Throughput Optimized Scheduler for Dispersed Computing Systems

Diyi Hu
University of Southern California
diyihu@usc.edu

Bhaskar Krishnamachari
University of Southern California
bkrishna@usc.edu

***Abstract*—Dispersed computing is promising paradigm to supplement the conventional cloud computing. Performing computation on the edge leads to significant reduction in communication with the remote cloud. However, challenges exist to fully exploit the advantages of dispersed computing systems: The edge devices are heterogeneous in computation and communication capacity; Tasks decomposed from the target applications can have complex precedence requirements captured by a Directed Acyclic Graph (DAG). With these challenges in mind, we propose a *throughput optimized* task scheduler, targeting at applications (such as computer vision and video processing) where input data are continuously and steadily fed into the execution pipeline. The scheduler incorporates two innovative techniques: task duplication and task splitting. To circumvent low bandwidth data links in the highly heterogeneous environment, we duplicate critical tasks to reroute communication paths. To load-balance the various tasks in complicated target applications, we split heavy-loaded tasks to allow cooperation of multiple nearby edge devices. Through simulation, we thoroughly evaluate the performance of the scheduler under various configuration of tasks and dispersed systems. Task duplication improves throughput of the baseline schedule significantly ($> 1.2\times$), for systems with large variance in data link bandwidth and tasks with large communication-to-computation ratio. Task splitting leads to significant throughput improvement ($> 1.25\times$) for systems with heterogeneous processing power and tasks with large variance in workload. On average, our scheduler improves throughput of the baseline schedule by $> 1.6\times$.**

***Index Terms*—Scheduling, task duplication, load balancing**

I. Introduction

Conventional cloud computing is a centralized computing paradigm. Data collected at the network edge are transmitted to cloud servers for processing. Recent years have witnessed increases in the end user demand and the number of mobile access devices. Especially for large-scale data analysis applications (e.g., surveillance, autonomous navigation and cybersecurity), sending all data to the remote cloud not only requires high bandwidth, but also results in long response time not tolerable by users [1]. On the other hand, dispersed computing is a distributed computation paradigm, which migrates cloud services closer to end users. It leverages the computation and storage capacity of edge devices (e.g., mobile user equipment, routers and IoT devices), so that data collected from end users are (partially) processed locally. Therefore, dispersed computing significantly reduces long-distance data transmission [2].

In general, applications running on dispersed systems can be decomposed into tasks with precedence requirements, captured by a Directed Acyclic Graph (DAG). Many research works [3] and industrial frameworks such as Apache Airflow and Google Cloud Composer focus on scheduling and deployment of task DAGs, enabling dispersed computing systems to run general, complicated applications efficiently.

To evaluate a dispersed computing scheduler, the important metrics include: resource consumption, makespan and throughput. Makespan and throughput are highly related. Makespan, defined as the end-to-end task DAG completion time for *one* input dataset, is critical for real-time applications such as health-care in smart homes [4]. On the other hand, throughput, defined as the average number of processed datasets per unit time, is a better metric for applications like video processing and computer vision [5]. When input to the application is a stream of datasets continuously generated from data source, a high throughput task schedule supports high input data rate, which then implies execution quality.

In this work, we propose a scheduler to optimize execution throughput. Our scheduler is *general* in that we take into account 1) heterogeneity of edge devices in both computation speed and data link bandwidth; 2) heterogeneity of tasks in both computation and communication workload; 3) arbitrary precedence requirements among tasks. In our model, throughput may potentially be bottle-necked by three factors: 1) bandwidth of slow links; 2) large volume of data transfer from parent tasks; 3) large computation workload of a task itself. The proposed scheduler addresses these bottlenecks for throughput optimization. The contributions of this paper are:

- We propose a low complexity scheduler to improve task execution throughput for dispersed systems, by:
  - *Task duplication*, which replaces low-bandwidth links with newly deployed high-bandwidth links, at the cost of some redundancy in computation.
  - *Task splitting*, which improves load-balancing of the overall execution by deploying idle devices to cooperate on heavy-loaded tasks.
- We perform extensive multi-factorial analysis to show the impact of system parameters and task parameters on the throughput improvement. The task duplication and splitting techniques are especially effective for systems and tasks with high heterogeneity.
- Simulation results show that, compared with baselines, our task scheduling achieves an average of $1.6\times$ through-

put improvement under variety of practical settings.

II. Background

A. Dispersed Computing Systems

In the era of IoT, numerous resource-constrained sensors are deployed dispersedly in smart cities. The large amount of data continuously collected from sensors are hard to be completely processed by the remote cloud, due to the expensive back-and-forth communication. Dispersed computing alleviates the burden of the cloud by local computation near the data source.

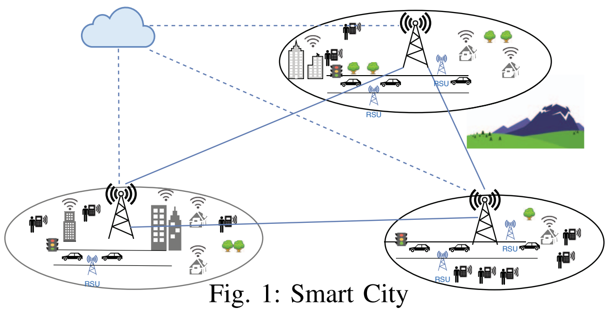


Fig. 1: Smart City

As illustrated by Figure 1, a smart city is divided into multiple blocks (the 3 circles). In each block, various types of sensors integrated into mobile phones, vehicles and smart homes keep collecting data from their vicinity. If the block containing the data source has ample processing power, the collected data may be completely processed by the IoT devices within such block. Otherwise, devices in other blocks or the cloud may be deployed for collaboration. The device network is highly heterogeneous. The devices themselves are of various types. Communication among devices can also be via wired or wireless channels, depending on distance and connectivity.

B. Related Work

1) Scheduling Algorithms for Grid Computing: For applications with precedence requirements (expressed as a DAG), scheduling algorithms for grid computing that minimize makespan can be categorized into three groups: List heuristics, duplication based algorithms and clustering heuristics. The most classic is Heterogeneous Earliest-Finish-Time (HEFT) [6], which is a list scheduling algorithm. HEFT computes a "rank" for each task, based on which the tasks are prioritized. Each task is scheduled greedily to minimize its earliest finish time with an insertion-based approach. Duplication based algorithms such as [7], [8] duplicate the predecessor of task t to the same processor where task t is executed. The basic idea is to save communication time by using idle computation resource on the processor that executes task t. Clustering algorithms partition the task DAG into clusters such that the inter-cluster data movement is minimized. Tasks in the same cluster are scheduled to the same processor if possible.

Apart from the above work in makespan reduction, there is a wealth of literature for boosting throughput [9], [10]. The work in [11] proposes multiple modifications to [6] so that throughput rather than earliest finish time is optimized.

2) Scheduling Algorithms for Dispersed / Mobile Computing: Mobile cloud computing [12] is a type of dispersed computing. By abstracting the remote cloud into virtual machines (VMs), the system consists of dispersed mobile devices and cloud VMs. Offloading is a scheduling approach widely studied for mobile cloud computing. By partitioning the application and sending computation-intensive portions to the cloud, offloading saves power and storage in mobile devices and reduces makespan [12], [2], [13]. The work in [3] provides a fully polynomial time approximation scheme of offloading that further improves the performance in latency (makespan). However, they either cannot handle applications with complex precedence requirements captured as a DAG, or only consider offloading from a *single* mobile device to the cloud. In addition, scheduling algorithms for more general dispersed computing systems have been proposed. In [14], the authors propose a "lowest latency policy" which greedily schedules the task to the edge device (fog node) that leads to minimum communication and computation latency. However, scheduler does not support tasks with precedence requirements.

Compared to latency, throughput is not well studied in dispersed computing. Nonetheless, throughput is a critical metric for streaming applications. For example, streaming signal processing and live video services require high throughput.

3) Task Duplication and Task Splitting: Although task duplication and splitting are known techniques to improve schedule quality, what we propose differ from existing approaches in various aspects. Most of the works [7], [8], [15] using task duplication aim at makespan minimization in the conventional grid computing environment (where homogeneity in either computation [8] or communication [15], [7] is implicitly assumed). Thus, they cannot be easily generalized to produce high throughput schedules in the highly heterogeneous dispersed computing environment. The work in [16] uses task replicas in vehicular networks. The scheduling scheme targets at service reliability improvement, and assumes a single task without precedence requirements. On the other hand, while the proposed task splitting and the known cloud offloading both improve load-imbalance due to heavy-loaded tasks, the two techniques apply to different environments. Existing offloading techniques [12], [2], [13] assume a *single*, resource constraint mobile device connected to the cloud with ample processing power. Task splitting in our case is designed for systems consisting of *large number* of low-end edge devices and the cloud. Thus, when task splitting initiates multiple nearby edge devices for cooperation, careful coordination and resource management is required to ensure high performance.

III. Problem Formulation

A. Objective

Given a set of tasks with precedence requirements and a dispersed computing system with heterogeneity in computation and communication, the proposed scheduler constructs a task execution pipeline and optimizes the steady state throughput when input data stream in at high enough data rate.

The proposed scheduler is a general one. First, it can handle complicated applications where the decomposed tasks can have arbitrary precedence requirements (captured by a DAG). Secondly, the scheduler is designed with various types of network heterogeneity in mind. For an input DAG, the scheduler constructs an execution pipeline, where each pipeline stage either computes a task or transfers data between a task pair. We aim at improving steady-state throughput. Such objective is reasonable when: 1) the pipeline is executed for long enough time, such that the initial warm-up of the pipeline is negligible; 2) input data can be generated by the sensors at a high enough rate, such that the input data stream can always saturate the processing capacity of the pipeline; 3) the environment around the dispersed computing system is stable during the time window of execution, so that the network connectivity (number of links, bandwidth) does not change dramatically.

B. System Model and Notation

We study dispersed computing systems composed of a large number of edge devices. We reasonably assume that each edge device has limited computation and communication capacity, able to execute no more than one task. The edge devices are geographically dispersed. In addition, we also assume the devices are *clusterized* — A cluster consists of devices close to each other. Devices within the same cluster have dedicated, high-speed communication channels with each other. Devices belonging to different clusters establish a communication link if and only if the corresponding clusters are not too far away. We assume heterogeneity in the computation capacity of edge devices, as well as bandwidth of inter-cluster data links.

We assign each cluster an ID (integer i), and each device an ID (tuple $\boldsymbol{m} := (i, n)$, where n is to index devices within cluster i). We capture cluster inter-connections into a graph $\mathcal{G}_{\text{clu}}(\mathcal{V}_{\text{clu}}, \mathcal{E}_{\text{clu}})$. A node $v_{\text{clu}}^{i} \in \mathcal{V}_{\text{clu}}$ represents a cluster i, and an edge $e_{\text{clu}}^{i;j} := \left(v_{\text{clu}}^{i}, v_{\text{clu}}^{j}\right) \in \mathcal{E}_{\text{clu}}$ if edge devices in clusters i and j establish communication links (i.e., clusters i, j are close to each other). Similarly, we use $\mathcal{G}_{\text{dev}}(\mathcal{V}_{\text{dev}}, \mathcal{E}_{\text{dev}})$ to represent the edge device interconnections. A node $v_{\text{dev}}^{\boldsymbol{m}} \in \mathcal{V}_{\text{dev}}$ is the edge device $\boldsymbol{m}$, and an edge $e_{\text{dev}}^{\boldsymbol{m};\boldsymbol{n}} := \left(v_{\text{dev}}^{\boldsymbol{m}}, v_{\text{dev}}^{\boldsymbol{n}}\right) \in \mathcal{E}_{\text{dev}}$ if the pair of edge devices $\boldsymbol{m}$, $\boldsymbol{n}$ have established a communication link. Note that any pair of two devices in the same cluster establish communication links. In addition, for the directed graphs $\mathcal{G}_{\text{clu}}$ and $\mathcal{G}_{\text{dev}}$, we assume the communication links are symmetric, i.e. same bandwidth in both directions of a link.

The attributes of nodes and edges in $\mathcal{G}_{\text{clu}}$ are defined below:

- $v_{\text{dev}}^{\boldsymbol{m}}.s_{\text{comp}}$ — A list, storing the computation speed of device $\boldsymbol{m}$ executing the tasks. The computation speed of the same device can vary for different types of tasks.
- $e_{\text{dev}}^{\boldsymbol{m};\boldsymbol{n}}.s_{\text{comm}}$ — A number, storing the communication speed (link bandwidth) between devices $\boldsymbol{m}$ and $\boldsymbol{n}$.

Finally, in our system model, we assume the network coherence time is large, considering environment with shadowing caused by a large obstruction such as a hill or large building obscuring the main signal path. We assume no outage in the network. Nevertheless, outage could be handled in our algorithm by setting the corresponding bandwidth to zero.

IV. Scheduling Algorithm

A scheduling algorithm maps tasks to edge devices. The scheduler generates a mapping $\mathcal{G}_{\text{map}}(\mathcal{V}_{\text{map}}, \mathcal{E}_{\text{map}})$, where $\mathcal{G}_{\text{map}}$ is a subgraph of $\mathcal{G}_{\text{dev}}$. A node $v_{\text{map}}^{\boldsymbol{m}} \in \mathcal{V}_{\text{map}}$ if and only if a device $\boldsymbol{m}$ is assigned to execute some task. An edge $e_{\text{map}}^{\boldsymbol{m};\boldsymbol{n}} \in \mathcal{E}_{\text{map}}$ if and only if the task executed by $\boldsymbol{m}$ is a parent of the task executed by $\boldsymbol{n}$. Let $v_{\text{map}}^{\boldsymbol{m}}.\text{ID}_{\text{task}}$ record the task ID of device $\boldsymbol{m}$. Let $v_{\text{map}}^{\boldsymbol{m}}.w_{\text{comp}}$ record the computation workload on $\boldsymbol{m}$. Let $e_{\text{map}}^{\boldsymbol{m};\boldsymbol{n}}.w_{\text{data}}$ record the data transfer load from $\boldsymbol{m}$ to $\boldsymbol{n}$.

Given $\mathcal{G}_{\text{map}}$, we construct an execution pipeline. A pipeline stage performs either computation by a node (device) in $\mathcal{V}_{\text{map}}$, or communication by an edge (data link) in $\mathcal{E}_{\text{map}}$. Throughput of a pipeline stage and the overall system are calculated as:

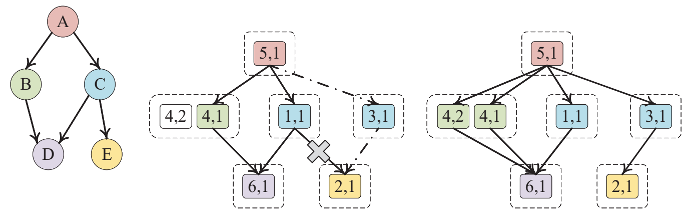
$$
T_{\text{comp}}^{\boldsymbol{m}} = \frac{v_{\text{dev}}^{\boldsymbol{m}}.s_{\text{comp}}\left[v_{\text{map}}^{\boldsymbol{m}}.\text{ID}_{\text{task}}\right]}{v_{\text{map}}^{\boldsymbol{m}}.w_{\text{comp}}}; \qquad T_{\text{comm}}^{\boldsymbol{m};\boldsymbol{n}} = \frac{e_{\text{dev}}^{\boldsymbol{m};\boldsymbol{n}}.s_{\text{comm}}}{e_{\text{map}}^{\boldsymbol{m};\boldsymbol{n}}.w_{\text{data}}}
$$

$$
T_{\text{sys}} = \min_{\boldsymbol{m},\boldsymbol{m}',\boldsymbol{n}'} \left\{T_{\text{comp}}^{\boldsymbol{m}}, T_{\text{comm}}^{\boldsymbol{m}';\boldsymbol{n}'}\right\} \tag{1}
$$

where the $[\cdot]$ operator returns an element in the list.

We propose techniques to address the potential system bottlenecks due to: 1) $e_{\text{dev}}^{\boldsymbol{m};\boldsymbol{n}}.s_{\text{comm}}$; 2) $e_{\text{map}}^{\boldsymbol{m};\boldsymbol{n}}.w_{\text{data}}$; 3) $v_{\text{map}}^{\boldsymbol{m}}.w_{\text{comp}}$.

Task duplication (Bottleneck 1): This step replaces the slow *inter-cluster* link with one or multiple fast links, by duplicating the source task of the slow link. In Figure 2, we study the scheduling of a simple task dag. The initial mapping of Figure 2b is represented by colored nodes and solid lines. Assume the system bottleneck is the link bandwidth between cluster 1 and 2. By duplicating task C to cluster 3, the slow link is completely avoided. Note that task duplication updates both $\mathcal{G}_{\text{task}}$ and $\mathcal{G}_{\text{map}}$. Here, a new task C_{dup} is added to G_{task}.



(a) Task DAG (b) Duplication (c) Splitting

Fig. 2: Illustration of task duplication and splitting

Task splitting (Bottlenecks 2,3): This step distributes workload of a task to multiple devices. Thus, a single, heavy-loaded task is "split" to improve load-balancing of the overall pipeline. In Figure 2c, assume task B (green) is computationally intensive. We improve overall throughput by deploying an idle device $(4, 2)$ to share the computation load. Different from task duplication, the splitting step updates $\mathcal{G}_{\text{map}}$ without changing $\mathcal{G}_{\text{task}}$. Here, after splitting, we have $v_{\text{map}}^{(4,1)}.\text{ID}_{\text{task}} = v_{\text{map}}^{(4,2)}.\text{ID}_{\text{task}} = B$. And $\mathcal{G}_{\text{task}}$ remains unchanged.

Overall workflow: As shown by Algorithm 1, the scheduler consists of three phases. Phase 1 (INITIAL_MAPPING) uses any existing task scheduling algorithm (e.g., [11]) to generate a one-to-one mapping from a task to an edge device.

Phase 2 (DUPLICATE_TASK) and Phase 3 (SPLIT_TASK) are outlined above. The returned $\mathcal{G}_{\text{map}}$ is the final schedule.

Note that our contributions, Phases 2 and 3, are general techniques, since they lead to incremental throughput improvement on *any* schedule returned by Phase 1. In our design, duplication is a global operation involving multiple clusters, and splitting is a local operation constrained in a single cluster. Scheduler with duplication and splitting can be extended to a distributed version so that little global coordination is needed (Section IX). Also, due to the introduction of "clusters", the overall scheduling complexity is low (Sections V, VI). Further discussion on the two techniques are presented in Section VII.

Algorithm 1 Overall scheduling algorithm

```
Input: Task DAG $\mathcal{G}_{\text{task}}$; Device graph $\mathcal{G}_{\text{dev}}$; Cluster $\mathcal{G}_{\text{clu}}$
Output: Mapping $\mathcal{G}_{\text{map}}$; Updated $\mathcal{G}_{\text{task}}$
1: $\mathcal{G}^{\text{init}}_{\text{map}} = \text{INITIAL\_MAPPING}(\mathcal{G}_{\text{task}}, \mathcal{G}_{\text{dev}})$
2: $\mathcal{G}^{\text{dup}}_{\text{map}}, \mathcal{G}^{\text{dup}}_{\text{task}} = \text{DUPLICATE\_TASK}(\mathcal{G}^{\text{init}}_{\text{map}}, \mathcal{G}_{\text{task}}, \mathcal{G}_{\text{clu}})$
3: $\mathcal{G}_{\text{map}} = \text{SPLIT\_TASK}(\mathcal{G}^{\text{dup}}_{\text{map}}, \mathcal{G}^{\text{dup}}_{\text{task}}, \mathcal{G}_{\text{dev}})$
4: return $\mathcal{G}_{\text{map}}$, $\mathcal{G}^{\text{dup}}_{\text{task}}$
```

V. Task Duplication

A. Algorithm Description

This phase takes as input an initial mapping $\mathcal{G}^{\text{init}}_{\text{map}}$, an initial task graph $\mathcal{G}_{\text{task}}$ and the cluster graph $\mathcal{G}_{\text{clu}}$. It outputs a mapping $\mathcal{G}^{\text{dup}}_{\text{map}}$ with higher throughput, and the updated task DAG $\mathcal{G}^{\text{dup}}_{\text{task}}$. As shown by Algorithm 2, this step involves 3 major steps:

1) **Identification of slow links** (Line 2): This step examines all the communication stages (i.e., $e \in \mathcal{E}^{\text{init}}_{\text{map}}$), and identifies the data links with low throughput as well as low bandwidth. In our system model, the returned "slow links" are inter-cluster links. Use *source* and *sink* cluster to denote the two ends of the link. Use *parent* clusters to refer to the clusters that transfer data to the source cluster.
2) **Initial screening for candidate clusters** (Lines 8 to 13): This step identifies candidates from all idle clusters. A candidate should be reachable by the sink and all parents via single-/multi-hop, high speed inter-cluster links.
3) **Final selection of the best-fit cluster** (Lines 14 to 28): This step selects the best cluster from all candidates. For each candidate, the scheduler finds vertex-disjoint paths from the parent and sink clusters to the currently selected candidate, and evaluate the resulting throughput. The best candidate leads to the most throughput improvement.

We explain details of Algorithm 2 with Figure 3 (In the example, assume that all links carry 1 unit of data to transfer). For step 1 (line 2), SLOW_LINKS returns a list of low bandwidth links. It first sorts all the inter-cluster links in $\mathcal{G}_{\text{map}}$ (solid blue lines) by ascending order of throughput. Then it picks the top K links in the sorted list whose link bandwidth is lower than a given threshold. Note from Equation 1 that, either low bandwidth or high data volume may cause the low throughput of a link. Function SLOW_LINKS only selects low

Algorithm 2 Task duplication algorithm

```
Input: Initial mapping $\mathcal{G}^{\text{init}}_{\text{map}}$; Task DAG $\mathcal{G}_{\text{task}}$; Cluster $\mathcal{G}_{\text{clu}}$
Output: Mapping graph $\mathcal{G}^{\text{dup}}_{\text{map}}$; Updated task graph $\mathcal{G}^{\text{dup}}_{\text{task}}$
 1: $\text{CID}^{\text{idle}} \leftarrow$ IDs of clusters where all devices inside are idle
 2: $\text{LID}^{\text{slow}} \leftarrow \text{SLOW\_LINKS}\left(\mathcal{G}^{\text{init}}_{\text{map}}, \mathcal{G}_{\text{clu}}\right)$
 3: $\mathcal{G}^{\text{dup}}_{\text{map}} \leftarrow \mathcal{G}^{\text{init}}_{\text{map}}$;    $\mathcal{G}^{\text{dup}}_{\text{task}} \leftarrow \mathcal{G}_{\text{task}}$;
 4: while $\text{CID}^{\text{idle}} \neq \emptyset$ and $\text{LID}^{\text{slow}} \neq \emptyset$ do
 5:     $(\boldsymbol{p}, \boldsymbol{q}) \leftarrow \text{LID}^{\text{slow}}.\text{POP}()$            ▷ $\boldsymbol{p}, \boldsymbol{q}$: Device IDs
 6:     $i \leftarrow \boldsymbol{p}[0]$;    $j \leftarrow \boldsymbol{q}[0]$;             ▷ $i, j$: Cluster IDs
 7:     $\text{CID}^{\text{par}} \leftarrow$ IDs of parent clusters of cluster $i$
 8:     $\text{CID}^{\text{cand}} \leftarrow \left\{i' \mid i' \in \text{CID}^{\text{idle}}\right\}$
 9:     for $\ell \in \text{CID}^{\text{par}}$ do   ▷ Initial screening on idle clusters
10:         $w^{\ell;i}_{\text{data}} \leftarrow$ Amount of data from cluster $\ell$ to $i$
11:         $\text{CID}^{\text{cand}} \leftarrow \text{CID}^{\text{cand}} \cap \text{BFS}_{\text{CLU}}(\ell, w^{\ell;i}_{\text{data}}, \mathcal{G}_{\text{clu}}, \text{CID}^{\text{idle}})$
12:     $w^{i;j}_{\text{data}} \leftarrow$ Amount of data from cluster $i$ to $j$
13:     $\text{CID}^{\text{cand}} \leftarrow \text{CID}^{\text{cand}} \cap \text{BFS}_{\text{CLU}}(j, w^{i;j}_{\text{data}}, \mathcal{G}_{\text{clu}}, \text{CID}^{\text{idle}})$
14:     $x \leftarrow \texttt{INV}$;    $\mathcal{P}^{\text{dup}} \leftarrow \emptyset$;    $\Delta \leftarrow 0$;
15:     for $c \in \text{CID}^{\text{cand}}$ do          ▷ Find the best candidate
16:         $\text{CID}^{\text{avail}} \leftarrow \left\{i' \mid i' \in \text{CID}^{\text{idle}}\right\}$
17:         $\mathcal{P} \leftarrow \emptyset$       ▷ Set of paths connecting candidate $c$
18:         for $\ell \in \text{CID}^{\text{par}}$ do
19:             $w^{\ell;i}_{\text{data}} \leftarrow$ Amount of data from cluster $\ell$ to $i$
20:             $p \leftarrow \text{ROUTE}_{\text{CLU}}(\ell, c, w^{\ell;i}_{\text{data}}, \mathcal{G}_{\text{clu}}, \text{CID}^{\text{avail}})$
21:             $\mathcal{P} \leftarrow \mathcal{P} \cup \text{SET}(p)$
22:             $\text{CID}^{\text{avail}} \leftarrow \text{CID}^{\text{avail}} \setminus \text{SET}(p)$
23:         $w^{c;j}_{\text{data}} \leftarrow$ Amount of data from cluster $c$ to $j$
24:         $p \leftarrow \text{ROUTE}_{\text{CLU}}(c, j, w^{c;j}_{\text{data}}, \mathcal{G}_{\text{clu}}, \text{CID}^{\text{avail}})$
25:         $\mathcal{P} \leftarrow \mathcal{P} \cup \text{SET}(p)$
26:         $\Delta' \leftarrow$ Throughput improvement after duplication
27:         if $\Delta < \Delta'$ then
28:             $\Delta \leftarrow \Delta'$;    $\mathcal{P}^{\text{dup}} \leftarrow \mathcal{P}$;    $x \leftarrow c$;
29:     $\mathcal{G}^{\text{dup}}_{\text{map}} \leftarrow$ Duplicate tasks on $x$; add connections by $\mathcal{P}^{\text{dup}}$
30:     $\mathcal{G}_{\text{task}} \leftarrow$ Add new nodes correspond to duplicated tasks
31:     $\text{CID}^{\text{idle}} \leftarrow \text{CID}^{\text{idle}} \setminus \left\{\text{IDs of clusters in } \mathcal{P}^{\text{dup}}\right\}$
32: return $\mathcal{G}^{\text{dup}}_{\text{map}}$, $\mathcal{G}^{\text{dup}}_{\text{task}}$
```

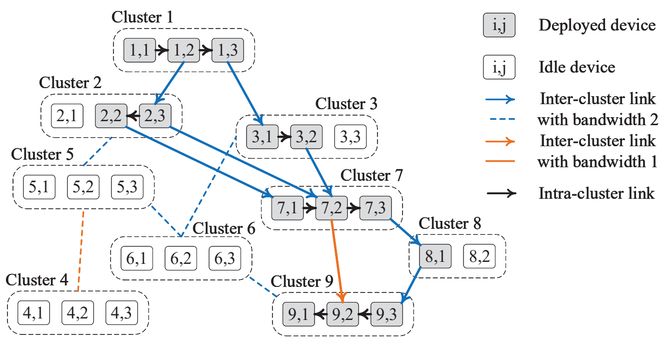


Fig. 3: An example of duplication

throughput links with low bandwidth. In Figure 3, the orange link $\big((7,2),(9,2)\big)$ with bandwidth of 1 is selected.

For step 2 (lines 8 to 13), considering a slow link $(\boldsymbol{p},\boldsymbol{q})$ connecting cluster i, j. The set CID^{par} contains IDs of all parent clusters of i. To determine if a cluster c is a candidate, we check if: 1) all the devices within cluster c are idle (line 8); 2) cluster c can receive data from the parent clusters via a high speed communication pipeline (lines 10, 11); 3) cluster c can transfer data to the sink cluster via a high speed communication pipeline (lines 12, 13) . Clusters satisfying the above conditions are returned by BFS_{CLU}, which takes four inputs: root cluster ID r, required data transfer workload from/to the root cluster w_{data}, cluster graph $\mathcal{G}_{\text{clu}}$, and the IDs of all idle clusters CID^{idle}. Function BFS_{CLU} performs a *conditional* breadth-first search on $\mathcal{G}_{\text{clu}}$ from root r. Suppose the scheduler expands a current leaf cluster f in the BFS tree. An un-visited neighbor n of f will be added into the BFS tree only if cluster n is idle and transfer of $w_{\text{data}}^{f;n}$ from f to n is sufficiently fast. To calculate the f to n data transfer throughput, BFS_{CLU} needs the inter-cluster bandwidth (retrieved from information stored in $\mathcal{G}_{\text{clu}}$) and the data load $w_{\text{data}}^{f;n}$. If cluster r is a parent cluster of i, then $w_{\text{data}}^{f;n}$ equals to the parent-to-source data load $w_{\text{data}}^{r;i}$. We calculate the amount of data that departs from parent clusters and eventually reaches the slow link source device $\boldsymbol{p}$ by backtracking on $\mathcal{G}_{\text{map}}$. Specifically, we build a dependency tree rooted at node $\boldsymbol{p}$, following the reverse direction of links in $\mathcal{G}_{\text{map}}$. We keep expanding the tree until all leaves enter the parent clusters. In Figure 3, the slow link is $\big((7,2),(9,2)\big)$, so $\boldsymbol{p}=(7,2)$, $\boldsymbol{q}=(9,2)$, $i=7$, $j=9$, $\text{CID}^{\text{par}}=\{2,3\}$. The backtracked dependency tree rooted at $\boldsymbol{p}=(7,2)$ has three leaves $(2,3)$, $(2,2)$ and $(3,2)$, making $w_{\text{data}}^{2;7}=1+1$ and $w_{\text{data}}^{3;7}=1$. Also, $w_{\text{data}}^{7;9}=1$. The conditional graph search on $\mathcal{G}_{\text{clu}}$ by BFS_{CLU} will reach clusters $\{5,6\}$ when rooted at $r=2$, clusters $\{6,5\}$ when rooted at $r=3$, and clusters $\{6,5\}$ when rooted at $r=9$. Thus, the candidates are $\{5,6\}$.

Step 3 (lines 14 to 28) selects the best cluster from all candidates. For each candidate, the scheduler finds routes on $\mathcal{G}_{\text{clu}}$ to reach the parent and sink clusters. Function $\text{ROUTE}_{\text{CLU}}$ takes five inputs — the starting cluster ℓ, the ending cluster c, the amount of data w_{data} flowing from ℓ to c, the cluster graph $\mathcal{G}_{\text{clu}}$ and the set of clusters $\text{CID}^{\text{avail}}$ available to connect ℓ and c. The router performs conditional BFS from cluster ℓ, and expand the available clusters until reaching the target c. The router adds a cluster n in its search tree if $n\in\text{CID}^{\text{avail}}$ and n receives data w_{data} fast enough — we calculate the communication throughput of the links entering n in a similar manner as Step 2. Note that the paths connecting the candidate currently under evaluation are vertex-disjoint. Thus, the set of available clusters need to be updated (line 22) for every new route found. The scheduler evaluates the throughput improvement after the routing associated with the current candidate. The candidate leading to the highest throughput is deployed to execute the duplicated tasks. In Figure 3, the best candidate is cluster 6. The routes found by $\text{ROUTE}_{\text{CLU}}$ are $\mathcal{P}_1=(2,5,6)$, $\mathcal{P}_2=(3,6)$ and $\mathcal{P}_3=(6,9)$.

The above 3 steps conclude one round of task duplication. In Figure 3, new inter-cluster links $(2,5)$, $(5,6)$, $(3,6)$, $(6,9)$ are added to replace the slow link $(7,9)$. The scheduler repeats the procedure for all links returned by SLOW_LINKS.

B. Analysis and Discussion

It is worth noticing that in task duplication, the scheduler:

1) traverses the graph (BFS_{CLU}, $\text{ROUTE}_{\text{CLU}}$) in the unit of clusters; and only duplicates to completely idle clusters;
2) may use multi-hop, fast, inter-cluster links to replace a single-hop, slow, inter-cluster link.

Observation 1 indicates that by clusterizing the edge devices, complexity of task duplication is low. Let $N:=|\mathcal{V}_{\text{clu}}|$, $S:=\frac{|\mathcal{V}_{\text{dev}}|}{|\mathcal{V}_{\text{clu}}|}$, the number of links in $\mathcal{G}_{\text{clu}}$ be $M:=|\mathcal{E}_{\text{clu}}|$ and the average degree of a task node be $D:=\frac{|\mathcal{E}_{\text{task}}|}{|\mathcal{V}_{\text{task}}|}$. Further, assume no isolated clusters in $\mathcal{G}_{\text{clu}}$. Thus, $N=\mathcal{O}(M)$. Finally, assume that average number of parent clusters is $\mathcal{O}(D)$.

BFS_{CLU} performs a conditional BFS search on $\mathcal{G}_{\text{clu}}$. Thus, the complexity is $\mathcal{O}(M)$. Similarly, the complexity of $\text{ROUTE}_{\text{CLU}}$ is $\mathcal{O}(M)$. Building the backtracking tree on $\mathcal{G}_{\text{map}}$ from the source device takes $\mathcal{O}(D\cdot S)$. For one round of task duplication, BFS_{CLU} is performed no more than D times on average, and $\text{ROUTE}_{\text{CLU}}$ is performed less than $D\cdot N$ times. In addition, note that one time backtracking tree construction returns all the relevant inter-cluster data load. Therefore, complexity of one round of task duplication is:

$$\mathcal{O}\big(D\cdot S+D\cdot M\cdot N\big) \tag{2}$$

Note that clusterizing the devices into $\mathcal{G}_{\text{clu}}$ makes Algorithm 2 of low complexity. If the duplication operation is in the granularity of a device rather than a cluster, the complexity for one round of duplication will increase from Equation 2 to $\mathcal{O}(DNM\cdot S^2)$. Another reason to perform duplication in the unit of a cluster is, such design avoids the resource contention between task duplication and task splitting. Detailed tradeoff analysis of the two proposed techniques are in Section VII.

Observation 2 indicates that duplication is well applicable to sparsely connected clusters. In general, since the clusters are far away from each other and the cluster graph is sparse, it is hard to find qualified candidate clusters if we require *direct* (single-hop) connection from the parent clusters to the candidate. In Figure 3, to replace one slow link $(7,9)$, we re-route data from cluster 2 to 6 via *indirect* links $(2,5,6)$. It is true that such duplication algorithm may improve system throughput at the cost of higher latency, due to more pipeline stages. On the other hand, we can easily make our scheduler a *latency aware throughput optimized* one — given the latency requirement, we can set an upper bound to the additional number of pipeline stages incurred by task duplication. Thus, BFS_{CLU} constructs *limited depth* BFS trees to select candidates.

In summary, we propose a low complexity task duplication algorithm, which augments throughput of a execution pipeline. The proposed algorithm addresses performance bottleneck due to slow inter-cluster links. In Section IX, we discuss how to extend the duplication algorithm into a distributed version.

VI. Task Splitting

A. Algorithm Description

The objective of task splitting is to improve the overall load-balancing of the pipeline. It deploys multiple devices and links to collaborate on computation intensive or communication intensive pipeline stages. The scheduler performing task splitting takes as input the mapping $\mathcal{G}_{\text{map}}^{\text{dup}}$ and the task graph $\mathcal{G}_{\text{task}}^{\text{dup}}$ returned by task duplication, as well as the device graph $\mathcal{G}_{\text{dev}}$ and cluster graph $\mathcal{G}_{\text{clu}}$. It outputs a mapping $\mathcal{G}_{\text{map}}^{\text{split}}$ with improved throughput. Algorithm 3 shows the main steps:

1) **Identification of bottleneck stages** (Line 6): This step examines all the pipeline stages and identifies the one with minimum throughput (bottleneck).
2) **Local deployment of new resources** (Lines 7 to 16): This step distribute the workload of the bottleneck to *nearby*, idle devices. The newly deployed devices and those already involved in the bottleneck stage lie in the same cluster, to avoid additional inter-cluster communication for the re-distributed workload.

Algorithm 3 Task splitting algorithm

```
Input: $\mathcal{G}_{\text{map}}^{\text{dup}}$; $\mathcal{G}_{\text{task}}^{\text{dup}}$; $\mathcal{G}_{\text{clu}}$; $\mathcal{G}_{\text{dev}}$
Output: $\mathcal{G}_{\text{map}}^{\text{split}}$
 1: $\mathcal{G}_{\text{map}}^{\text{temp}} \leftarrow \mathcal{G}_{\text{map}}^{\text{dup}}$        ▷ Temporary mapping after duplication
 2: $\Delta \leftarrow 1$        ▷ Initialization to some positive value
 3: $T_{\text{sys}} \leftarrow$ Overall throughput of $\mathcal{G}_{\text{map}}^{\text{dup}}$
 4: while $\Delta > 0$ do
 5:     $\mathcal{G}_{\text{map}}^{\text{split}} \leftarrow \mathcal{G}_{\text{map}}^{\text{temp}}$
 6:     $S_{\text{bneck}} \leftarrow$ BOTTLENECK$(\mathcal{G}_{\text{map}}^{\text{split}}, \mathcal{G}_{\text{dev}})$
 7:     if $S_{\text{bneck}}$ is a communication stage then
 8:         $(\boldsymbol{p}, \boldsymbol{q}) \leftarrow$ IDs of the two devices involved in $S_{\text{bneck}}$
 9:         $\mathcal{G}_{\text{map}}^{\text{temp}}, T'_{sys} \leftarrow$ SPLIT$(\boldsymbol{p}, S_{\text{bneck}}, \mathcal{G}_{\text{map}}^{\text{split}}, \mathcal{G}_{\text{dev}}, \mathcal{G}_{\text{clu}})$
10:         if $T'_{sys} > T_{sys}$ then
11:             $\Delta \leftarrow T'_{\text{sys}} - T_{\text{sys}}$;        $T_{\text{sys}} \leftarrow T'_{\text{sys}}$;
12:             continue
13:         $\mathcal{G}_{\text{map}}^{\text{temp}}, T'_{sys} \leftarrow$ SPLIT$(\boldsymbol{q}, S_{\text{bneck}}, \mathcal{G}_{\text{map}}^{\text{split}}, \mathcal{G}_{\text{dev}}, \mathcal{G}_{\text{clu}})$
14:     else        ▷ $S_{\text{bneck}}$ is a compucation stage
15:         $\boldsymbol{p} \leftarrow$ ID of the device involved in $S_{\text{bneck}}$
16:         $\mathcal{G}_{\text{map}}^{\text{temp}}, T'_{sys} \leftarrow$ SPLIT$(\boldsymbol{p}, S_{\text{bneck}}, \mathcal{G}_{\text{map}}^{\text{split}}, \mathcal{G}_{\text{dev}}, \mathcal{G}_{\text{clu}})$
17:     $\Delta \leftarrow T'_{\text{sys}} - T_{\text{sys}}$;        $T_{\text{sys}} \leftarrow T'_{\text{sys}}$;
18: return $\mathcal{G}_{\text{map}}^{\text{split}}$
```

Below we explain the details of Algorithm 3. First the scheduler calls the BOTTLENECK function, which traverses each node and link of $\mathcal{G}_{\text{map}}$ and identifies the stage with minimum throughput. It then calls the SPLIT function to deploy new devices to address the bottleneck. The SPLIT function takes four input arguments: ID $\boldsymbol{p}$ of the device executing the task to be split, the mapping $\mathcal{G}_{\text{map}}^{\text{split}}$, the device graph $\mathcal{G}_{\text{dev}}$ and the bottleneck S_{bneck}. The scheduler only splits tasks locally within the cluster. Suppose the scheduler selects an idle device $\boldsymbol{p}'$ from the candidate devices. Then $v_{\text{map}}^{\boldsymbol{p}'}$ is added to the mapping graph $\mathcal{G}_{\text{map}}$, and $v_{\text{map}}^{\boldsymbol{p}'}.\text{ID}_{\text{task}} = v_{\text{map}}^{\boldsymbol{p}}.\text{ID}_{\text{task}}$. The workload distribution between $\boldsymbol{p}$ and $\boldsymbol{p}'$ is discussed in the following paragraphs. Note that for a communication bottleneck stage, the SPLIT function first tries to split the source task. If such splitting does not lead to throughput improvement, the scheduler then splits the sink task.

The above steps conclude one round of splitting. The scheduler keeps splitting the bottleneck tasks until no further improvement to the overall system throughput can be made by deploying idle resources. Throughout the splitting procedure, the task graph $\mathcal{G}_{\text{task}}$ keeps unchanged, while new nodes and edges are added into the mapping $\mathcal{G}_{\text{map}}$.

Note that for task splitting to take place, we are assuming an execution pipeline performing *batch processing*. Suppose within each pipeline stage, a batch of b (independent) datasets are fed into the pipeline (b can be of a small value, depending on the latency requirement). We need to enforce a *data re-distribution policy* to avoid data mismatch when the task DAG has complicated precedence requirements.

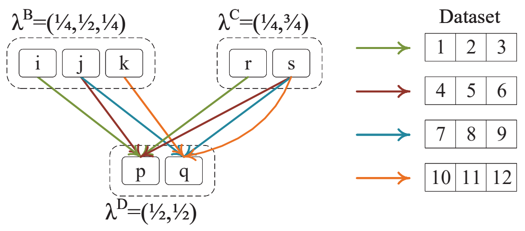


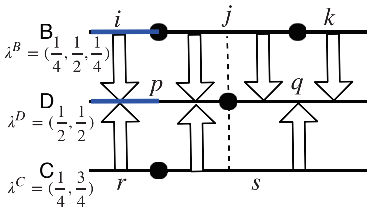
Fig. 4: Example showing the data re-distribution policy

To formalize the data re-distribution policy, we first introduce a parameter $\boldsymbol{\lambda}^D$, which is a length ℓ vector defining the workload partition among the ℓ devices collaborating on a single task D. Let the IDs of these devices be $\boldsymbol{p}_1, \boldsymbol{p}_2, ..., \boldsymbol{p}_\ell$. So in each pipeline stage, device $\boldsymbol{p}_i$ receives $b \cdot \lambda_i^D$ number of datasets within a batch. Correspondingly, $v_{\text{map}}^{\boldsymbol{p}_i}.w_{\text{comp}} = \lambda_i^D b \cdot v_{\text{task}}^D.w_{\text{comp}}$, and $\sum_{i=1}^{\ell} \lambda_i^D = 1$. We describe the data re-distribution policy by the example in Figure 4. Suppose task D is split between two devices $\boldsymbol{p}$ and $\boldsymbol{q}$, where $\boldsymbol{\lambda}^D = (\frac{1}{2}, \frac{1}{2})$. The two parents B and C are split by three devices $\boldsymbol{i}$, $\boldsymbol{j}$, $\boldsymbol{k}$ and two devices $\boldsymbol{r}$, $\boldsymbol{s}$ respectively, where $\boldsymbol{\lambda}^B = (\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$ and $\boldsymbol{\lambda}^C = (\frac{1}{4}, \frac{3}{4})$. Further suppose the batch size is $b = 12$. An example of valid dataflow from parents B, C to child D is:

- Datasets $1, 2, 3$ flow from devices $\boldsymbol{i}, \boldsymbol{r}$ to $\boldsymbol{p}$;
- Datasets $4, 5, 6$ flow from devices $\boldsymbol{j}, \boldsymbol{s}$ to $\boldsymbol{p}$;
- Datasets $7, 8, 9$ flow from devices $\boldsymbol{j}, \boldsymbol{s}$ to $\boldsymbol{q}$; and
- Datasets $10, 11, 12$ flow from devices $\boldsymbol{k}, \boldsymbol{s}$ to $\boldsymbol{q}$.

In general, any valid dataflow should satisfy the following condition: if a device executing parent task B and a device executing parent task C both output datasets of index k, then the two datasets k should flow into the same downstream device executing the child task D. We develop a simple yet efficient data re-distribution policy that guarantees valid parent-child dataflow, as illustrated by Figure 5.

In Figure 5, we abstract the weight vector $\boldsymbol{\lambda}^t$ (where $t = B, C, D$ in this case) as a unit-length line partitioned into segments where length of the a^{th} segment is proportional to the value of $\boldsymbol{\lambda}_a^t$. To assign the parent-child links, first we further divide the weight vector of parent tasks by the division points in child task's weight vector (dashed line). Then we compute the weight of each segment in the further divided weight vectors. One such weighted segment corresponds to one communication link in $\mathcal{G}_{\text{map}}^{\text{split}}$. The source device of the link is the index annotated on the segment ($\boldsymbol{i}$ in the example). The sink device of the link is the annotated index on the projection of the segment on the child's weight vector ($\boldsymbol{p}$ in the example). The number of datasets transmitted on the link in each pipeline stage is $b \cdot \omega$ where ω is the length of the further partitioned segment (blue segment). In the figure, the link corresponding to the blue segment is $e_{\text{map}}^{\boldsymbol{i},\boldsymbol{p}}$. And $b \cdot \omega = 12 \cdot \frac{1}{4} = 3$ datasets are transmitted via this $(\boldsymbol{i}, \boldsymbol{p})$ link. Following such procedure, we come up with the aforementioned example dataflow which transmits $b = 12$ datasets via 7 parent-child links. In summary, given any weights $\boldsymbol{\lambda}^*$, our data re-distribution policy incurs *linear* time complexity to derive a valid parent-child dataflow.



B
$\lambda^B = (\frac{1}{4}, \frac{1}{2}, \frac{1}{4})$
D
$\lambda^D = (\frac{1}{2}, \frac{1}{2})$
C
$\lambda^C = (\frac{1}{4}, \frac{3}{4})$
i
j
k
p
q
r
s

Fig. 5: Data re-distribution after splitting

B. Analysis and Discussion

Task splitting is performed within the cluster to avoid additional inter-cluster communication cost. Additionally, such restriction reduces the time complexity of the scheduler. For the analysis below, we use the same notations as in Section V. Further, we assume a simple heuristic to calculate the vector $\boldsymbol{\lambda}^t$ as $\boldsymbol{\lambda}_a^t = \frac{1}{|\boldsymbol{\lambda}^t|}$, where $1 \leq a \leq |\boldsymbol{\lambda}^t|$. For each round of splitting, there are at most S idle devices within the cluster. Once selecting an idle device, the time for enforcing the data re-distribution policy is $\mathcal{O}(S)$. In total, to split a bottleneck task once, the scheduler performs $\mathcal{O}(S^2)$ amount of work.

VII. Comparison of Duplication and Splitting

We remove low bandwidth inter-cluster links by *duplicating* tasks to an idle cluster. We *split* the workload of a bottleneck task, to devices within the same cluster. The design choices such that duplication operates across clusters and splitting operates within a cluster, are complementary. In this way, we avoid possible resource contention across the two phases.

The potential "contention" mentioned above is due to the fact that both techniques use additional resources to improve throughput. An alternative to perform duplication is to duplicate the associated task to some *partially* occupied cluster X. As a result, in task splitting, tasks originally distributed to cluster X may not be easily split, since the previous duplication step may take up all the idle resources in X.

Benefit in throughput from the two techniques varies depending on scenarios. Theoretically, in the dispersed system where inter-cluster link bandwidth has large variance, there will be higher throughput improvement. It is because when the link quality is highly heterogeneous, a good link that substitute a poor inter-cluster link can significantly boost throughput. Similarly, in dispersed system with smaller cluster size, duplication is favorable because there are more inter-cluster links. This leads to higher chance of encountering a low bandwidth link. The conclusions are verified by experiments.

VIII. Experimental Results

A. Experimental Setup

We implement a software simulator in `Python3`, which takes as input user customized configuration for task graph, device graph and cluster graph. The simulator randomly generates $\mathcal{G}_{\text{task}}$, $\mathcal{G}_{\text{dev}}$ and $\mathcal{G}_{\text{clu}}$ based on the user configuration, and schedules tasks based on Algorithm 1. For random task graph generation, we follow the generator design in [8]. The supported parameters to specify $\mathcal{G}_{\text{task}}$, $\mathcal{G}_{\text{dev}}$ and $\mathcal{G}_{\text{clu}}$ are summarized in the three parts of Table I, respectively. Note that the mean of parent-child task data transfer volume in $\mathcal{G}_{\text{task}}$ is calculated by: $\mu(w_{\text{data}}) = \gamma_{\text{ccr}} \cdot \mu(w_{\text{comp}})$. We assume the parameters w_{comp}, w_{comm}, s_{comp}, s_{comm} are independent random variables following Gaussian distributions defined by the corresponding mean $\mu(\cdot)$ and variance $\sigma(\cdot)$. Under the clusterized system model, we assume the bandwidth of device-device links follows two separate Gaussian distributions, corresponding to intra-cluster bandwidth (defined by μ_1, σ_1) and inter-cluster bandwidth (defined by μ_2, σ_2) respectively.

TABLE I: Parameters to configure $\mathcal{G}_{\text{task}}$, $\mathcal{G}_{\text{dev}}$ and $\mathcal{G}_{\text{clu}}$

Name	**Meaning**
α, β	Depth, width of task DAG
δ	Average degree of a task node
γ_{ccr}	**C**ommunication-**C**omputation **R**atio (CCR)
$\sigma(w_{\text{data}})$	Variance of parent-child data transfer load
$\mu(w_{\text{comp}})$, $\sigma(w_{\text{comp}})$	Mean, variance of task computation workload
$\mu(s_{\text{comp}})$, $\sigma(s_{\text{comp}})$	Mean, variance of device computation speed
$\mu_1(s_{\text{comm}})$, $\sigma_1(s_{\text{comm}})$	Mean, variance of intra-cluster link bandwidth
N	Number of clusters
S	Average size of a cluster
D	Average degree of a cluster
$\mu_2(s_{\text{comm}})$, $\sigma_2(s_{\text{comm}})$	Mean, variance of inter-cluster link bandwidth

We evaluate the effect of the parameters (in Table I) on the throughput improvement. For each configuration, the simulator runs the scheduler 2500 times, each time on a newly generated $\mathcal{G}_{\text{task}}$, $\mathcal{G}_{\text{dev}}$, $\mathcal{G}_{\text{clu}}$ confined to the configuration. The simulator returns the average throughput improvement due to duplication only, splitting only and duplication plus splitting. The initial mapping (INITIAL_MAPPING, Algorithm 1) is based on the throughput optimized scheduler in [11]. We show that in general, our techniques significantly improve throughput of an already highly optimized baseline schedule [11].

B. Impact of Task DAG Parameters

We evaluate the impact of task parameters $\sigma(w_{\text{comp}})$, γ_{ccr} and $\sigma(w_{\text{data}})$ on throughput improvement. We vary the parameters $\sigma(w_{\text{comp}})$ from 0.1 to 1, γ_{ccr} from 0.001 to 15.625, $\sigma(w_{\text{data}})$ from 0.1 to 0.7, and fix the other parameters to $\alpha = 6$, $\beta = 4$, $\delta = 4$, $\mu(w_{\text{comp}}) = 2$, $\mu(s_{\text{comp}}) = 10$, $\sigma(s_{\text{comp}}) = 3$, $\mu_1(s_{\text{comm}}) = 50$, $\sigma_1(s_{\text{comm}}) = 5$, $N = 80$, $S = 4$, $D = 7$, $\mu_2(s_{\text{comm}}) = 10$, $\sigma_2(s_{\text{comm}}) = 5$.

Figure 6 shows the effect of $\sigma(w_{\text{comp}})$, γ_{ccr} and $\sigma(w_{\text{data}})$. The y-axes correspond to the throughput improvement due to the duplication and splitting techniques. Figure 6.A and 6.C show that duplication and splitting are both very effective in improving throughput, even when the tasks are highly heterogeneous in computation and communication. Duplication leads to at least $1.2\times$ throughput improvement and splitting leads to at least $1.25\times$ improvement. The overall effect is an average of $1.6\times$ higher throughput compared with the baseline. Figure 6.B shows that 1) When the tasks are computation intensive (small γ_{ccr}), duplication is not helpful since even very slow inter-cluster links are unlikely to become the overall system bottleneck. Splitting is still effective, since it can better balance the computation load. 2) When the tasks are communication intensive (large γ_{ccr}), both duplication and splitting are effective. The overall improvement is around $1.6\times$.

We conclude the duplication and splitting techniques are suited for mobile-cloud environment. The scheduler is robust and flexible to boost throughput of complicated applications.

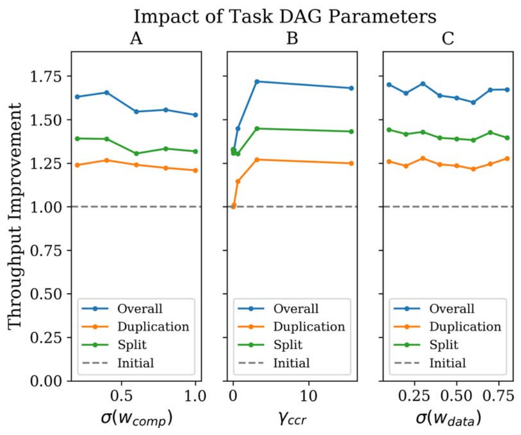


Fig. 6: Impact of $\sigma(w_{\text{comp}})$, γ_{ccr} and $\sigma(w_{\text{data}})$

C. Impact of System Parameters

We evaluate the impact of three device/cluster graph parameters D, S, $\sigma_2(s_{\text{comm}})$ on throughput improvement. We vary the parameters $\sigma_2(s_{\text{comm}})$ from 10 to 70, S from 3 to 9, D from 5 to 11, and fix the other parameters to $\alpha = 6$, $\beta = 4$, $\delta = 4$, $\mu(w_{\text{comp}}) = 2$, $\sigma(w_{\text{comp}}) = 0.3$, $\mu(s_{\text{comp}}) = 10$, $\sigma(s_{\text{comp}}) = 3$, $\mu_1(s_{\text{comm}}) = 50$, $\sigma_1(s_{\text{comm}}) = 5$, $N = 80$, $\mu_2(s_{\text{comm}}) = 10$.

Figure 7 shows the effect of D, S and $\sigma_2(s_{\text{comm}})$. Figure 7.A shows that the connectivity of the clusters has insignificant impact on the throughput improvement. For well connected clusters, the initial mapping likely finds a good enough scheduling scheme, limiting the benefit of the subsequent duplication and splitting steps. On the other hand, for such clusters with higher degree D, task duplication will have higher chance of finding routes to the candidate clusters. Considering these effects altogether, the duplication and splitting techniques work well for both sparsely and densely connected clusters. Figure 7.B shows that, for a given task DAG with fixed size, task duplication works well for relatively small clusters. If the cluster size is very large, then the initial mapping algorithm tends to fit most of the task nodes within the same cluster, and thus reduces the chance of encountering a slow inter-cluster link. In practice, it is often not feasible to group many devices closely together into a large cluster. On the other hand, splitting leads to more consistent throughput improvement ($1.25\times$ on average) for various cluster sizes. For larger clusters, there is also higher probability that task splitting algorithm can find a suitable idle device nearby the bottlenecked device. Figure 7.C shows the effect of variance of inter-cluster link bandwidth. If the link is highly heterogeneous, task duplication will likely find better routes that will bypass the slow inter-cluster links. Similarly, task splitting will better re-distribute the load due to "imbalance" of the link speed.

Overall, the scheduler leads to significantly higher throughput when the link bandwidth is highly heterogeneous. The techniques incorporated into the scheduler is thus very general.

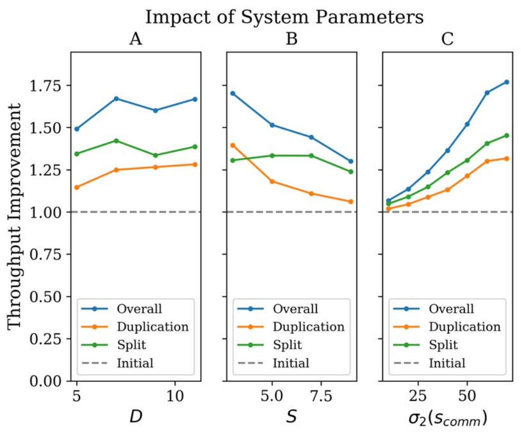


Fig. 7: Impact of D, S and $\sigma_2(s_{\text{comm}})$

D. Comprehensive Multi-Factor Design Analysis

To systematically analyze the importance of the numerous parameters defined in Table I, we present comprehensive multi-factor design analysis in this section. We use the Central-Composite Design (CCD) methodology [17] for design space exploration, and the ANOVA [18] model for fitting the Table I parameters under a quadratic model.

The CCD method generates two types of design points for experiments: "factorial" points by taking two values (low/high) of each design parameter, and $2k$ number of "axial" points for the k design parameters. For each generated design points, we run the simulator to obtain the value of throughput improvement. Based on the collected data, ANOVA ranks the importance of each parameter by assigning a p-value. Smaller p-value indicates more significance on throughput improvement. Table II lists the 5 parameters with smallest p-value. Consistent with the observations in Sections VIII-B and VIII-C, we conclude that cluster size S, variance of inter-cluster bandwidth $\sigma_2(s_{\text{comm}})$ and task CCR γ_{ccr} are the most important factors influencing throughput improvement.

TABLE II: Parameters with lowest p-values

	$\sigma_2(s_{\text{comm}})$	S	γ_{ccr}	D	$\sigma(w_{\text{data}})$
p-value	$< \mathbf{0.0001}$	$< \mathbf{0.0001}$	$< \mathbf{0.0001}$	0.0158	0.3412

Figure 8 verifies the conclusion that $\sigma_2(s_{\text{comm}})$ and S are significant factors. The contour heat map shows the throughput improvement due to duplication under various $\sigma_2(s_{\text{comm}})$ and S settings. The four dots at the center of each plot show the design points for actual measurement. The contours and the complete heat map are interpolated based on the ANOVA fitted model. First of all, we observe that higher throughput improvement by duplication is achieved when the inter-cluster bandwidth is more heterogeneous, and the cluster size is relatively small. This is in line with the conclusion in Section VIII-C. Secondly, we observe that when varying $\sigma_2(s_{\text{comm}})$ and S, both two plots show similar trends of throughput improvement, even through these plots correspond to experiments with drastically different settings of other parameters (such as D, $\sigma(w_{\text{data}})$, δ, etc.). This indicates that $\sigma_2(s_{\text{comm}})$ and S are indeed design factors having dominant impact on performance.

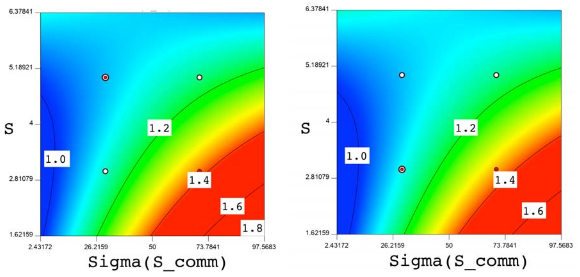


Fig. 8: Throughput improvement under various S, $\sigma_2(s_{\text{comm}})$

IX. Conclusion and Future Work

We proposed a throughput optimized scheduler for dispersed computing systems. Task duplication addressed the bottleneck of slow data links. Task splitting improved load-balancing in computation workload and data transfer. Applying these two techniques, our scheduler generated high throughput schedules in polynomial time. We also performed thorough multi-factorial analysis to understand the impact of system parameters on throughput improvement.

We will extend our scheduler to a *distributed* one by the following modifications. First, to identify bottleneck, we can assign each non-entry task with a controller, following routine similar to [19]. Then each device in $\mathcal{G}_{\text{map}}$ reports the minimum of its computation throughput and outgoing link throughput to its controller. Such information will eventually be propagated to the device executing the entry task, which will subsequently identify the overall bottleneck. The second step will invoke distributed BFS search. In task duplication, the BFS is initiated by the root nodes in parallel without global coordination.

Acknowledgment

This material is based upon work supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0053. Any views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] C.-S. Yang, R. Pedarsani, and S. Avestimehr, "Communication-aware scheduling of serial tasks for dispersed computing," *arXiv:1804.06468*.
[2] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *arXiv:1702.05309*, 2017.
[3] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE Transactions on Mobile Computing*, 2017.
[4] H. Wang, J. Gong, Y. Zhuang, H. Shen, and J. Lach, "Healthedge: Task scheduling for edge computing with health emergency and human behavior consideration in smart homes," in *Big Data (Big Data), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1213–1222.
[5] Y. Gu, C. Q. Wu, X. Liu, and D. Yu, "Distributed throughput optimization for large-scale scientific workflows under fault-tolerance constraint," *Journal of grid computing*, vol. 11, no. 3, pp. 361–379, 2013.
[6] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, Mar 2002.
[7] M. Sajid and Z. Raza, "Turnaround time minimization-based static scheduling model using task duplication for fine-grained parallel applications onto hybrid cloud environment," *IETE Journal of Research*.
[8] D. Bozdag, F. Ozguner, E. Ekici, and U. Catalyurek, "A task duplication based scheduling algorithm using partial schedules," in *Parallel Processing, 2005. ICPP 2005. International Conference on*. IEEE.
[9] S. G. Ahmad, C. S. Liew, M. M. Rafique, E. U. Munir, and S. U. Khan, "Data-intensive workflow optimization based on application task graph partitioning in heterogeneous computing systems," in *Big Data and Cloud Computing (BdCloud), 2014 IEEE Fourth International Conference on*. IEEE, 2014, pp. 129–136.
[10] X. Wei, Y. Liang, T. Wang, S. Lu, and J. Cong, "Throughput optimization for streaming applications on CPU-FPGA heterogeneous systems," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 488–493.
[11] M. Gallet, L. Marchal, and F. Vivien, "Efficient scheduling of task graph collections on heterogeneous resources," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–11.
[12] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*.
[13] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan, "Odessa: enabling interactive perception applications on mobile devices," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 43–56.
[14] K. Intharawijitr, K. Iida, and H. Koga, "Analysis of fog model considering computing and communication latency in 5g cellular networks," in *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on*. IEEE, 2016, pp. 1–4.
[15] S. Ranaweera and D. P. Agrawal, "A task duplication based scheduling algorithm for heterogeneous systems," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, May.
[16] Y. Sun, J. Song, S. Zhou, X. Guo, and Z. Niu, "Task replication for vehicular edge computing: A combinatorial multi-armed bandit based approach," *CoRR*, vol. abs/1807.05718, 2018.
[17] N. R. Draper, "Central composite designs," *Wiley StatsRef: Statistics Reference Online*, 2014.
[18] L. St, S. Wold *et al.*, "Analysis of variance (anova)," *Chemometrics and intelligent laboratory systems*, vol. 6, no. 4, pp. 259–272, 1989.
[19] P. Sakulkar, P. Ghosh, B. Krishnamachari, S. Avestimehr, M. Annavaram, and etc., "Wave: A distributed scheduling framework for dispersed computing," in *Technical Report*. USC ANRG, 2018.