Ditto: Efficient Serverless Analytics with Elastic Parallelism

Chao Jin
Peking University

Zili Zhang
Peking University

Xingyu Xiang
Peking University

Songyun Zou
Peking University

Gang Huang
Peking University

Xuanzhe Liu
Peking University

Xin Jin
Peking University

ABSTRACT

Serverless computing provides *fine-grained* resource elasticity for data analytics—a job can flexibly scale its resources for *each* stage, instead of sticking to a fixed pool of resources *throughout* its lifetime. Due to different data dependencies and different shuffling overheads caused by intra- and inter-server communication, the best degree of parallelism (DoP) for each stage varies based on runtime conditions.

We present Ditto, a job scheduler for serverless analytics that leverages fine-grained resource elasticity to optimize for job completion time (JCT) and cost. The key idea of Ditto is to use a new scheduling granularity—*stage group*—to decouple parallelism configuration from function placement. Ditto bundles stages into stage groups based on their data dependencies and IO characteristics. It exploits the parallelized time characteristics of the stages to determine the parallelism configuration, and prioritizes the placement of stage groups with large shuffling traffic, so that the stages in these groups can leverage zero-copy intra-server communication for efficient shuffling. We build a system prototype of Ditto and evaluate it with a variety of benchmarking workloads. Experimental results show that Ditto outperforms existing solutions by up to 2.5× on JCT and up to 1.8× on cost.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Networks** → **Cloud computing**.

KEYWORDS

Serverless computing, data analytics, task scheduling

ACM Reference Format:
Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10–14, 2023, New York, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3603269.3604816

ACM SIGCOMM '23, September 10–14, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3604816

1 INTRODUCTION

Serverless computing is a new paradigm for cloud computing [5, 10, 13, 23, 37]. It provides *fine-grained* resource elasticity for cloud applications. Cloud resources are allocated to applications in the form of lightweight stateless *functions*, which can be started in sub-seconds and elastically scaled out based on the application demands. Fine-grained resource elasticity also comes with fine-grained billing. Users are only charged for the resources consumed during the execution of functions, excluding the idle time.

The execution and pricing model of serverless computing makes it a good fit to data analytics jobs [1, 11, 28, 38, 43, 45, 51]. In the realm of data analytics, a job is typically represented as a directed acyclic graph (DAG) based on the data dependencies between different execution stages. Each stage consists of a number of tasks that execute in parallel. Traditional serverful solutions necessitate users to provision a fixed pool of resources. A job sticks to this resource pool *throughout* its entire lifetime, regardless of the diverse characteristics and varying resource demands of each stage [43, 45]. Consequently, under-provisioning resources increases job completion time (JCT), while over-provisioning resources results in low resource utilization and unnecessary high cost.

Fine-grained resource elasticity, offered by serverless computing, enables a data analytics job to dynamically scale its resources for *each* stage to optimize performance and cost efficiency. Specifically, each individual task is assigned to a serverless function, and the number of parallel tasks (i.e., the degree of parallelism or DoP) of each stage can be flexibly determined according to resource demand. This inherent scalability benefits *both* JCT and cost. The stages with high resource demands can leverage higher degrees of parallelism for faster execution and reduction in JCT, while those with low resource demands can use fewer parallel tasks to save cost.

A natural solution is to adjust the DoP for each stage based on the amount of data it processes. This approach is adopted by the state-of-the-art solution NIMBLE [51]. Intuitively, the amount of input data for a stage is correlated with its resource demand. Allocating more parallel tasks to a stage with a large amount of input data can effectively reduce the execution time of this stage and potentially reduce the total JCT. Additionally, it is not necessary to allocate too many parallel tasks to a stage with a small input data size to prevent resource waste.

Unfortunately, this solution suffers from two problems. First, the amount of input data for a stage cannot accurately reflect its actual resource consumption (§ 4.2). Adjusting the parallelism configuration based on the input data size ignores the relationships between different stages in the DAG, and the resource usage is a better and

more relevant metric to configure the parallelism when optimizing for cost. Second, this solution fails to consider the impact of data communication between stages (a.k.a., shuffling) due to different function placement. In early serverless systems, function placement do not affect communication overheads, as all data exchange is forced via the external storage (e.g., S3 or Redis) [27, 28, 45, 51]. However, recent advancements in serverless systems enable efficient communication through shared memory for functions on the same server [46, 48]. Given the ever-varying runtime conditions, the cluster is not always able to accommodate two stages on the same server when they both use a high DoP. Using a low DoP and placing the two stages on the same server introduces both lower JCT and cost due to efficient shared memory.

We present Ditto, a serverless analytics system that exploits fine-grained elastic parallelism to minimize JCT and cost. The key idea of Ditto is to use a new scheduling granularity—*stage group*—to *decouple* parallelism configuration from function placement. Ditto analyzes the DAG and bundles stages into stage groups, taking into account their data dependencies and IO characteristics. Functions are placed on servers at the group granularity, obviating the need of considering intra- and inter-server communication overheads for parallelism configuration.

There are three technical challenges in realizing Ditto. First, predicting the execution time of each stage under different DoPs is critical for the estimation of both JCT and cost. Existing schedulers [45, 51] rely on historical job data to estimate the execution time, which works well when the parallelism configuration is fixed in previous executions. However, it is not adequate when it comes to handling elastic parallelism, where the execution time of a stage varies widely as the DoP changes. To address this problem, we propose a step-based time model to capture the relationship between the execution time and the degree of parallelism.

Second, adjusting the parallelism configuration based on the amount of input data is suboptimal. We design a *DoP ratio computing* algorithm to efficiently calculate the optimal ratios of DoPs between stages under a given function placement plan. To minimize JCT, *DoP ratio computing* recalibrates the DoPs of the stages according to the data dependencies and the parallelized time characteristics. This adjustment balances the execution times of the paths in the DAG and minimizes the execution time of the critical path. To minimize cost, it transforms the cost optimization for a general DAG into the JCT optimization for a single-path DAG.

Third and most importantly, co-scheduling the parallelism configuration and function placement under the given resource distribution is challenging. We propose a heuristic *greedy grouping* algorithm that bundles stages into groups in descending order. Stages with large shuffling traffic are grouped preferentially so that they can efficiently shuffle data through zero-copy shared memory. Ditto combines *greedy grouping* with *DoP ratio computing*, and forms a joint iterative optimization method to reduce JCT and cost.

We implement a system prototype of Ditto and evaluate it under a variety of benchmarking workloads. The experimental results show that Ditto outperforms NIMBLE by up to 2.5× on JCT and up to 1.8× on cost. We also verify the accuracy of the step-based time model and the effectiveness of *DoP ratio computing* and *greedy grouping*. Besides, Ditto is able to schedule jobs within one millisecond, and the offline time for building the time model is under 0.3 seconds.

In summary, we make the following contributions.

- We present Ditto, a serverless system that exploits fine-grained elastic parallelism to minimize JCT and cost.
- We propose a new scheduling granularity for serverless analytics jobs—*stage group*—to decouple parallelism configuration from function placement.
- We design a *greedy grouping* algorithm to bundle stages with large shuffling overheads into stage groups and a *DoP ratio computing* algorithm to determine the optimal parallelism configuration under the given stage groups. Combination of the two algorithms achieves significant reduction on JCT and cost.
- We implement a Ditto prototype, and experiments show that Ditto outperforms existing solutions by up to 2.5× on JCT and up to 1.8× on cost.

2 BACKGROUND AND MOTIVATION

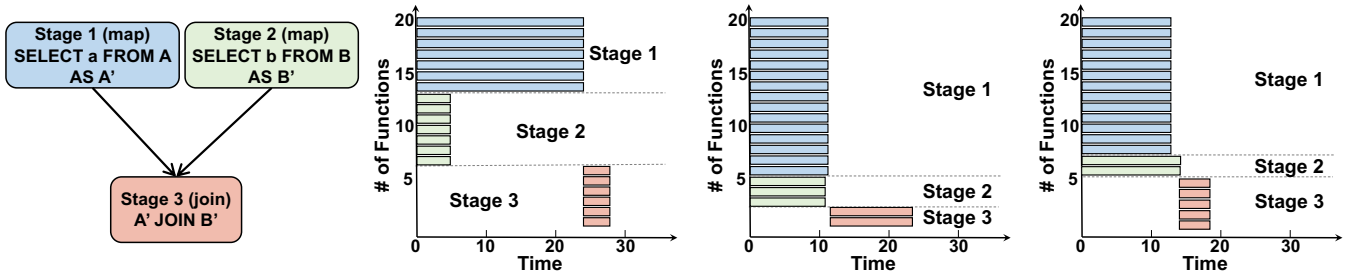
In this section, we first introduce the background of serverless analytics. Then we describe the impact of degree of parallelism on serverless analytics jobs and the existing solution to configure it. Finally, we summarize the challenges to exploit elastic parallelism for serverless analytics, which motivates our design of Ditto.

2.1 Background on Serverless Analytics

Data analytics. Data analytics workloads are an important type of workloads in the cloud [1, 11, 26, 28]. The execution of a data analytics job is divided into a DAG of stages, each comprising multiple parallel tasks. Figure 1a shows an example DAG with three stages. Stage 1 and stage 2 are responsible for executing separate `map` operations, involving the selection of specific columns from Table A and B, respectively. Subsequently, stage 3 performs a `join` operation that joins the two output tables from the preceding two stages. A task scheduler is responsible for scheduling the tasks in each stage based on the entire DAG structure and the runtime conditions of the cluster.

From serverful to serverless analytics. Traditional serverful solutions for data analytics [8, 16, 26, 50] provision a fixed pool of compute and storage resources to facilitate the execution of jobs. Due to the nature of data analytics, different stages of a job typically exhibit diverse resource demands [43, 45]. For example, as a job progresses, the later stages usually process far less data than the initial stages after multiple `filter` and `join` operations. Consequently, right sizing the resource pool is hard for serverful analytics. But no matter how the pool is sized, since a job sticks to a fixed pool throughout its lifetime, resource under-provisioning or over-provisioning cannot be totally eliminated.

Serverless computing provides fine-grained resource elasticity to address the resource provisioning problem. Each task can be mapped to a serverless function, and a stage in the DAG corresponds to a *configurable* number of functions, which can be different across stages. The task scheduler has the freedom to decide the degree of parallelism for each stage to optimize for JCT or cost. Serverless computing platforms charge users based on the actual resource usage. In terms of serverless analytics jobs, the cost of a job is calculated by the cumulative sum of the product of the running time and resource usage of each task.



(a) DAG of join. (b) Fixed DoP. (c) DoP based on data size. (d) Optimal DoP.

Figure 1: Impact of degree of parallelism (DoP) on job completion time (JCT).

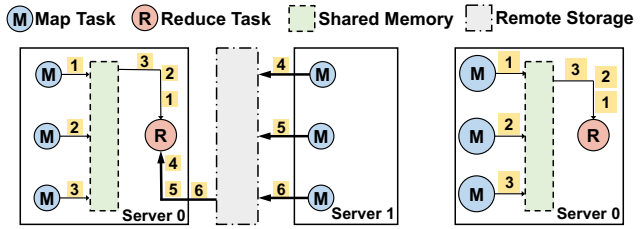
2.2 Benefits and Challenges

Benefits of fine-grained elastic parallelism. Fine-grained elastic parallelism provided by serverless computing enables *each* stage to use a different degree of parallelism, which provides an opportunity for serverless analytics to minimize JCT and cost. Figure 1 shows an example to illustrate the benefits of fine-grained elastic parallelism. Assume there are 20 functions slots available to execute the job with three stages. The naive solution in Figure 1b distributes function slots evenly across all stages (i.e., allocates six or seven slots to each stage), leading to a JCT of 28 time units. Different stages have different execution speeds. Stage 2 and stage 3 executes much faster than stage 1, implying that the resources are over-provisioned to the two stages. Figure 1c allocates fewer function slots to stage 2 and stage 3, but allocates more slots to stage 1. The JCT becomes 23 time units, which is 18% lower. In practice, the execution times of different tasks in a stage can differ due to the data skew. This example is simplified to show the benefits of fine-grained elasticity.

Limitations of prior work. A natural solution is to adjust the number of functions (i.e., the degree of parallelism) for each stage based on its input data size. NIMBLE [51], the state-of-the-art scheduler for serverless analytics, adopts this policy. Intuitively, this solution is reasonable, as data IO typically dominates the execution time for data analytics workloads. However, this solution is suboptimal since it ignores the data dependencies between stages. Figure 1c and Figure 1d compares the DoP configurations of NIMBLE and the optimal solution, respectively. Adjusting the number of functions in proportional to the input data size makes the execution time of each stage almost equal, which is optimal for the first two parallel stages. For consecutive stages with data dependencies, however, the solution allocates too many function slots to the stage with larger input data size, leading to a higher JCT. Figure 1d increases the degree of parallelism of stage 3 to five and reduces the JCT to 19 time units, which is 17% lower than that of Figure 1c.

NIMBLE's solution is also not suitable when considering the impact of function placement. Early serverless computing systems force all data exchange between functions via external storage (e.g., S3 or Redis) [27, 28, 45, 51]. As such, function placement does not affect shuffling overheads, because all shuffling traffic goes through the external storage.

However, this is no longer true with advancements in serverless computing platforms [36, 40, 46, 48]. The communication inefficiency for serverless functions is a known problem, and several solutions [46, 48] have been proposed to improve it. For example,



(a) (b)

Figure 2: Impact of function placement on the best DoP configuration. (a) High DoP with heavy data shuffling time. (b) Low DoP with almost zero data shuffling time.

SPRIGHT [46] is a recent solution that enables functions to exchange data via local shared memory, instead of relying on remote external storage. This makes function placement relevant for shuffling overheads. In Figure 2, the map functions can communicate with the local reduce function via shared memory. When the cluster lacks enough free slots for six map functions and one reduce function on one server, using a smaller degree of parallelism and placing them on the same server can be more efficient than using a higher degree of parallelism and placing them on different servers.

Challenges. The impact of data dependencies on job performance and the impact of function placement on communication overhead create three challenges for fine-grained parallelism configuration. First, parallelism configuration requires accurate estimation of the execution time for each stage under different degrees of parallelism. Analytics jobs in production workloads tend to be recurring [18, 25, 31–33, 45]. Existing schedulers for serverless analytics [45, 51] rely on job history to estimate execution time. This is no longer accurate with the introduction of elastic parallelism and efficient intra-server communication. The fluctuating runtime resource availability constrains parallelism configuration and function placement, and different choices of parallelism configuration and function placement yield varying execution times.

Second, the optimal parallelism configuration depends on the data dependencies between stages. Stages in general DAGs exhibit complex data dependencies. Specifically, a stage can consume data from multiple upstream stages, and the data dependencies can cascade to downstream stages. Understanding and effectively managing these data dependencies is crucial in achieving an efficient and well-optimized parallelism configuration.

Third, coupled with function placement, the optimization of parallelism configuration is challenging. The naive solution is to enumerate all possible combinations to find the best parallelism configuration and placement plan. However, a data analytics job contains a DAG of stages and the datacenter cluster involves numerous servers. As a result, the search space of enumeration is huge, as the decisions for the degree of parallelism and placement for different stage are supposed to be jointly considered.

3 DITTO OVERVIEW

We present Ditto, a serverless analytics system that exploits elastic parallelism to achieve efficient and cost-effective execution for data analytics jobs. Ditto achieves so by dynamically adjusting the parallelism configuration and function placement based on the two key techniques: *DoP ratio computing* and *greedy grouping*. *DoP ratio computing* finds out the optimal parallelism configuration given the execution characteristics of stages (§ 4.2), and *greedy grouping* bundles the stages with large shuffling traffic into groups to reduce the data communication overhead (§ 4.3).

Architecture overview. Figure 3 shows the overall architecture of Ditto. Ditto is a serverless system that contains a cluster of servers to execute functions and an external storage service to provide data exchange between functions. Functions on the same server leverage local shared memory for efficient data exchange (e.g., SPRIGHT [46]). Each server receives function execution requests from the control plane and then executes the corresponding functions. The number of functions held on each server is limited by the hardware capability (e.g., CPU cores). In addition, each server accommodates a runtime monitor to track the runtime statistics and the execution results of each function.

Ditto components. After submitting an analytics job, Ditto takes the job DAG and the available resources as input. Users can specify the optimization objective as either minimizing JCT or cost. Ditto then calculates the parallelism configuration and task placement plan for the given job.

Execution time predictor. Ditto builds an execution time model (§ 4.1) based on the job profiles. The approach of leveraging job history is inline with other data analytics schedulers [33, 45, 51]. The main difference is that Ditto considers both degree of parallelism and task placement to predict the execution time of a stage. Ditto updates the model periodically as new job profiles are generated.

Elastic parallelism scheduler. The scheduler consists of three parts: *DoP ratio computing* (§ 4.2), *greedy grouping* (§ 4.3) and placement check (§ 4.4). The scheduler first groups two consecutive stages with a heuristic greedy policy that leverages local shared memory as much as possible. After grouping, the scheduler computes the optimal DoP for each stage with the accurate DoP ratio computing algorithm. Finally, it uses the *best fit* method to check whether the parallelism configuration and the placement plan are feasible for the available resources. If feasible, the scheduler keeps the two stages in a group; otherwise, it backtracks to abandon grouping the two stages. The scheduler repeats the above steps until no more stages can be grouped.

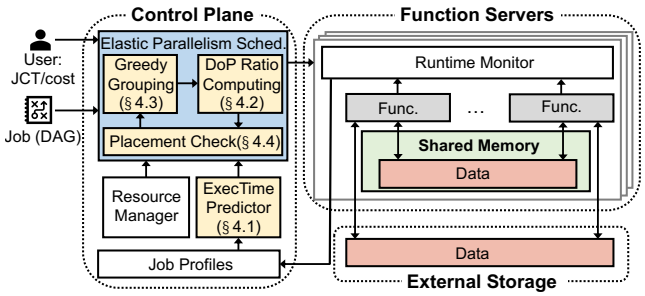


Figure 3: Ditto overview.

4 DITTO DESIGN

In this section, we first model the execution time of the stages in analytics jobs (§ 4.1). Then we introduce the *DoP ratio computing* algorithm to find the best parallelism configuration given the execution characteristics of the stages (§ 4.2). Next, we propose *greedy grouping* algorithm to bundle stages into groups for placement optimization (§ 4.3). Finally, we combine the two algorithms to jointly optimize parallelism configuration and task placement (§ 4.4).

4.1 Execution Time Model

Step-based time model. To accurately profile the execution time of a stage, we apply the step model in NIMBLE [51]. Typically, a stage consists of three steps, i.e., read, compute and write. The time of the three steps are denoted as $R(s,d,\mathcal{P})$, $C(s,d)$ and $W(s,d,\mathcal{P})$, where d represents the number of tasks and $\mathcal{P}$ represents the placement policy. The read and write time are affected by both d and $\mathcal{P}$, while the compute time is only affected by d. We then derive the execution time of the entire stage s:

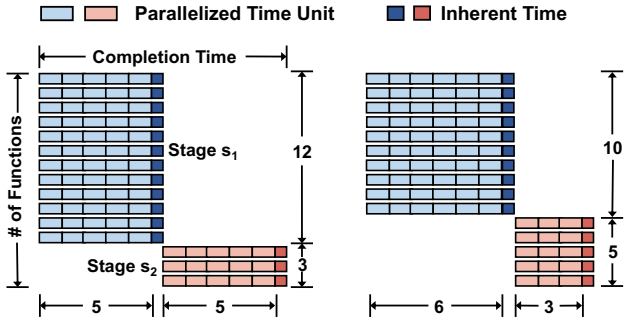
$$T(s,d,\mathcal{P}) = R(s,d,\mathcal{P}) + C(s,d) + W(s,d,\mathcal{P}) \tag{1}$$

The read and write steps can be further divided into multiple fine-grained steps, each of which relates to a data dependency of the stage. Consequently, we can estimate $T(s,d,\mathcal{P})$ by profiling the fine-grained steps. We model the execution time of each step in the stage as $\alpha/d+\beta$, where d is the degree of parallelism, and α and β are fitted offline. α/d is the parallelized time of the step, which represents the execution time benefited from high parallelism, while β is the inherent execution overhead of the step. Assume the number of steps in stage s_i is m, and we derive that:

$$T(s_i,d_i,\mathcal{P}) = \sum_{k=1}^{m}\left(\frac{\alpha_{i_k}}{d_i}+\beta_{i_k}\right) = \frac{\alpha_i}{d_i}+\beta_i \tag{2}$$

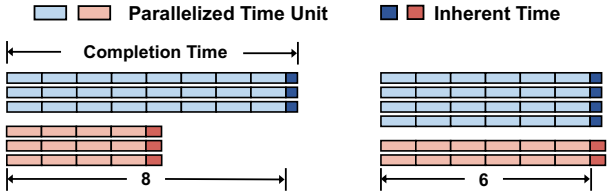
Modeling the shared memory. In the execution time model 2, α and β of the I/O steps are influenced by the task placement policy $\mathcal{P}$. Tasks on the same server exchange data with local shared memory, which eliminates the data transmission and serialization/deserialization time. For example, SPRIGHT [46] achieves zero-copy data exchange with microsecond-level latency, no matter the data size is small or large. Therefore, α and β of the I/O steps are set to zero when $\mathcal{P}$ places the corresponding tasks on the same server.

Modeling stragglers. A stage involves multiple parallel tasks, and the execution time of the stage is determined by its slowest task. Formula 2 is effective when each task processes the same amount



(a) DoP based on data size. (b) Optimal DoP.

Figure 4: Intra-path DoP ratio.



(a) Unbalanced DoP. (b) Balanced DoP.

Figure 5: Inter-path DoP ratio.

of data, which is impossible in practice. We apply a scaling factor to adjust the average execution time when fitting Formula 2. In practice, the scaling factor is dynamically tuned according to the profiled job history.

4.2 DoP Ratio Computing

We first discuss how to find the optimal parallelism configuration with a given task placement plan. There are $\binom{C-1}{n-1}$ possible parallelism configurations, where C is the total number of available function slots and n is the number of the stages. However, the parallelized time characteristics of stages provide the opportunity to simplify the problem. The core idea of Ditto is to find the ratio of the degree of parallelism between stages, denoted as the *DoP ratio*. We observe that the best parallelism configuration changes as the available resources varies but the optimal DoP ratio between stages remains. Therefore, we design the *DoP ratio computing* algorithm to efficiently find the optimal DoP ratio for JCT and cost optimization.

Optimizing JCT. We first consider the case of optimizing JCT. To ease the analysis, we start with tree-like DAGs and then extend to general DAGs. We refer to a sequence of stages connected by data dependencies as a *path*. The JCT is the execution time of the critical path in the DAG. The main idea of *DoP ratio computing* is to minimize the execution time of the critical path by calculating the *intra-path DoP ratio* and balance the execution times of all paths by calculating the *inter-path DoP ratio*.

We first demonstrate the intra-path and inter-path DoP ratio between two stages. Then we merge the two stages into a single virtual stage based on the execution time model. Finally, we combine the DoP ratio between two stages and the stage merging process into a bottom-up approach to calculate the DoP ratios from the leaves (i.e., the initial stages) to the root (i.e., the final stage).

Intra-path DoP ratio. Figure 4 shows an example of the execution of two consecutive stages (i.e., parent-child stages). Assume there are 15 function slots available. The completion time of the two stages is the sum of their execution times. Since the inherent time of the two stages (i.e., β_1 and β_2) are constant, we omit them in the completion time calculation. The parameter α in the parallelized time is represented by the total number of time units in the stage, which also reflects the amount of data processed by the stage. α_1 and α_2 are 60 and 15, respectively. In Figure 4, stage s_1 processes 4× as much data as stage s_2, so the solution based on data size (Figure 4a) assigns 4× more function slots to s_1 than s_2 (i.e., 12 function slots to s_1 and 3 function slots to s_2). The completion time of the two stages is 10 time units.

However, this method allocates too many function slots to s_1, leading to the slow down of s_2. The optimal solution (Figure 4b) distributes function slots according to $\sqrt{\alpha_1/\alpha_2}$, which is 2 in this case. It assigns 10 function slots to s_1 and 5 function slots to s_2. After the adjustment, the completion time is 9. We extend the intra-path DoP ratio to n consecutive stages and prove the optimality according to the Cauchy-Schwarz inequality in Appendix A.1.

Inter-path DoP ratio. Figure 5 shows the execution of two stages as siblings in a tree, which means they have the same parent (i.e., the downstream stage). The completion time is the maximum execution time of the two stages. The intuition to minimize the completion time is to balance the execution time across the two stages by adjusting their DoPs. We prove that the balanced structure is optimal for n sibling stages in Appendix A.2.

As Figure 5a shows, stage s_1 and stage s_2 both have three tasks, leading to the completion time of 8 time units without considering the small inherent time. In this case, α_1 is 24 time units and α_2 is 12 time units. And the inter-path DoP ratio is defined as α_1/α_2, which is 2. So we balance the execution time by assigning four and two function slots to stage s_1 and s_2, respectively. Therefore, the completion time is 6 time units. Note that the theoretical optimal solution is an *entirely* balanced structure, which is not practical. We omit the influence of the small inherent time with the observation that most of the execution time can benefit from higher parallelism, i.e., $\alpha_i/d_i \gg \beta_i$.

Stage merging. Due to the inverse function form of the parallelized time, two sibling stages or parent-child stages with the optimal DoP ratio can be merged into a virtual stage while still conforming to the execution time model. Let A be the array that stores the parallelized time parameters (α) of different stages and r_{ij} be the DoP ratio between stage s_i and s_j. The function MERGE_STAGES in Algorithm 1 merges two sibling stages (line 8–9) or parent-child stages (line 5–6) into a virtual stage s and calculates the parallelized time parameter of s (line 6 and line 9). Then, it replaces s_i and s_j in the DAG with the virtual stage s (line 10).

The concrete calculation of the virtual stage's parallelized time parameter is as follows. Assume that s_i and s_j are merged into s. Let d_i, d_j be the DoPs of stage s_i and s_j, respectively. Let d be the DoP of the new stage s, meaning that $d_i + d_j = d$. Let the execution times of stage s_i and s_j be $\alpha_i/d_i + \beta_i$ and $\alpha_j/d_j + \beta_j$, respectively.

For intra-path stage merging, according to the intra-path DoP ratio (i.e., $d_i/d_j = \sqrt{\alpha_i}/\sqrt{\alpha_j}$), we have

$$d_i = \frac{\sqrt{\alpha_i}}{\sqrt{\alpha_i}+\sqrt{\alpha_j}}d, \quad d_j = \frac{\sqrt{\alpha_j}}{\sqrt{\alpha_i}+\sqrt{\alpha_j}}d$$

The completion time of stage s is the sum of the completion time of s_i and s_j, which is

$$\frac{\alpha_i}{d_i}+\beta_i+\frac{\alpha_j}{d_j}+\beta_j = \frac{\alpha_i(\sqrt{\alpha_i}+\sqrt{\alpha_j})}{\sqrt{\alpha_i}d}+\beta_i+\frac{\alpha_j(\sqrt{\alpha_i}+\sqrt{\alpha_j})}{\sqrt{\alpha_j}d}+\beta_j$$
$$= \frac{(\sqrt{\alpha_i}+\sqrt{\alpha_j})^2}{d}+(\beta_i+\beta_j) \tag{3}$$

For inter-path stage merging, according to the inter-path DoP ratio (i.e., $d_i/d_j = \alpha_i/\alpha_j$), we have

$$d_i = \frac{\alpha_i}{\alpha_i+\alpha_j}d, \quad d_j = \frac{\alpha_j}{\alpha_i+\alpha_j}d$$

The completion time of stage s is the larger completion time of s_i and s_j, which is

$$\max\left\{\frac{\alpha_i}{d_i}+\beta_i, \frac{\alpha_j}{d_j}+\beta_j\right\} = \max\left\{\frac{\alpha_i}{d_i}, \frac{\alpha_j}{d_j}\right\}+\max\{\beta_i,\beta_j\}$$
$$= \frac{\alpha_i+\alpha_j}{d}+\max\{\beta_i,\beta_j\} \tag{4}$$

The first equality is due to the balanced structure (i.e., $\alpha_i/d_i = \alpha_j/d_j$). Consequently, the execution time of the virtual stage s conforms to the execution time model.

Bottom-up approach. Algorithm 1 shows the pseudo-code of the detailed DoP ratio computing algorithm for JCT optimization. Algorithm 1 merges the stages in a bottom-up manner and calculates the optimal DoP ratios during the stage merging process. In this way, it balances the execution time across paths and minimizes the completion time of the critical path.

Let $\mathcal{V}$ be the set of all stages and $\mathcal{E}$ be the set of all data dependencies in the DAG. The function BOTTOM_UP_DOP computes the DoP ratios from the stages with the largest depth to the root stage with depth zero, which follows the layer-by-layer pattern. For stages with the same depth, it first merge the sibling stages into an equivalent virtual stage by stage merging (line 18–23). The algorithm then merge the virtual stage with its parent stage into a new virtual stage, decrementing the total depth of the DAG by one (line 24–26). In this way, the DAG is reduced to a single stage step by step, and the optimal DoP ratios are obtained one by one according to the optimality of the intra-path DoP ratio and the inter-path DoP ratio. With the DoP ratios r_{ij}, we can assign the C function slots to n stages. Each stage is merged once, so the time complexity of Algorithm 1 is $O(|\mathcal{V}|)$.

Extend to General DAGs. Each stage in tree-like DAGs has only one parent stage. However, in general DAGs, a stage may have multiple parent stages. We emphasize that the above algorithm can be extended to general DAGs, because: (*i*) The objective functions are both to minimize the critical path in tree-like and general DAGs. (*ii*) The basic idea is to balance the paths as much as possible. Therefore, when facing a general DAG, merging sibling stages first and then merging parent-child stages is still the correct strategy for computing the optimal DoP ratios.

Algorithm 1 Bottom-up DoP ratio computing algorithm

```
1:  function MERGE_STAGES(s_i, s_j, A)
2:      // Array A[s] stores the parallelized time parameters (α) of all stages
3:      α_i ← A[s_i], α_j ← A[s_j], s ← new stage
4:      if ∃(s_i, s_j) ∈ E then
5:          r_ij ← sqrt(α_i/α_j)                          ▷ Intra-path DoP ratio
6:          A[s] ← (sqrt(α_i) + sqrt(α_j))^2
7:      else
8:          r_ij ← α_i/α_j                                ▷ Inter-path DoP ratio
9:          A[s] ← α_i + α_j
10:     Change the DAG (V, E) with the new stage s
11:     return s
12:
13: function BOTTOM_UP_DOP(V, E, A)
14:     max_depth ← the maximum depth of all stages
15:     for i = max_depth → 1 do
16:         V(i) ← all stages with depth i in V
17:         while V(i) ≠ ∅ do
18:             Select a stage s from V(i)
19:             s_p ← the parent stage of s, Sib ← the sibling stages of s
20:             while |Sib| > 1 do
21:                 Select two stages s_x, s_y from Sib
22:                 s ← MERGE_STAGES(s_x, s_y, A)
23:                 Sib ← (Sib − {s_x, s_y}) ∪ {s}
24:             s ← the only stage in Sib
25:             MERGE_STAGES(s, s_p, A)
26:             V(i) ← all stages with depth i in V
```

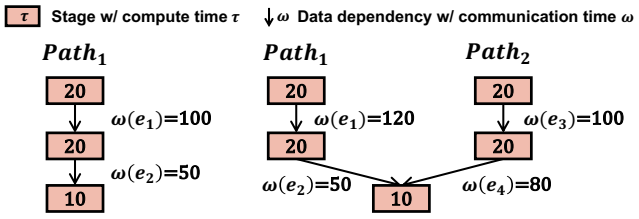
Optimizing cost. Let n be the number of stages and $M(s_i, d_i)$ be the resource usage of stage s_i with the degree of parallelism d_i. $M(s_i, d_i)$ is a linear function of d_i:

$$M(s_i, d_i) = \rho_i + \sigma_i d_i, \quad i \in \{1, 2, \ldots, n\} \tag{5}$$

where ρ_i stands for the resource usage related to the data processing of stage s_i, and $\sigma_i d_i$ represents the resource overhead of launching d_i functions. The cost of stage s_i is $M(s_i, d_i) \times T(s_i, d_i, \mathcal{P})$, which can be rewritten as $\frac{\rho_i\alpha_i}{d_i} + \sigma_i\beta_i d_i + \rho_i\beta_i + \sigma_i\alpha_i$ according to Formula 2 and Formula 5. The cost of the DAG is the sum of the cost of all stages. For an analytics task, resources used to process the big data are typically far larger than the resource consumption by the function itself. Then we can ignore $\sigma_i d_i$ and the problem shares the same form as the JCT optimization for intra-path DoP ratio. Minimizing the cost of the DAG is equivalent to minimizing the JCT of a single-path DAG with n stages, where the new parallelized time of stage s_i is $\rho_i\alpha_i/d_i$. Then we obtain the optimal DoP ratio as $d_i/d_j = \sqrt{\rho_i\alpha_i}/\sqrt{\rho_j\alpha_j}$.

4.3 Greedy Grouping

The second problem is to decide the placement of different stages. As described in §2, task placement affects the performance of data transmission between stages (the write step of the upstream stage and the read step of the downstream stage). Due to the limited resources of a single server, it is impossible to place all stages on one server and achieve zero-copy shuffling for all intermediate data. Since shuffle is an all-to-all communication, we need to group stages (i.e., place their tasks together) and only place the stages within a group on the same server. In this subsection, we optimize JCT or cost through grouping stages. We emphasize that the grouping algorithm only finds the appropriate grouping order, and the final stage groups are decided in collaboration with DoP ratio computing and placement check (§ 4.4).



(a) Single path. (b) Multiple paths.

Figure 6: Greedy grouping.

The granularity of grouping is one stage, i.e., a node in the DAG. The consecutive stages (s_i, s_j) have a data dependency, i.e., an edge. The weights of nodes and edges depend on the optimization objective. Let $R(s), C(s), W(s)$ be the read, compute, and write time of stage s, respectively. Let $M(s)$ be the resource usage of stage s. For JCT optimization, the weight of node s_i is $C(s_i)$, and the weight of (s_i, s_j) is $W(s_i) + R(s_j)$. For cost optimization, the node weight is $M(s_i)C(s_i)$, and the edge weight is $M(s_i)W(s_i) + M(s_j)R(s_j)$. If the intermediate data is transmitted through shared memory, the edge weight is nearly zero due to the zero-copy mechanism.

Optimizing JCT. To optimize JCT, we need to consider the stages of the critical path. We first introduce the greedy algorithm to group the stages of a single path, and then extend it to general DAGs. The execution time of a path is the sum of the weights of all nodes and edges in the path. Enumerating all possible grouping strategies is intractable for a large DAG at runtime since the time complexity is $O(2^{|\mathcal{E}|})$, where $\mathcal{E}$ is the set of all edges. Therefore, we propose the greedy method, which groups the consecutive stages with the largest weight first. This heuristic method is similar to the greedy solution of the packing problem and provides more opportunities to reduce the transmission time.

As shown in Figure 6a, the edges are denoted as e_1 and e_2 and $\omega(e)$ represents the weight of the corresponding edge. For single path, the algorithm simply traverses the edges in descending order, i.e., order $[e_1, e_2]$.

As for multiple paths, the JCT is the completion time of the critical path. Thus, we adjust the greedy algorithm to balance the completion time of each path as much as possible. The algorithm first finds the critical path through the runtime profile, selects the edge with the largest weight in the critical path and groups the stages connected by it. Then, it re-profiles the DAG and finds the new critical path and repeats the grouping process. The loop stops when all edges are traversed. The time complexity of the algorithm is $O(|\mathcal{E}| \log |\mathcal{E}| + |\mathcal{E}||\mathcal{V}|)$, because the sort complexity is $O(|\mathcal{E}| \log |\mathcal{E}|)$, and the placement check algorithm needs to examine at most $|\mathcal{V}|$ stage groups in each iteration. The pseudo-polynomial time complexity is negligible compared to the data processing time.

Figure 6b shows the grouping order for multiple paths. For simplicity, we assume the sum of the node weights is identical for all paths. The algorithm first profiles the DAG and selects $Path_2$ as the critical path. Then, it traverses the edge e_3 with the largest weight in $Path_2$. After grouping the corresponding stages, the algorithm re-profiles the DAG ($\omega(e_3) = 0$) and selects the new critical path $Path_1$. Then, it traverses the edge e_1 with the largest weight in $Path_1$. Finally, the traversing order is $[e_3, e_1, e_4, e_2]$.

Algorithm 2 Greedy grouping

```
1:  function GREEDY_GROUP(V, E, R, obj, DoP)
2:      // DoP stores the DoP of each stage
3:      E_g ← ∅
4:      while E ≠ ∅ do
5:          if obj is JCT then
6:              CP ← the critical path of the DAG (V, E)
7:              (s_i, s_j) ← the edge with the largest weight in CP
8:          else
9:              (s_i, s_j) ← the edge with the largest weight
10:         // Try grouping s_i and s_j, and ω_ij is the weight of (s_i, s_j)
11:         ω_ij ← 0, E_g ← E_g ∪ {(s_i, s_j)}
12:         if CAN_PLACE(DoP, E_g, R) is false then
13:             E_g ← E_g − {(s_i, s_j)}
14:         E ← E − {(s_i, s_j)}
```

Optimizing cost. In serverless computing, the cost of the stage is in proportion to the product of its resource consumption and duration. Minimizing the transmission cost in the DAG is different from minimizing the transmission time. Thus, we adjust the weight of the edge to the product of the read/write time and the average resource usage. The communication cost is the sum of the cost of each edge, so the greedy algorithm traverses every edge of the entire DAG in descending order by weight.

Algorithm 2 shows the pseudo-code of the greedy grouping algorithm. The function GREEDY_GROUP finds the grouping strategy based on the aforementioned greedy policy. It uses $\mathcal{E}_g$ to record the edges whose related stages are grouped. The while loop traverses all edges in the order based on the optimization objective. In each iteration, the function finds the edge with the largest weight in the critical path for JCT optimization and that with the largest weight for cost optimization. We can also reduce the time complexity of deciding traverse order to $O(|\mathcal{E}| \log |\mathcal{E}|)$ by profiling the DAG and sorting all edges in advance, instead of finding one edge in each iteration. The function CAN_PLACE checks whether the parallelism configuration and placement plan satisfy the resource constraints $\mathcal{R}$. If grouping the stages connected by this edge satisfies the resource constraints, the edge is added to $\mathcal{E}_g$. The algorithm finishes when all edges are traversed.

4.4 Combination

Placement check. Given the deterministic stage groups and the DoP of each stage, the subsequent problem is to *place* the stage groups on the physical servers with different number of function slots. Ditto adopts the best-fit algorithm to place the stage groups. The best-fit algorithm first sorts the stage groups in descending order according to the number of function slots that the group requires. Then, it traverses the sorted stage groups and places the group on the server with the nearest function slot number. If there is no enough resources to place a group, the placement check fails for the given groups.

Joint optimization. Algorithm 3 shows the pseudo-code of the joint optimization for JCT. It co-optimizes the parallelism configuration and the stage grouping in an iterative manner. Initially, each stage is regarded as a group. In each iteration, the algorithm follows the *greedy grouping* order to traverse the edges in the DAG. It attempts to group the two stages connected by an edge. Then, it

Algorithm 3 Joint optimization for JCT

```
1:  function JOINT_ITERATIVE_OPTIMIZATION(V, E, R)
2:      // Initialize DoP and update the parameters (A, ω) based on DoP
3:      DoP ← BOTTOM_UP_DOP(V, E, A)
4:      UPDATE_PARAMS(A, ω, DoP)
5:      // E_g and E_u store grouped and ungrouped edges, respectively
6:      E_g ← ∅, E_u ← E
7:      while E_u ≠ ∅ do
8:          Sort E_u in greedy grouping order mentioned in § 4.3
9:          for (s_i, s_j) ∈ E_u do
10:             // Try grouping s_i and s_j
11:             ω_ij ← 0, E_g ← E_g ∪ {(s_i, s_j)}
12:             DoP ← BOTTOM_UP_DOP(V, E, A)
13:             if CAN_PLACE(DoP, E_g, R) then
14:                 UPDATE_PARAMS(A, ω, DoP)
15:                 E_u ← E_u − {(s_i, s_j)}
16:                 break
17:             else
18:                 // Undo grouping s_i and s_j, and restore DoP
19:                 Undo line 11 and 12
20:         if No edge in E_u is grouped in the above loop then
21:             break
```

uses the *DoP ratio computing* algorithm, implemented by the function BOTTOM_UP_DOP as shown in Algorithm 1, to figure out the optimal parallelism configuration based on the stage groups and the parallelized time characteristics. The function CAN_PLACE implements the *best fit* algorithm to check the placement feasibility under the resource constraints $\mathcal{R}$. After the stage grouping and parallelism configuration are determined, the algorithm tries to place the stage groups onto the physical servers. If the placement check fails, the algorithm backtracks and breaks the group into the original stages. Otherwise, the new group is retained. Subsequently, it traverses other consecutive stages in next iterations and repeats the above process until no stages can be grouped.

The joint optimization for cost is similar to that for JCT. The only difference is that the optimal DoP ratios and the greedy grouping order are computed based on the cost model, as described in § 4.2 and § 4.3, respectively.

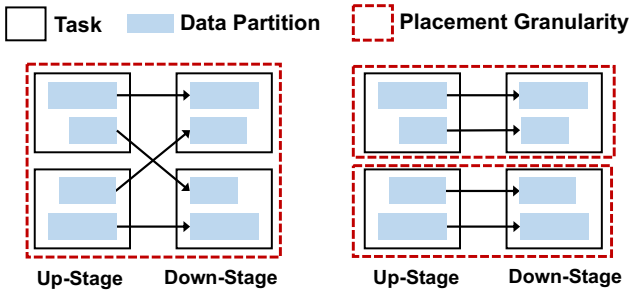
Remark. Since each step in the joint optimization algorithm takes pseudo-polynomial time, the time complexity of the algorithm is pseudo-polynomial with respect to the number of nodes and edges in the DAG. The algorithm guarantees that the objective function (i.e., JCT or cost) is non-increasing during the iterations. Let $F(D_i, P_i)$ be the value of the objective function after the i-th iteration, where D_i and P_i represents the DoP configuration and the placement plan, respectively. Then we have the following inequalities:

$$F(D_{i+1}, P_{i+1}) \leq F(D_i, P_{i+1}) \leq F(D_i, P_i) \tag{6}$$

The first inequality is due to the optimality of the *DoP ratio computing* algorithm in the i+1-th iteration. The second inequality is because the objective function monotonously decreases as more stages are grouped under the same DoP configuration, following the greedy grouping order.

4.5 Practical Issues

Pipelined execution. NIMBLE [51] proposes a pipelining mechanism to overlap the steps of different stages, which influences the execution time model. Therefore, Ditto adjusts the profile by reading the pipelining annotation and modifies the time model accordingly.



(a) Shuffle. (b) Gather.

Figure 7: Decomposing stage groups to task groups.

In detail, the execution time of the downstream stage only involves the non-overlapping steps while ignoring the overlapping steps.

DoP rounding. The DoP ratio computing algorithm calculates the degree of parallelism d_i as a real number instead of an integer in real world. Thus, we round down the real number to the nearest integer and set d_i to one if $d_i < 1$. It guarantees that each stage involves at least one task for execution. Besides, the round down method aims to guarantee the resource constraint, i.e., $\sum_{i=1}^{n} d_i \leq C$.

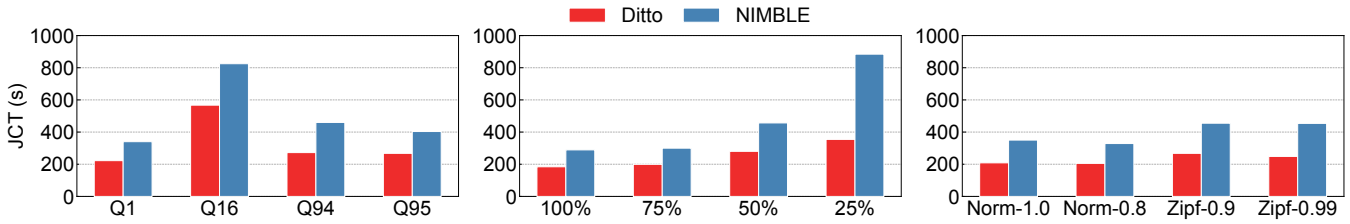
Stage group decomposition. For data communications in the DAG, we replace shuffle with a new primitive, gather, to decompose stage groups into fine-grained groups while maintaining correctness. Figure 7 shows a stage group with two stages (an upstream stage and a downstream stage), each of which has two tasks. In Figure 7a, each upstream task shuffles its data partitions to two downstream tasks. Due to the full collection, the four tasks are supposed to be placed on the same server. However, gather (Figure 7b) allows one upstream task to transmit data to only one downstream task. The stage group is divided into two fine-grained task groups, which improves the possibility of successful placement.

Resource utilization. We assume all available resources at the job arrival can be used throughout its lifetime, excluding the resource sharing across jobs. It may cause resource under-utilization because the stages in a job may not overlap in time, leaving the corresponding function slots idle. We emphasize that Ditto focuses on optimizing JCT and cost for an *individual* job, which are the most *user-concerned* objectives in serverless computing. However, maximizing the resource utilization for a serverless cluster requires co-design of inter-job resource allocation and intra-job scheduling by *cluster providers*. We leave this study as future work.

5 IMPLEMENTATION

We implement a system prototype of Ditto with ~3000 lines of code in C++. Our implementation is based on SPRIGHT [46], a serverless framework that allows high-performance data communication through shared memory. We use Amazon S3 [7] and provision Redis [15] nodes on Amazon ElastiCache [2] as our external storage.

Analytics queries with Ditto. We implement a data analytics execution engine atop SPRIGHT. The engine integrates a set of SQL operators (e.g., `join` and `groupby`) for analytics queries. It also provides data communication APIs (e.g., shuffle and broadcast)



(a) TPC-DS queries. **(b) Different slot usage.** **(c) Different slot distributions.**

Figure 8: Performance when optimizing JCT.

that transparently dispatch I/O requests to shared memory or external storage, according to the co-location of the upstream and downstream tasks.

Scheduler. We implement the algorithms described in § 4 for the scheduler. The scheduler takes the job DAG as input and performs the joint optimization of parallelism configuration and task placement. Then, it sends the optimized execution requests to the function servers. Ditto specifies a set of tasks of the same stage in the execution request, which decreases the network overhead. Besides, the execution request also includes the upstream and downstream task information that are necessary for the transparent data communication APIs.

Task launch time. Task launch time affects job performance. Starting a task too early incurs unnecessary costs for waiting for upstream tasks, while starting it too late increases the JCT. NIMBLE [51] focuses on estimating the right time to launch tasks, and we employ the NIMBLE algorithm in Ditto to decide the task launch time. The NIMBLE algorithm is triggered after the parallelism configuration and task placement are determined, and Ditto uses a timer to send execution requests at the calculated task launch times.

6 EVALUATION

In this section, we first use the TPC-DS benchmark to evaluate the overall performance of Ditto over the state-of-the-art serverless analytics scheduler, NIMBLE (§ 6.1, § 6.2 and § 6.3). Next, we dive into Ditto to analyze the effectiveness of its components (§ 6.4). Finally, we demonstrate the system overhead of Ditto (§ 6.5).

Experimental Setup. All experiments are conducted on AWS. We deploy the control plane of Ditto on one m6i.4xlarge instance configured with 16 vCPUs and 64 GB memory. We use eight m6i.24xlarge instances for function execution, with 96 vCPUs and 384 GB memory each. Each function slot is configured with one CPU core and each function uses memory based on the task demand. We use Amazon S3, a widely-used elastic object storage in serverless computing, as the main external storage in the experiments. Redis [15] is adopted as the supplemental external storage for the evaluation. We use two cache.r5.4xlarge instances on ElastiCache [2] as Redis nodes, with 16 vCPUs and 114 GB memory each.

Benchmark. We evaluate Ditto with the TPC-DS analytics benchmark [17] which provides representative workloads for retail product suppliers. The TPC-DS benchmark involves 100 standard decision support queries that vary in terms of computation and I/O loads. We select four representative queries (Q1, Q16, Q94 and Q95) with different performance characteristics to perform our experiments. In most experiments, the scale factor of the benchmark is configured to 1000, which generates 1TB data of several tables, and the input data size of the four queries ranges from 33 GB to 312 GB. While evaluating the overall performance with Redis as the external storage, the scale factor is set to 100, because Redis is typically used to speed up access to small intermediate data and has limited capacity. We split the input data files into partitions of equal size to enable parallel execution.

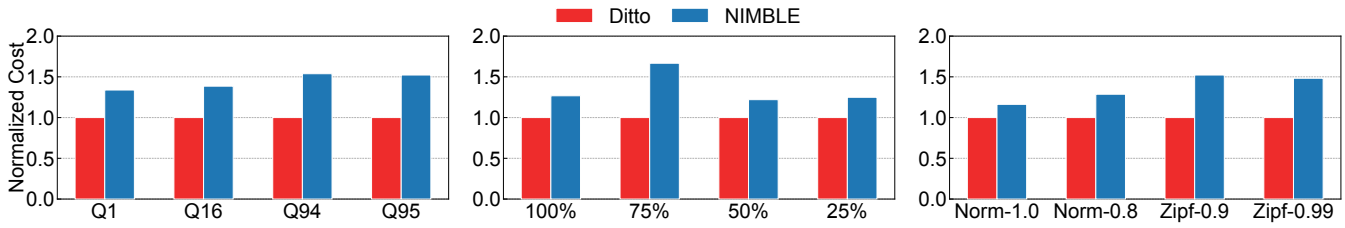
Metrics. We focus on JCT and cost as the metrics to evaluate Ditto. For a given query, JCT is defined as the duration from the submission of the query to the completion of its last task. Cost is defined as the aggregation of the memory footprint multiplied by the execution time of each task, which is in line with major cloud providers' billing policies [5, 10, 13]. To guarantee that all the task are executed favorably, we use the maximum theoretical memory footprint to represent its actual memory footprint. We consider the cost of data persistence in shared memory and Redis, while ignoring that in S3 [7]. This is because memory cost dominates in serverless computing [51], and S3 is priced >1000× less per GB per time unit [3, 6] than memory.

Baseline. We compare Ditto to NIMBLE, the state-of-the-art scheduler for serverless analytics. NIMBLE tunes the degree of parallelism of each stage in proportion to the input data size, and randomly places tasks on function servers.

6.1 Overall Performance under JCT

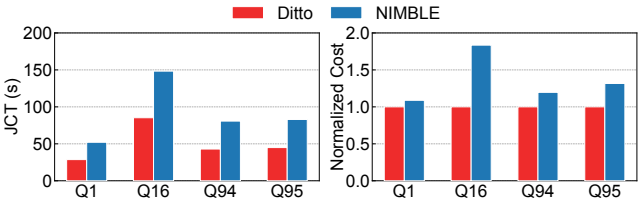
We first compare the overall performance of Ditto to NIMBLE under the metric JCT. To simulate the realistic scenarios, we restrict the number of function slots on each function server and use the function slot distribution to represent the available resources. We configure S3 as the default external storage. Figure 8 demonstrates the JCT reduction of Ditto against NIMBLE under different settings.

JCT under different queries. We set the function slot distribution to Zipf-0.9 and evaluate Ditto with the four TPC-DS queries. Figure 8a indicates that Ditto outperforms NIMBLE by 1.26-1.69× in JCT across all the four queries, which have different characteristics of computation and I/O. This is because Ditto uses *greedy grouping* to reduce the data shuffling time between large stages and uses *DoP ratio computing* to obtain the best parallelism configuration under the given execution time model. Ditto combines these two techniques to further reduce the JCT.



(a) TPC-DS queries. (b) Different slot usage. (c) Different slot distributions.

Figure 9: Performance when optimizing cost.



(a) JCT. (b) Cost.

Figure 10: Performance under Redis.

JCT under different slot usage. We vary the function slot usage to show that Ditto consistently outperforms NIMBLE. The function slot usage is defined as the ratio of the number of available function slots to the maximum number of slots on each server. We configure the function slot usage from 100% to 25% with a step of 25%, and each function server holds the same number of function slots. Ditto runs Q95 under the four resource configurations, and Figure 8b shows that Ditto reduces the JCT by 1.5-2.5× compared to NIMBLE.

JCT under different slot distributions. We also evaluate Ditto under different function slot distributions. The normal distribution and the Zipf distribution are used, for each we specify two skewness parameters. For normal distribution, we symmetrically sample eight probabilities with a fixed step from the standard normal distribution $N(0, 1)$ and its variant $N(0, 0.8)$. The probabilities are the ratios of the permitted number of function slots to the maximum number of that on each server. For Zipf distribution, we use the probabilities calculated from Zipf-0.9 and Zipf-0.99. Figure 8c indicates that Ditto achieves 1.51-1.83× lower JCT than NIMBLE.

6.2 Overall Performance under Cost

The experimental settings are the same with § 6.1. Figure 9 demonstrates the normalized cost of Ditto and NIMBLE under the different function slot distributions. When optimizing cost, Ditto outperforms NIMBLE by 1.16-1.67×. There are two reasons that the cost reduction is smaller than that of JCT. First, for cost optimization, adjusting the degree of parallelism based on the input data size is closer to the optimal solution. The best parallelism configuration when optimizing cost is strongly linearly correlated to the square root of the product of a stage's resource usage and its execution time, as described in § 4.2. And both the resource usage and execution time are related to the input data size. Therefore, the relationship between the best parallelism configuration and the input data size is nearly linear. However, the input data size cannot simply replace the two factors, so Ditto still achieves a comparable cost saving.

Second, Ditto schedules more stages to exchange data through shared memory compared with NIMBLE, increasing the shared memory cost caused by data persistence. Note that eliminating the data persistence in the shared memory is orthogonal to our work and is not trivial. It requires extremely accurate prediction of the reasonable task launch time, so that the data can be immediately consumed by downstream tasks once it is produced. We employ NIMBLE to decide the task launch time to minimize the shared memory cost as much as possible.

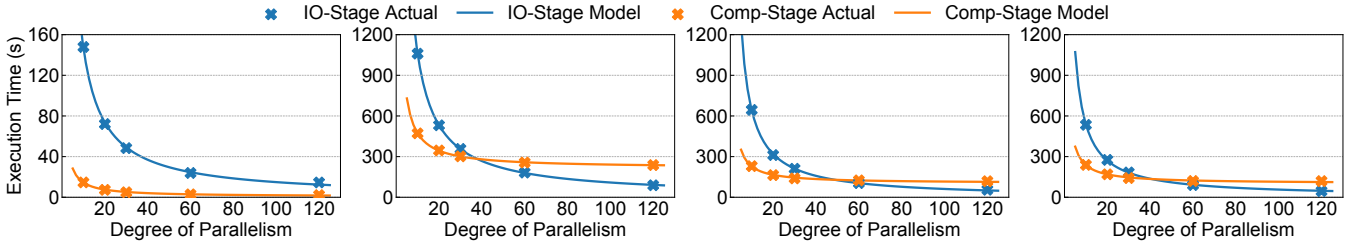
6.3 Performance under Redis

Existing serverless analytics frameworks can employ a small amount of fast external storage to further reduce JCT and cost [45]. We perform an experiment to verify that serverless analytics also benefits from Ditto when using fast external storage. In this experiment, we use Redis, a fast in-memory storage system, to provide the external storage service. We adapt the total data size of the benchmark to 100 GB, which can be accommodated by the Redis server. We configure the function slot distribution as Zipf-0.9 and run Q95 with NIMBLE and Ditto. Figure 10 demonstrates the appreciable performance improvement achieved with Ditto under Redis. In comparison to NIMBLE, Ditto consistently reduces the JCT by 1.74-1.88× and the cost by 1.09-1.83×.

6.4 Deep Dive of Ditto

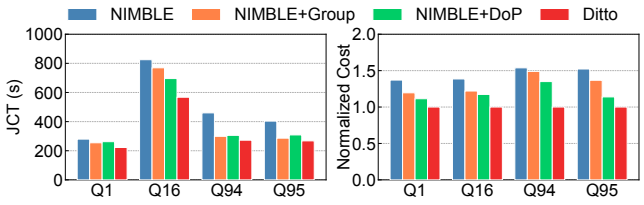
We conduct a deep dive into Ditto to verify the effectiveness of its three components: the execution time predictor, the *greedy grouping* algorithm, and the *DoP ratio computing* algorithm. We also illustrate the execution breakdown of Ditto to gain a better understanding of the benefits from elastic parallelism.

Execution time predictor. We use the four queries with different characteristics to evaluate Ditto's execution time predictor. For each query, we execute one compute-intensive stage and one IO-intensive stage under different degrees of parallelism. The external storage is S3. The results are shown in Figure 11. We plot the average execution time of all tasks in a stage as points, while the lines represent the predicted execution time of the stage based on the execution time model. The gap between the predicted and actual execution time is within 6% for all stages except for Q1's IO-intensive stage. When the degree of parallelism grows to 120, the predicted execution time is 15% larger than the actual execution time. This is because tasks in Q1's IO-intensive stage processes



(a) Q1. (b) Q16. (c) Q94. (d) Q95.

Figure 11: Effectiveness of Ditto's execution time model.



(a) JCT. (b) Cost.

Figure 12: Effectiveness of greedy grouping and DoP ratio computing.

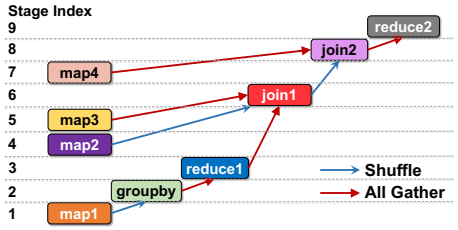


Figure 13: The DAG structure of Q95.

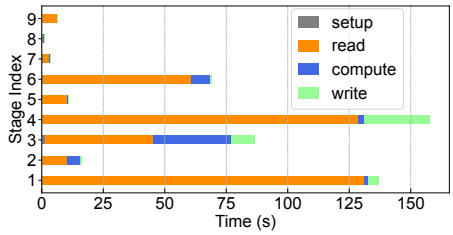


Figure 14: Time breakdown for Q95.

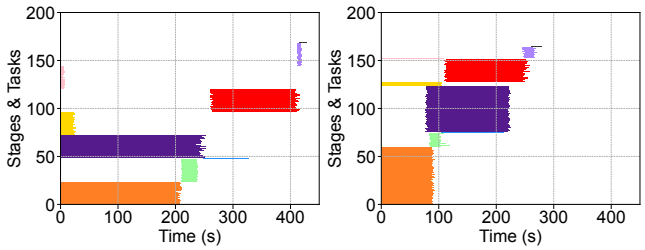
5-10× less data than that in other queries' IO-intensive stages. Due to the higher execution time variance of smaller tasks, the accuracy of the execution time model is lower.

Greedy grouping and DoP ratio computing. To evaluate the effectiveness of *greedy grouping* and *DoP ratio computing*, we compare the performance of the following four approaches: (1) NIMBLE; (2) NIMBLE+Group, which uses *greedy grouping* to bundle stages into groups under the parallelism configuration of NIMBLE; (3) NIMBLE+DoP, which uses *DoP ratio computing* to adjust the parallelism configuration without bundling stages; and (4) Ditto. We run the four queries with the four approaches under the Zipf-0.9 function slot distribution. Figure 12 shows the results. NIMBLE+Group decreases the data shuffling time by co-locating stages on the same function server, so it reduces the JCT by 1.07-1.36× and the cost by 1.2-1.49× compared to NIMBLE. NIMBLE+DoP achieves 1.12-1.23× reduction in JCT and 1.11-1.35× reduction in cost over NIMBLE by using *DoP ratio computing* to adjust to a better parallelism configuration for the specific optimization objective. By combining *greedy grouping* and *DoP ratio computing*, Ditto achieves considerable performance gains over NIMBLE.

Execution breakdown. To illustrate the benefit from elastic parallelism, we compare Ditto to an approach with a fixed parallelism configuration that all stages have the same degree of parallelism. We show the DAG structure of Q95 in Figure 13, where the shuffle operation means the upstream tasks transfer data to their intended downstream tasks. The all-gather operation means each downstream task receives a copy of the data from all upstream tasks. Figure 14 shows the execution time breakdown across different stages with the fixed degree of parallelism (40 in this case). Figure 15a and Figure 15b demonstrate the Q95 execution breakdown under the Zipf-0.9 function slot distribution with fixed parallelism and elastic parallelism, respectively.

When optimizing JCT, Ditto identifies the most time-consuming stage in each path and expands the parallelism of such stage to reduce the execution time of the critical path. In the context of the provided example and the corresponding execution time breakdown illustrated in Figure 14, stage 1 (orange) and stage 4 (purple) emerge as the most time-consuming stages in their respective paths. To address this, Ditto expands the parallelism of stage 1 from 24 to 60, and the parallelism of stage 4 from 24 to 48. It is worth noting that the execution times of stage 5 (yellow) and stage 7 (pink) increase due to the shrinking parallelism of these shorter stages. However, the JCT remains unaffected by the change because the execution of the two stages still overlaps with the critical path.

Figure 15 also shows the benefit derived from Ditto's stage grouping. The execution time of stage 2 (green) decreases even if the degree of parallelism reduces. This is because Ditto bundles stage 1 and stage 2 into a stage group and the results of stage 1 are transferred to stage 2 through zero-copy shared memory, which eliminates the data transmission overhead. The combined effect of elastic parallelism and stage grouping in Ditto significantly reduces the JCT without increasing the total number of functions used.



(a) Fixed parallelism. (b) Elastic parallelism.

Figure 15: Execution breakdown for Ditto when optimizing JCT.

6.5 Ditto Overhead

Scheduling overhead. To evaluate the runtime overhead of Ditto, we measure the scheduling time for each query under different resource usage. Table 1 shows that the scheduling time is sub-millisecond for all queries, which is negligible compared to the query processing time (hundreds of seconds). As the usage of function slots on each node varies from 25% to 100%, the scheduling time remains nearly constant. This is because the time complexity of Ditto is independent of the total number of function slots and is only related to the structure of the query DAG.

Building execution time model. We evaluate the offline time to build the execution time model. For each stage in a query, Ditto collects the profiles under five different degrees of parallelism and uses least squares method to fit the execution time model of the stage. Table 2 shows that the time to build the execution time model is within 0.3 seconds for all queries, which is relatively small compared to the queries with JCTs of hundreds of seconds.

	Scheduling Time			
Query	**25%**	**50%**	**75%**	**100%**
Q1	235 μs	195 μs	264 μs	208 μs
Q16	210 μs	211 μs	227 μs	257 μs
Q94	247 μs	207 μs	169 μs	174 μs
Q95	201 μs	206 μs	221 μs	187 μs

Table 1: Scheduling overhead of Ditto under different resource usage.

Query	Model Building Time
Q1	216 ms
Q16	208 ms
Q94	194 ms
Q95	197 ms

Table 2: Execution time model building overhead of Ditto for different queries.

7 RELATED WORK

Serverless frameworks. With the rise of serverless computing, cloud providers and open-source communities have launched serverless frameworks for specific applications (e.g., SQL analytics [4, 11, 12], video processing [21, 28] and machine learning [20, 47]) and general-purpose computing [5, 9, 10, 13, 14]. Functions in these frameworks communicate with each other in different ways. AWS Lambda [5] uses shared storage to exchange data, while Knative [14] allows direct communication over network. Recent works, SPRIGHT [46] and Pheromone [48], introduce shared memory to accelerate data communication within the node. We remark that Ditto's design is suitable for the two types of remote communication, since the execution time model also adapts to the data transmission over network, where the time is almost proportional to the data size.

Intra-job parallelism. Existing analytics schedulers [31–33, 42, 51] largely side-step the problem of scheduling intra-job parallelism for data analytics jobs. Users are burdened with the choice of the parallelism of the stages in their jobs. Decima [42] focuses on inter-job parallelism and leverages reinforcement learning to allocate parallel workers across jobs. However, it is designed for serverful deployments without fine-grained resource elasticity, where tasks of a stage are executed in multiple waves by a fixed number of workers. NIMBLE [51] manually adjusts the degree of parallelism based on the input data size of each stage, leading to performance degradation. The intra-job parallelism scheduling problem is also related to the parallel query scheduling in databases [24, 29, 30]. Unlike Ditto, however, algorithms proposed for parallel query scheduling optimize for serverful objectives, such as the average JCT and resource utilization under the fixed resource constraints.

Task placement. The task placement optimization aims to leverage the data locality to minimize the data transfer overhead [35, 49]. It is well studied in the context of geo-distributed data analytics [34, 39, 44], and is also studied in the context of serverless computing [19, 22, 41]. However, unlike Ditto, these works do not consider the joint impact of task placement and parallelism configuration.

8 CONCLUSION

We present Ditto, a serverless system that harnesses the elastic parallelism to minimize JCT and cost for data analytics jobs. Ditto efficiently determines the appropriate parallelism configuration and jointly schedules parallelism with task placement to reduce data shuffling overhead. Extensive experiments show that Ditto outperforms the state-of-the-art scheduler by up to 2.5× on JCT and up to 1.8× on cost, while making scheduling decisions within one millisecond.

This work does not raise any ethical issues.

Acknowledgments. This work was supported by the National Key Research and Development Program of China under the grant number 2022YFB4500700.

We sincerely thank the anonymous reviewers for their valuable feedback on this paper. Xin Jin is the corresponding author. Chao Jin, Zili Zhang, Gang Huang, Xuanzhe Liu and Xin Jin are also with the Center for Data Space Technology and System, Peking University and the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

REFERENCES

[1] 2023. Amazon Aurora Serverless. https://aws.amazon.com/rds/aurora/serverless.
[2] 2023. Amazon ElastiCache. https://aws.amazon.com/elasticache/.
[3] 2023. Amazon ElasticCache Pricing. https://aws.amazon.com/elasticache/pricing/.
[4] 2023. Amazon Glue. https://aws.amazon.com/glue/.
[5] 2023. Amazon Lambda. https://aws.amazon.com/lambda/.
[6] 2023. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/.
[7] 2023. Amazon simple storage service (S3). https://aws.amazon.com/s3/.
[8] 2023. Apache Hive. https://hive.apache.org/.
[9] 2023. Apache OpenWhisk. https://openwhisk.apache.org/.
[10] 2023. Azure Functions. https://azure.microsoft.com/products/functions/.
[11] 2023. Azure Synapse Analytics. https://azure.microsoft.com/products/synapse-analytics/.
[12] 2023. Google BigQuery. https://cloud.google.com/bigquery/.
[13] 2023. Google Cloud Functions. https://cloud.google.com/functions/.
[14] 2023. Knative. https://knative.dev/.
[15] 2023. Redis. https://redis.io/.
[16] 2023. Spark SQL. https://spark.apache.org/sql/.
[17] 2023. TPC-DS. https://www.tpc.org/tpcds/.
[18] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-Optimizing Data-Parallel Computing. In *USENIX NSDI*.
[19] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *USENIX ATC*.
[20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2019. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*.
[21] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *ACM Symposium on Cloud Computing*.
[22] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: A Scalable and Locality-Enhanced Framework for Serverless Parallel Computing. In *ACM Symposium on Cloud Computing*.
[23] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* (2019).
[24] Chandra Chekuri, Waqar Hasan, and Rajeev Motwani. 1995. Scheduling Problems in Parallel Query Optimization. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.
[25] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *USENIX NSDI*.
[26] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI*.
[27] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *USENIX ATC*.
[28] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI*.
[29] Minos N. Garofalakis and Yannis E. Ioannidis. 1996. Multi-Dimensional Resource Scheduling for Parallel Queries. *ACM SIGMOD Rec.* (1996).
[30] Minos N. Garofalakis and Yannis E. Ioannidis. 1997. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In *Proceedings of the VLDB Endowment*.
[31] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-Resource Packing for Cluster Schedulers. In *ACM SIGCOMM*.
[32] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic Scheduling in Multi-Resource Clusters. In *USENIX OSDI*.
[33] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *USENIX OSDI*.
[34] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. 2018. Wide-Area Analytics with Multiple Resources. In *EuroSys*.
[35] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*.
[36] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *ACM ASPLOS*.
[37] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja J. Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *Commun. ACM* (2019).
[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *USENIX OSDI*.
[39] Konstantinos Kloudas, Margarida Mamede, Nuno Preguiça, and Rodrigo Rodrigues. 2015. Pixida: Optimizing Data Parallel Jobs in Wide-Area Data Analytics. In *Proceedings of the VLDB Endowment*.
[40] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *USENIX ATC*.
[41] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *USENIX ATC*.
[42] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *ACM SIGCOMM*.
[43] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *ACM SIGMOD*.
[44] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-Distributed Data Analytics. In *ACM SIGCOMM*.
[45] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *USENIX NSDI*.
[46] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *ACM SIGCOMM*.
[47] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM*.
[48] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *USENIX NSDI*.
[49] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*.
[50] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud Workshop*.
[51] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *USENIX NSDI*.

Appendices are supporting material that has not been peer-reviewed.

A APPENDIX

A.1 Optimality of the Intra-path DoP Ratio

For a single-path DAG with n stages, let $\frac{\alpha_i}{d_i}+\beta_i$ be the execution time of the i-th stage according to the execution time model. Let C be the number of function slots. The problem of minimizing the JCT can be expressed as

$$\min_{(d_1,d_2,\ldots,d_n)} \sum_{i=1}^{n}\left(\frac{\alpha_i}{d_i}+\beta_i\right) \tag{7}$$

$$\text{s.t.} \sum_{i=1}^{n} d_i = C \tag{8}$$

Since the term $\sum_{i=1}^{n}\beta_i$ is constant, we can omit it from the objective function. Replace $\frac{\alpha_i}{d_i}$ with $\left(\sqrt{\frac{\alpha_i}{d_i}}\right)^2$ and d_i with $(\sqrt{d_i})^2$, and we can bound the simplified objective function $\sum_{i=1}^{n}\frac{\alpha_i}{d_i}$ using the Cauchy-Schwarz inequality as follows.

$$\left[\sum_{i=1}^{n}\left(\sqrt{\frac{\alpha_i}{d_i}}\right)^2\right]\cdot\left[\sum_{i=1}^{n}(\sqrt{d_i})^2\right] \geq \left(\sum_{i=1}^{n}\sqrt{\frac{\alpha_i}{d_i}}\cdot\sqrt{d_i}\right)^2$$

$$\Leftrightarrow \left(\sum_{i=1}^{n}\frac{\alpha_i}{d_i}\right)\cdot\left(\sum_{i=1}^{n} d_i\right) \geq \left(\sum_{i=1}^{n}\alpha_i\right)^2 \tag{9}$$

$$\Rightarrow \sum_{i=1}^{n}\frac{\alpha_i}{d_i} \geq \frac{1}{C}\left(\sum_{i=1}^{n}\alpha_i\right)^2 \tag{10}$$

The equality in inequality 9 holds if and only if

$$\frac{\sqrt{\alpha_1/d_1}}{\sqrt{d_1}} = \frac{\sqrt{\alpha_2/d_2}}{\sqrt{d_2}} = \cdots = \frac{\sqrt{\alpha_n/d_n}}{\sqrt{d_n}}$$

$$\Rightarrow \frac{d_i}{d_j} = \sqrt{\frac{\alpha_i}{\alpha_j}}, \quad \forall i,j \in \{1,2,\ldots,n\} \tag{11}$$

So the intra-path DoP ratio is optimal, which concludes the proof.

A.2 Optimality of the Balanced Structure in the Inter-path DoP Ratio

We now prove that the completion time for n sibling stages is minimized when the execution time of each stage is balanced. Let the execution time of the i-th stage be $\frac{\alpha_i}{d_i}+\beta_i$. All stages start at the same time, because the function BOTTOM_UP_DOP in Algorithm 1 merges stages in a layer-by-layer manner. Every time the inter-path DoP ratio computing is invoked, the n sibling stages, which may be virtual stages, have no upstream stages.

We prove the optimality of the balanced structure by induction on n. Let C be the total number of function slots. For the base case (i.e., $n=2$), the problem of minimizing the completion time of the two stages can be expressed as

$$\min_{(d_1,d_2)} \max\left\{\frac{\alpha_1}{d_1}+\beta_1, \frac{\alpha_2}{d_2}+\beta_2\right\} \tag{12}$$

$$\text{s.t. } d_1+d_2 = C \tag{13}$$

We define (d_1^*, d_2^*) as the solution subject to $\frac{\alpha_1}{d_1^*}+\beta_1 = \frac{\alpha_2}{d_2^*}+\beta_2$ and then prove that (d_1^*, d_2^*) is the optimal solution. An arbitrary solution (d_1, d_2) subject to $d_1+d_2=C$ either allocates more function slots to the first stage or to the second stage, and we have

$$\begin{cases}\frac{\alpha_1}{d_1}+\beta_1 \geq \frac{\alpha_1}{d_1^*}+\beta_1, & \text{if } d_1 \leq d_1^* \wedge d_2 \geq d_2^* \\ \frac{\alpha_2}{d_2}+\beta_2 \geq \frac{\alpha_2}{d_2^*}+\beta_2, & \text{if } d_1 \geq d_1^* \wedge d_2 \leq d_2^*\end{cases}$$

$$\Rightarrow \max\left\{\frac{\alpha_1}{d_1}+\beta_1, \frac{\alpha_2}{d_2}+\beta_2\right\} \geq \max\left\{\frac{\alpha_1}{d_1^*}+\beta_1, \frac{\alpha_2}{d_2^*}+\beta_2\right\} \tag{14}$$

The completion time under (d_1, d_2) is no less than that under (d_1^*, d_2^*), so (d_1^*, d_2^*) is the optimal parallelism configuration.

For the induction step, we assume that the optimal structure for $n-1$ sibling stages satisfies $\frac{\alpha_1}{d_1^*}+\beta_1 = \frac{\alpha_2}{d_2^*}+\beta_2 = \cdots = \frac{\alpha_{n-1}}{d_{n-1}^*}+\beta_{n-1}$. For n sibling stages, minimizing completion time is formulated as

$$\min_{(d_1,d_2,\ldots,d_n)} \max\left\{\frac{\alpha_1}{d_1}+\beta_1, \frac{\alpha_2}{d_2}+\beta_2, \ldots, \frac{\alpha_n}{d_n}+\beta_n\right\} \tag{15}$$

$$\text{s.t.} \sum_{i=1}^{n} d_i = C \tag{16}$$

Taking the first $n-1$ stages as a whole, we can bound the above objective function as

$$\begin{aligned}&\max_{i\in\{1,2,\ldots,n\}}\left\{\frac{\alpha_i}{d_i}+\beta_i\right\} \\ &= \max\left\{\max_{i\in\{1,2,\ldots,n-1\}}\left\{\frac{\alpha_i}{d_i}+\beta_i\right\}, \frac{\alpha_n}{d_n}+\beta_n\right\} \\ &\geq \max\left\{\min_{(d_1,d_2,\ldots,d_{n-1})}\max_{i\in\{1,2,\ldots,n-1\}}\left\{\frac{\alpha_i}{d_i}+\beta_i\right\}, \frac{\alpha_n}{d_n}+\beta_n\right\}\end{aligned} \tag{17}$$

The equality in inequality 17 holds when the completion time of the first $n-1$ stages (i.e., the first term on the right-hand side of inequality 17) is minimized.

Similar to the base case, let $(d_1^*, d_2^*, \ldots, d_n^*)$ be the solution subject to $\frac{\alpha_1}{d_1^*}+\beta_1 = \frac{\alpha_2}{d_2^*}+\beta_2 = \cdots = \frac{\alpha_n}{d_n^*}+\beta_n$. For an arbitrary solution $(d_1, d_2, \ldots, d_n)$, we adjust $(d_1, d_2, \ldots, d_{n-1})$ to balance the execution time of the first $n-1$ stages. Let $(d_1', d_2', \ldots, d_n')$ be the adjusted solution subject to $\frac{\alpha_1}{d_1'}+\beta_1 = \frac{\alpha_2}{d_2'}+\beta_2 = \cdots = \frac{\alpha_{n-1}}{d_{n-1}'}+\beta_{n-1}$ and $d_n' = d_n$. According to the induction hypothesis and the inequality 17, the completion time under $(d_1', d_2', \ldots, d_n')$ is less than or equal to that under $(d_1, d_2, \ldots, d_n)$. The solution $(d_1', d_2', \ldots, d_n')$ either allocates more function slots to the first $n-1$ stages or to the n-th stage compared to $(d_1^*, d_2^*, \ldots, d_n^*)$, and we have

$$\begin{cases}\frac{\alpha_n}{d_n'}+\beta_n \geq \frac{\alpha_n}{d_n^*}+\beta_n, & \text{if } d_n' \leq d_n^* \\ \max_{i=1}^{n-1}\left\{\frac{\alpha_i}{d_i'}+\beta_i\right\} \geq \max_{i=1}^{n-1}\left\{\frac{\alpha_i}{d_i^*}+\beta_i\right\}, & \text{if } d_n' \geq d_n^*\end{cases}$$

Note that $\sum_{i=1}^{n-1} d_i' \leq \sum_{i=1}^{n-1} d_i^*$ in the second case, so for every $i \in \{1,2,\ldots,n-1\}$, $d_i' \leq d_i^*$. This is because $(d_1', d_2', \ldots, d_{n-1}')$ and $(d_1^*, d_2^*, \ldots, d_{n-1}^*)$ both satisfy the induction hypothesis, and increasing one of d_i' but decreasing another violates the equality constraint for the execution time of the first $n-1$ stages. Consequently, the completion time under $(d_1', d_2', \ldots, d_n')$ is no less than that under $(d_1^*, d_2^*, \ldots, d_n^*)$, and $(d_1^*, d_2^*, \ldots, d_n^*)$ is the optimal solution. Hence by induction, the balanced structure is optimal and we can obtain the inter-path DoP ratio after omitting the small inherent time β_i. This concludes the proof.