SGD: Shortest-Job-First and Greedy Depth Based Hierarchical Programming Algorithm

An Efficient Solution to Multi-Job Scheduling with Max-Min Fairness

Qi Liu, Yanjie Ze, Renyang Guan
Student ID:{519021910529, 519021910706, 519021911058}
{purewhite, zeyanjie, guanrenyang}@sjtu.edu.cn

Department of Computer Science,
Shanghai Jiao Tong University, Shanghai, China

Abstract. Big data processing flourishing in recent years requires data and computing tasks to be stored and processed in geographically distributed datacenters. Even though the wasted time on transferring data among these datacenters is inevitable, but we have no idea neglecting so great cost. The trade-off between data size and transferring capacity requires us to figure out a method to schedule tasks on computing slots distributed among datacenters. Driven by this realistic and important problem, we propose an algorithm, SGD, to solve this scheduling problem while maintaining max-min fairness. First of all, we formalize the scheduling problem as a linear programming problem and prove that this problem is NP-completeness. Then we abstract the process of scheduling as 3-stage hierarchical process, and design our SGD algorithm accordingly. We also analyze the time complexity of it. At last, we do tests on toy data and generated data, verifying the efficiency of the algorithm. Our source code is available here.
Keywords: Multi-stage Job Scheduling, Breadth-First Search, Shortest Job First, Integer Linear Programming, Greedy, NP-completeness

1 Introduction

In support of big data processing, which means applying machine learning algorithms alike to process large volumes of data, it has become a trend that large volumes of data has been stored in geographically distributed data centers around the world. Big data processing has a wide range of applications in regard of social media and Internet advertising. As a result of this, it is of great significance to solve the problem of scheduling *jobs* across geographically distributed data centers.

A data analytical *job* is usually processed in more than one stages because of *precedence constraints*. In other words, formally, a *job* is separated to several *tasks* which must be performed in a certain order, because some tasks that must be executed later would require the results or data of the former tasks. In order to describe the serial and parallelism of *jobs*. We define *stage* as the collection of *tasks* which could be executed in parallelism. Different *stages* must be processed in serial.

Each of the geo-distributed datacenters has two types of resources: *slot* and *dataset*. *Slots* could be understood as processors where each *task* has to be put and processed. Some *tasks* need to obtain some certain amount of data from some *datasets*. Additionally, these *datasets* may be distributed in the same *datacenters* or in the same *datacenter*.

The intuition of gathering all the data in one datacenter and then processing them is not advisable. One of the reasons is that some *slots* in other *datacenters* are wasted. Since the datacenters are far apart geographically, the other reason which is more important is that the cost of transferring data is extremely expensive and unacceptable. Consequently, the method of randomly assigning *tasks* to *slots* to deal with the problem of low efficiency is irrational, too.

In order to specifically express the restrictions in the process of transmission, we define the concept of *bandwidth*. *Bandwidth* is the transmission rate of data between different *datacenters* or between different *slots* and *datasets* in one *datacenter*, with **MB/s** as the unit. In accordance with our common sense, the *bandwidth* between *datacenters* is much larger than that inside a *datacenter*.

Based on the concepts and brief analysis above, the **core of the problem** is to **find a placement** of data(in *datacenters*), intermediate results(of previous *tasks*) and computation tasks(in *slots*), **so that the average completion time of the set of jobs is minimized.**

However, since several *tasks* are processed in concurrently, there is a high possibility that they would compete for limited resources, such as *slots, bandwidth, etc*. Hence our solution of the algorithm has to maintain *max-min fairness*, which means trying to minimize the *job* completion times among all concurrent *jobs*.

Existing works in literature.[3], [4] only deal with scheduling single data analytical *job* while those in literature.[2] don't consider the inter-datacenter *bandwidth* and fail to figure out a good solution to pick appropriate number of *tasks* from a *task* pool. On the basis of the essential formulation and intra-stage scheduling algorithm literature.[2], we come up with a **Shortest-Job-First and Greedy-Depth-Based Hierarchical Programming** algorithm to solve the multi-job scheduling problem.

Our Work focuses on the **computational complexity analysis** and proposed a **3-phase model** to achieve the globally optimal scheduling problem. The architecture of our work is shown in Fig 1.

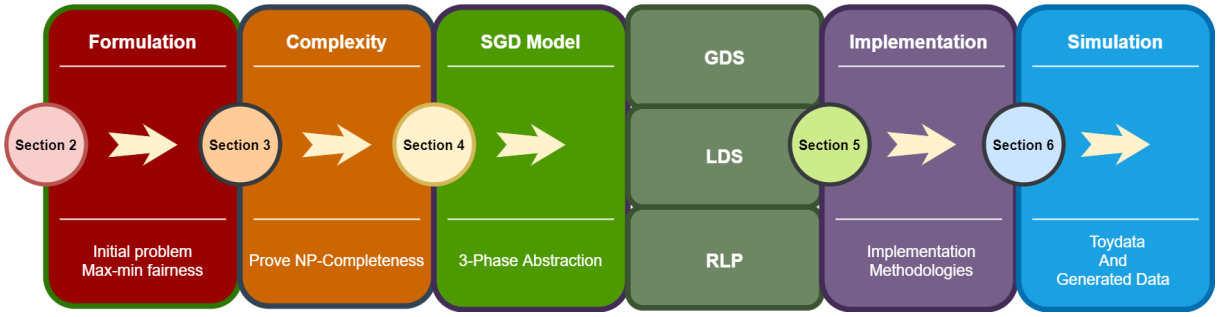

Fig. 1. The architecture of our work

First of all, we observe that the computational complexity of the global scheduling problem is **NP-Complete** and give a detailed proof by reducing ***Geo distributed Job Scheduling Problem*** $\leqslant_P$ ***General Scheduling Problem***, which is discussed in `section 3`.

In `section 4`, we describe our **SGD** algorithm in detail. In order to **reduce computational complexity** while **ensuring maximum parallelism and minimum data dependence**, we **change the smallest unit involved in scheduling from *task* to *taskset***, where *taskset* refers to set collection of tasks that belong to the same job and can be executed in parallel. Furthermore, In order to achieve optimal performance while allowing computing resources to be evenly distributed among jobs, we maintain ***Max-min fairness*** in the **schedule** phase.

In `section 5`, we describe how to implement the algorithm and how to construct the whole program structure based on Python. To be more precise, we propose several **basic basic structures** and two **Schedulers**, and construct the automatic process of solving this problem, which **needs no handcraft**. In `section 6`, we test our SGD algorithm and Baseline on several data, including **Toy Data** and our **Generated Data** based on **Gaussian Distribution**. We can compare the performance of SGD and Baseline clearly, from 3 groups of experiments, which show that SGD is much more efficient than Baseline.

To summarize, our contributions are:

- Formulating the problem **Multi-Job Scheduling with Max-Min Fairness** and proving its **NP-Completeness**
- Proposing Shortest-Job-First and Greedy Depth Based Hierarchical Programming Algorithm, short for **SGD**, which is an efficient solution to Multi-Job Scheduling with Max-Min Fairness, and analysing its complexity
- Experimenting with several groups of data, to test the efficiency of our SGD algorithm
- Performing nice visualization of data and the algorithm process

2 Problem Formulation

2.1 Notation List

Tab. 1 gives a brief table of notations used in our formulation.
Detailed explanation is given in section 2.4.

Table 1. Notation List for Formulation

Notation	Explanation
$\mathcal{K}$	The set of *jobs*, $\mathcal{K} = \{J_1, J_2, \cdots, J_K\}$
$\mathcal{D}$	The DAG of *datacenters*, $\mathcal{D} = (D, E^*)$
D	The set of *datacenters*, each labelled with natural numbers from 1 to $\|D\|$
E^*	The set of connections between datacenters,
	$e = (i, j) \in E^*$ represents for the connection from datacenter i to datacenter j
a_i	The capacity of available *slots* in *datacenter* i
$b_{i,j}$	BandWidth from datacenter i to dataceter j, where $\{i, j\} \in E^*$ and $s \neq j$
$b_{i,i}$	BandWidth of intra-datacenter transferring of datacenter i, where $\{i, i\} \in E^*$
J_i	The DAG of job i, $J_i = (V_i, E_i), V_i = T_i \cup D_i$
T_i	The set of tasks in job i, $T_i = \{1, 2, \cdots, \|T_i\|\}$, each task is each labelled with integers from 1 to $\|T_i\|$
D_i	The set of initial data in job i, $D_i \subseteq \mathbb{N}^+, D_i \cap T_i = \varnothing$
E_i	the data dependencies within job i
	$e = (i, j) \in E_i$ indicates that there is a data dependency from i to j
$e_{i,j}$	The execution time of task i of job j
$d^k_{i,j}$	The amount of data needed to transfer from task (or initial datum) i to task j of job k
$x_{i,j}$	The datacenter that task i of job j is deployed in
$d_{i,j}$	The deployment time of task i of job j
$c_{i,j} = t$	The completion time of task i of job j
$r_{i,j} = t$	The removal time of task i of job j
$C_j = t$	The completion time of job j
$x_d^{-1}(t) = S$	The set of tasks that are executing on datacenter d at time t.

2.2 Definition

Definition 1. *Let $\langle \boldsymbol{v} \rangle_k$ denote the kth ($1 \leqslant k \leqslant K$)largest element of $\boldsymbol{v} \in \mathbb{Z}^K$, implying $\langle \boldsymbol{v} \rangle_1 \geqslant \langle \boldsymbol{v} \rangle_2 \geqslant \cdots \geqslant \langle \boldsymbol{v} \rangle_K$. Intuitively, $\langle \boldsymbol{v} \rangle = (\langle \boldsymbol{v} \rangle_1, \langle \boldsymbol{v} \rangle_2, \cdots, \langle \boldsymbol{v} \rangle_K)$represents the non-increasing sorted version of $\boldsymbol{v}$.*

Definition 2. *For any $\boldsymbol{\alpha} \in \mathbb{Z}^K$ and $\boldsymbol{\beta} \in \mathbb{Z}^K$, if $\langle \boldsymbol{\alpha} \rangle_1 < \langle \boldsymbol{\beta} \rangle_1$ or $\exists k \in \{1, 2, \cdots, K\}$ such that $\langle \boldsymbol{\alpha} \rangle_k < \langle \boldsymbol{\beta} \rangle_k$ and $\langle \boldsymbol{\alpha} \rangle_i = \langle \boldsymbol{\beta} \rangle_i$, $\forall i \in \{1, \cdots, k-1\}$, then $\boldsymbol{\alpha}$ is lexicographically smaller than $\boldsymbol{\beta}$, represented as $\boldsymbol{\alpha} \prec \boldsymbol{\beta}$. Similarly, if $\langle \boldsymbol{\alpha} \rangle_k = \langle \boldsymbol{\beta} \rangle_k$, $\forall k \in \{1, 2, \cdots, K\}$ or $\boldsymbol{\alpha} \prec \boldsymbol{\beta}$, then $\boldsymbol{\alpha}$ is LEXICOGRAPHICALLY NO GREATER than $\boldsymbol{\beta}$, represented as $\boldsymbol{\alpha} \preceq \boldsymbol{\beta}$.*

Definition 3. *$\underset{\boldsymbol{x}}{\text{lexmin}} \boldsymbol{f}(\boldsymbol{x})$ represents the lexicographical minimization of the vector $\boldsymbol{f} \in \mathbb{R}^N$, which consists of N objects functions of $\boldsymbol{x}$. To be particular, the optimal solution $\boldsymbol{x}^* \in \mathbb{R}^K$ achieves the optimal $\boldsymbol{f}^*$, in the sense that $\boldsymbol{f}^* = \boldsymbol{f}(\boldsymbol{x}^*) \preceq \boldsymbol{f}(\boldsymbol{x}), \forall \boldsymbol{x} \in \mathbb{R}^K$*

Definition 4 (*Max-min fairness*[1]). *Vector $\boldsymbol{f}^*$ achieves max-min fairness if and only if it satisfies such property:*

For and allocation vector $\boldsymbol{f} \in F$, where F is the solution space of $\boldsymbol{f}$, and for any element $\langle \boldsymbol{f} \rangle_k$ such that $\langle \boldsymbol{f} \rangle_k > \langle \boldsymbol{f}^* \rangle_k$ there exists an element $\langle \boldsymbol{f} \rangle_{k'}$ such that $\langle \boldsymbol{f} \rangle_{k'} < \langle \boldsymbol{f}^* \rangle_{k'} < \langle \boldsymbol{f}^* \rangle_k$.

2.3 Assumptions

In order to quantify the complex and systematic scheduling problem and limit the scope of our discussion, we make the following assumptions:

Assumption 1:(General Slot Assumption)

Assume that all the slots distributed geographically in datacenters are general computing processors, which means each of them is able to handle any task of any job. Moreover, none of the slots show an advantage in handling specific tasks or jobs.

Assumption 2:(Incomplete Graph Assumption)

Given a directed graph $G = (\mathcal{D}, \mathcal{L})$, where $\mathcal{D} = 1, 2, \cdots, J$ denotes the set of datacenters and the J is the total number of datacenters. $\forall l_{s,j} \in \mathcal{L}$ if and only if there exists a data path transferring data from datacenter s to datacenter j, where $s, t \in \mathcal{D}$. We assume that G is not necessarily a binary directional complete graph, which means $\exists s, j \in \mathcal{D}$ such that $l_{s,j} = 0$ or $l_{j,s} = 0$. Furthermore, no data

path transferring data from datacenter s to j is equivalent to the bandwidth $b_{s,j}$ from s to j equalling to 0.

Assumption 3:(Direct Transferring Assumption)
Data and computing results could only be transferred from source datacenter to destination datacenter, with transferring through other datacenters not permitted. That is to say, if $b_{s,j} = 0$ then we can't transfer data from datacenter s to datacenter j. Even though $b_{s,t} > 0$ and $b_{t,j} > 0$, transferring data from datacenter s to datacenter j passing by datacenter k is not allowed.

Assumption 4:(Arbitrary Dataset Placement Assumption)
Assume that any one of the datasets each of whom is specified for some particular job could place in any datacenter. There is no limitation of the number of datasets which one datacenter could contain. Moreover, the placement of all the datasets is predefined before scheduling.

Assumption 5:(Fixed Bandwidth Assumption)
Assume that the bandwidths between datacenters and those intra-datacenter are fixed and predefined before scheduling. Furthermore, the bandwidth is the transmission rate of data and is measured by **MB/s**. There is no upper bound of the capacity of each data path. This assumption makes sense because the transmission rate is dependent on infrastructures rather than soft wares or algorithms.

Assumption 6:(Identical Intra-datacenter Bandwidth Assumption)
Assume that the bandwidth between different slot belonging to the same datacenter and the bandwidth from one dataset to slots of the same datacenter is identical. This assumption makes sense for the reason that the intra-datacenter bandwidth are far smaller than that of inter-datacenter bandwidth. However, we assign a relative small number to intra-datacenter bandwidth in order to represent the cost of assigning tasks to different slots

Assumption: 7:(Execution After Transmission Assumption)
Assume that each task must gather all the data it may use before starting execution. In other words, the action of execution and that of tranferring data must be done in sequence.

Assumption 8:(No Cache Assumption)
Assume that tasks are not allowed to store temporary data which will be used by later tasks into datasets. For example, supposing tasks t_{A1}, t_{A2} of job A such that t_{A1} precedes t_{A2} and t_{A2} requires the result of t_{A1} to proceed, t_{A1} could not leave the slot but to stay in a slot with t_{A2} simultaneously until the process of data transferring is done.

Assumption 9:(Task Parallelism Assumption)
Assume that the tasks which could be executed in parallel of job k, denoted as the set $\mathcal{T}_k = \{1, 2, \cdots, n_k\}$, must be launched on available computing slots in these datacenters and executed simultaneously.

2.4 Notation Explanation

For the convenience of our research, we denote the set of jobs as $\mathcal{K} = \{J_1, J_2, \cdots, J_K\}$, where the total number of jobs is $|\mathcal{K}| = K$.

Geo-distributed datacenters is modeled by a general graph $\mathcal{D} = (D, E^*)$ where $D \subseteq \mathbb{N}^+$ is the set of datacenters and each one is labelled with natural numbers from 1 to $|D|$. $e = (i, j) \in E^*$ represents for the connection from datacenter i to datacenter j. We define 2 functions to show their attributes.

- $a : D \mapsto \mathbb{N}^+$
  a_i indicates the number of slots of datacenter i
- $b : E^* \mapsto \mathbb{R}^+$
  $b_{i,j}$ indicates the bandwidth to transfer from data center i to datacenter j

Each job consists of a directed acyclic graph of tasks to execute, all tasks and initial data are labelled with natural numbers. Concretely, for job i, the DAG is denoted with $J_i = (V_i, E_i), V_i = T_i \cup D_i$, where

$T_i = \{1, 2, \cdots, |T_i|\}$ denotes the set of tasks in job i, $D_i \subseteq \mathbb{N}^+$ denotes the set of initial data in job i, and $e = (i, j) \in E_i$ indicates that there is a data dependency from i to j. We also define 2 weight functions to show their attributes.

- $e : \bigcup_{i=1}^{K} T_i \times \mathcal{K} \mapsto \mathbb{R}^+$
  $e_{i,j}$ indicates the execution time of task i of job j
- $d : \bigcup_{i=1}^{K} E_i \times \mathcal{K} \mapsto \mathbb{R}^+$
  $d_{i,j}^k$ indicates the amount of data needed to transfer from task (or initial datum) i to task j of job k

In the scheduling process, we also define

- $x : \bigcup_{i=1}^{K} T_i \times \mathcal{K} \mapsto D$
  $x_{i,j} = d_i$ indicates that task i of job j is deployed in datacenter d_i.
- $d : \bigcup_{i=1}^{K} T_i \times \mathcal{K} \mapsto \mathbb{R}^*$
  $d_{i,j} = t$ indicates that task i of job j is deployed at time t.
- $c : \bigcup_{i=1}^{K} T_i \times \mathcal{K} \mapsto \mathbb{R}^*$
  $c_{i,j} = t$ indicates that task i of job j is completed at time t.
- $r : \bigcup_{i=1}^{K} T_i \times \mathcal{K} \mapsto \mathbb{R}^*$
  $r_{i,j} = t$ indicates that task i of job j is removed at time t.
- $x^{-1} : D \times \mathbb{R}^* \mapsto 2^{\bigcup_{i=1}^{K} T_i \times \mathcal{K}}$
  $x_d^{-1}(t) = S$ indicates that a set S of tasks are executing on datacenter d at time t.

Therefore, we have the following **constraints**.

- Each datacenter has a limited number of slots.

$$|x_d^{-1}(t)| \leqslant a_d, \forall d \in \mathcal{D}, t \in \mathbb{R}^* \tag{1}$$

- Each task is completed after all the data needed is fetch and the task itself is executed.

$$c_{i,j} = d_{i,j} + e_{i,j} + \max_{\{i',i\} \in E_j} \frac{d_{i,i'}^j}{b_{x_{i,j}, x_{i',j}}}, \forall j \in \mathcal{K}, i \in J_j \tag{2}$$

- Each task can be deployed only after all of its precedent tasks are completed so it can get intermediate data from them.

$$d_{i,j} \geqslant c_{i',j}, \forall j \in \mathcal{K}, \forall \{i', i\} \in E_j \tag{3}$$

- Each task can be removed from the slot only after it is completed.

$$r_{i,j} \geqslant c_{i,j}, \forall j \in \mathcal{K}, i \in J_j \tag{4}$$

- Each task should be removed from the slot only after all of its data are transferred to needed tasks. And naturally, we want to release the slot as early as possible. Therefore,

$$r_{i,j} = d_{i',j} + \max_{\{i,i'\} \in E_j} \frac{d_{i,i'}^j}{b_{x_{i,j}, x_{i',j}}}, \forall j \in \mathcal{K}, i \in E_j \tag{5}$$

According to analysis above, we only need to schedule 2 functions, x and d.
Our objective is that

$$\min_{x,d} \frac{1}{K} \sum_{j=1}^{K} \max_{i \in J_j} c_{i,j} \tag{6}$$

However, function. 6 only ensures that the total completion time of all the jobs is optimal. Since the computing resources are limited and jobs may compete for them. In other words, the completion time of the jobs other than the longest job may vary greatly, which means the performance of them is extremely uneven. In order to solve the problem of unbalanced performance, we have to modify our optimizing goal to achieve ***max-min fairness***.

2.5 Max-Min Fairness

Mathematical proof

Let C_j denote the completion time of job j and $\mathcal{C}$ denote the set of C_j, $\forall j \in \mathcal{K}$. Since $c_{i,j}$ is the completion time of task i from job j, we have

$$\mathcal{C} = \{C_j : C_j = \max_{i \in \mathcal{T}_j} c_{i,j}, \forall j \in \mathcal{K}\}$$

Let vector $\langle C \rangle$ denote the permutation of $\mathcal{C}$ in descending order. In other words, vector $\langle C \rangle$ satisfies definition. 1. Let $\langle C^* \rangle$ denote the *lexicographically minimum* of all possible vector $\langle C \rangle$, which means $\langle C^* \rangle$ satisfies definition. 2. We claim that ***max-min fairness*** holds if and only if the linear programming(LP). 7 holds. Furthermore, if LP. 7 guarantees that LP 6 holds statistically.

$$\begin{aligned} &lexmin_{x,d} \langle C \rangle = (\langle C \rangle_1, \langle C \rangle_2, \cdots, \langle C \rangle_K) \\ &s.t. \text{ Constraints 1, 2, 3, 4 and 5} \end{aligned} \tag{7}$$

It is obvious that the only solution of LP. 7 is $\langle C^* \rangle$ mentioned in the last paragraph. Let us prove that C^* satisfies ***Max-min fairness***.

Proof (Result of LP 7 is max-min fairness).

For any other possible $\langle C \rangle$ except for $\langle C^* \rangle$, suppose the element $\langle C \rangle_k$ exists such that $\langle C \rangle_k > \langle C^* \rangle_k$. Since $\langle C^* \rangle$ is the *lexicographically minimum*, there exist $\langle C \rangle_{k'}$, where $k' > k$, such that $\langle C \rangle_{k'} < \langle C^* \rangle_{k'}$. Moreover, $\langle C^* \rangle_{k'} < \langle C^* \rangle_k$ holds because $\langle C \rangle$ is in descending order.

In conclusion, $\langle C \rangle_{k'} < \langle C^* \rangle_{k'} < \langle C^* \rangle_k$ holds and $\langle C* \rangle$ satisfies **max-min fairness**.

Vivid example

Then let us show a vivid example to explain the concept of ***max-min fairness***. As is shown in Fig. 2, the left picture shows an ordinary feasible solution with $< \langle C_1 \rangle = (6, 5, 3)$ and the right one is a better solution with $\langle C_2 \rangle_k = (5, 5, 5)$. $< \langle C_1 \rangle$ is inferior to $\langle C_2 \rangle$ because $< \langle C_1 \rangle$ do not even satisfy LP. 6.

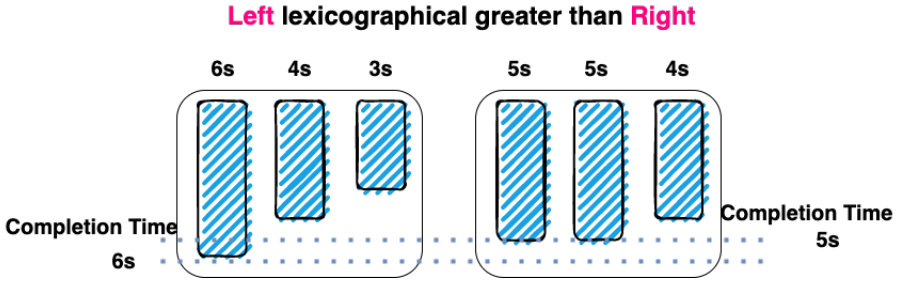

Fig. 2. Illustration of Max-Min Fairness

However, the effects of the two allocation plans are different even if they have the same total completion time. As is shown in Fig 3, the total completion time of $\langle C_1 \rangle$ is 5s, which is greater than that of $\langle C_2 \rangle$. But we claim that $\langle C_1 \rangle$ is better than $\langle C_2 \rangle$, because $\langle C_2 \rangle$ spends too many resources to execute job 3, which is **unfair** for job 2.

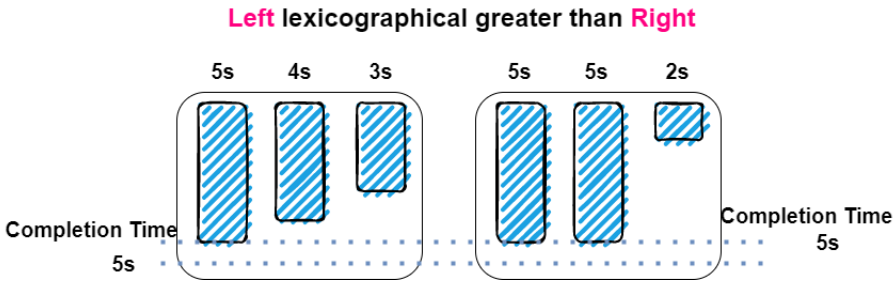

Fig. 3. Illustration of Max-Min Fairness

2.6 Conclusion

Based on the analysis above, we can naturally formulate the problem:

(**P1**) *Geo-distributed Job Scheduling Problem.* Given a set $\mathcal{K}$ of jobs with functions a and b, a set $\mathcal{D}$ of geo-distributed data centers with functions e and d, find two functions x and d such that

$$\min_{x,d} \frac{1}{K} \sum_{j=1}^{K} \max_{i \in J_j} c_{i,j}$$
$$s.t. \text{ Constraints 1, 2, 3, 4 and 5}$$

which is statistically equivalent to

$$lexmin_{x,d} \langle C \rangle = (\langle C \rangle_1, \langle C \rangle_2, \cdots, \langle C \rangle_K)$$
$$s.t. \text{ Constraints 1, 2, 3, 4 and 5}$$

3 Computational Complexity

In this section, we analyze the hardness of this problem. Intuitively, we may think this problem is NP-complete. Now we prove its NP-completeness.

Firstly, we formalize the decision version of this optimization problem.

(**P1′**) ***Geo-distributed Job Scheduling Problem.*** Given a set $\mathcal{K}$ of jobs with functions a and b, a set $\mathcal{D}$ of geo-distributed datacenters with functions e and d, and a time limit t, with constraints 1, 2, 3, 4 and 5, does there exist two functions x and d such that $\frac{1}{K} \sum_{j=1}^{K} \max_{i \in J_j} c_{i,j} \leqslant t$?

We use a simpler NP-complete scheduling problem to indirectly prove the NP-completeness of P1.

(**P2**) ***General Scheduling Problem.*** Given a set S of n jobs, a partial order $\prec$ on S, a weighting function W, a number of processors k and a time limit t, does there exist a total function f from S to $\{0, 1, \ldots, t-1\}$ such that

i if $J \prec J'$, then $f(J) + W(J) \leqslant f(J')$
ii for each J in S, $f(J) + W(J) \leqslant t$, and
iii for each $i, 0 \leqslant i < t$, there are at most k values of J for which $f(J) \leqslant i < f(J) + W(J)$

According to [5], **P2** is in $\mathscr{NP}$ if the encoding is such that at least n symbols are required to represent any n job problem. Therefore, we can prove the NP-completeness of **P1** polynomial transforming **P2** into **P1′**.

Given an instance of **P1**, $\Phi = (S, n, \prec, W, k, t)$, we construct an instance of Φ' **P2**, and prove that Φ is specifiable iff Φ' is satisfiable.

Let

$$\mathcal{J} = \{J_1\}, J_1 = (V_1 \cup D_1, E_1), \quad V_1 = \{1, 2, \ldots, n\}, D_1 = \varnothing, E_1 = \{\{i, j\} | i \prec j\}$$
$$e_{i,1} = W(i), d^1_{i,j} = 0, \quad \forall i, j \in J_1$$
$$\mathcal{D} = (D, E^*), \quad D = \{1\}, E^* = \{\{1, 1\}\}$$
$$a_1 = k, b_{1,1} = \infty$$
$$\Phi' = (\mathcal{K}, a, b, \mathcal{D}, e, d, t)$$

Therefore, obviously, the *Geo-distributed Job Scheduling Problem* is reduced to an *General Scheduling Problem*, and obviously, Φ is satisfiable iff Φ' is satisfiable.

Therefore, we have **P2** $\leqslant_p$ **P1′**. According to the **Self-Reducibility** of $\mathscr{NP}$, **P1′** $\leqslant_p$ **P1**. Due to the transitivity of $\mathscr{NP}$, **P2** $\leqslant_p$ **P1**, indicating that **P1** is NP-complete.

4 Model

The purpose of our model is to schedule multiple tasks belonging to different jobs into limited slots which are distributed among several geographically distributed datacenters, in order to minimize the average completion time and maintain **max-min fairness**.

However, due to the NP-Hardness of this whole problem, it is almost impossible to effectively obtain the optimal schedule of tasks in poly-time. Fortunately, through careful analysis of the DAG of jobs, we find that data dependency is a critical constraint which greatly degrades the parallelism of tasks. An inspiration naturally come up that we can consider data dependency functions as a "barrier" separating tasks into **stages**, which are sets of tasks that can be executed in parallel, and data dependencies exist between neighboring stages, indicating they can't be deployed on the datacenters at the same time. Therefore, we can hierarchically solve the whole optimization problem by transforming the minimum schedule unit from *task* to *tasksets*.

After transforming the *tasks* to *tasksets*, we choose some appropriate tasksets to form a *set collection*. Tasks executed in parallel under the arrangement of our scheduling algorithm form a *set collection*.

Because *set collection* represents the maximum parallelism, we assume that if we achieve our optimization goal for each set collection while maintaining **max-min fairness** for each *set collection*, then we achieve the global optimization goal. As a result of this, we are able to eliminate the dimension of time to establish a series simplified notations to help us representing our model. The list of the simplified notation is shown in Tab. 2.

Table 2. Simplified Notation List

Notation	Explanation
$\mathcal{K}$	The set of *jobs*, $\mathcal{K} = \{1, 2, \cdots, K\}$
$\mathcal{D}$	The set of *datacenters*, $\mathcal{D} = \{1, 2, \cdots, J\}$
$\mathcal{T}_k$	The set of parallel *tasks* of *job* i, $\mathcal{T}_k = \{1, 2, \cdots, n_k\}$
a_j	The capacity of available *slots* in *datacenter* j
$c^k_{i,j}$	The complete time of *task* i of *job* k, placed in *datacenter* j, where $i \in \mathcal{T}_k, j \in \mathcal{D}, k \in \mathcal{K}$
$e^k_{i,j}$	The execution time of *task* i of *job* k, placed in *datacenter* j, where $i \in \mathcal{T}_k, j \in \mathcal{D}, k \in \mathcal{K}$
S^k_i	The set of *datecenters* that *task* i of *job* k will use.
$d^{k,s}_i$	The input data that is stored in *datacenter* s of *task* i of *job* j, where $s \in S^k_i$
$b_{s,j}$	BandWidth from datacenter S to dataceter j, where $s, j \in \mathcal{D}$ and $s \neq j$
$b_{j,j}$	Bandwidth intra-dataceter j
$x^k_{i,j}$	A binary variable indicating if *task* i of *job* k is assgned to *dataceter* j
τ_k	Completion time of job k

A *taskset* is a set of tasks belonging to the same job. We consider that the tasks from the same taskset must be executed in parallel, because a particular task could be execute if and only if all the tasks from its predecent tastset have been executed.

We denote the set $\mathcal{T}_k$ as the *taskset* of job k, with $|\mathcal{T}_k| = K$

The completion time of each task i is jointly determined by two factors, one of which is the transferring time of the task's input data and the other is the execution time of it. we introduce two simplified variables $c_{i,j}^k$ and $e_{i,j}^k$ to represent the two deciding factors, respectively.

In respect to the execution time, $e_{i,j}^k$ denotes the execution time for the particular task labelled as i, placed in datacenter j and belonging to job k.

As for the transferring time, let S_i^k denote the set of datacenters where the input data of task i from job k are stored. Then, we use the variable $d_i^{k,s}$ to denote the amount of data stored in datacenter $s \in S_i^k$, which task i from job j needs to read. Since the rate of transferring data block $d_i^{s,k}$ from datacenter s to datacenter j is represented by $b_{s,j}$, we could generate the completion time calculation function

$$c_{i,j}^k = \begin{cases} 0 & S_i^k = \varnothing \\ \max\limits_{s \in S_i^k} \dfrac{d_i^{k.s}}{b_{s,j}} & otherwise. \end{cases} \tag{8}$$

This function indicates that when $S_i^k = \varnothing$, which means the particular i doesn't need input data to execute, so the transferring time is 0. Otherwise, the transferring time is decided by the longest transferring time of all the needed data. In the function. 8, $\frac{d_i^{k.s}}{b_{s,j}}$ denotes the transferring time of each data block and the maximum of them is the transferring time of the network.

The assignment of the each task is represented with a binary variable $x_{i,j}^k$. $x_{i,j}^k = 1$ means that task i from job k is assigned to datacenter j. Since the completion time of a job is deciding by its slowest task and the completion time of a task is composed of its transferring time and execution time, the completion time of job k, which is denoted by τ_k, is expressed as follows:

$$\tau_k = \max_{i \in \mathcal{T}_k, j \in \mathcal{D}} x_{i.j}^k (c_{i,j}^k + e_{i,j}^k) \tag{9}$$

Since all the slots in datacenters are general processors and shared among all the tasks from different jobs, the cases may exist when different tasks run out of all available resources and when they compute for some particular resources. Furthermore, because each task must remain in the slot until the transfer of data is done, deadlock may occur. For example, all the tasks already done conquer all the slots and need to send data, making that there is no empty slot for the next task to receive data. The two systematic problems are partially solved by maintaining *max-min fairness* and *deadlock-prevention*.

According to the analysis above, our work is divided into three main parts listed as follows:

Optimization Goal	Minimizing the completion time of all jobs
Constraints:	Maintaining max-min fairness
	Preventing deadlock

To some extent, the deadlock prevention is of greater significance than maintaining max-min fairness, for the reason that max-min fairness ensures efficiency while deadlock prevention ensures safety. In the case of deadlock, all the tasks already done have no idea of transferring data out, so at least one task must leave the slot without sending data to make room for the next task to execute. In such a case, the job which the task previously mentioned belongs to has to restart again, making huge resource waste. Generally, the process of scheduling job among geographically distributed datacenters could be divided into three main phases listed below. The division is for the purpose of optimization goal and the two main constraints are ensured in different phases of scheduling.

Three phases:

Phase 1: Organize tasks from the same job into a DAG and decide parallelism of tasks

Phase 2: Merge the parallel task sets into a pool and select tasks to execute concurrently

Phase 3: Schedule the selected task among geo-distributed computing slots

The whole process of our model is shown in Fig. 4.

Let us claim that we ensure deadlock prevention in phase 2 and maintain max-min fairness in phase 3. Later we will use three sub-sections to describe the solutions of these three phases in detail.

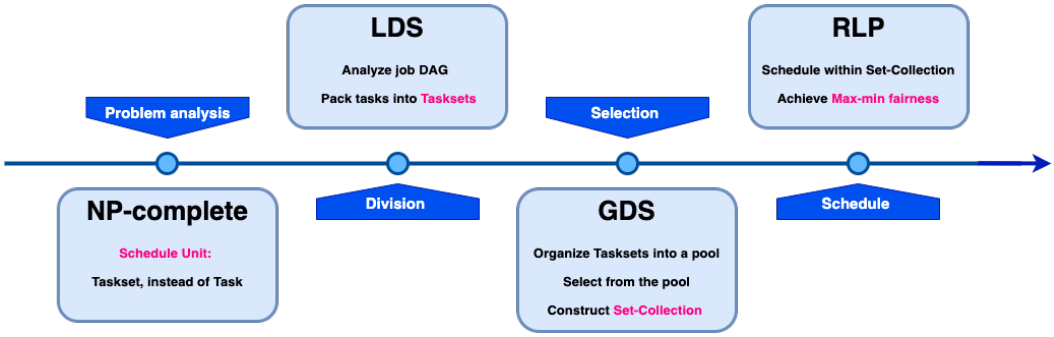

Fig. 4. Whole Process of Our Model

4.1 Phase 1(Division) LDS: Largest Depth Search Algorithm

In the first step, we divide tasks from the same job k, where $k \in \mathcal{K}$, into different task sets based on the precedent constraints. Because the tasks performed in one round do not overlap with the tasks performed in the next round, which means that for two task sets $\mathcal{T}_k$ and $\mathcal{T}_k'$ such that $\mathcal{T}_k \cap \mathcal{T}_k' = \varnothing$, we use $\mathcal{T}_k$ to denote the set of tasks that are executed concurrently for convenience.

The prior information we know before scheduling is the set of jobs $\mathcal{D}$ and the tasks each job contains. Let $\mathcal{W}_k$ denote the set of tasks from job k regardless of particular stage. For a single data analytical job k, which is essentially a parallel program, can be represented by a directed acyclic graph(DAG) $G = (\mathcal{W}_k, E_k)$, where $\mathcal{W}_k$ is the set of tasks and E_k is the set of directed edge that represents precedent constraints. Edge (i, j) exists if and only task j needs data from task i. Fig. 5 is a vivid example. Suppose $\mathcal{W}_k = \{t_1, \cdots, t_9\}$, which means job k has nine tasks to execute. We divide the parallel tasks into a *taskset* and we call the parallel process of execution as *stage*. For instance, *taskset* $\{t_1\}$ executes in *stage* 1 in parallel, *taskset* $\{t_2, t_3, t_4\}$ executes in *stage* 2 in parallel, and so on. What needs to be emphasized is that the stage of each task depends on the latest predecessor task. In the example, t_8 needs data from task t_3, t_5 and t_6, which have their own precedent constraints. t_8 has to be divided into *stage* 4 because its latest predecessors t_5 and t_6 are in *stage* 3.

Our goal is to analyze the DAG to divide the different tasks of each job into different *tasksets* that can be executed in parallel. We propose the **Largest Depth Search Algorithm** to solve this problem, as shown in Alg. 1.

The traditional breadth-first search algorithm is an algorithm for traversing a graph data structure. It starts at the some arbitrary node of a graph and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. Our modification is that **a node could be explored if and only if all of its predecessor node are explored**, which is consistent with our concept of *stage*.

We still use Fig. 5 as an example. Firstly, we add task t_1 into *taskset* 1. After that, we explore all its neighbours t_2, t_3, and t_4. So do t_5, t_6 and t_7. Our modification is reflected on exploring t_8. Traditionally, we consider t_8 is the successor node of t_3 and add t_8 into *taskset* 3. In our **LDS** algorithm, we check whether all of t_8's predecessors are explored after exploring t_3. At this moment, it is impossible for t_6 to be explored, so t_8 can't be explored either. Task t_8 is explored when all its predecessors t_3, t_5, and t_6 have been explored.

In phase 1, we apply the **LDS** algorithm to each job $k \in \mathcal{K}$ and generate the *tasksets* of each particular job. Then we find the union of these *tasksets* and get the collection of *tastsets* from all jobs. After that, we enter the second phase to select an appropriate combination of *tasksets* to execute in parallel.

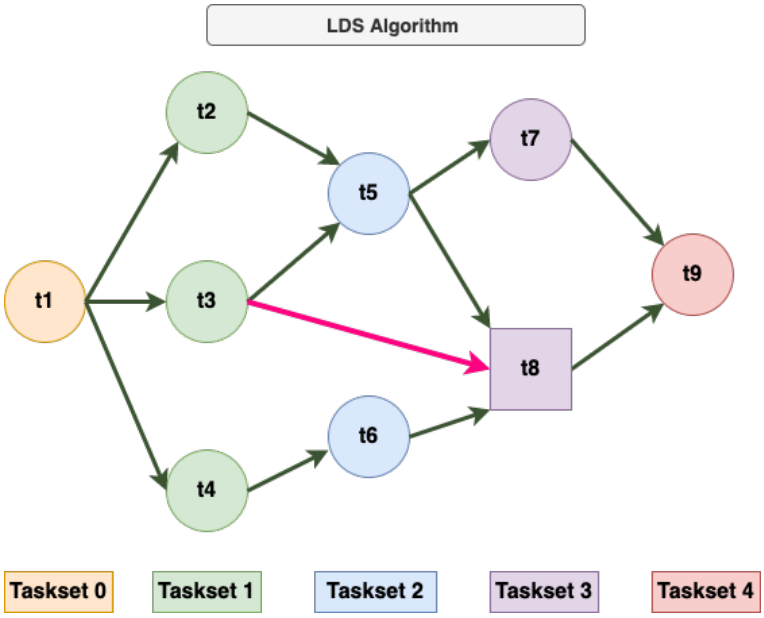

Fig. 5. Example of Largest Depth Search Algorithm

Algorithm 1: Largest Depth Search Algorithm

Input: Graph $G = (V, E)$, *source* s.
Output: Collection $\boldsymbol{T} = \{\boldsymbol{T}_1, \boldsymbol{T}_2, \cdots, \boldsymbol{T}_n\}$ of sets of tasks of the same depth.

foreach $u \in V$ **do**
  Depth(u)$\leftarrow \infty$;
end
Depth(s) $\leftarrow 0$;
$Q \leftarrow [s]$;
$n \leftarrow 0$;
while Q *is not empty* **do**
  $u \leftarrow$ DEQUEUE(Q);
  if *Depth*(u) $> n$ **then**
    $n \leftarrow$ Depth(u);
  end
  foreach $(u, v) \in E$ **do**
    if *Depth*(v) $= \infty$ **then**
      ENQUEUE(Q, v);
    end
    if *Depth*(v) $>$ *Depth*(u) $+ 1$ **then**
      Depth(v) $\leftarrow$ Depth(u) $+ 1$;
    end
  end
end
foreach $u \in V$ **do**
  $\boldsymbol{T}_{\text{Depth}(u)} \leftarrow \boldsymbol{T}_{\text{Depth}(u)} \cup \{u\}$;
end
OUTPUT($\boldsymbol{T} = \{\boldsymbol{T}_1, \boldsymbol{T}_2, \cdots, \boldsymbol{T}_n\}$);

Complexity Analysis.

1. **Time Complexity.** This algorithm will access each node twice and only twice, and each edge will be visited exactly once. As for the construction of $\boldsymbol{T_i}$ and $\boldsymbol{T}$, the amortized complexity is $O(1)$, and the total complexity is $(|V|)$. Therefore, the time complexity of this algorithm is $O(|V| + |E|)$.

2. **Spatial Complexity.** This algorithm needs to store data structures merely about the vertices, thus the complexity is $O(|V|)$ (the size of input data is neglected).

4.2 Phase 2(Selection) GDS: Greedy-based Deadlock-free Selection Algorithm

In this phase, we choose tasksets from *scheduling pool*, which stores tasksets that do not depend on unfinished tasks and have no mutual dependency. After that, we pack these chosen tasksets into a *set collection* and submit it to *RLP* in *Phase 3*. Now the point is how can we choose tasksets in every step that finally reaches optimum. Before solving this problem, we work on a problem caused by **Assumption 8***(No Cache Assumption)*, which is named ***deadlock*** by us.

Definition 5. ***Deadlock*** *is a circumstance that the number of tasks to be deployed plus the number of tasks waiting in the slots to provide data is greater than the number of slots, resulting in a schedule failure.*

We name this phenomenon *deadlock* because it quite assembles to the definition of *deadlock* in *Process Schedule Problem*, which has 4 necessary conditions:

1. *Mutual Exclusion.* Only one task at a time can use a slot.
2. *Hold and Wait.* A task holding the slot is waiting to transfer intermediate data to its targets, waiting until all its targets have fetch the data.
3. *No Preemption.* A slot can be released only voluntarily by the task holding it, after it has completed execution and transferred all intermediate data.
4. *Circular Wait.* There may exist a circle t_1, t_2, t_1 that t_1 is waiting for a slot to deploy whole t_2 is waiting to transfer data to t_1 thus can't release the slot immediately.

To solve this problem and prevent failure in *Phase 3*, we should examine the sufficient condition of *deadlock* during taskset-choosing time and avoid its occurrence.

Here, for simplicity, we temporarily use $d_{i,k}$ to denote the number of job i's tasks which need to reside in slots to provide data just before taskset k of job i, s_j denotes the stage that job j is at this time, and $\boldsymbol{T}_j, j \in \{1, 2, \cdots, m\}$ denotes the j-th taskset that will be deployed at this time. Suppose current time is t, let ϵ be the minimal time left of tasks in slots to complete. Then, during time $(t, t+\epsilon)$, the number of parallely executing jobs is

$$N = \sum_{j=1}^{m} |\boldsymbol{T}_j| + \sum_{j=1}^{K} d_{j,s_j}$$

According to constraint 1, we should ensure

$$N \leqslant \sum_{i=1}^{|D|} a_i$$

that is

$$\sum_{j=1}^{m} |\boldsymbol{T}_j| + \sum_{j=1}^{K} d_{j,s_j} \leqslant \sum_{i=1}^{|D|} a_i \tag{10}$$

Satisfying 10, we will definitely not fail in *Stage 3*.

Baseline: Threshold Selection Algorithm

To make the execution of all the tasks feasibly and avoid the deadlock, one trivial way to schedule the task sets is to set a threshold and select the task sets into a set collection based on a multi-level queue.

Thus, we propose **Threshold Selection Algorithm**, which is a **baseline** algorithm in our model, which ensures the basic performance on this scheduling problem. The whole algorithm is shown in Alg. 2.

The key idea of Alg. 2 is to main a queue to collect the set collection. If we set the threshold reasonably, the algorithm works as the baseline of the problem.

Algorithm 2: Threshold Selection Scheduler

Input: $\boldsymbol{Q} = (\boldsymbol{T_1}, \boldsymbol{T_2}, \cdots, \boldsymbol{T_n})$ is an array of tasksets; threshold td; $count$

```
Define SQ as a scheduling queue;
Define i = 0;
while i ⩽ threshold do
    SQ.enqueue(T_count);
    i+ = 1;
    count+ = 1;
end
OptimalAlgorithm(SQ);
SQ.clear();
if count ≠ n then
    Threshold Selection Scheduler(Q, td, count);
end
```

Complexity Analysis.

1. **Time Complexity.** Each queue operation like **enqueue** and **dequeue** is $O(1)$. Thus, the total time complexity is $O(n)$, showing this is a linear algorithm.
2. **Spatial Complexity.** Maintaining a queue leads to $O(threshold)$ space consumption. Thus the space complexity is $O(n)$, since $threshold \leqslant n$.

One problem of the baseline Threshold Selection Algorithm is that **the value of the threshold heavily depends on the data**. If the threshold is too small, the time consumption is large. If the threshold is too large, the deadlock may occur, which limits the value of the threshold. Thus, it's necessary for us to design a algorithm, which is more efficient, more flexible and more robust.

Greedy-based Deadlock-free Selection Algorithm

Obviously, schedule according to the order that tasksets enqueue can't fully utilize the parallelism and potential of these datacenters and the independency between different jobs. With more careful investigation and several additional assumptions based on facts, we develop a coarse-grained statistically optimal algorithm.

1. **Shortest-taskset-first Schedule Algorithm**

   Assumption 10:*(Task i.i.d. Assumption) Assume that the execution times of different tasks are independently identically distributed. That is to say,* $\forall j \in \mathcal{K}, i \in J_j$ *such that* $e_{i,j}$ *is independently identically distributed.*

   Therefore, for taskset $\mathcal{T}_j^k$, which denotes the k-th taskset of job j, its *run time*, that is, the time needed to finish it since it is scheduled is

$$e_j^k = \max_{j \in \mathcal{T}_j^k} e_{i,j} + \max_{\{i',i\} \in E_j} \frac{d_{i,i'}^j}{b_{x_{i,j}, x_{i',j}}}$$

   The latter term depends on the previous state when the taskset is assigned, whose difference can be neglected under statistical significance. The former term is a constant, which further simplifies our induction. Hence due to the max non-linearity, **the expectation of e_i^k monotonically increases as $|\mathcal{T}_i^k|$ increases**. So we can estimate e_i^k with the number of tasks job i have. Apply the idea of **greedy-max**, we can get the following algorithm which has the potential to minimize the average finish time of each taskset, which targets at a local optimality.

Algorithm 3: Shortest-taskset-first Schedule Algorithm

Input: $\boldsymbol{Q} = (\boldsymbol{T_1}, \boldsymbol{T_2}, \cdots, \boldsymbol{T_n})$ is an array of tasksets

```
Sort Q in ascending order of the number of tasks each taskset has;
T ← ∅;
for i ← 1; i ⩽ n; i ← i + 1 do
    if |T| + Σ_{j=1}^{|K|} d[j][c_j] ⩽ # Σ_{j=1}^{|D|} a_j then
        T ← T ∪ T_i;
    end
end
OptimalAlgorithm(T);
```

Complexity Analysis.

(a) **Time Complexity.** Sort takes $O(|\boldsymbol{T}| \log |\boldsymbol{T}|)$. While in each iteration, the algorithm needs to do a determination, which is $O(|\boldsymbol{T}|)$, and a $O(|\boldsymbol{T}|)$ union. There are in total $|\boldsymbol{T}|$ iterations. Therefore, the time complexity is $O(|\boldsymbol{T}|^2)$.

(b) **Spatial Complexity.** This algorithm needs to store data structures merely about the number of tasksets, thus the complexity is $O(|\boldsymbol{T}|)$ (the size of input data is neglected).

On the one hand, we greedily pick as many tasksets as possible; one the other hand, we pick those with as few tasks as possible. Therefore we can not only fully utilize the parallelism of slots but shorten the average run time for each taskset. Whereas our goal is to minimize the average run time of each job, which is nonequivalent to minimizing the average run time of tasksets. Therefore, based on the facts, we propose a new algorithm:

2. **Shortest-job-first Taskset Schedule Algorithm**
Assumption 11: *(Taskset i.i.d. Assumption) Assume that the numbers of tasks of different tasksets are independently identically distributed. That is to say,* $\forall i \in \mathcal{K}, k \in \{1, 2, \cdots, |\mathcal{T}_i|\}, |\mathcal{T}_i^k|$ *is independently identically distributed.*
According to this assumption and **Assumption 10**, we can conclude that e_i^k **is also i.i.d.** (This corollary neglects the mutual dependency of tasksets, however, under statistical significance, it is almost true). Therefore, a job's *run time*, that is, the time needed to finish a job continuously is

$$e_i = \sum_{k=1}^{\mathcal{T}_i} e_i^k$$

According to the **Central Limit Theorem**, the distribution of e_i approximates a normal distribution as the $|\mathcal{T}_i|$ becomes larger, that is,

$$e_i \xrightarrow{d} N(|\mathcal{T}_i|\mu, |\mathcal{T}_i|\sigma^2)$$

where μ is the mean of e_i^k, $\sigma^2 \leqslant \infty$ is the variance of e_i^k.
Thus we can estimate e_i according to $|\mathcal{T}_i|$. In other words, jobs have more tasksets are have longer expected time to run. Take the idea of **greedy-max**, we can get the following algorithm which has the potential to minimize the average finish time of each job, which targets at a global optimality (which is of coarse-grained optimality).

Algorithm 4: Shortest-job-first Taskset Schedule Algorithm

Input: $\boldsymbol{Q} = (\boldsymbol{T_1}, \boldsymbol{T_2}, \cdots, \boldsymbol{T_n})$ is an array of tasksets

```
Sort Q in ascending order of the number of stages the corresponding job has;
T ← ∅;
for i ← 1; i ⩽ n; i ← i + 1 do
    if |T| + Σ_{j=1}^{|K|} d[j][c_j] ⩽ # Σ_{j=1}^{|D|} a_j then
        T ← T ∪ T_i;
    end
end
OptimalAlgorithm(T);
```

Complexity Analysis.

(a) **Time Complexity.** Sort takes $O(|\boldsymbol{T}| \log |\boldsymbol{T}|)$. While in each iteration, the algorithm needs to do a determination, which is $O(|\boldsymbol{T}|)$, and a $O(|\boldsymbol{T}|)$ union. There are in total $|\boldsymbol{T}|$ iterations. Therefore, the time complexity is $O(|\boldsymbol{T}|^2)$.

(b) **Spatial Complexity.** This algorithm needs to store data structures merely about the number of tasksets, thus the complexity is $O(|\boldsymbol{T}|)$ (the size of input data is neglected).

In this algorithm, we greedily choose tasksets whose corresponding job has as few tasksets as possible. Thus apart from greatly utilizing parallelisms, we can shorten the average run time for each job, which corresponds with our goal. And as is proven that *SJF* is optimal for *Process Scheduling Problem*, *Shortest-job-first Taskset Schedule Algorithm* is also **statistically optimal**. However, since it doesn't directly manage the assignment of each taskset (which is done by *RLP*), this optimality is **coarse-grained**.

4.3 Phase 3(Schedule) RLP: Recursive Linear Programming Algorithm

After the **selection** phase, we get the union of *tasksets*, that is $\mathcal{T} = \bigcup_{k=1}^{K} \mathcal{T}_k$, where $K = |\mathcal{K}|$. In a separate stage, $\mathcal{T}_k$ is the *taskset* of job k. In the current **schedule** phase, our scheduler would decide the assignment of all the task in $\mathcal{T}$, aiming to optimize the worst performance achieved among all the jobs with respect to their job completion times, and then optimize the next worst performance without impacting the previous one, and so on. This recursive process of optimization obtains an optimal task assignment without resource capacities. Furthermore, we maintain max-min fairness in the **schedule** stage.

Based on the our definition. 1, 2, and 3, such an objective can be rigorously formulated as a *lexicographical minimization* problem shown below:

$$\underset{\boldsymbol{x}}{lexmin} \quad \boldsymbol{f} = (\tau_1, \tau_2. \cdots .\tau_K) \tag{11}$$

$$s.t. \quad \tau_k = \max_{i \in \mathcal{T}_k, j \in \mathcal{D}} x^k_{i.j}(c^k_{i,j} + e^k_{i,j}), \forall k \in \mathcal{K} \tag{12}$$

$$\sum_{k \in \mathcal{K}} \sum_{i \in \mathcal{T}_k} x^k_{i,j} \leqslant a_j, \forall j \in \mathcal{D} \tag{13}$$

$$\sum_{j \in \mathcal{D}} x^k_{i,j} = 1. \forall i \in \mathcal{T}_k, \forall k \in \mathcal{K} \tag{14}$$

$$x^k_{i,j} \in \{0, 1\}, \forall i \in \mathcal{T}_k, \forall j \in \mathcal{D}, \forall k \in \mathcal{K} \tag{15}$$

Constraint.12 is the definition of τ_k, which means the completion of the *taskset* $\mathcal{T}_k$ from job k in this stage. Constraint. 13 is ensures that the total number of tasks assigned to datacenter j, whichever jobs they are from, will not exceed the number of slots in datacenter j. Constraints. 14 makes sure that the particular task i from job k is assigned to just one datacenter. Constraint. 15 means the particular task i from job k is either assigned to datacenter j or not.

The optimization objective is a vector $\boldsymbol{f} \in \mathbb{R}^K$ with K elements. Each of them, that is τ_k, indicates the completion time of a particular job $k \in \mathcal{K}$. According to definition. 3, the linear programming shown above generates the optimal solution $\boldsymbol{f}^*$ which is lexicographically no greater than any $\boldsymbol{f}$ obtained with a

feasible assignment. The goal of optimization implies that the first largest element of $\boldsymbol{f}^*$ is the minimum among all $\boldsymbol{f}$. For an optimal solution $\boldsymbol{x}^* = \{\tau_1, \tau_2, \cdots, \tau_K\}$, the *taskset* with the first larger completion time is optimal. Among all the feasible *tasksets*, the *taskset* with the second larger completion time is also optimal in the solution $\boldsymbol{x}^*$. The goal of optimization ensures that for any $k \in \mathcal{K}$ such that τ_k is optimal.

In order to solve vector optimization with multiple objectives in function. **??**, we first consider the primary sub-problem of it, which is a linear programming optimization with a single objective. The primary sub-problem is shown below:

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & \max_{k\in\mathcal{K}}(\tau_k) \\ s.t. \quad & Constraints\ 12,\ 13,\ 14, and\ \ 15 \end{aligned} \tag{16}$$

Let us claim that **we could solve the linear programming(LP). 11 by recursively calling LP. 16**, which is name as **RLP-Algorithm**. After solving the sub-problem, the optimal worst completion time is achieved by job k^*, whose slowest task i^* is assigned to datacenter j^*. Then we fix the computed assignment of the slowest task of job k^*. After fixing k^*, the slowest task $i*$ from job k^* is assigned to datacenter j^*, which means the corresponding schedule variable $x^{k^*}_{i*,j*}$ should be removed from constraints. Meanwhile, all the other assignment variable corresponding to datacenters excluding j^* ought to be set to 0, which means for $\forall j \neq j^*, j \in \mathcal{D}$ such that $x^{k^*}_{i*,j} = 0$.

Since we have fixed the slowest task i^* of job k^*, which will occupy a slot of datacenter j^*, the corresponding capacity of datacenter j^* should be updated in the following rounds. For example, if $x^{k^*}_{i*,j*} = 1$ in the current round, $x^{k^*}_{i*,j*}$ is no longer a variable and $x^{k^*}_{i*,j} = 0$ for $j \neq j^*, j \in \mathcal{D}$ in the next round.

Literature [2] proposed a method to transform the LP. 16 to an equivalent linear programming 17 using *separable convex objective* and *unimodular linear constraint*. LP. 17 is a programmable linear programming and literature [2] proposes a algorithm, which is shown in Alg.5 to solve the original multiple-objective linear programming.

$$\begin{aligned} \min_{\boldsymbol{x},\boldsymbol{\lambda}} & \sum_{k\in\mathcal{K}}\sum_{i\in\mathcal{T}_k}\sum_{j\in\mathcal{D}}(\lambda^{k,0}_{i,j} + M^{c^k_{i,j}+e^k_{i,j}}\lambda^{k,1}_{i,j}) \\ s.t. & x^k_{i,j} = \lambda^{k,1}_{o,j}, \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & \lambda^{k,0}_{i,j} + \lambda^{k,1}_{i,j} = 1, \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & \lambda^{k,0}_{i,j}, \lambda^{k,1}_{i,j}, x^k_{i,j} \in \mathbb{R}^+, \forall k \in \mathcal{K}, i \in \mathcal{T}_k, j \in \mathcal{D} \\ & Constraints 12, 13 \end{aligned} \tag{17}$$

Algorithm 5: Recursive Linear Programming Algorithm

Input: Input data size:$d^{k,s}_i$; Bandwidth:$b_{s,j}$; Execution time $e^k_{i,j}$; Slot number:a_j;
Output: Task assignment $x^k_{i,j}, \forall k \in \mathcal{K}, \forall i \in \mathcal{T}_k, \forall j \in \mathcal{D}$;

```
K' ← K;
while K' ≠ ∅ do
    Solve the LP Problem 17 to obtain the solution x;
    Obtain x^{j*}_{k*,i*} =argmax_{x^k_{i,j}∈x} φ(x^k_{i,j});
    Fix x_{i*,j*}, j ∈ D;
    remove them from variable set x;
    Update the corresponding resource capacities in Constraints13;
    Set φ(x^{k*}_{i,j}) = x^{k*}_{i,j}(c^{k*}_{i*,j*} + e^{k*}_{i*,j*}), ∀i ∈ T_{k*}, ∀j ∈ D;
    Remove k* from K';
end
```

Complexity Analysis.

1. **Time Complexity.** Depending on the specific implementation of the LP algorithm, the time complexity differs. Let us denote it as $\mathscr{P}$, because overall it is polynomial. Other operations are of at most $O(|\mathcal{K}|^2) < \mathscr{P}$ complexity thus neglected. Since there are $O(|\mathcal{K}|)$ calls of LP solving, the time complexity is $O(\mathcal{K}\mathscr{P})$.

2. **Spatial Complexity.** Depending on the specific implementation of the LP algorithm, the space complexity differs. Let us denote it as $\mathscr{P}'$, because overall it is polynomial. Since at the same time only one LP is running, the spacial complexity is $\mathscr{P}\prime$. (Other operations are omitted since their complexities are at most $O(|\mathcal{K}|^2) < \mathscr{P}$)

5 Implementation Methodologies

In this section, we will briefly introduce the code frame and structure in implementing the algorithms we proposed, and show the full process of the execution of the whole program.

Basically, we use **Python 3** as the programming language, and utilize some common packages, including Numpy, Pandas, Pulp.

First, we will construct some **basic data structures** to make use of data in convenience.

Second, we introduce how we implement the two schedulers: **DAG Scheduler** and **Task Scheduler**. The two schedulers are corresponding to **Phase 1** and **Phase 2**.

Third, we are ready to implement the scheduling algorithm, as **Phase 3** describes. We propose **Solver**, a class that implements the algorithm **RLP**(Recursive Linear Programming).

5.1 Basic Data Structure

To utilize the data in **ToyData.xlsx**, we design several basic data structures. Three of them are the most significant **Job, Task, DataCenter**, since we will use them in each part of the implementation: , which are shown in Fig. 6.

- `Job`
  `Job` is a class that have several elements, such as **job name, task dictionary and DAG**.
  `Job name` functions as the name of the current job, to help us distinguish between different jobs. For example, 'A' and 'B' are two job's names.
  `Task dictionary` stores the tasks the job includes. We make use of **dictionary** in Python to construct the storage element, and this is why is is so called. For example, job 'A' has tasks 'tA1' and 'tA2', then we use the key **'tA1'** to get the corresponding task, represented by the class **Task**.
  `Directed Acyclic Graph(DAG)` keeps the DAG of one job, which enables the scheduler decode the DAG and assign different task sets. **DAG Scheduler** mainly depends on this element.
- `Task`
  `Task` is a class simulating a task, with **required data**, **execution time** and **task name**. This class is actually a fundamental data structure in our program, which is often nested in other data structures like **Job** and **DAG**.
- `DataCenter`
  `DataCenter` is a class including the information of data centers, such as **bandwidth**, **available slots**, **data center name**. Each of them is decoded from the data provided in **ToyData.xlsx**.

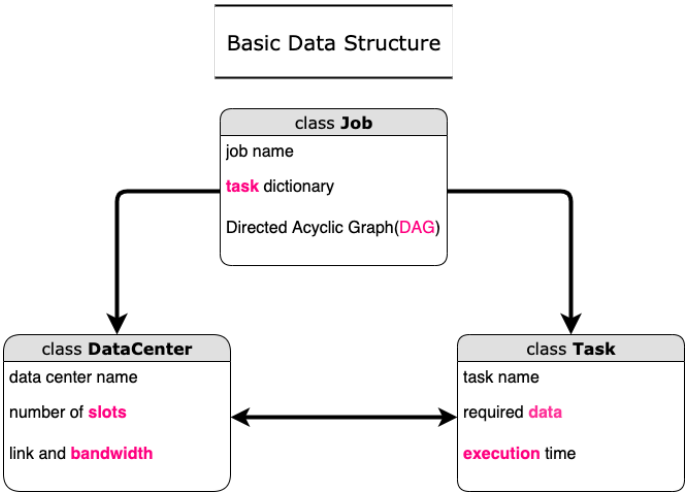

Fig. 6. Basic Data Structure

5.2 Schedulers

Since we have prepared data and data structures, we are ready to assign these tasks and jobs. To form the tasks into task sets and then set collections, we first construct **DAG Scheduler** to divide the tasks in jobs into different stages and form task sets, then construct **Task Scheduler** to collect the task sets into set collections. The process is shown in Fig. 7.

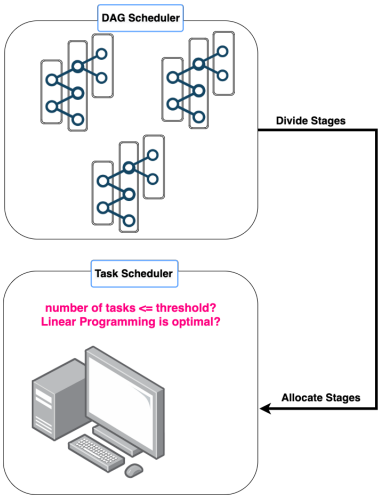

Fig. 7. DAG Scheduler and Task Scheduler

In **DAG Scheduler**, we divide stages and form them into different task sets based on the algorithm **LDS**(Largest Depth Search). Note that Task set is defined as a set of tasks in the same stage of one job. To be more specific on LDS, we utilize the adjacent matrix to represent the connection.

In **Task Scheduler**, we continue to using the task sets and form them into set collections. Set Collection is defined as a collection of task sets. We may do some judgments to collect the task sets, such as judging whether to reach the **threshold** or whether linear programming can be optimal.

Note that the difference of our baseline algorithm and our Shortest-Job-First Greedy Depth Based algorithm lies in the task scheduler. Therefore, in **Task Scheduler**, we implement two task allocation methods.

5.3 Solver

The core part of the structure is **Solver**, which implements **RLP**(Recursive Linear Programming).

Solver consists of two parts, also shown in Fig. 8:

- **Linear Programming Formulator**. Taking the set collection and data centers as the input, the formulator will formulate the input into a Linear Programming problem, as shown in Phase 3.

  To make the process of the formulation more automatic, we construct a full pipeline to decode data and transform data into the objective function and constraints, by the help of the Python package Pulp. After the transformation, Solver use the algorithm **RLP** and outputs the placement of the tasks and the finish time.
- **Data Center Updator**. After the placement outputs, we update the data center based the need of task data. This is very essential since many tasks have data dependence.

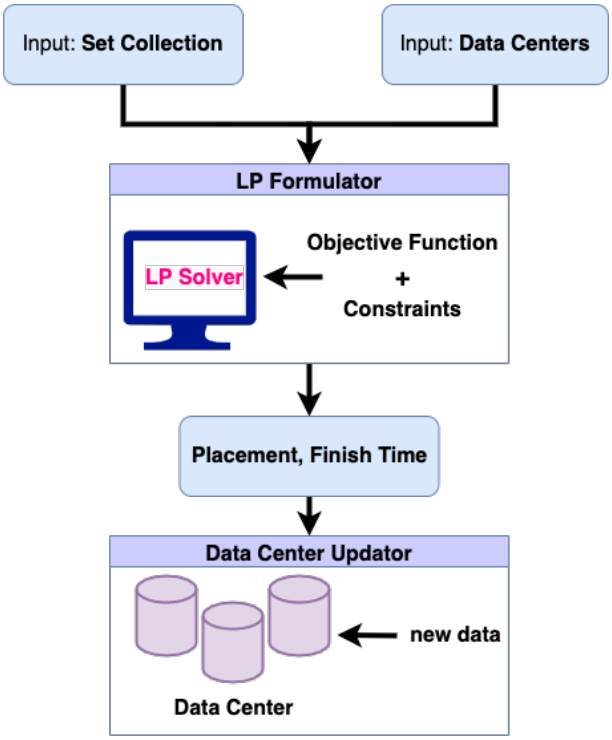

Fig. 8. Solver consists of LP Formulator and Data Center Updator.

6 Simulation

In this section, we will the simulation results of our program on different data, from **Toy Data** provided to **Random Data** we generate.

To make the illustration of the usage of different algorithms direct, we use the threshold selection algorithm as our **Baseline** and the Shortest-Job-First and Greedy Depth Based algorithm as our improved version, called **SGD**.

6.1 Toy Data

In the toy data, we set the threshold of our baseline as **5**, which means the maximal size of each set collection is smaller than 5. This setting of the threshold is based on the constraints. From the experiment, large thresholds may lead to the infeasible solution of programming, and small thresholds may lead to the too much consuming time. Thus we have $threshold = 5$ in the baseline.

The **final completion time** of all jobs:

- Baseline: 33.358333s
- SGD: 22.416667s

The **average completion time** of all jobs:

- Baseline:
$$t_{avg} = \frac{4.25 + 14.25 + 17.78 + 22.03 + 25.11 + 33.358}{6} = 19.463$$
- SGD:
$$t_{avg} = \frac{5.0 + 10.0 + 16.0 + 16 + 18.75 + 22.417}{6} = 14.693$$

Thus, our algorithm SGD has much improvement compared with the baseline algorithm.

Fig. 9 shows the two algorithms' execution steps. Baseline needs 7 steps to finish all the jobs, while SGD only needs 4 stages.(Each step represents a set collection). The advantage of SGD is shown clearly.

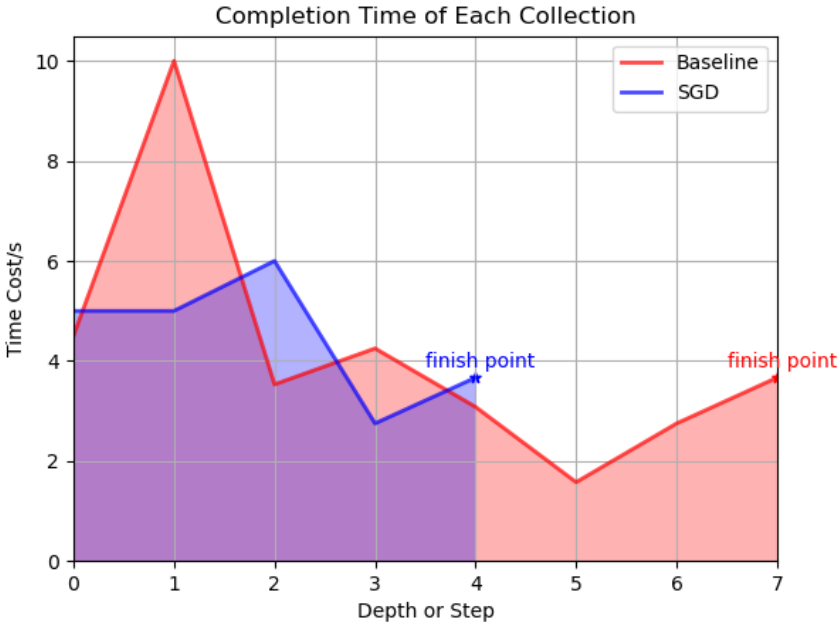

Fig. 9. Completion Time of Each Collection

6.2 Random Data Generation

From the intuition that **Simple Copy-Paste is a Strong Data Augmentation Method** [8], we generate some new data based on copy-and-paste. What's more, to induce the fluctuation of the execution, we use **Gaussian Distribution** to create new data.

One reasonable explanation of our data generation is that the data in the natural world mostly distributes in Gaussian Distribution.

We first do estimation on the current Execution Time, to get a simple view of the data, as shown in Fig. 10. Since the data is in small scale and the distribution is not clear, we treat the mean and the variance of the data as the parameter in Gaussian Distribution. Thus, The generation execution time data follows the distribution:

$$t_{ex}\ N(\mu, \sigma^2), where\ \mu = 2.09, \sigma^2 = 0.945$$

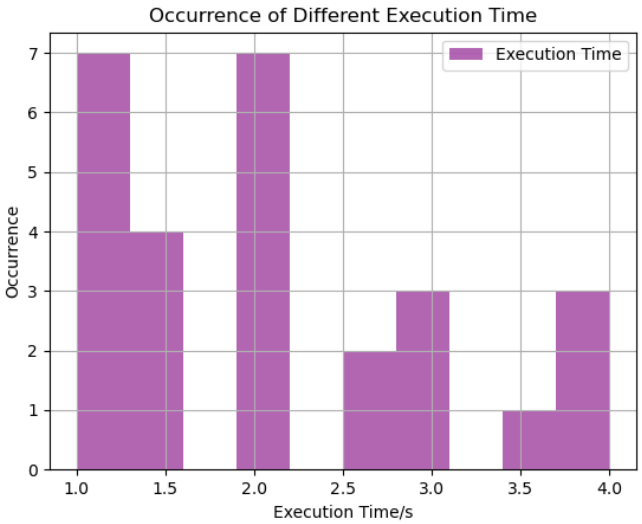

Fig. 10. Occurrence of Execution Time

We use the random Execution Time simulator to experiment more, to test the performance of our Baseline and SGD.

6.3 More Experiments

Based on the Execution Time Simulator, we have more experiments, which robustly show that our SGD Algorithm is better than Baseline.

First three experiments have **Gaussian Distribution**:

$$t_{ex}\ N(\mu, \sigma^2), where\ \mu = 2.09, \sigma^2 = 0.945$$

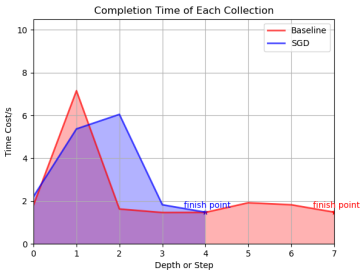

Fig. 11. Experiment 1

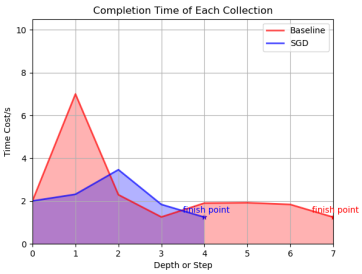

Fig. 12. Experiment 2

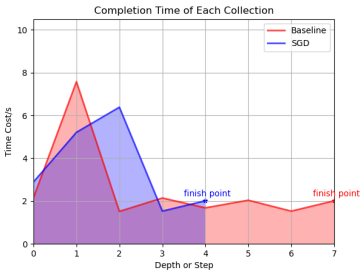

Fig. 13. Experiment 3

The following three experiments test the performance of the algorithms on Gaussian Distribution with **High Variance**:

$$t_{ex}\ N(\mu, \sigma^2), where\ \mu = 2.09, \sigma^2 = 6$$

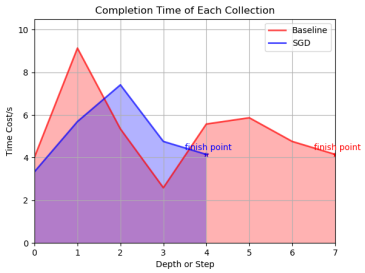

Fig. 14. Experiment 4

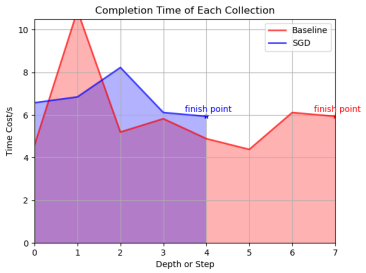

Fig. 15. Experiment 5

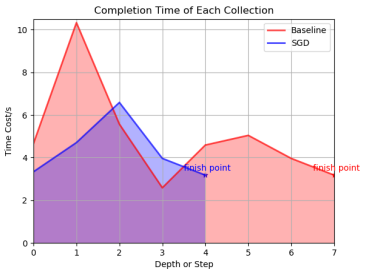

Fig. 16. Experiment 6

The last three experiments test the performance of the algorithms on Gaussian Distribution with **High Mean** and **High Variance**:

$$t_{ex}\ N(\mu, \sigma^2), where\ \mu = 10, \sigma^2 = 6$$

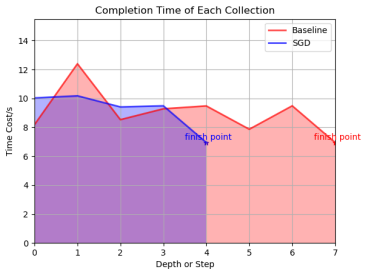

Fig. 17. Experiment 7

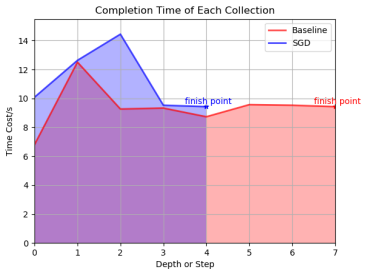

Fig. 18. Experiment 8

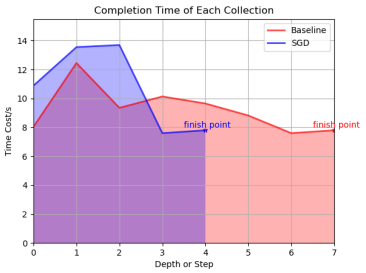

Fig. 19. Experiment 9

In conclusion, SGD algorithm is much more efficient than Baseline. However, we also see a trend that as the execution time becomes larger and more unstable(high variance), Baseline Algorithm is a more stable version, which makes the completion of each stages not too late. Thus, this is a trade-off actually: **Fairness or Efficiency? That's a question.**

7 Conclusion

In this paper, we have conducted a hierarchical study of the Multiple Multi-stage Geo-distributed Job Scheduling Problem, which is a task assignment problem competing independent jobs consisting of exclusive and mutual independent tasks among computing slots in datacenters. To solve this complex problem, we first **formulated** the task into a **optimization problem**, which is a mixture of real-value term and integer constraints, posing great difficulty to efficiently solve. We proved the **NP-completeness** of this problem. Inspired by the inner dependency of this problem, we developed **SGD**, *Shortest-Job-First and Greedy Depth Based Hierarchical Programming Algorithm*, to solve it nearly optimally. We started from using **LDS**, *Largest Depth Search Algorithm* to divide jobs into tasksets based on data dependency. At the top level, we implemented **GDS**, *Greedy-based Deadlock-free Selection Algorithm*, to carefully choose tasksets to schedule with coarse-grained optimality while avoiding *deadlock*. Then in the bottom level, we use **RLP**, *Recursive Linear Programming Algorithm*, to optimally schedule tasks in our chosen tasksets to achive optimal completion time and max-min fairness. Apart from those, we also analyzed the complexity of our algorithm, which is **polynomial**. We finally tested our algorithm on toy data and random data generated based on Gaussian Distribution. Well visualization is also done.

Acknowledgements

- Great thanks to Professor.Xiaofen with Gao and Professor.Lei Wang for teaching us with the valuable knowledge necessary for our research and giving us valuable instructions.
- Thanks for CS214: Algorithm and Complexity. This course gives us a great opportunity to work as a team and perform advanced algorithm on the problem in reality.
- Thanks Ta.Haolin Zhou and Ta. Yihao Xie for explaining the problem to us with great patience.
- Thanks Department of Computer Science, Shanghai Jiao Tong University for opening this very rewarding course *Algorithm and Complexity* and giving us the chance to improve ourselves.

References

1. Bo Li, Scheduling Jobs across Geo-Distributed Data Centers
2. Li Chen, Shuhao Liu, Baochun Li, and Bo Li, Scheduling Jobs across Geo-Distributed Datacenters with MaxMin Fairness, IEEE Transactions on Network Science and Engineering (TNSE) Journal, **6**(3), 488–500 (2019)
3. Q. Pu, et al., Low latency geo-distributed data analytics, Proc. ACM Conf. Special Interest Group Data Commun., pp. 421–434 (2015)
4. Z. Hu, B. Li, and J. Luo, Flutter: Scheduling tasks closer to data across geo-distributed datacenters, Proc. IEEE Conf. Inf. Comput. Commun., pp. 1–9 (2016)
5. J.D. Ullman, NP-complete scheduling problems, Journal of Computer and System Sciences, Volume 10, Issue 3, 1975, pp. 384–393, ISSN 0022-0000
6. Bingchuan Tian, Chen Tian, Bingquan Wang, Bo Li, Zehao He, Haipeng Dai, Kexin Liu, Wanchun Dou, Guihai Chen, Scheduling dependent coflows to minimize the total weighted job completion time in datacenters, Computer Networks, Volume 158, 2019, pp. 193–205, ISSN 1389-1286,
7. Yue Zeng, Baoliu Ye, Bin Tang, Songtao Guo, Zhihao Qu, Scheduling coflows of multi-stage jobs under network resource constraints, Computer Networks, Volume 184, 2021, 107686, ISSN 1389-1286,
8. Ghiasi, Golnaz, et al. Simple Copy-Paste is a Strong Data Augmentation Method for Instance Segmentation. arXiv preprint arXiv:2012.07177 (2020).

Appendix

Python Code for LDS

```
def BFS_mark(self , job_name):
        """
        Use BFS to mark the tasks in job, give depth to each task
        """
        gmatrix = self.job_dict[job_name].graph_matrix

        cur_depth = 0
        for i in range(gmatrix.shape[0]):
            # get current node's depth, or update it from -1 to 0
            if(self.job_dict[job_name].depth_vector[i]==-1):
                self.job_dict[job_name].depth_vector[i] = cur_depth
            else:
                cur_depth = self.job_dict[job_name].depth_vector[i]
            # update others
            for j in range(gmatrix.shape[1]):
                if gmatrix[i][j]==1:
                    if self.job_dict[job_name].depth_vector[j] < cur_depth + 1:
                        self.job_dict[job_name].depth_vector[j] = cur_depth + 1
```

Test Results on Toy Data

Baseline:

```
step 0 cost time 4.500000
step 1 cost time 10.000000
step 2 cost time 3.533333
step 3 cost time 4.250000
step 4 cost time 3.083333
step 5 cost time 1.575000
step 6 cost time 2.750000
step 7 cost time 3.666667
Step0:
Task tA1, Execution Time 1, need data: ['A1', 'A2'] place in DC[4]
Task tA2, Execution Time 1, need data: ['A1', 'A2'] place in DC[4]
Task tB1, Execution Time 2, need data: ['B1', 'B2'] place in DC[1]
Task tC1, Execution Time 4, need data: ['C1', 'C2'] place in DC[2]
Step1:
Task tB2, Execution Time 1, need data: ['B1', 'B2', 'tB1'] place in DC[3]
Task tC2, Execution Time 1, need data: ['C2', 'tC1'] place in DC[2]
Task tD1, Execution Time 2, need data: ['D1', 'D2', 'D3'] place in DC[9]
Task tD2, Execution Time 2, need data: ['D1', 'D2', 'D3'] place in DC[9]
Task tE1, Execution Time 4, need data: ['E1', 'E2', 'E3'] place in DC[4]
Step2:
Task tC3, Execution Time 2, need data: ['C1', 'tC1', 'tC2'] place in DC[2]
Task tD3, Execution Time 1, need data: ['D2', 'tD1', 'tD2'] place in DC[7]
Task tD4, Execution Time 1, need data: ['D1', 'D3', 'tD2'] place in DC[5]
Task tE2, Execution Time 2, need data: ['E2', 'E4', 'tE1'] place in DC[1]
Task tE3, Execution Time 3, need data: ['E1', 'E3', 'tE1'] place in DC[8]
Step3:
Task tD5, Execution Time 1, need data: ['D2', 'tD3'] place in DC[7]
Task tE4, Execution Time 2, need data: ['E2', 'tE2'] place in DC[6]
Task tE5, Execution Time 1, need data: ['E3', 'E4', 'tE3'] place in DC[8]
Task tF1, Execution Time 4, need data: ['F1'] place in DC[9]
Step4:
Task tE6, Execution Time 3, need data: ['E1', 'tE3', 'tE5'] place in DC[4]
Task tF2, Execution Time 2, need data: ['F2', 'tF1'] place in DC[1]
Task tF3, Execution Time 1, need data: ['F3', 'tF1'] place in DC[9]
Task tF4, Execution Time 3, need data: ['F4', 'tF1'] place in DC[9]
Step5:
Task tF5, Execution Time 1, need data: ['F1', 'F2', 'F3', 'tF2', 'tF3'] place in DC[9]
Task tF6, Execution Time 1, need data: ['F1', 'F4', 'tF4'] place in DC[9]
Step6:
Task tF7, Execution Time 2, need data: ['F3', 'F5', 'tF5'] place in DC[9]
Task tF8, Execution Time 2, need data: ['F2', 'F5', 'tF5', 'tF6'] place in DC[9]
Step7:
Task tF9, Execution Time 3, need data: ['F1', 'F3', 'F4', 'F5', 'tF7', 'tF8']
place in DC[9]
-----Baseline Finish. Final Time:33.358333----
-----Execution Time of  Baseline: 0.635820-------
```

SGD:

```
depth 0 cost time 5.000000
depth 1 cost time 5.000000
depth 2 cost time 6.000000
depth 3 cost time 2.750000
depth 4 cost time 3.666667
Depth0:
```

```
Task tA1, Execution Time 1, need data: ['A1', 'A2'] place in DC[4]
Task tA2, Execution Time 1, need data: ['A1', 'A2'] place in DC[1]
Task tB1, Execution Time 2, need data: ['B1', 'B2'] place in DC[1]
Task tC1, Execution Time 4, need data: ['C1', 'C2'] place in DC[2]
Task tD1, Execution Time 2, need data: ['D1', 'D2', 'D3'] place in DC[7]
Task tD2, Execution Time 2, need data: ['D1', 'D2', 'D3'] place in DC[9]
Task tE1, Execution Time 4, need data: ['E1', 'E2', 'E3'] place in DC[4]
Task tF1, Execution Time 4, need data: ['F1'] place in DC[9]
Depth1:
Task tB2, Execution Time 1, need data: ['B1', 'B2', 'tB1'] place in DC[3]
Task tC2, Execution Time 1, need data: ['C2', 'tC1'] place in DC[2]
Task tD3, Execution Time 1, need data: ['D2', 'tD1', 'tD2'] place in DC[7]
Task tD4, Execution Time 1, need data: ['D1', 'D3', 'tD2'] place in DC[5]
Task tE2, Execution Time 2, need data: ['E2', 'E4', 'tE1'] place in DC[1]
Task tE3, Execution Time 3, need data: ['E1', 'E3', 'tE1'] place in DC[8]
Task tF2, Execution Time 2, need data: ['F2', 'tF1'] place in DC[1]
Task tF3, Execution Time 1, need data: ['F3', 'tF1'] place in DC[9]
Task tF4, Execution Time 3, need data: ['F4', 'tF1'] place in DC[9]
Depth2:
Task tC3, Execution Time 2, need data: ['C1', 'tC1', 'tC2'] place in DC[2]
Task tD5, Execution Time 1, need data: ['D2', 'tD3'] place in DC[7]
Task tE4, Execution Time 2, need data: ['E2', 'tE2'] place in DC[6]
Task tE5, Execution Time 1, need data: ['E3', 'E4', 'tE3'] place in DC[8]
Task tF5, Execution Time 1, need data: ['F1', 'F2', 'F3', 'tF2', 'tF3'] place in DC[9]
Task tF6, Execution Time 1, need data: ['F1', 'F4', 'tF4'] place in DC[9]
Depth3:
Task tE6, Execution Time 3, need data: ['E1', 'tE3', 'tE5'] place in DC[4]
Task tF7, Execution Time 2, need data: ['F3', 'F5', 'tF5'] place in DC[9]
Task tF8, Execution Time 2, need data: ['F2', 'F5', 'tF5', 'tF6'] place in DC[9]
————Depth Based Baseline Finish. Final Time:22.416667————
————Execution Time of Depth Based Baseline: 0.599426————
```