Cross-layer Scheduling for MapReduce-based Big Data Workflows in Heterogeneous Hadoop Systems

Yijie Zhang and Chase Q. Wu
Department of Data Science
New Jersey Institute of Technology
Newark, NJ 07102, USA
Email: {yz829, chase.wu}@njit.edu

Aiqin Hou
School of Information Science and Technology
Northwest University
Xi'an, Shaanxi 710127, China
Email: houaiqin@nwu.edu.cn

***Abstract*—The performance of big data workflows depends on both the workflow mapping scheme, which determines task assignment and container allocation in Hadoop, and the on-node scheduling policy, which governs resource allocation and container provisioning. Most research on big data workflow scheduling focuses solely on workflow mapping, achieving only limited success. We conduct an in-depth investigation into the impact of node-level scheduling on overall workflow performance and explore the benefits of combining these two levels of scheduling (workflow- and node-level). We formulate a generic problem that considers cross-layer scheduling to minimize the end-to-end delay of MapReduce-based big data workflows in the Hadoop system. The efficacy of our proposed solution, compared with existing methods, is demonstrated through extensive simulations and proof-of-concept experiments using real-life big data workflows deployed on a real-life cluster.**

***Index Terms*—Big data, cross-layer scheduling, Hadoop**

I. Introduction

With rapidly growing data volumes, the big data technology has become a cornerstone for many artificial intelligence (AI), and machine learning (ML) applications [1], [2]. As datasets expand, the efficiency of workflows handling such massive data becomes paramount. The optimization of these workflows hinges on the intricate interplay between two critical components: workflow mapping and node-level scheduling within Hadoop environments [3], [4].

Existing research predominantly focuses on workflow mapping schemes for task assignment and container allocation, but the significant impact of node-level scheduling on overall workflow performance remains underexplored [5]–[8]. Recognizing this gap, our study delves into a comprehensive examination of how node-level scheduling intricacies influence the makespan performance of MapReduce-based big data workflows in Hadoop systems [9]–[13]. We advocate for the integration of both workflow- and node-level scheduling strategies to enhance overall efficiency.

In this paper, we present a generic problem formulation that encompasses both workflow mapping and on-node scheduling, aimed at minimizing end-to-end delays of big data workflows. We propose an integrated heuristic solution, offering a novel approach to optimize workflow performance by bridging the gap between theoretical insights and practical applicability. The efficacy of our proposed solution is demonstrated through extensive simulations and real-life experiments conducted on a physical cluster, illustrating tangible improvements over existing methodologies.

Through this endeavor, we aim to contribute to the evolving landscape of big data analytics by elucidating the critical role of node-level scheduling and advocating for integrated solutions that transcend traditional boundaries. This approach would ultimately pave the way for more efficient and scalable big data workflows in Hadoop environments.

The rest of this paper is organized as follows. Sec. II surveys related work. Sec. III defines a container provisioning problem. Sec. IV details the design of an approximation algorithm. Sec. V evaluates algorithm performance. Sec. VI concludes our work and sketches a plan of future work.

II. Related work

We conduct a brief survey on workflow mapping and container provisioning on different clusters.

In [4], an approach to Big Data task scheduling for IoT cloud computing applications is introduced. Leveraging a hybrid implementation of the Dragonfly Algorithm's optimization technique inspired by the swarming behaviors of dragonflies, the authors designed the MHDA algorithm to minimize makespan and maximize resource utilization. However, this method only considers workflow scheduling in clouds, where virtual machines can be provisioned by different physical machines, and it does not account for on-node scheduling.

In [2], the intricate challenge of scheduling workflows within cloud environments is formulated as a multi-objective optimization problem and addressed by a novel algorithm based on decomposition. Notably, this approach also does not account for on-node scheduling.

In [14], a pipeline-based dynamic modular arithmetic-based scheduler (PMOD) is introduced. It re-schedules streams distributed across a network of nodes and tasks in response to changes in system parameters, such as the number of tasks, executors, or nodes. By organizing essential operations within a pipeline framework, PMOD effectively minimizes overall processing time. However, it does not consider the structure of big data workflows.

In this paper, we design an approximation algorithm using linear programming (LP)-based rounding to guide workflow mapping and container provisioning with a rigorous performance guarantee.

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

III. Problem formulation

In this section, we construct cost models and formally define a big data workflow optimization problem that takes both workflow mapping and on-node scheduling into consideration.

A. A Three-Layer Workflow Execution Architecture

We consider a three-layer architecture for MapReduce workflow execution on a physical cluster, where a workflow is mapped to a physical cluster through a container layer. We consider both workflow mapping and on-node scheduling to minimize the end-to-end delay of the entire workflow.

In the top workflow layer, as in the majority of the previous work, we model a big data workflow as a weighted directed acyclic graph (DAG) $G_w(V_w, E_w)$, with $|v_w| = m$ modules and $|E_w|$ edges $e_{p,q}$, each of which represents the execution dependency between two adjacent modules w_p and w_q, and whose weight represents the data size $DS_{p,q}$ transferred between them. For each module w_i, it has an input data size DIN_i and an output data size $DOUT_i$.

In the middle container layer, a set of containers are used to execute the modules mapped to them. We use a complete weighted graph $G_c(V_c, E_c)$ to model this layer, where each node v_i is associated with a resource profile that defines memory size m_i and the number n_i of cores.

In the bottom physical cluster layer, we model it as a complete weighted graph $G_p(V_p, E_p)$ consisting of a limited number of physical machines (PMs or nodes) connected via a high-speed switch. Each node v_i is associated with a resource profile that defines memory size m_i, I/O speed $r_{I/O,i}$, CPU speed $r_{cpu,i}$, the number n_i of cores, and a Network Interface Card (NIC) with upload bandwidth BW_{up} and download bandwidth BW_{down}. Without loss of generality, we assume that all nodes have the same upload and download bandwidths as they are connected via the same switch, i.e., $BW_{up} = BW_{down} = b$. Also, the network delay between any two nodes is considered identical, i.e., $d_{p,q} = d$. In the I/O queue, the tasks are first come first served.

B. Time Cost Models

In a big data workflow, each edge represents a data transfer and each node represents a module to be executed. For a MapReduce job, it contains a shuffling process, which can also be considered as a data transfer process. Hence, we focus only on module execution time and data transfer time.

1) Time Cost of Data Transfer

There is a data transfer process between two modules of two neighboring MapReduce jobs in the DAG, denoted as w_p and w_q, connected with a directed edge whose weight represents the data size $DS_{p,q}$. Actually, w_p can be considered as the end module of the preceding MapReduce job and w_q can be considered as the start module of the succeeding MapReduce job. The data transfer time between w_p and w_q is modeled as:

$$T_{p,q} = \frac{DS_{p,q}}{b} + d.$$

2) Time Cost of a MapReduce Job

A MapReduce job has three phases of data processing: Mapping, data shuffling from Mapper to Reducer, and Reducing.

a) Time Cost of Mapper Modules

A mapper module involves three steps: read data from disk to memory, execute the mapper function, write data back from memory to disk. We use $DS_{mapper,i}$ to denote the data size ($DS_{mapper,i} = DIN_i$). According to the implementation of the read and write of the mapper module, e.g., fread and fwrite, the data block size of mapper module w_i (for the whole workflow, each module has an ID i) is denoted as $DS_{fread,i}$ (or $DS_{fwrite,i}$) every time read (or written) by fread (or fwrite) instruction. Every time fread reads in a data block, the mapper module processes it, e.g., scanning the input data, and then uses fwrite to write back the output from memory to disk. For a mapper module, the time is dominated by reading data from disk to memory. The time cost of Mapper module w_i executed on node $v_{i''}$ can be modeled as (suppose that mapper module w_i is mapped to container $v_{i'}$, and container $v_{i'}$ is provisioned on physical node $v_{i''}$):

$$T_{mapper,i} = \frac{DS_{mapper,i}}{r_{I/O,i''}} + \frac{DS_{fread,i}}{r_{cpu,i''}} + \frac{DS_{fwrite,i}}{r_{I/O,i''}}.$$

b) Time Cost of Shuffling Process

The shuffling process of a MapReduce job is actually data copy from Mapper modules to Reducer modules. For the shuffling process $e_{p,q}$ between mapper module w_p and reducer module w_q, the data transfer time can be computed as:

$$T_{p,q} = \frac{DS_{p,q}}{b} + d.$$

c) Time Cost of Reducer Modules

A reducer module has two steps: read data from disk to memory and execute the reducer module. We use $DS_{reducer,i}$ to denote the data size ($DS_{reducer,i} = DIN_i$). Similarly, according to the implementation of the read and write of the mapper module, e.g., fread and fwrite, the data block size of reducer module w_i (for the whole workflow, each module has an ID i) is denoted as $DS_{fread,i}$ (or $DS_{fwrite,i}$) every time read (or written) by fread (or fwrite) instruction. A data block read in by fread is processed by the reducer module, which merges the input data. The time cost of a Reducer module is dominated by the time for reading data from disk to memory. The time cost of Reducer module w_i executed on node $v_{i''}$ is calculated as (suppose that reducer module w_i is mapped to container $v_{i'}$, and container $v_{i'}$ is provisioned on physical node $v_{i''}$):

$$T_{reducer,i} = \frac{DS_{reducer,i}}{r_{I/O,i''}} + \frac{DS_{fread,i}}{r_{cpu,i''}}.$$

d) Time Cost of Start and End Module of MapReduce Job

Every MapReduce job has a start module and an end module, which perform initialization, data merging, or some simple tasks (for example, sorting), whose time can be ignored compared with the execution time of the MapReduce job.

e) Time Cost of Edge between Start and Mapper Modules

For a MapReduce job, the data is typically partitioned offline in HDFS. We also ignore the time of the edge between the start module and the mapper modules connected to it.

f) Time Cost of Data Copy from Reducer to End Module

After the execution of a reducer module, data is copied from memory to the end module (in reality, the data is copied to the destination node in the succeeding MapReduce job, and because the bandwidth and delay are identical, this process can be simplified as this model). The data transfer time between reducer module w_p and end module w_q is calculated as:

$$T_{p,q} = \frac{DS_{p,q}}{b} + d.$$

C. Workflow Mapping Scheme

Based on the above models, we may use an existing workflow mapping algorithm to compute a mapping scheme under the follow constraints on workflow mapping and execution/transfer precedence: (1) Each module/edge is required to be mapped to only one node/link. (2) A computing module cannot start execution until all of its required input data arrives. (3) A dependency edge cannot start data transfer until its preceding module finishes.

We define a workflow mapping scheme $M_{G_w \to G_c}$ that maps G_w to G_c as follows:

$$M_{G_w \to G_c} = w \to v, \forall w \in V_w, \exists v \in V_c.$$

For any module w in workflow G_w, there must be a container v in the container set G_c, to which w is mapped $(w \to v)$.

D. On-container Scheduling Policy

In a scheduling round, a subset of modules $\{w_1, w_2, \ldots, w_k\}$ in a workflow G_w are mapped to a container v and there may exist resource competition among these modules. An On-container Scheduling Policy for $\{w_1, w_2, \ldots, w_k\}$ on v, denoted as $S_{\{w_1,w_2,\ldots,w_k\} \to v}$, is a k-dimensional vector, in which each element $s_i (1 \leq i \leq k)$ is the percentage usage of CPU time for module w_i in this scheduling round. Note that the sum of all the elements in this vector is 100%.

For each container v_i, we have a n-dimensional vector S_i (n is large enough) to demonstrate the on-container scheduling policy on container v_i. This method allows us to demonstrate any possible scheduling policy. For example, for FCFS scheduling on modules $\{w_1, w_2, \ldots, w_k\}$ executing on node v_i in k-dimensional array S_i, $s_1 = 1$ and others are zero. When w_1 is done, $w_2 = 1$, others are zero, and so on. For RR scheduling on modules $\{w_1, w_2, \ldots, w_k\}$ executing on node v_i in k-dimensional array S_i, $s_i = 1/k (1 \leq i \leq k)$. This k-dimensional array is able to express any possible scheduling policy in every scheduling round.

Since an on-container scheduling policy is container-specific according to the above definition, we use $S_{G_w \to G_c}$ to denote a full set of on-container scheduling policies for the entire workflow G_w over container set G_c.

E. Container Set Mapping Scheme

Based on the above models, we wish to determine a mapping scheme, where each container is required to be mapped to only one physical node. We define a container mapping scheme $M_{G_c \to G_p}$ that maps G_c to G_p as follows:

$$M_{G_c \to G_p} = w \to v, \forall w \in V_c, \exists v \in V_p.$$

For any container w in the container set G_c, there must be a physical node v in the physical node set G_p, to which w is mapped $(w \to v)$.

F. On-node Scheduling Policy

In a scheduling round, if a subset of containers $\{w_1, w_2, \ldots, w_k\}$ in the container set G_c are mapped to a computer node v, which means that these containers are provisioned on this physical node, and there exists resource competition among these containers. An On-node Scheduling Policy for $\{w_1, w_2, \ldots, w_k\}$ on v, denoted as $S_{\{w_1,w_2,\ldots,w_k\} \to v}$, is denoted as a k-dimensional vector, in which each element $s_i (1 \leq i \leq k)$ is the percentage usage of CPU time for module w_i in this scheduling round. Similarly, the sum of all the elements in this vector is 100%.

For each physical node v_i, we have an n-dimensional vector S_i (n is large enough) to demonstrate the on-node scheduling policy on physical node v_i.

Since an on-node scheduling policy is node-specific according to the above definition, we use $S_{G_c \to G_p}$ to denote a full set of on-node scheduling policies for the entire container set G_c over physical cluster G_p. Once $M_{G_w \to G_c}$, $S_{G_w \to G_c}$, $M_{G_c \to G_p}$ and $S_{G_c \to G_p}$ are determined for all modules, containers and nodes, we calculate the execution time of every workflow module and the End-to-end Delay (ED) of the entire workflow, as defined below.

G. End-to-end Delay (ED)

The End-to-end Delay (ED) of a workflow $G_w(V_w, E_w)$ that is mapped to a container set G_c by $M_{G_w \to G_c}$ and scheduled by $S_{G_w \to G_c}$, which is further mapped to a physical cluster G_p by $M_{G_c \to G_p}$ and scheduled by $S_{G_c \to G_p}$, denoted as $ED_{(G_w, G_c, G_p, M_{G_w \to G_c}, S_{G_w \to G_c}, M_{G_c \to G_p}, S_{G_c \to G_p})}$, is the time duration from the time point when the first module w_0 in G_w starts execution to the time point when the last module $w_{|V_w|-1}$ in G_w is completed.

H. End-to-end Workflow Optimization Problem

We formally define an End-to-end Workflow Optimization Problem as follows:

Definition 1: Given a DAG-structured workflow $G_w(V_w, E_w)$, and a heterogeneous physical cluster $G_p(V_p, E_p)$, we wish to find a workflow mapping scheme $M_{G_w \to G_c}$ that maps the workflow to a container set $G_c(V_c, E_c)$ and an overall on-container scheduling policy $S_{G_w \to G_c}$, where the container set is provisioned on the physical cluster by a container mapping scheme $M_{G_c \to G_p}$ and an on-node scheduling policy $S_{G_c \to G_p}$, such that the workflow's end-to-end delay $ED_{(G_w, G_c, G_p, M_{G_w \to G_c}, S_{G_w \to G_c}, M_{G_c \to G_p}, S_{G_c \to G_p})}$ is minimized.

The general workflow mapping problem is well known to be NP-complete, and so is the on-node scheduling problem. As a combined optimization problem, this problem with preemptive scheduling is also NP-complete.

IV. Algorithm Design

We first focus on container provisioning in one round (task), where there are a set of containers needed to be provisioned, which could be regarded as a set of jobs.

A. Container Provisioning with Only One Task

In this problem, we consider a complete bipartite graph $G = \{I \cup J, E\}$, where I represents a set of n machines and J denotes a set of m independent jobs that are unrelated to the machines. Each edge $e = (i, j)$ is associated with a cost c_{ij}. We assume that job arrivals follow a certain distribution. We define the performance of the offline optimal solution over the set $\mathcal{S}$ of all possible arrival sequences as $\mathbb{E}_{\mathcal{S}}[\text{OPT}(\mathcal{S})]$.

A straightforward approach to bound the offline OPT is to use linear programming (LP). However, such an LP solution can be arbitrarily poor because it may not be tight. If the job arrival distribution is known in advance, the competitive ratio can be defined as: $\frac{\mathbb{E}[\text{ALG}(\mathcal{S})]}{\mathbb{E}[\text{OPT}(\mathcal{S})]}$. It is crucial to define the offline OPT appropriately; otherwise, it does not differ from the scenario where no information about the distribution is available. The offline OPT is actually $\mathbb{E}[\max_i \ell_i]$, where ℓ_i represents the load on machine i after all jobs have been assigned. If x_{ij} denotes the probability of assigning job j to machine i, then: $\mathbb{E}[\ell_i] = \sum_{j=1}^{m} x_{ij} c_{ij}$.

We can use LP to bound $\mathbb{E}[\ell_i]$, but our goal is to bound $\mathbb{E}[\max_i \ell_i]$. Therefore, the problem now is to find the relationship between $\mathbb{E}[\max_i \ell_i]$ and $\max_i \mathbb{E}[\ell_i]$. Unfortunately, we find that $\mathbb{E}[\max_i \ell_i]$ can be either $\max_i \mathbb{E}[\ell_i]$ or $n \cdot \max_i \mathbb{E}[\ell_i]$ in different scenarios. For instance, with n machines and n jobs, if $x_{ij} = \frac{1}{n}$ for $i = j$ and $p_{ij} = n$ if $i = j$ and $p_{ij} = 0$ otherwise, then if at any time all jobs are assigned to a single machine, the probability distribution holds. In this case, $\mathbb{E}[\ell_i] = 1$ and $\mathbb{E}[\max_i \ell_i] = n$. This implies that the LP solution could be $\frac{1}{n} \cdot \mathbb{E}_{\mathcal{S}}[\text{OPT}(\mathcal{S})]$ or $\mathbb{E}_{\mathcal{S}}[\text{OPT}(\mathcal{S})]$ depending on the scenario.

This analysis indicates that the LP is not tight. Regarding the idea of Feasibility-LP, due to the expectation of the load, we cannot simply discard the assignment $j \to i$ if $p_{ij} > T$.

B. LP Relaxation of the Problem

Consider the following linear programming (LP) relaxation for the problem involving a single task:

$$\begin{aligned} \min \quad & \gamma, \\ \text{s.t.} \quad & \sum_{i \in I} x_{ij} = 1, \quad \forall j \in J, \\ & \sum_{j \in J} p_{ij} \cdot x_{ij} \le \gamma, \quad \forall i \in I, \\ & x, \gamma \ge 0, \end{aligned}$$

where γ denotes the makespan and x_{ij} is a binary variable indicating whether job j is assigned to node i.

We denote this LP problem as **LP-1**. Despite its significant integrality gap, **LP-1** proves useful in our approximation approach once it is suitably strengthened.

Theorem 1: There exists a polynomial-time algorithm that, given T, either produces a solution with a makespan of at most $2T$ or reports the absence of a feasible solution. If $T \ge \text{OPT}$, the algorithm guarantees a makespan of at most $2T$.

The key to proving Theorem 1 is the strengthened version of **LP-1**, termed **Feasibility-LP**. Note that while **Feasibility-LP** is not an LP in the strictest sense (as it lacks an objective function), it serves the purpose of determining feasibility.

In the illustrative example where the integrality gap is problematic, **Feasibility-LP** has no feasible solution. We focus on proving the following result:

Theorem 2: If **Feasibility-LP** lacks a feasible solution, it implies that OPT $> T$. Conversely, if **Feasibility-LP** has a feasible solution, an integer solution with a makespan of at most $2T$ can be found in polynomial time.

Clearly, when $T \ge$ OPT, the natural binary solution corresponding to the optimum is feasible for the LP. We demonstrate that if a feasible solution exists, we can construct an assignment $\phi : J \to I$ with a makespan of at most $2T$.

During the algorithm, some potential $j \to i$ assignments are discarded, and certain machines are removed from consideration after job assignments. Thus, it is more advantageous to analyze the problem in a broader context.

Let $G = (J \cup I, E)$ be a bipartite graph, where each $i \in I$ has a target running time bound T_i. We then consider the generalized feasibility LP, referred to as **Feasibility-LP2**:

$$\begin{aligned} & \sum_{i:(i,j)\in E} x_{ij} = 1, \quad \forall j \in J, \\ & \sum_{j:(i,j)\in E} p_{ij} \cdot x_{ij} \le T_i, \quad \forall i \in I, \\ & x \ge 0. \end{aligned}$$

We develop an LP-based iterative rounding algorithm for the container provisioning problem, denoted as LPCP, outlined in Algorithm 1. This algorithm alternates between rounding some variables and solving the residual problem for unrounded variables. At each step, **Feasibility-LP2** is referenced concerning the current bipartite graph G in the algorithm.

We now demonstrate that each iteration of the **while** loop guarantees that **Feasibility-LP2** remains feasible and that at least one job or edge is removed from G during each iteration. Consequently, the number of iterations is at most $|E| + |J|$, ensuring the algorithm runs in polynomial time. The following lemma also holds:

Lemma 1: If LPCP terminates, then the assignment will have a makespan of at most $2T$.

Proof: We have:

$$\begin{aligned} \sum_{j'' \in J_i} p_{ij''} &\le T_i' + p_{ij} + p_{ij'} \\ &= T_i' + (1 - \bar{x}_{ij}) \cdot p_{ij} + (1 - \bar{x}_{ij'}) \cdot p_{ij'} + \bar{x}_{ij} \cdot p_{ij} \\ &\quad + \bar{x}_{ij'} \cdot p_{ij'} \\ &\le T_i' + (2 - \bar{x}_{ij} - \bar{x}_{ij'}) \cdot T + \bar{x}_{ij} \cdot p_{ij} + \bar{x}_{ij'} \cdot p_{ij'} \\ &\le T_i' + T + T_i \\ &= T_i' + T + (T - T_i') \\ &= 2 \cdot T. \end{aligned}$$

Algorithm 1 An LP-based iterative rounding algorithm for container provisioning

Input: $J \cup I, \{(i,j) : p_{ij} \leq T\}, T$
Output: a container provisioning strategy: $J \to I$

$G = (J \cup I; \{(i,j) : p_{ij} \leq T\})$;
for each $i \in I$, $T_i = T$;
if Feasiblity-LP2 is infeasible **then**
  return no feasible solution;
while J is not empty **do**
  For **Feasibility-LP2** obtain an extreme point solution $\bar{x}$;
  Delete every edge (i,j) from E where $\bar{X}_{ij} = 0$;
  if $\bar{x}_{ij} = 1$ **then**
    provision container j by machine i and decrease T_i by p_{ij};
    delete container j from J;
  if some machine i has at most 1 neighbour in G **then**
    if machine i has a neighbour job j in G **then**
      provision container j by machine i and delete container j from G;
    delete machine i from G;
  if some machine i has exactly two neighbors j and j', and $x_{ij} + x_{ij'} \geq 1$. **then**
    provision container j and container j' to i;
    delete machine i, container j, and container j' from G;

Here, the second inequality holds because (i,j) and (i,j') are elements of E, implying p_{ij} and $p_{ij'}$ are bounded by T as per Step 1. The third inequality follows from the feasibility of $\bar{x}$ as a solution to **Feasibility-LP2**. ∎

Lemma 2: In each iteration, **Feasibility-LP2** has a feasible solution, and the total number of edges plus jobs, $|E| + |J|$, strictly decreases.

Proof: We start by establishing that **Feasibility-LP2** consistently produces a feasible solution in each iteration. Initially, this condition is satisfied; otherwise, LPIR-BR would have indicated the absence of a feasible solution. In each iteration, beginning with a feasible solution $\bar{x}$ for the LP over the graph G, the removal of certain edges and nodes from G results in a subgraph G'. It is straightforward to verify that $\bar{x}$, restricted to the remaining edges and nodes, continues to be a feasible solution for the subgraph G'. Thus, each subsequent iteration also maintains a feasible LP solution.

Next, we demonstrate that an edge or job node must be removed from G in each iteration. Suppose, for the sake of contradiction, that at the start of an iteration with the extreme point $\bar{x}$, every $(i,j) \in E$ satisfies $0 < \bar{x}_{ij} < 1$, and no $i \in I$ has degree one in G (otherwise, an edge or job node would have been removed). We show that there exists some $i \in I$ with exactly two neighbors j and j' such that $\bar{x}_{ij} + \bar{x}_{ij'} \geq 1$, leading to the removal of both j and j'.

Given that every $i \in I$ has $\deg(i) \geq 2$, and for each $j \in J$, $\sum_{i:(i,j)\in E} \bar{x}_{ij} = 1$, with no $\bar{x}_{ij}$ being exactly 1, it follows that $\deg(j) \geq 2$ as well. By the characterization of extreme points, there are at least $|E|$ tight constraints under $\bar{x}$. Since none of these tight constraints are non-negativity constraints and there are $|J| + |I|$ other constraints, we have:

$$\begin{aligned} |J| + |I| &\geq \text{Number of tight constraints} \\ &\geq |E| \\ &= \frac{\sum_{j\in J} \deg(j) + \sum_{i\in I} \deg(i)}{2} \\ &\geq \frac{2|J| + 2|I|}{2} = |J| + |I|. \end{aligned}$$

Each inequality must hold with equality, implying that every node in G has degree exactly 2. Consequently, $2 \cdot |J| = |E| = 2 \cdot |I|$, which leads to $|J| = |I|$.

Thus, we observe:

$$|I| = |J| = \sum_{j\in J} \sum_{i:(i,j)\in E} \bar{x}_{ij} = \sum_{i\in I} \sum_{j:(i,j)\in E} \bar{x}_{ij}.$$

In particular, there must be some $i \in I$ such that $\sum_{j:(i,j)\in E} \bar{x}_{ij} \geq 1$. This completes the proof that there exists a machine i with degree 2 and $\sum_{j:(i,j)\in E} \bar{x}_{ij} \geq 1$. ∎

Therefore, we have a 2-approximation algorithm based on the approach of "guessing" the target makespan via binary search. It is important to note that this algorithm is designed for the container provisioning problem with a single task. However, it can be extended to the general problem of workflow mapping and container provisioning with multiple tasks by applying the same algorithm to each task S_k within job $\mathcal{S}$.

V. Performance evaluation

In this section, we present results from three sets of experiments conducted in a heterogeneous computing environment. The experimental setting comprises a cluster of six physical machines: three Dell PowerEdge R740 machines, each with 20 CPUs (Intel(R) Xeon(R) Silver 4114 CPU @ 2.20 GHz) and 256 GB of memory; two Dell Precision Rack 7910 machines, each with 6 CPUs (Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40 GHz) and 16 GB of memory; and one Dell PowerEdge R730 machine with 8 CPUs (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz) and 16 GB of memory. We utilize VMware ESXi to provision a Hadoop cluster comprising 10 virtual machines (VMs) across these physical machines. Each VM runs Ubuntu 20.04 LTS and Hadoop 2.6.5. On this cluster, one VM is designated as the master/name node, and the remaining nodes function as slave/data nodes. VMs from the same physical machine are grouped into a single rack. In Hadoop's configuration, we set the number of data replicas in HDFS to three and the data block size to 128 MB.

In the first set of experiments, we evaluate 10 nodes and 100 modules, where the execution time p_{ij} of each module ranges from 1 to 10 seconds. We generate 100 random big data processing jobs, each consisting of a varying number of tasks, each requiring the execution of a set of modules. In each trial, we compare the performance of the LPCP algorithm and a Greedy algorithm based on the module execution time. The results are illustrated in Figure 1.

The results indicate that LPCP performs comparably to Greedy. However, LPCP guarantees a constant approximation ratio of 2, while Greedy's performance can be arbitrarily poor. To highlight this difference, we design a second set of experiments.

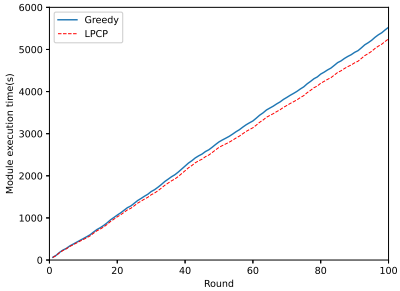

Fig. 1. Job module execution time of Greedy and LP-based rounding (LPCP).

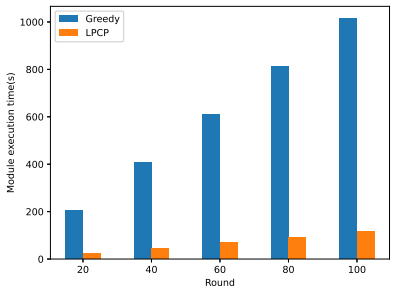

Fig. 2. Job module execution time of Greedy and LP-based rounding (LPCP) in the extreme case.

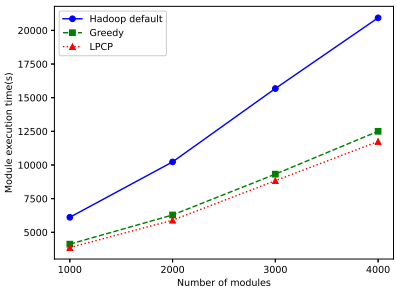

Fig. 3. Job module execution time of Hadoop default, Greedy and LP-based rounding (LPCP) in different load environments.

In the second set of experiments, we consider a scenario where module j incurs a cost of j on machine j, a cost of $1+\epsilon$ on machine $j-1$, and an infinite cost on all other machines (i.e., $p_{j-1}(j) = 1+\epsilon$, $p_j(j) = j$, and $p_i(j) = \infty$ for $i \neq j-1, j$). For clarity, we also refer to machine n as machine 0. Here, ϵ is an arbitrarily small positive constant to avoid ties. The optimal offline algorithm can schedule all modules with a maximum load of $1+\epsilon$ by assigning module j to machine $j-1$.

Conversely, the Greedy algorithm assigns module 1 to machine 1, resulting in a load of 1 (as opposed to $1+\epsilon$ on machine n or ∞ elsewhere). When job 2 arrives, Greedy assigns it to machine 2, resulting in a load of 2, instead of assigning it to machine 1, which would produce a load of $2+\epsilon$. This pattern continues for subsequent jobs, leading to a maximum load of n on machine n and a performance ratio of $n/(1+\epsilon)$, which can approach n as ϵ approaches 0.

For an illustrative example, consider $n = 10$ with p_{ij} as described. In this setting, Greedy achieves a module execution time of 10 seconds, while LPCP achieves $1+\epsilon$ seconds, which is the optimal performance. Consequently, the module execution time of Greedy is nearly 10 times that of LPCP. This disparity is shown in Figure 2.

In the third set of experiments, we compare the default Hadoop workflow scheduling strategy (Hadoop default), Greedy, and LPCP across four different load environments with varying numbers of modules. Each job executes all modules, and the performance of each scheduling strategy in these environments is plotted in Figure 3. The results demonstrate that LPCP consistently achieves superior performance.

VI. Conclusion and Future Work

We formulated a workflow optimization problem that encompasses both workflow mapping and node scheduling and designed an approximation algorithm, LPCP, based on linear programming (LP) rounding. Experimental results demonstrate the effectiveness of LPCP and confirm our theoretical analysis.

The primary contribution of this study lies in the integration of workflow mapping with node scheduling and the development of an approximation algorithm that provides performance guarantee. Future research will be conducted to harness historical data for predictive modeling to refine and optimize cross-layer scheduling.

References

[1] Y. Hu, H. Wang, and W. Ma, "Intelligent cloud workflow management and scheduling method for big data applications," *Journal of Cloud Computing*, vol. 9, no. 1, p. 39, 2020.

[2] E. Bugingo, D. Zhang, Z. Chen, and W. Zheng, "Towards decomposition based multi-objective workflow scheduling for big data processing in clouds," *Cluster Computing*, vol. 24, no. 1, pp. 115–139, 2021.

[3] J. Liu, S. Lu, and D. Che, "A survey of modern scientific workflow scheduling algorithms and systems in the era of big data," in *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 2020, pp. 132–141.

[4] L. Abualigah, A. Diabat, and M. A. Elaziz, "Intelligent workflow scheduling for big data applications in iot cloud computing environments," *Cluster Computing*, vol. 24, no. 4, pp. 2957–2976, 2021.

[5] I. A. T. Hashem, N. B. Anuar, M. Marjani, E. Ahmed, H. Chiroma, A. Firdaus, M. T. Abdullah, F. Alotaibi, W. K. M. Ali, I. Yaqoob *et al.*, "Mapreduce scheduling algorithms: a review," *The Journal of Supercomputing*, vol. 76, pp. 4915–4945, 2020.

[6] V. R. Pillareddy and G. R. Karri, "Monws: Multi-objective normalization workflow scheduling for cloud computing," *Applied Sciences*, vol. 13, no. 2, p. 1101, 2023.

[7] F. Ding and M. Ma, "Data locality-aware and qos-aware dynamic cloud workflow scheduling in hadoop for heterogeneous environment," *International Journal of Web and Grid Services*, vol. 19, no. 1, pp. 113–135, 2023.

[8] S. Hedayati, N. Maleki, T. Olsson, F. Ahlgren, M. Seyednezhad, and K. Berahmand, "Mapreduce scheduling algorithms in hadoop: a systematic study," *Journal of Cloud Computing*, vol. 12, no. 1, p. 143, 2023.

[9] J. Cao, J. Xu, and B. Wang, "Service capability aware big data workflow scheduling approach in cloud datacentre," *International Journal of Intelligent Systems Technologies and Applications*, vol. 22, no. 1, pp. 1–15, 2024.

[10] S. Zhang, Y. Xue, H. Zhang, X. Zhou, K. Li, and R. Liu, "Improved hungarian algorithm–based task scheduling optimization strategy for remote sensing big data processing," *Geo-spatial information science*, pp. 1–14, 2023.

[11] S. Pal, N. Jhanjhi, A. S. Abdulbaqi, D. Akila, F. S. Alsubaei, and A. A. Almazroi, "An intelligent task scheduling model for hybrid internet of things and cloud environment for big data applications," *Sustainability*, vol. 15, no. 6, p. 5104, 2023.

[12] D. Wu, X. Wang, X. Wang, M. Huang, R. Zeng, and K. Yang, "Multi-objective optimization-based workflow scheduling for applications with data locality and deadline constraints in geo-distributed clouds," *Future Generation Computer Systems*, 2024.

[13] N. Ahmed, A. L. Barczak, M. A. Rashid, and T. Susnjak, "A parallelization model for performance characterization of spark big data jobs on hadoop clusters," *Journal of Big Data*, vol. 8, no. 1, p. 107, 2021.

[14] S. Souravlas and S. Anastasiadou, "Pipelined dynamic scheduling of big data streams," *Applied Sciences*, vol. 10, no. 14, p. 4796, 2020.