignment α'. If there is one unit of flow through the path $u \to t \to s \to v$ in f'_α, we assign task t is to server s in α'. This conversion from a network flow to a partial assignment can always be done, because the flow is integral and all edges between tasks and servers have capacity 1. Therefore, there is a one-to-one correspondence between a unit flow through the path $u \to t \to s \to v$ and the assignment of task t to its local server s. It follows that $|f'_\alpha| = \ell^{\alpha'}$. By Fact 5.1, the Ford-Fulkerson algorithm computes a maximum flow that respects the capacity constraint τ. Thus, the following lemma is immediate.

Lemma 5.2 *Let α^τ be the partial assignment computed by max-cover at iteration τ, and β be any partial assignment such that $k^\beta \le \tau$. Then $\ell^{\alpha^\tau} \ge \ell^\beta$.*

5.1.2 Bal-assign

Definition 13 *Let β and β' be partial assignments, t a task and s a server. We say that $\beta \xrightarrow{t:s} \beta'$* is a step that assigns t to s *if t is unassigned in β and $\beta' = \beta \cup \{(t,s)\}$. We say $\beta \to \beta'$* is a step, *if $\beta \xrightarrow{t:s} \beta'$ for some t and s.*

A sequence of steps $\alpha = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_u$ is a trace *if for each $i \in [1,u]$, if $\alpha_{i-1} \xrightarrow{t:s} \alpha_i$ is a step, then $V_s^{\alpha_{i-1},\alpha} \le V_{s'}^{\alpha_{i-1},\alpha}$ for all $s' \ne s$.*

Given two partial assignments α_{i-1} and α_i in a trace such that $\alpha_{i-1} \xrightarrow{t:s} \alpha_i$, it follows that

$$\begin{aligned} V_s^{\alpha_i,\alpha} &\le V_s^{\alpha_{i-1},\alpha} + w_{\text{rem}}^\alpha \\ V_s^{\alpha_i,\alpha} &= V_{s'}^{\alpha_{i-1},\alpha} \text{ for all } s' \ne s \end{aligned}$$

The following lemma is immediate.

Lemma 5.3 *Let $u = u^{\alpha^\tau}$ and $\alpha^\tau = \alpha_0^\tau \subseteq \alpha_1^\tau \subseteq \cdots \subseteq \alpha_u^\tau$ be a sequence of partial assignments generated by bal-assign at iteration τ. This sequence is a trace that ends in a complete assignment $B^\tau = \alpha_u^\tau$.*

5.2 Main Result

It is obvious that Algorithm 2 is optimal for $n = 1$ since only one assignment is possible. Now we show that for $n \geq 2$, Algorithm 2 computes, in polynomial time, assignments that are optimal to within an additive constant. The result is formally stated as Theorem 5.4.

Theorem 5.4 *Let $n \geq 2$. Given an HMR-system with m tasks and n servers, Algorithm 2 computes an assignment A in time $O(m^2n)$ such that $L^A \leq L^O + \left(1 - \frac{1}{n-1}\right) \cdot w^O_{\text{rem}}$.*

Lemma 5.5 *Algorithm 2 runs in time $O(m^2n)$.*

Proof. By Fact 5.1, we know that the Ford-Fulkerson algorithm takes time $O(|E|\cdot|\Delta_f|)$ to augment the network flow by $|\Delta_f|$. At iteration $\tau = 1$, max-cover takes time $O(|E| \cdot |f_1|)$, where $|f_1| \leq n$. Then at iteration $\tau = 2$, max-cover takes time $O(|E| \cdot (|f_2| - |f_1|))$, where $|f_2| \leq 2n$. The same process is repeated until $|f_m| = m$. The total running time of max-cover for all iterations thus adds up to $O(|E|\cdot(|f_1|+|f_2|-|f_1|+|f_3|-|f_2|+\cdots+|f_m|)) = O(|E|\cdot|f_m|) = O(|E|\cdot m) = O(m^2n)$.

We implement the greedy algorithm in the bal-assign phase with a priority queue. Since there are n servers, each operation of the priority queue takes $O(\log n)$ time. During the bal-assign phase at each iteration, at most m tasks need to be assigned. This takes time $O(m \log n)$. The total running time of bal-assign for all iterations is thus $O(m^2 \log n)$.

Combining the running time of the two phases for all iterations gives time complexity $O(m^2n)$. □

Lemma 5.5 suggests the max-cover phase is the main contributor to the time complexity of Algorithm 2. However, in a typical Hadoop system, the number of replicas for each data block is a small constant, say 2 or 3. Then the degree of each $t \in G$ is bounded by this constant. In this case, the placement graph G is sparse and $|E| = O(m + n)$. As a result, max-cover runs in time $O(m(m + n))$. Therefore the bal-assign phase might become the main contributor to the time complexity.

5.2.1 Properties of optimal assignments

In order to prove the approximation bound, we first establish some properties of optimal assignments.

Definition 14 *Given an HMR-system, let $\mathcal{O}$ be the set of all optimal assignments, i.e., those that minimize the maximum load. Let $r_{\min} = \min\{r^A \mid A \in \mathcal{O}\}$ and let $\mathcal{O}_1 = \{O \in \mathcal{O} \mid r^O = r_{\min}\}$.*

Lemma 5.6 *Let $O \in \mathcal{O}_1$. If $\ell^O_s = k^O$, then $r^O_s = 0$ and $L^O_s = K^O$.*

Proof. Let $\ell^O_s = k^O$ for some server s. Assume to the contrary that $r^O_s \geq 1$. Then $L^O_s \geq K^O + w^O_{\text{rem}}$. Let t be a remote task assigned to s by O. By definition 3, $\rho(t, s')$ holds for at least one server $s' \neq s$.

Case 1: s' has at least one remote task t'. Then move t' to s and t to s'. This results in another assignment B. B is still optimal because $L^B_s \leq L^O_s$, $L^B_{s'} \leq L^O_{s'}$, and $L^B_{s''} = L^O_{s''}$ for any other server $s'' \in S - \{s, s'\}$.

Case 2: s' has only local tasks. By the definition of k^O, s' has at most k^O local tasks assigned by O. Then move t to s'. This results in another assignment B. B is still optimal because $L^B_s < L^O_s$, $L^B_{s'} = K^O + w_{\text{loc}} \leq K^O + w_{\text{rem}} \leq L^O_s$, and $L^B_{s''} = L^O_{s''}$ for any other server $s'' \in S - \{s, s'\}$.

In either case, we have shown the new assignment is in $\mathcal{O}$. However, since t becomes local in the new assignment, fewer remote tasks are assigned than in O. This contradicts that $O \in \mathcal{O}_1$. Thus, s is assigned no remote tasks, so $L^O_s = k^O w_{\text{loc}} = K^O$. □

Definition 15 *Let* $O \in \mathcal{O}_1$. *Define* $M^O = \frac{H^O - K^O}{n-1}$.

Lemma 5.7 $L^O \geq M^O$.

Proof. Let s_1 be a server of maximal local load in O, so $k^O_{s_1} = k^O$. Let $S_2 = S - \{s_1\}$. By Lemma 5.6, $L^O_{s_1} = K^O$. The total load on S_2 is $\sum_{s \in S_2} L^O_s = H^O - K^O$, so the average load on S_2 is M^O. Hence, $L^O \geq \max_{s \in S_2} L^O_s \geq \text{avg}_{s \in S_2} L^O_s = M^O$. □

5.2.2 Analyzing the algorithm

Assume throughout this section that $O \in \mathcal{O}_1$ and $\alpha = \alpha_0 \to \alpha_1 \to \ldots \to \alpha_u = B$ is a trace generated by iteration $\tau = k^O$ of the algorithm. Virtual loads are all based on α, so we generally omit explicit mention of α in the superscripts of v and V.

Lemma 5.8 $w_{\text{loc}} \leq w^B_{\text{rem}} \leq w^\alpha_{\text{rem}} \leq w^O_{\text{rem}}$.

Proof. $w_{\text{loc}} \leq w^B_{\text{rem}}$ follows from the definition of a Hadoop cost function. Because $B \supseteq \alpha$, $r^B \leq r^\alpha + u^\alpha$. By Lemma 5.2, $\ell^\alpha \geq \ell^O$, so $r^\alpha + u^\alpha = m - \ell^\alpha \leq m - \ell^O = r^O$. Hence, $w_{\text{rem}}(r^B) \leq w_{\text{rem}}(r^\alpha + u^\alpha) \leq w_{\text{rem}}(r^O)$ by monotonicity of $w_{\text{rem}}(\cdot)$. It follows by definition of the w^β_{rem} notation that $w^B_{\text{rem}} \leq w^\alpha_{\text{rem}} \leq w^O_{\text{rem}}$. □

Lemma 5.9 $k^\alpha = k^O$.

Proof. $k^\alpha \leq k^O$ because no server is assigned more than τ local tasks by max-cover at iteration $\tau = k^O$. For sake of contradiction, assume $k^\alpha < k^O$. Then $u^\alpha > 0$, because otherwise $\alpha = B$ and $L^B = k^\alpha \cdot w_{\text{loc}} < K^O \leq L^O$, violating the optimality of O. Let t be an unassigned task in α. By definition, $\rho(t, s)$ holds for some server s. Assign t to s in α to obtain a new partial assignment β. We have $k^\beta \leq k^\alpha + 1 \leq k^O = \tau$. By Lemma 5.2, $\ell^\alpha \geq \ell^\beta$, contradicting the fact that $\ell^\beta = \ell^\alpha + 1$. We conclude that $k^\alpha = k^O$. □

Lemma 5.10 $L^B \leq V^B$.

Proof. By definition, $L^B_s = \sum_{t:B(t)=s} w(t, B)$ and $V^B_s = \sum_{t:B(t)=s} v(t, B)$. By Lemma 5.8, $w^B_{\text{rem}} \leq w^\alpha_{\text{rem}}$, and thus $w(t, B) \leq v(t, B)$. It follows that $\forall s \in S$, $L^B_s \leq V^B_s$. Therefore $L^B \leq V^B$, because $L^B = \max_s L^B_s$ and $V^B = V^B = \max_s V^B_s$. □

For the remainder of this section, let s_1 be a server such that $\ell^\alpha_{s_1} = k^O$. Such a server exists by Lemma 5.9. Let $S_2 = S - \{s_1\}$ be the set of remaining servers. For a partial assignment $\beta \supseteq \alpha$, define N^β to be the average virtual load under β of the servers in S_2. Formally,

$$N^\beta = \frac{\sum_{s \in S_2} V^\beta_s}{|S_2|} = \frac{\ell^\beta w_{\text{loc}} + r^\beta w^\alpha_{\text{rem}} - V^\beta_{s_1}}{n-1}$$

To obtain the approximation bound, we compare N^β with the similar quantity M^O for the optimal assignment. For convenience, we let $\delta = w^O_{\text{rem}}/(n-1)$.

Lemma 5.11 *Let $\beta = \alpha_i \xrightarrow{t:s} \alpha_{i+1} = \beta'$. Then*

$$V^\beta_s \le N^\beta \le M^O - \delta.$$

Proof. Proof is by a counting argument. By Lemma 5.9, we have $k^\alpha = k^O$, so $\ell^\beta_{s_1} \ge \ell^\alpha_{s_1} = k^O$. Hence, $V^\beta_{s_1} \ge K^O$. By Lemma 5.2, we have $\ell^\alpha \ge \ell^O$. Let $d = \ell^\beta - \ell^O$. $d \ge 0$ because $\ell^\beta \ge \ell^\alpha \ge \ell^O$. Because $\ell^\beta + r^\beta + u^\beta = m = \ell^O + r^O$, we have $r^\beta + u^\beta + d = r^O$. Also, $u^\beta \ge 1$ since t is unassigned in β. Then by Lemma 5.8,

$$\begin{aligned}
(n-1)N^\beta &= \ell^\beta w_{\text{loc}} + r^\beta w^\alpha_{\text{rem}} - V^\beta_{s_1} \\
&= (\ell^O + d) w_{\text{loc}} + (r^O - u^\beta - d) w^\alpha_{\text{rem}} - V^\beta_{s_1} \\
&\le \ell^O w_{\text{loc}} + (r^O - u^\beta) w^O_{\text{rem}} - K^O \\
&\le (n-1)M^O - w^O_{\text{rem}}.
\end{aligned}$$

Hence, $N^\beta \le M^O - \delta$.

Now, since β is part of a trace, we have $V^\beta_s \le V^\beta_{s'}$ for all $s' \in S$. In particular, $V^\beta_s \le N^\beta$, since N^β is the average virtual load of all servers in S_2. We conclude that $V^\beta_s \le N^\beta \le M^O - \delta$. □

Proof of Theorem 5.4: Lemma 5.5 shows that the time complexity of Algorithm 2 is $O(m^2 n)$. Now we finish the proof for the approximation bound.

Let s be a server of maximum virtual load in B, so $V^B_s = V^B$. Let i be the smallest integer such that $\alpha_i|_s = B|_s$, that is, no more tasks are assigned to s in the subtrace beginning with α_i.

Case 1: $i = 0$: Then $\ell^{\alpha_0}_s \le k^{\alpha_0} = k^O$ by Lemma 5.9, and $r^{\alpha_0} = 0$, so $V^B = V^{\alpha_0}_s \le K^O$. Hence, $V^B \le K^O \le L^O$.

Case 2: $i > 0$: Then $\beta = \alpha_{i-1} \xrightarrow{t:s} \alpha_i = \beta'$ for some task t. By lemma 5.11, $V^\beta_s \le M^O - \delta$, so using Lemma 5.8,

$$V^{\beta'}_s \le V^\beta_s + w^\alpha_{\text{rem}} \le M^O - \delta + w^O_{\text{rem}}.$$

Then by Lemma 5.7,

$$V^B = V^B_s = V^{\beta'}_s \le M^O + w^O_{\text{rem}} - \delta \le L^O + w^O_{\text{rem}} - \delta.$$

Both cases imply that $V^B \le L^O + w^O_{\text{rem}} - \delta$. By Lemma 5.10, we have $L^B \le V^B$. Because the algorithm chooses an assignment with least maximum load as the output A, we have $L^A \le L^B$. Hence,

$$L^A \le L^O + w^O_{\text{rem}} - \delta = L^O + \left(1 - \frac{1}{n-1}\right) \cdot w^O_{\text{rem}}$$

□

6 Conclusion

In this paper, we present an algorithmic study of the task assignment problem in the Hadoop MapReduce framework and propose a mathematical model to evaluate the cost of task assignments. Based on this model, we show that it is infeasible to find the optimal assignment unless $\mathcal{P} = \mathcal{NP}$. Theorem 3.1 shows that the task assignment problem in Hadoop remains hard even if all servers have equal capacity of 3, the cost function only has 2 values in its range, and each data block has at most 2 replicas.

Second, we analyze the simple round robin algorithm for the UHTA problem. Theorem 4.1 reveals that the intuition is wrong that increasing the number of replicas always helps load balancing. Using round robin task assignment, adding more replicas into the system can sometimes result in worse maximum load. Theorems 4.2 and 4.3 show there could be a multiplicative gap in maximum load between the optimal assignment and the assignment computed by Algorithm 1.

Third, we present Algorithm 2 for the general HTA problem. This algorithm employs maximum flow and increasing threshold techniques. Theorem 5.4 shows that the assignments computed by Algorithm 2 are optimal to within an additive constant that depends only on the number of servers and the remote cost function.

There are many interesting directions for future work. We have sketched a proof of a matching lower bound to Theorem 5.4 for a class of Hadoop cost functions. We plan to present this result in followup work. Sharing a MapReduce cluster between multiple users is becoming popular and has led to recent development of multi-user multi-job schedulers such as fair scheduler and capacity scheduler. We plan to analyze the performance of such schedulers and see if the optimization techniques from this paper can be applied to improve them.

7 Acknowledgments

We would like to thank Avi Silberschatz, Daniel Abadi, Kamil Bajda-Pawlikowski, and Azza Abouzeid for their inspiring discussions. We are also grateful to the anonymous referees for providing many useful suggestions that significantly improved the quality of our presentation.

References

[1] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3):486–504, 1997.

[2] Y. Azar, J. S. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proceedings of the 3rd Annual ACM-SIAM symposium on Discrete algorithms*, pages 203–210. SIAM Philadelphia, PA, USA, 1992.

[3] K. Birman, G. Chockler, and R. van Renesse. Towards a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[4] E. Bortnikov. Open-source grid technologies for web-scale computing. *SIGACT News*, 40(2):87–93, 2009.

[5] R. E. Burkard. Assignment problems: Recent solution methods and applications. In *System Modelling and Optimization: Proceedings of the 12th IFIP Conference, Budapest, Hungary, September 2-6, 1985*, pages 153–169. Springer, 1986.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms, 2nd ed.* MIT press Cambridge, MA, 2001.

[7] M. de Kruijf and K. Sankaralingam. MapReduce for the Cell B. E. architecture. *University of Wisconsin Computer Sciences Technical Report CS-TR-2007*, 1625, 2007.

[8] J. Dean. Experiences with MapReduce, an abstraction for large-scale computation. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM New York, NY, USA, 2006.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation, San Francisco, CA*, pages 137–150, 2004.

[10] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[11] M. R. Garey, D. S. Johnson, et al. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman San Francisco, 1979.

[12] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[13] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, pages 416–429, 1969.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269. ACM New York, NY, USA, 2008.

[15] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics*, 52(1), 2005. Originally appeared in *Naval Research Logistics Quarterly,* 2, 1955, 83–97.

[16] R. Lämmel. Google's MapReduce programming model—Revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

[17] J. K. Lenstra, D. B. Shmoys, and E. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1):259–271, 1990.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society Washington, DC, USA, 2007.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation, San Diego, CA*, 2008.