Aladdin+H: Efficient Flow-based Scheduling of Long Running tasks in Heterogeneous Production Clusters

Abstract

Tasks with multidimensional resource requirements in long running tasks (LRTs) usually face complex placement constraints and heterogeneous machines (e.g., the number of CPU cores, RAM size). Existing efforts may perform poorly since they cannot provide an elegant mechanism to choose the heterogeneous machines for each task, especially when considering various constraints. Therefore, scheduling LRTs with multidimensional resource requirements on heterogeneous machines brings new challenges to attain global optimization objectives, such as high resource efficiency and minimized constraint violations.

We build Aladdin+H, a novel flow-based scheduling method with the awareness of heterogeneous machines to address these challenges: (i) it extends the flow's capacity to express multidimensional resource requirements, and proposes a clustering-based cost assignment mechanism to support heterogeneous machine awareness. (ii) it designs an optimized minimum cost maximum flow (MCMF) algorithm to achieve a high resource efficiency without any constraint violations. Experiments on an Alibaba production trace show that Aladdin+H increases resource efficiency by as much as 30% and reduces violated constraints by up to 10% compared with state-of-the-art schedulers.

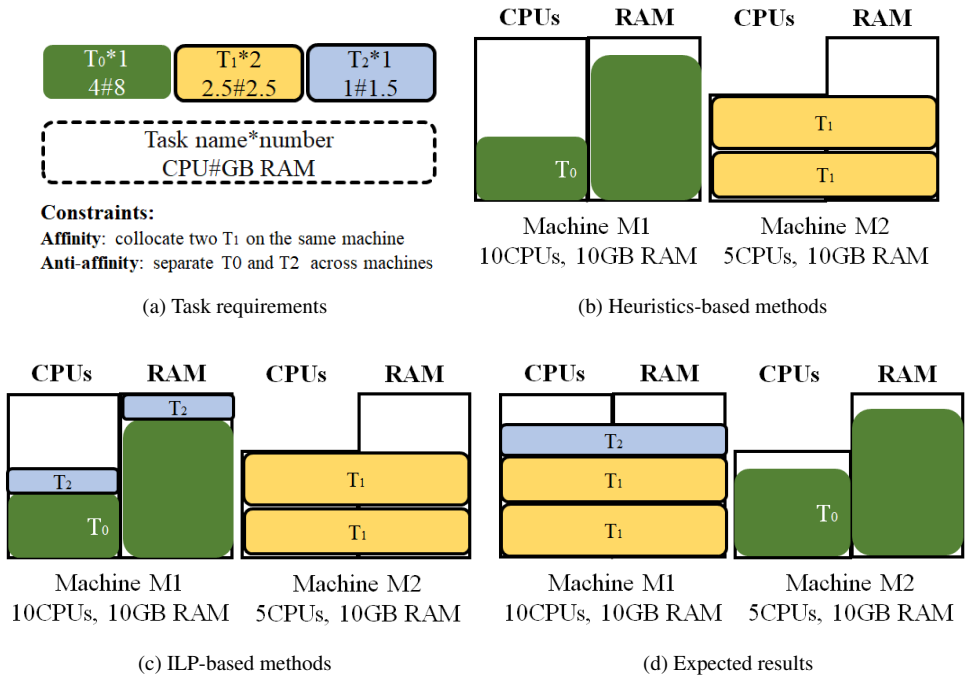
1 Introduction

Today's production clusters usually consist of **heterogeneous machines** with different number of CPU cores and RAM size because enterprises need to continue to replace unusable and old machines. In addition, those clusters should host various long running tasks (LRTs) [1] with multi-resource requirements [4, 16, 40] (e.g., the number of CPU cores, RAM size and bandwidth) and meet complex placement constraints [17]. For example, a typical Alibaba cluster includes more than 20 kinds of heterogeneous machines, some of them have 28 CPUs and 80,392 MB RAM while some other ones have 88 CPUs and 354,544 MB RAM [1]. Furthermore, the **"machine heterogeneity"** [37] brings new challenges for clusters to attain global optimization objectives [13], such as high resource efficiency and minimized constraint violations.

Current schedulers do not fully support **machine heterogeneity**. (i) Some of them focus on scheduling massive tasks simultaneously [15, 24, 36] based on **heuristic rules** [16–18]. They always assume machines are homogeneous so that the machine selection of packing tasks has no impact on resource efficiency [16]. However, this assumption leads to severe resource fragmentation and low resource efficiency on heterogeneous environments. As shown in Figure 1b, It may first schedule T_0 to M_1 because a machine with more available resources can host more tasks theoretically, this placement may minimize the number of used machines and increase resource efficiency. Unfortunately, after scheduling the two tasks T_1 to M_2, it cannot deploy T_2 because both M_1 and M_2 encounter **resource fragmentation** and none of them can satisfy T_2's resource requirements. Conversely, if T_0 selects M_2 instead of M_1, it can get an optimal result as shown in Figure 1d. This implies that **using a preference order for each task to choose heterogeneous machines can minimize resource fragmentation.** (ii) Some other studies employ **integer linear program (ILP)** to address this challenges [12, 13]. They balance resource efficiency, the number of used machines and constraint violations by different weights [12, 13]. However, these studies cannot work well on heterogeneous machines. As shown in Figure 1c, it may deploy T_0 and T_2 on M_1 to increase resource efficiency [13, 18, 32, 40], but this packing violates the constraint of separating T_0 and T_2 across machines (Figure 1a). To make matter worse, it cannot discover the violated constraint before completing all tasks' placement, although swapping T_0 and T_1 can get the optimal result during scheduling. This implies that **employing a kind of task migration mechanism can reduce constraint violations.** In summary, ignoring the above two new factors can cause resource fragmentation and constraint violations for state-of-the-art schedulers to support heterogeneous environments. And our analysis shows that the average CPU utilization of Alibaba's production clusters is just about 30% [1].

In our view, existing heuristic-based and ILP-based approaches cannot fully support machine heterogeneity because of the disadvantage of model expression [19, 25, 31]. In an ideal scheduling model, it can express the preference

[1] Such as machine learning and microservices



(a) Task requirements

(b) Heuristics-based methods

(c) ILP-based methods

(d) Expected results

Figure 1: An illustrative example. All tasks must be scheduled at the same time. Two tasks T_1 must be deployed on the same machine.Tasks T_0, T_2 must be deployed on different machines.

order for each task to choose heterogeneous machines to minimize resource fragmentation, and it can detect violated constraints and repack the tasks to get optimal results. Fortunately, flow network model is a feasible method to support the two requirements [5, 15]. Scheduling LRTs with multi-resource requirements on heterogeneous machines can convert to a minimum cost maximum flow (MCMF) problem [15, 22]. In this context, each task or machine is a vertex, and the edge from a task to a machine means a possible placement decision. Moreover, (i) we can use various edge costs for each task to express the order how to select heterogeneous machines to minimize resource fragmentation [5], and we call this **fragmentation awareness**. (ii) We can use negative cycle optimality to support task migration and repacking when we encounter constraint violations at runtime [5](Section 2.1 for detail), and we call this **task repacking with conflict detection**.

As far as we know, the fragmentation awareness problem based on flow network models has not been discussed. While the latter one just studied [15, 22] to support fairness, priority, and data locality. But in our scenario, we should support the tasks with multi-resource requirements, and avoid any task's resource interference. For CPU and RAM resources, we use a reservation mechanism to avoid interference. For the other resources, we plan to employ the term of "affinity" and "anti-affinity" to minimize interference [1, 3]. (i)**Anti-affinity**. It may be desirable to place tasks on different machines to minimize resource competition (e.g., disk, network) among tasks. (ii)**Affinity**. It is often beneficial to collocate the tasks of the same machine, e.g., to reduce the network traffic and minimize network competition.

We believe our work makes the following advancements.

- We extend flow's capacity to express multidimensional resource requirements, and apply a clustering-based cost assignment mechanism to express the preference order between tasks and heterogeneous machines to minimize resource fragmentation.
- We support affinity and anti-affinity, and design an optimized minimum cost maximum flow algorithm to support task repacking with conflict detection, which make high-quality placements by maximizing resource efficiency and avoiding constraint violations.
- We compare our approach to state-of-the-art schedulers with an Alibaba trace. Our experiments show that Aladdin+H increases resource efficiency by 30% and reduces violated constraints by 10% with sub-second placement latency.

The rest of the paper is organized as follows. Section 2 presents the problem statement and analysis, and Section 3 provides the design of Aladdin+H with an optimized minimum cost maximum flow algorithm. The evaluation

of Aladdin+H is in Section 4, followed by a discussion of related work in Section 5. Finally, the conclusion of our work is in Section 6.

2 Problem statement and analysis

We describe the scheduling problem based on a flow network model through an example, and illustrate why we can meet Alibaba's scenario.

2.1 Classical flow network

As shown in Figure 2, a MCMF algorithm takes a directed flow network $GN = (V, E)$ as input [5]. Each edge $(i, j) \in E$ has a cost c_{ij} and a maximum multidimensional and linear capacity u_{ij}. In addition, there are four kinds of vertices ($v \in V$): the source $s \in V$, the task $T \in V$, the machine $M \in V$ and the sink $t \in V$, as shown in Table 1.

Table 1: Symbols in the flow network

Symbol	Description
s	the starting vertex
T_i	each task is a vertex T_i
M_k	each machine is a vertex M_k
t	the ending vertex
c_{ij}	the edge cost between vertex i and vertex j
u_{ij}	the edge capacity between vertex i and vertex j
f_{ij}	the flow from vertex i to vertex j

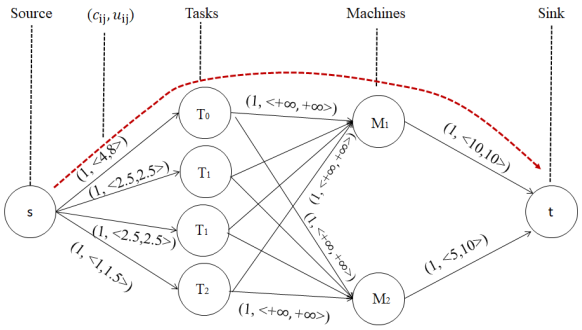


Figure 2: A flow network example for Figure 1. We assume the cost of each edge is 1. The edge (s, T_i)'s capacity is the task T_i's resource requirements. While the edge (M_j, t)'s capacity is the machine M_j's available resources.

In this paper, if we can find a path like $s \rightarrow T_0 \rightarrow M_1 \rightarrow t$ without exceeding the capacity constraints on any edges, we call it a flow or a placement decision $f_{T_0M_1}$. For instance, the task T_0 will be scheduled on the machine M_1 (dot line in Figure 2). Informally, a MCMF algorithm must optimally route all available flows from the source s to the sink t to complete the placement decisions of all tasks. To understand the algorithm, we formalize the goal to meet Equation 1, while respecting the flow feasibility constraints of **flow conservation** (Equation 2 [6]) and **capacity constraint** (Equation 3 [6]). Here, the term "**maximize**" represents the goal of deploying as many tasks as possible, and the term "**minimize**" means the minimum number of used machines (for all deployed tasks without constraint violations). In addition, both Equation 2 and 3 mean that the flow must be non-negative, and one task's resource demands should be no more than the available resources of its host machine.

$$Minimize(\sum_{(i,j)\in E} c_{ij} \cdot (Maximize\ f_{ij}))\ subject\ to \quad (1)$$

$$\sum_{k:(j,k)\in\{G,M,t\}} f_{jk} - \sum_{i:(s,i)\in\{s,T\}} f_{si} \geq 0, \forall j \in V \quad (2)$$

$$and\ 0 < f_{ij} \leq u_{ij}, \forall (i,j) \in E \quad (3)$$

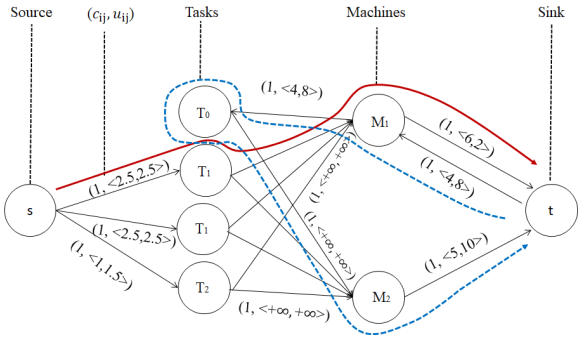


Figure 3: A residual network for Figure 2. We also assume the cost of each edge is 1.

Figure 3 shows a residual network after a placement decision. We deleted some edges and updated the capacity. For example, we deleted edge (s, T_0) since we scheduled the task T_0. We updated edge (M_1, t)'s capacity from $< 10, 10 >$ to $< 6, 2 >$ because running the task T_0 on the machine M_1 would reduce some available resources. Moreover, we had new edges (t,M_1) and (M_1,t_0) with capacity $< 4, 8 >$ [2], and these new edges could introduce a new cycle (dot line in Figure 3). This implied that the task T_0 could migrate from M_1 to M_2.

Then we continued our MCMF algorithm to schedule the task T_1, we found that M_1 did not have enough RAM resource ($2 < 2.5$). Then our MCMF algorithm may migrate the task T_0 from M_1 to M_2 and complete the placement of T_1. In fact, this is the concept of **negative cycle optimality** in the flow network theory [11, 13, 16]. Finally,

[2] It is equals T_0's resource requirements.

our MCMF algorithm would pack tasks T_0 and T_2 to the machine M_2, while packing two tasks T_1 to M_1.

In this case, the classical MCMF algorithm does not concern much about resource fragmentation, and always encounter conflicts when repacking tasks. we call them as **fragmentation unawareness** and **task repacking without conflict detection**, respectively.

- **Fragmentation unawareness**. As shown in Figure 1(d), we found that the ratio of RAM to CPU for M_1 and M_2 are 1 and 2, respectively. If T_0 (its ratio is 2) prefers M_2, two tasks T_1 (its ratio is 1) prefer M_1, then we can deploy all tasks without repacking extra tasks.
- **Task repacking without conflict detection**. We may deploy tasks T_0 and T_2 on the same machine M_2 and violate the anti-affinity constraint [3]. Because classical MCMF algorithms only check the capacity constraint (Equation 3) to make a placement decision. Unfortunately, we cannot express the constraints of affinity and anti-affinity by using various edge costs since these constraints are not linear relation.

2.2 Fragmentation Awareness

Figure 4 shows our idea to support fragmentation awareness in flow network. We employ a new kind of vertex G_i to represent the machine group, and the machines in the same machine group vertex have the same or similar resource characteristics, such as the ratio of RAM to CPU.

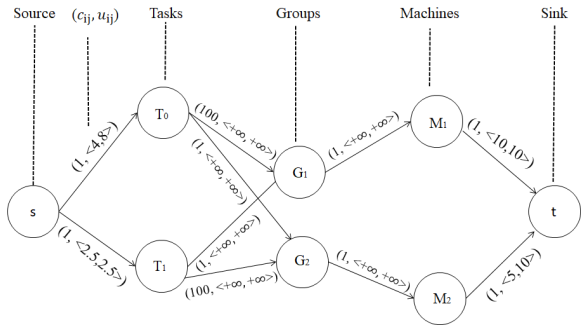


Figure 4: Introducing machine group vertices to support fragmentation awareness. We just consider T_0 and T_1 in Figure 1

Obviously, we can find two flows by using MCMF algorithms. One is $s \rightarrow T_0 \rightarrow G_2 \rightarrow M_2 \rightarrow t$, the other is $s \rightarrow T_1 \rightarrow G_1 \rightarrow M_1 \rightarrow t$. The reason why T_0 prefers G_2 to G_1 is that the cost of edge (T_0, G_2) is less than the edge (T_0, G_1)'s ($1 < 100$), and MCMF algorithms prefer the flow with the minimum cost. Further, we find that those placement decisions are optimal because we do not need to migrate any tasks if we continue to deploy the remaining tasks as shown in Figure 1.

Therefore, **we think the machine group vertices with the optimal cost can express the preference order for each task to choose heterogeneous machines, which can minimize resource fragmentation.**

2.3 Task Repacking with Conflict Detection

.

Figure 5 shows our idea to support task repacking with conflict detection in flow network. We employ a blacklist for each machine vertex M_i to represent the tasks with constraint conflicts. After completing the placement of T_0, we would add T_2 to M_2's blacklist since we have a constraint that T_0 and T_2 cannot be deployed on the same machine. Further, MCMF algorithms would ignore the flow $s \rightarrow T_2 \rightarrow G_2 \rightarrow M_2 \rightarrow t$, although T_2 prefers G_2 to G_1 because of the cost of edge (T_2, G_2) is less than the edge (T_2, G_1)'s.

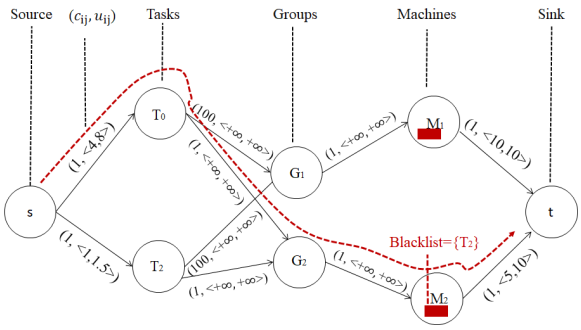


Figure 5: Introducing blacklist to support task repacking with conflict detection. Considering T_0 and T_2 in Figure 1

Therefore, **we think using the set theory to express the constraint conflicts, and extending machine vertex design with blacklist is an elegant way to support task repacking with conflict detection.**

3 Aladdin+H Design

As described in Section 2, we can extend flow network with fragmentation awareness and task repacking with conflict detection to meet Alibaba's requirements. Then we need to deal with the following challenges. (i) How to generate machine group vertices with optimal costs? (ii) How to express affinity and anti-affinity?

3.1 Architecture

As shown in Figure 6, after Aladdin+H collects machine runtime statuses by the **Node Monitor**, the **Machine**

[3] T_2 can deploy to either M_1 (the ratio is 1) or M_2 (the ratio is 2), since the ratio of RAM to CPU of T_2 is 2/3.

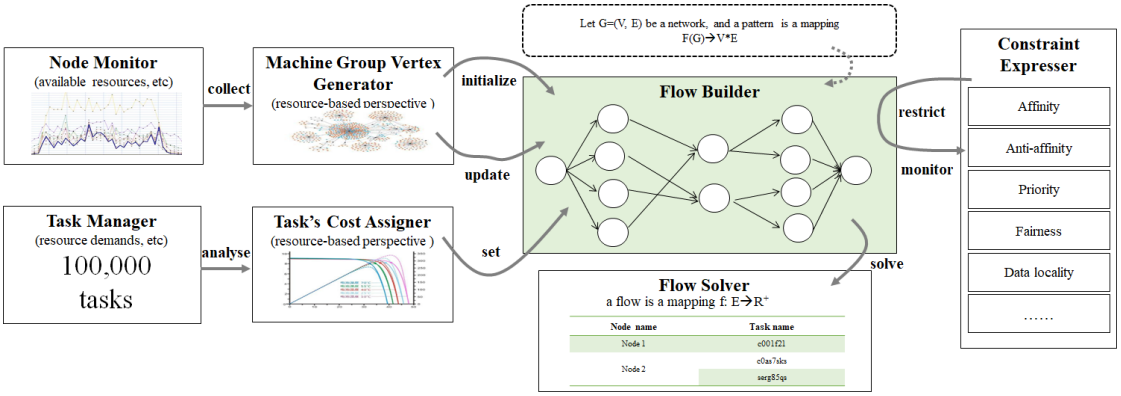


Figure 6: Aladdin architecture

Group Vertex Generator divides the heterogeneous machines into several classifications from a resource-based perspective. When massive tasks $\{T_1, T_2, ..., T_i\}$ arrive at the **Task Manager**, the **Task's Cost Assigner** automatically sets each edge with a non-negative cost in order to support fragmentation awareness. Moreover, the **Flow Builder** constructs the flow network from the outputs of the **Machine Group Vertex Generator** and the **Task's Cost Assigner** during the initialization phase. Further, the **Constraint Expresser** is used for discovering and reducing violated constraints at runtime. At last, Aladdin+H employs the **Flow Solver** to solve the flow network problem based on an optimized MCMF algorithm. Meanwhile, the **Constraint Expresser** and the **Machine Group Vertex Generator** update the flow network structure during the execution.

3.2 Machine Group Vertex Generator

To make the optimized MCMF algorithm works well on heterogeneous machines for those tasks with multi-resource requirements, we first extend the flow network's capacity to support multi-resources. Then we generate machine group vertices automatically to be aware of the heterogeneity by proposing a clustering-based method.

Extend the flow network's capacity. The capacity u_{ij} in the maximum flow can be denoted as N-tuples $(u_{ij}^1, u_{ij}^2, \cdots, u_{ij}^n)$. Formally, let $\mathbb{R}$ denotes the set of real numbers, let $\mathbb{R}^n$ denotes the vector space of real n-vectors. Then the capacity u_{ij} can be denoted by Equation 4.

$$u_{ij} = \begin{bmatrix} u_{ij}^1 \\ u_{ij}^2 \\ u_{ij}^3 \\ . \\ u_{ij}^n \end{bmatrix}, u_{ij}^k \in \mathbb{R} \tag{4}$$

$$u_{ij}^{'} = u_{ij} - f_{ij}, \forall k, u_{ij}^k > f_{ij}^k \tag{5}$$

In this context, every element in this tuple is a non-negative number, and the flow value should not exceed the allowed capacity in any dimensions between vertices i and j. Here, we refer to u_{ij}^k as the k^{th} dimension of u_{ij}.

Generate machine group vertices during initialization phase. We divide heterogeneous machines into several classifications [4] by using a K-means algorithm [23].

Given a set of n data points (machine instances) $D = \{M_1, ..., M_n\}$ in $\mathbb{R}^d$ and an integer K, the **K-means problem** is how to determine a set of K centroids $G = \{G_1, ..., G_k\}$ in $\mathbb{R}^d$ so as to minimize the following **error function** in Equation 6.

$$E(C) = \sum_{M \in D}^{k=1,\ldots,K} min \left\| M_j - G_k \right\|^2 \tag{6}$$

This is a combinatorial optimization problem since it is equivalent to finding the partition of the n instances in K groups. Informally, a typical K-means algorithm includes the following steps:

1. Randomly choose an initial K centers, $G_s = \{G_1, ..., G_k\}$.
2. For each $k \in \{1, ..., n\}$, set the cluster G_k to be the set of points in M_j (machines) that are closer to G_k than they are to G_p, for all $k \neq p$.
3. For each $k \in \{1, ..., n\}$, reset G_k to be the center of all points in G_k: $G_k = \frac{1}{|G_k|} \sum_{x \in G_j} x$.
4. Repeat steps 2 and 3 until G no longer changes.

Finally, we have a set of K centroids $G = \{G_1, ..., G_k\}$ in $\mathbb{R}^d$, and each element in the set G is a machine group vertex.

[4] As described above, we should support the tasks with multi-resource requirements, and avoid any task's resource interference. For CPU and RAM resources, we use a reservation mechanism to avoid interference. For the other resources, we plan to employ the term of "affinity" and "anti-affinity" to minimize interference (Section 3.4).

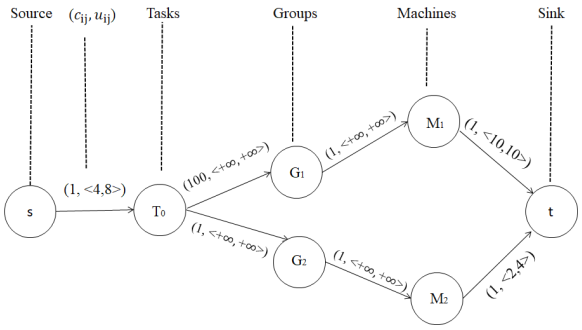


Figure 7: A scheduling example of a two-machine cluster and one task.

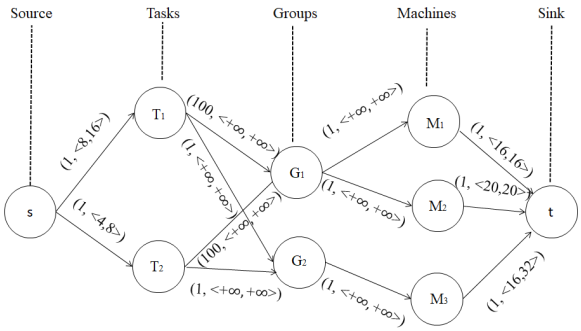


Figure 8: A scheduling example of a three-machine cluster and two tasks.

3.3 Task's Cost Assigner

When massive tasks arrive, we employ $d\left(T_i, G_j\right)$ denotes the distance between the task vertices $\{T_i\}$ and the machine group vertices $\{G_j\}$. Equation 7 is a Euclidean distance which is used to evaluate the difference between two vertices, function ch represents the resource characteristics for T_i^k and G_j^k, such as ratio of CPU to RAM. k is the k^{th} dimension of each vertex. In addition, G_j represents a machine group, which is the result from the K-means algorithm. In this context, we get the distance from a task T_i to each machine group G_j, and sort them in an ascending order. Then we apply a geometric sequence to model the value from a specified T_i to each G_j, as shown in Equation 8. In this paper, the smaller distance means the task and the machine have the similar resource characteristics, such as the ratio of RAM to CPU.

$$d(T_i, G_j) = \sqrt{\sum_{k=1}^{N} (ch(T_i^k) - ch(G_j^k)^2} \tag{7}$$

In Equation 8, x is N^{th} element in the distance list from a specified T_i to all G_j using Equation 7. It expresses the preference order for each task to choose heterogeneous machines, and the smaller distance means the better placement. As shown in Figure 8, the ratio of RAM to CPU of T_1 and T_2 is 2, and it is the same as the G_2's. So the cost from T_1 to G_1, G_2 is 100 and 1 according to Equation 8. In this context, T_1 and T_2 prefer M_3 and we get optimal placements. Because we only need one machine to host T_1 and T_2. Otherwise, we need two machines to host them.

$$cost(T_i, G_j) = 100^{x-1} \tag{8}$$

In addition, after completing a placement, we notice the $MachineGroupVertexGenerator$ to update flow network structure. As shown in Figure 7, the cost from T_1 to G_1, G_2 is 100 and 1, respectively. And the default cost of all other edges are 1.

- **If M_2 has enough resources**. Our algorithm would choose M_2 to host T_1 since the our algorithm prefers the flow with the minimum cost. And it is an optimal placement and it preserves the M_2's resource characteristics (the ratio of RAM to CPU is still equal to 2).

- **If M_2 cannot provide enough resources**. There are no available resources on M_2. So we have to deploy T_1 on M_1. However, this placement would affect the M_1's characteristics, in which the ratio of RAM to CPU has changed from 1 to $2/3$. In this case, we should update the related G's value or add a new machine group G with the value of $2/3$ [5].

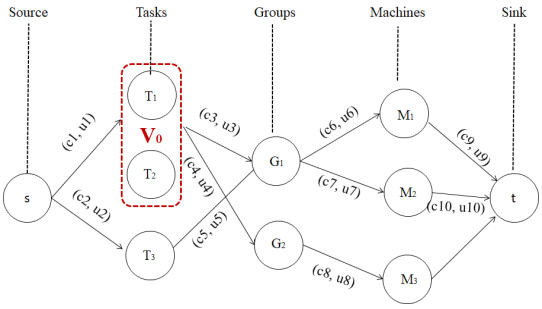
Briefly, **we think classifying machines and setting the preference order for each task can be used to support fragmentation awareness and avoid resource fragmentation.**

3.4 Constraint Expresser

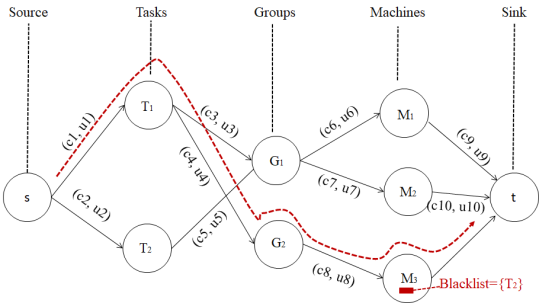
The next question is how to update the flow network structure at runtime by considering affinity and anti-affinity. As shown in Figure 6, typical constraints in production include fairness, data locality, task dependency, priority, affinity and anti-affinity [11, 15, 18]. The first three were expressed in a previous work [15]. In this paper, we focus on how to extend the flow network to support affinity and anti-affinity.

Affinity. Collocating the tasks of the same machine can reduce the network traffic and minimize network competition. As shown in Figure 9 (a), we introduce a new virtual vertex V_j to replace a group of tasks with the affinity constraint. Affinity follows the "all or nothing" principle: these tasks should be deployed at the same time, or none of them should be deployed.

[5] after completing the placement of T_1, the available resources of M_1 is 12 CPUs with 8GB RAM



(a) Affinity



(b) Anti-affinity

Figure 9: Affinity and anti-affinity constraints

Anti-affinity. Placing tasks on different machines can minimize resource competition (e.g., disk, network) among tasks. We introduce a blacklist for each machine vertex. As shown in Figure 9 (b), we assume T_1 and T_2 cannot be collocated, and denoted by an anti-affinity constraint $p = \{T_1, T_2, 0\}$. After we have deployed T_1 on M_3, Aladdin+H added T_2 to M_3's blacklist by Equation 9. Then, we continue to deploy T_2, and we cannot deploy it to M_3 by Equation 10, even if M_3 has enough resources.

$$blacklist(M_i).add(p.get(key)) \tag{9}$$

$$deployed(T_i) = \begin{cases} 1, T_i \notin blacklist(M_i) \\ 0, T_i \in blacklist(M_i) \end{cases} \tag{10}$$

Briefly, **we think introducing blacklist for machine vertex to adding this detection during algorithm runtime is an effective extension of flow network. It can be used for discovering and reducing violated constraints.**

3.5 Optimized MCMF

So far, we have explained why Aladdin+H can increase resource efficiency and reduce constraint violations. Then, we implemented an optimized MCMF algorithm. Table 2 summarizes the worst-case complexities of those frequently-used MCMF algorithms. The number of edges is much larger than the number of vertices in Aladdin+H. For example, if we have 100,000 tasks and 10,000 machines in a cluster, the number of E and V could be 10^9 and $110,000$, respectively. We think the shortest augmenting path algorithm can outperform the others because of its lowest time complexity.

In this context, Aladdin+H finds the shortest path from the vertex s, through T_i, G_j and M_k, to the vertex t. As described above, G_j can represent the similar resource characteristics for both tasks and machines. So if T_i has been deployed on M_k, the available resources of M_k still have the similar characteristics for other tasks having the same resource demands as T_i's. So M_k is not likely unbalanced among multi-resources. Hence, our flow network can avoid resource fragmentation in heterogeneous machines.

Table 2: Worst-case time complexities for MCMF. V is the number of vertices, E the number of edges.

Algorithm	Worst-case complexity
Cycle canceling	$O(VE^2)$ [10]
Shortest augmenting path	$O(VE \log V)$ [10]
Capacity scaling	$O(E^2 \log V)$ [6]
Cost scaling	$O(V^2E)$ [6]

As shown in Algorithm 1, our MCMF algorithm, which is based on the shortest augmenting path, takes a directed flow network $GN = (V, E)$ as input (see Lines 1-2). Then we use a clustering-based cost assignment method in flow network to identify heterogeneous machines (see Lines 15-22). Furthermore, we design an optimized MCMF algorithm based on the shortest augmenting path with constraint awareness to get global optimization objectives (see Lines 3-14 and 23-39).

3.6 Discussion

Robustness is one of the most important reasons that flow-based schedulers can automatically optimize placements even if they get near optimal intermediate results as shown in Figure 1(b). But unlike the classical flow network, a task with 4CPUs cannot be broken down into two or more smaller tasks. Although our MCMF algorithm's robustness has not been proven theoretically, it is still working well in the Alibaba trace.

In addition, modern clusters support a diverse mix of workloads. They consists of many short-lived tasks and a number of LRTs. Aladdin+H can also use a flow-based scheduling method considering data locality, task dependency and fairness for short-lived tasks [15], although this paper focuses on how to support LRTs.

Further, we can support the scenarios for task arrivals and completions by using incremental MCMF algorithms, although this paper mainly talks a one-time assignment.

Algorithm 1 Minimum cost maximum flow algorithm

Require:
All machines M
All tasks T
All constraints $Cons$={affinity, anti-affinity}
Ensure:
the relationship between task and machine (aka flow)
init G = getClassificationVertics(M)
init GN = createGraphSupportMutiResources (s, t, initCosts(T), G, M)
init flow $f = 0$
while getShortestPath() is Not NULL **do**
  p = getShortestPath()
  $f = f$ + getFlow(p)
end while
Function initCosts
SORTING($DistanceList$)
for Vertex ver in C **do**
  x=DistanceList.getIndex(ver)
  $cost(ver) = 100^{x-1}$
end for
END Function
Function getClassificationVertics
init k = groupSize
init C = Random selection K points
$G^{`} = \frac{1}{|C|} \sum_{x \in G_j} x$
while $\sum_{M \in D}^{k=1,\ldots,K} min \left\| G_k^{`} - G_k \right\|^2$ **do**
  update G with $G^{`}$
end while
return G
END Function
Function getShortestPath
init $minDist$ = INFINITY
init $Q = graph$.getAllVertices()
while $Q \neq \phi$ **do**
  Vertex $current = Q$.dequeue()
  for vertex $next \in$ neigbors(current) **do**
    if dist(s, next) < dist(s,current) + dist(current,next) **then**
      $minDist$ = dist (s, next)
    end if
    if !meet($Cons$) **then**
      roll back
    end if
  end for
end while
return $path$
END Function

4 Evaluation

We now evaluate how well Aladdin+H meets its goals. (i) Can Aladdin+H reduce fragmentation and improve resource efficiency on heterogeneous environments compared with state-of-the-art schedulers? (ii) Can Aladdin+H avoid constraint violations with sub-second placement latency?

4.1 Methodology

Workload. We replay a production workload trace from a 12,000-machine Alibaba cluster [1]. Table 3 shows the original number of machine types is 23, and the type is named by the CPU number and the RAM size. We observe that there is a significant difference among these machine types. For instance, one type has 28 CPUs and 80,392 MB RAM, while another has 88 CPUs and 354,544 MB RAM. In addition, the machines having almost the same resources may belong to different machine types. For example, some machines have 32 CPUs and 128,740 MB RAM, while some others have 32 CPUs and 129,089 MB RAM.

Table 4 shows the number of task types is 31, and the type is also named by the CPU number and the amount of RAM. In addition, we find that all the tasks and machines have the similar resource characteristics. For example, some tasks have 4 CPUs and 16 GB (16,384 MB) RAM, while some machines have 32 CPUs and 128 GB (128,740 MB) RAM. In this case, the ratio of RAM to CPU for both tasks and machines is equal.

Besides, we find that approximately 50% and 20% of LRTs have anti-affinity and affinity constraints, respectively. In addition, we find that several LRTs cannot be co-located with at least other 2,000 tasks due to anti-affinity constraints [1].

Table 3: Machine types. The key is the CPU number and the amount of RAM (the unit of RAM is MB).

Machine type	Number	Machine type	Number
28-80,392	1,040	28-112,646	868
32-128,952	112	64-258,356	320
40-96,517	344	24-64,265	88
32-128,740	74	56-160,038	3,670
16-64,266	1,152	84-338,160	49
32-129,089	84	32-128,739	121
64-257,665	212	64-258,360	168
32-96,518	84	32-128,713	122
88-354,544	1,956	64-193,716	192
40-96,683	160	56-225,705	52
36-515,820	240	56-467,436	52
56-225,531	2,604		

Simulation. We deploy Aladdin+H's codes on a machine with Intel(R) Xeon(R) CPU E5-2682 v4

Table 4: Task types. The key is the CPU number and the amount of RAM (the unit of RAM is MB).

Task type	Task number	Task type	Task number
8-16,384	30,048	8-32,768	7,827
4-10,240	1,966	2-976	51
16-32,768	2,103	16-61,440	2,868
4-16,384	396	28-79,872	2,919
8-31,744	858	4-15,360	36
32-216.753	249	16-49,152	84
32-98,304	915	16-63,488	927
1-1,024	11,629	2-8,192	12,302
16-65,536	1,821	28-112,646	3,023
4-8,192	23,345	32-96,256	90
28-110,592	765	6-8,192	240
56-204,800	3,120	8-1,024	600
4-4,096	3,012	28-110,000	1,512
32-128,947	2,357	8-128,113	334
8-8,192	96	8-24,576	804
28-80,000	117		

CPU(2.50GHz), 32GB RAM, and 1TB SSD disk, merely stubbing out RPCs and tasks execution. Note that in the Alibaba workload trace, the metrics including CPU, RAM, disk, network and so on. In this paper, we use the ratio of RAM to CPU to express the machine's heterogeneous, and use anti-affinity to express the resource completion of disk and network.

4.2 Resource Fragmentation and Constraint Violations

Metrics. We count the used machines $num(sched)$ for 100,000 tasks without constraint violations by Equation 11. Here, $sched$ is the scheduler name, res can be either CPU or RAM, $utilization(res)$ is the total tasks' resource demands. In this context, the larger value of $rs(sched, res)$ means higher resource efficiency and lower resource fragmentation.

$$rs(sched, res) = \frac{utilization(res)}{num(sched)} \tag{11}$$

Environments. We take Graphene [18], Tetris [16], Medea [13] and Firmament [15] for a comparison.

- Both Graphene and Tetris will first schedule the tasks with the larger amount of resource demands, but they cannot support affinity and anti-affinity. So in order to make a fair comparison, we introduce a rescheduling mechanism when some tasks have violated the above constraints. Here, we use i to express the maximum number of rescheduling tasks on the same machine if we encounter some constraint conflicts, and we consider the case where i is 1, 4, respectively.
- Medea uses an integer linear program (ILP) method to place as many LRTs as possible. It minimizes constraint violations and avoids resource fragmentation using three weights. Each weight is normalized to a value ranging from 0 to 1. Here, we use (a, b, c) to express Medea's weights and just set the case where (a, b, c) is $(1, 1, 1)$, $(1, 0.5, 0.5)$,$(0.5, 0.5, 0.5)$, respectively.
- Firmament is a flow-based scheduler and it can only manage one kind of resource. But Firmament can run on multidimensional resource scenarios [15]. There are eight scheduling policies in the Firmament code base currently, and we only select the two most used policies. Firmament-TRIVIAL's goal is to minimize the number of used machines, while Firmament-OCTOPUS aims to use all machines with balanced resource efficiency. In this paper, we employ Firmament's algorithm to manage CPU (Firmament_CPU) and RAM (Firmament_RAM), respectively.
- In Aladdin+H, we consider the situation that All machines belong to 2, 4, or 8 classes.

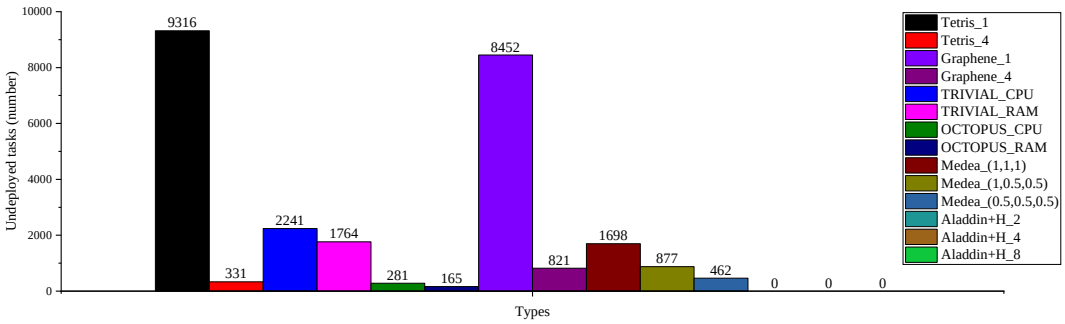
As shown in Figure 10(a), Aladdin+H_2, Aladdin+H_4 and Aladdin+H_8 [6] outperforms other schedulers. Given 12,000 machines, these methods can deploy all tasks without any constraint violations, it can reduce violated constraints by up to 10% (the number of constraint violations of Tetris_1 is almost 10,000). This indicates that our algorithm supports affinity and anti-affinity. And our algorithm increases resource efficiency with fragmentation awareness. Moreover, the number of constraint violations of Tetris_1 is more than 9,000, but Tetris_4's [7] is only 331. One important reason is that Tetris cannot support affinity and anti-affinity. And the more times we try to reschedule the tasks with violated constraints, the more probability we can reduce it. Graphene is an extension of Tetris, so the results of them are similar. Furthermore, we cannot avoid the violated constraints of Medea_(1,1,1) [8], Medea_(1,0.5,0.5) and Medea_(0.5,0.5,0.5). Although we have tried three different weights, it is really a challenge to get the optimized weights. TRIVIAL_CPU [9] and TRIVIAL_RAM try to minimize used machines, so they encounter a lot of conflicts. OCTOPUS_CPU and OCTOPUS_RAM try to balance all machines' resources. They have less violated constraints than TRIVIAL_CPU and TRIVIAL_RAM, because they use more machines which can reduce the probability of

[6] Aladdin+H_2 means using the laddin+H's algorithm, and all machines belong to 8 classifications

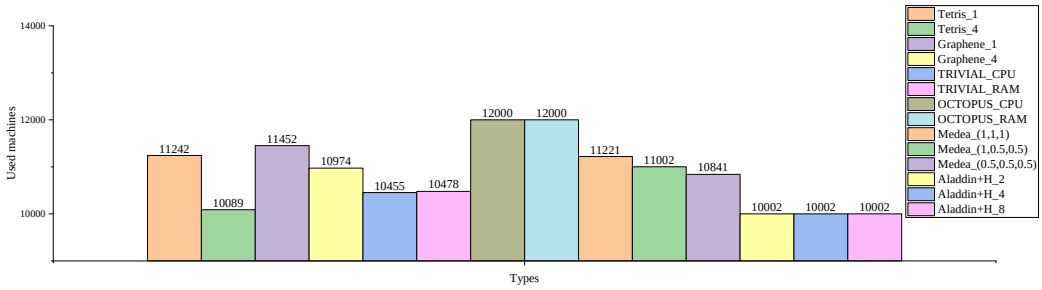
[7] Tetris_4 means using the Tetris's algorithm. If we encounter constraint conflicts, we can reschedule the task up to 4 times

[8] Medea_(1,1,1) means using the Medea's algorithm with the specified weights (1,1,1)

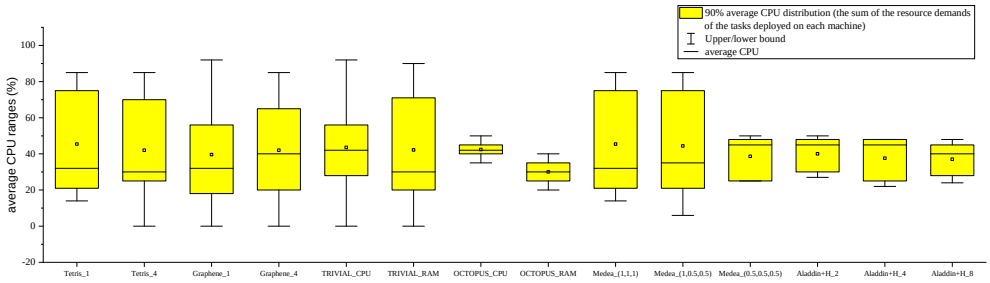
[9] TRIVIAL_CPU means using the TRIVIAL's algorithm, and we just consider CPU resource



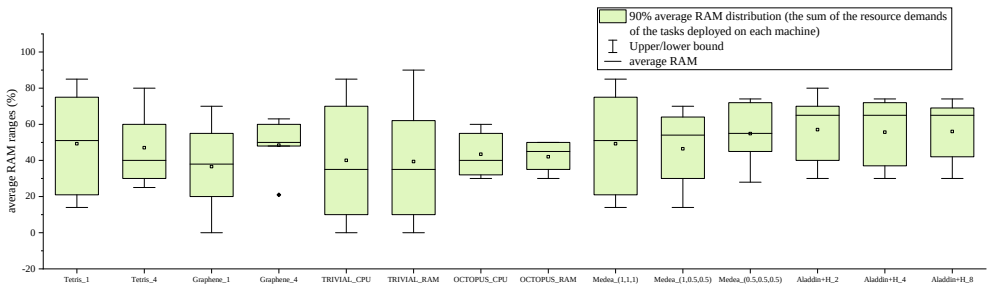
(a) The number of violated constraints for different flow-based schedulers



(b) Used machines for different flow-based schedulers



(c) Average CPU ranges for different schedulers



(d) Average RAM ranges for different schedulers

Figure 10: Resource efficiency for different schedulers

conflicts, as shown in Figure 10(b). Correspondingly, Aladdin+H_2, Aladdin+H_4 and Aladdin+H_8 can minimize the number of used machines.

Furthermore, we find the CPU efficiency ranges of OCTOPUS_CPU, OCTOPUS_RAM, Aladdin+H_2, Aladdin+H_4 and Aladdin+H_8 are much more stable, as shown in Figure 10(c). Here, a smaller median range value represents a more stable algorithm result. The reason is that both OCTOPUS_CPU and OCTOPUS_RAM use a load balancing strategy. Aladdin+H's clustering mechanism can be essentially regarded as a load balancing mechanism. Unfortunately, other schedulers want to minimize the number of used machines so that their CPU efficiency fluctuate. For example, some machines are idle and some others have heavy loads when using TRIVIAL_CPU and TRIVIAL_RAM schedulers. Besides, we find the average CPU efficiency (Tetris_1, Graphene_1) is only about 32%. But if we use the clustering mechanism, the average CPU efficiency (Aladdin+H_2, Aladdin+H_4 and Aladdin+H_8) can achieve up to 42%. It can increase resource efficiency by as much as 30%. The reason is that both Tetris and Graphene need more machines to deploy all tasks without any constraint violations (Figure 10(a) shows that more than 8,000 tasks of Tetris_1 and Graphene_1 violate the affinity or anti-affinity constraints).

Finally, we find the trend of RAM efficiency ranges is similar to the CPU's, as shown in Figures 10(c) and 10(d). We find that RAM is the bottleneck resource because its average efficiency is more than 65%.

4.3 Placement latency

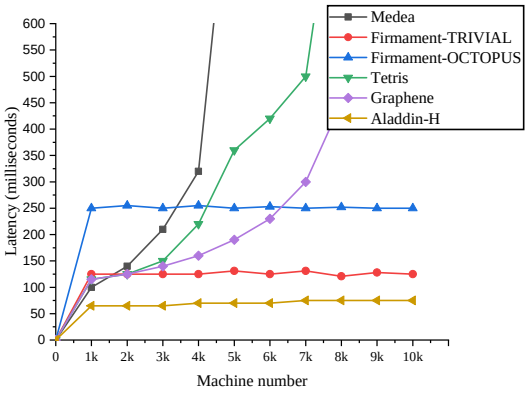
Metrics. Placement latency is the time period between the submission of all tasks and the completion of placements. It also indicates the algorithm overhead. To make a fair comparison with the state-of-the-art schedulers, we calculate the placement latency per task (aka average placement latency) by using Equation 12.

$$latency = \frac{\sum time(T_i)}{total\_tasks} \tag{12}$$

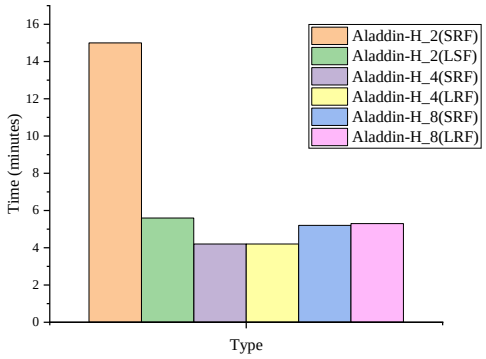
Here, $latency$ means the average placement latency of 120,000 tasks with a specific scheduler i.

As shown in Figure 11(a), the average placement latency per task is still sub-second for Aladdin+H, Firmament-OCTOPUS and Firmament-TRIVIAL, although the cluster size is larger than 10,000. the latency of Aladdin+H is just a half of Firmament-OCTOPUS's and Firmament-TRIVIAL's. We think the reason is that Firmament only considers one kind of resource, which may increase resource fragmentation So they have to migrate some tasks and these migrations increase the latency.

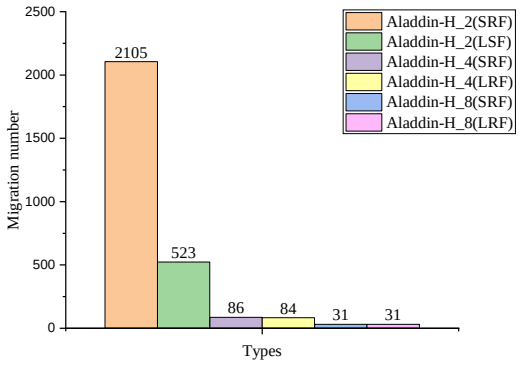
Figure 11(a) also shows that Medea, Tetris and Graphene cannot scale to over ten thousand machines at sub-second



(a) Average placement latency



(b) Aladdin+H's algorithm runtime



(c) The number of migration

Figure 11: Algorithm overhead

placement latency. We find that the latency of Medea is about 300ms when the cluster size is less than 3,000 (the latency's conclusion is similar to those in [13]). But when the cluster size is more than 5,000, the latency can be increased to more than 1 second. We also find that the latency of Graphene is better than Tetris's, we think the reason is that Tetris tries to find troublesome tasks one by one, while Graphene tries to find a subset of ones.

Figure 11(b) shows that Aladdin-2(SRF) needs about 15 minutes to complete the placement decision of 100,000 tasks in the worst case. The reason is that if we schedule the tasks with multidimensional resource demands in heterogeneous environments, we may encounter continuous conflicts, and we have to migrate many tasks from one machine to another. Such migrations in Aladdin-2(SRF) slow down its placement decision and its performance is not as good as Aladdin-4's. In addition, we find that the performance of Aladdin-4 is the best. Comparing with Aladdin-4, Aladdin-8 may increase the edge number in flow network, which result in higher latency since edge number affects the searching space of the MCMF algorithm.

Moreover, we evaluate MCMF's algorithm performance. And we consider two kinds of task arrival characteristics in our paper: (i) tasks with a large amount of resource demands first (aka LRF); (ii) tasks with a small amount of resource demands first (aka SRF). In this context, the type "Aladdin-2(SRF)" in Figure 11 means we divide all machines into 2 classifications, and when massive tasks arrive at the same time, Aladdin+H schedules the tasks with a large amount of resource demands first.

As shown in Figure 11(b), We find that the algorithm runtime of Aladdin+H_4(SRF) and Aladdin+H_4(LRF) are the best. Scheduling 100,000 tasks takes only nearly 4 minutes. Aladdin+H_2(SRF) costs more than 15 minimizes. The reason is that the number of migration is more than 2,000 shown in Figure 11(c). Moreover, we find the migration number of Aladdin+H_8(SRF) is the lowest, but the algorithm runtime is not the best, we think the reason is that increasing the edge number in flow network which result in higher latency since the number of classifications affects the searching space of the MCMF algorithm.

5 Related work

Flow-based schedulers. Flow-based schedulers convert the mapping of tasks and machines into the problem of graph searching. They aim to achieve **low placement latency** by express various constraints in the flow network.

Quincy [22] creatively expresses the data locality and fairness by different costs in flow network, it employs a min-cost flow algorithm to optimize placement latency. Firmament [15] generalizes Quincy , which represents the scheduling problem as a min-cost max-flow (MCMF) optimization over a graph. It can improve placement latency by 20× over Quincy. HaaS [20] uses sub-graphs to optimize placement latency for stream processing tasks. However, the main limitation of current flow-based methods is they only consider one-dimensional resources. Because the capacity of the classical flow theory is only one-dimensional, it is hard to support multi-resources. Aladdin+H extends the capacity to support multi-dimensional resources. Meanwhile, it can scale to over ten thousand machines with sub-second placement latency.

Schedulers with heterogeneous resources support. Some approaches use diverse combinatorial theory (Integer Linger Program (ILP) [13] and pack [33] to solve the scheduling problem. They express various constraints and heterogeneity in clusters by using the mathematical formulations. But they are usually time-consuming (high latency) to analyse a large quantity of data to get optimized results. Their simple bin packing rules cannot deal with the heterogeneity well and usually get a low resource efficiency. Recently, some efforts using machine learning to address the scheduling problem [7, 21, 28], but these methods can only fit the scenarios that have plenty of historical data. Otherwise they may encounter low resource efficiency. In this paper, Aladdin+H can achieve high resource efficiency with low placement latency in heterogeneous clusters even though we have few historical data.

Schedulers with low latency support. Some works characterize specific workloads (eg., task dependencies, workload predication) to accelerate data analytics frameworks [1–3, 8, 9, 13, 14, 17, 35, 40], distributed machine learning [30, 39], deep learning [29, 38], continual learning [34] and reinforce learning [26]. Thus these works can get low latency with workload characteristics awareness based on algorithm optimization. These works have limited applicability in heterogeneous production clusters because their workloads do not have an obvious and fine-grained pattern. Some efforts introduce the distributed architecture to get low latency. Sparrow [27], as a distributed scheduler, can support a high throughput of very short, sub-second tasks. Mercury [24] is a hybrid scheduler that makes centralized, high-quality assignments for tasks. Currently, Aladdin+H is implemented by a centralized architecture and it can be integrated into a distributed scheduler.

6 conclusion

Aladdin+H is the first system that fully supports heterogeneous production clusters based on the flow network theory. Experiments with an Alibaba workload trace from a 12,000-machine cluster show that Aladdin+H can increase resource efficiency by as much as 30% and reduce violated constraints by up to 10% compared with state-of-the-art schedulers.

References

[1] Alibaba cluster data. `https://github.com/alibaba/clusterdata`.

[2] Azure public dataset. `https://github.com/Azure/AzurePublicDataset`.

[3] Google cluster data. `https://github.com/google/cluster-data`.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] R. K. Ahuja. *Network flows: theory, algorithms, and applications*. Pearson Education, 2017.

[6] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows: theory, algorithms, and applications. *Journal of the Operational Research Society*, 45(11):791–796, 1993.

[7] B. Berg, J.-P. Dorsman, and M. Harchol-Balter. Towards optimality in parallel job scheduling. *ACM SIGMETRICS Performance Evaluation Review*, 46(1):116–118, 2019.

[8] W. Chen, A. Pi, S. Wang, and X. Zhou. Characterizing scheduling delay for low-latency data analytics workloads. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 630–639. IEEE, 2018.

[9] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. Hug: Multi-resource fairness for correlated and elastic demands. In *NSDI*, pages 407–424, 2016.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 2009.

[11] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 177–192, Boston, MA, 2019. USENIX Association.

[12] A. P. Esa Hyytia, Samuli Aalto. Minimizing slowdown in heterogeneous size-aware dispatching systems. *ACM SIGMETRICS Performance Evaluation Review*, 40(1):29–40, 2012.

[13] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao. Medea: scheduling of long running applications in shared production clusters. In *EuroSys*, pages 4–1, 2018.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[15] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. Usenix, 2016.

[16] R. Grandl, U. G. Ananthanarayanan, I. S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *acm special interest group on data communication*, 44(4):455–466, 2015.

[17] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *OSDI*, pages 65–80, 2016.

[18] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, 2016. USENIX Association.

[19] L. Han, V. Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. *A generic approach to scheduling and checkpointing workflows*. PhD thesis, Inria, 2018.

[20] J. He, Y. Chen, T. Z. Fu, X. Long, M. Winslett, L. You, and Z. Zhang. Haas: Cloud-based real-time data analytics with heterogeneity-aware scheduling. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1017–1028. IEEE, 2018.

[21] C. Imes, S. Hofmeyr, and H. Hoffmann. Energy-efficient application resource scheduling using machine learning classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*, page 45. ACM, 2018.

[22] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[23] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[24] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya,

R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*, pages 485–497, 2015.

[25] L. Liu and H. Xu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 107–120. ACM, 2018.

[26] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[27] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 69–84. ACM, 2013.

[28] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, page 2. ACM, 2018.

[29] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, page 3. ACM, 2018.

[30] A. Qiao, A. Aghayev, W. Yu, H. Chen, Q. Ho, G. A. Gibson, and E. P. Xing. Litz: Elastic framework for high-performance distributed machine learning. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 631–644, 2018.

[31] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 36. ACM, 2016.

[32] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, 2018. USENIX Association.

[33] H. Sun, R. Elghazi, A. Gainaru, G. Aupy, and P. Raghavan. *Scheduling Parallel Tasks under Multiple Resources: List Scheduling vs. Pack Scheduling*. PhD thesis, Inria Bordeaux Sud-Ouest, 2018.

[34] H. Tian, M. Yu, and W. Wang. Continuum: A platform for cost-aware, low-latency continual learning. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 26–40. ACM, 2018.

[35] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, pages 363–378, 2016.

[36] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[37] W. Wang, B. Liang, and B. Li. Multi-resource fair allocation in heterogeneous cloud computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 26(10):2822–2835, 2015.

[38] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[39] P. Xie, J. K. Kim, Q. Ho, Y. Yu, and E. Xing. Orpheus: Efficient distributed machine learning via system and algorithm co-design. In *Symposium of Cloud Computing*, volume 1, pages 1–2, 2018.

[40] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, B. C. Ooi, and J. Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161. ACM, 2018.