DynDL: Scheduling Data-Locality-Aware Tasks with Dynamic Data Transfer Cost for Multicore-Server-Based Big Data Clusters

Jiahui Jin 1, Qi An 2, Wei Zhou 2 and Jiakai Tang 1 and Runqun Xiong 1,*

1 School of Computer Science and Engineering, Southeast University, Nanjing 211189, China; jjin@seu.edu.cn (J.J.); tangjk@seu.edu.cn (J.T.)
2 School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China; anqii@seu.edu.cn (Q.A.); zhou_weii@seu.edu.cn (W.Z.)
* Correspondence: rxiong@seu.edu.cn; Tel.: +86-25-52091025

Received: 25 October 2018; Accepted: 8 November 2018; Published: 10 November 2018

Featured Application: This work is applicable to most state-of-the-art data-parallel frameworks, such as Hadoop, Spark, Pregel, and Tensorflow, to improve task-scheduling performance.

Abstract: Network bandwidth is a scarce resource in big data environments, so data locality is a fundamental problem for data-parallel frameworks such as Hadoop and Spark. This problem is exacerbated in multicore server-based clusters, where multiple tasks running on the same server compete for the server’s network bandwidth. Existing approaches solve this problem by scheduling computational tasks near the input data and considering the server’s free time, data placements, and data transfer costs. However, such approaches usually set identical values for data transfer costs, even though a multicore server’s data transfer cost increases with the number of data-remote tasks. Eventually, this hampers data-processing time, by minimizing it ineffectively. As a solution, we propose DynDL (Dynamic Data Locality), a novel data-locality-aware task-scheduling model that handles dynamic data transfer costs for multicore servers. DynDL offers greater flexibility than existing approaches by using a set of non-decreasing functions to evaluate dynamic data transfer costs. We also propose online and offline algorithms (based on DynDL) that minimize data-processing time and adaptively adjust data locality. Although DynDL is NP-complete (nondeterministic polynomial-complete), we prove that the offline algorithm runs in quadratic time and generates optimal results for DynDL’s specific uses. Using a series of simulations and real-world executions, we show that our algorithms are 30% better than algorithms that do not consider dynamic data transfer costs in terms of data-processing time. Moreover, they can adaptively adjust data localities based on the server’s free time, data placement, and network bandwidth, and schedule tens of thousands of tasks within subseconds or seconds.

Keywords: data locality; task scheduling; multicore server; load balancing; bipartite graph; big data; cloud computing

1. Introduction

Data-parallel frameworks such as MapReduce [1], Hadoop [2], Spark [3], Pregel [4], and TensorFlow [5] have emerged as important components in big data-processing ecosystems. For example, the Spark deployment at Facebook processes tens of petabytes of newly-generated data every day, and a single job can process hundreds of terabytes of data [6]. Because data-parallel frameworks process terabytes or petabytes of data on hundreds or thousands of servers, the costs of transferring data between
servers significantly affect the frameworks’ performance. Hence, data locality becomes a fundamental problem for all data-parallel frameworks.

Data locality means moving computational tasks rather than moving data to save the network bandwidth. In data-parallel frameworks such as Hadoop or Spark, data files are divided into fixed-size data blocks and stored on distributed file systems. (On the Hadoop distributed file system, a data block’s default size is 128 megabytes (MB).) For fault tolerance, each block has multiple replicas that spread over the servers. To process the file, a job is divided into data-parallel tasks (e.g., Hadoop’s map tasks and Spark’s narrow transformation tasks), each of which is assigned to an idle server and processes a data block on the server. As data blocks are distributed over servers, a task reads the input data blocks from either local memories/disks (i.e., data local) or remote servers through networks (i.e., data remote). To reduce data transfer costs, data-locality-aware task scheduling that assigns tasks close to the input data block is used widely by existing data-parallel frameworks. Although the idea is quite simple, it is non-trivial to design an effective and efficient data-locality-aware task scheduler, because it should consider the time for waiting for free servers, the time for transferring data blocks, and the data blocks’ placements. At the same time, it is required to handle the scheduling instances that contain a large number of tasks and servers in a timely fashion. Thus, an ideal data-locality-aware task scheduler is expected to minimize job completion time, adaptively adjust data locality, and timely schedule tens of thousands or more tasks.

Although researchers proposed many data-locality-aware scheduling algorithms in the past decade, it is difficult to design an ideal data-locality-aware task scheduler for multicore servers. A multicore server contains two or more identical processor cores that are connected to a single shared main memory, and has full access to all input and output devices. In a modern data center, the number of processor cores on a single server can be tens or more. For example, the Amazon Elastic Compute Cloud (EC2) r5.24xlarge instances (https://aws.amazon.com/ec2/instance-types) support up to 96 processor cores. Because of the multiple cores, multiple tasks (which compete for limited network bandwidth) can simultaneously run on the same server. Figure 1 illustrates this problem. The Hadoop default scheduler assigns two data-remote tasks to the second server. However, for an optimal assignment, it assigns only a data-remote task to the server. Its data transfer cost and makespan are much lower than the Hadoop default scheduler’s, but this leads to an onerous challenge: How can we design an effective and efficient data-locality-aware task scheduler for large-scale multicore servers?

- First, the data transfer cost on a multicore server is dynamic. It changes with the number of concurrent data-remote tasks on the server. For example, if \( k \) data-remote tasks on the same server transfer data simultaneously, the data transfer costs are almost \( k \) times larger than the data transfer cost of a single data-remote task. This significantly increases the difficulties in minimizing job completion time. Suppose a server is running \( k \) data-remote tasks. If we assign additional data-remote tasks to the server, all the tasks’ (including the existing \( k \) data-remote tasks) data transfer costs must be recomputed. To minimize job completion time, the scheduler needs more adjustment and recomputation, which significantly increases time complexity.

- Second, the scheduling instance is large. A production data-parallel system might even contain tens of thousands of multicore servers. Because each server contains tens of processor cores, the total number of task slots can be hundreds of thousands. Meanwhile, the number of data-parallel tasks is also large. For example, assuming a data block’s size is 128 MB, the scheduler needs to assign about 8000 tasks to process a 1-terabyte file. Because the data transfer cost is dynamic and the scheduling instance is large, it is difficult to design an effective scheduler that has a low scheduling latency.
Keeping this in mind, here we study a data-locality-aware task-scheduling problem, taking the dynamic data transfer costs into account. Although many have proposed data-locality-aware task-scheduling algorithms [7–12], usually researchers define data transfer costs as static values, rather than making them robustly adaptable or representative of changes in costs. Some recent research does consider dynamic data transfer costs. For instance, the Hadoop task-assignment problem [13,14] uses a non-decreasing function to evaluate data transfer costs such that the data transfer cost increases with the total number of data-remote tasks. Pandas [15] models data-remote tasks’ runtime as random variables. However, these works either set an identical cost for all data-remote tasks or assume all data-remote tasks’ runtime follows the same stochastic model, which does not fit our situation (where data transfer costs on different servers are quite different). Previous approaches also discussed both offline and online scheduling modes. First, offline scheduling uses elaborate algorithms to find high-quality assignments [8,9,13,14], but most of them have high scheduling latencies. One approach, Firmament [9], is a low-latency offline scheduling algorithm, but its sophisticated model is hard to extend to dynamic data transfer costs. Second, online scheduling [2,7] is based on a server-by-server policy. Once a server is idle, the scheduler scans the task queue to select the best task for the server. Compared to the offline scheduling algorithms, the online scheduling algorithms usually possess low scheduling latencies, but they generate suboptimal assignments. Moreover, to the best of our knowledge, dynamic data transfer costs are not considered by existing online scheduling algorithms.

To address these problems, we propose a novel task-scheduling model, called DynDL (Dynamic Data Locality), for assigning data-locality-aware tasks on multicore servers. Inspired by the Hadoop Task Assignment (HTA) model, we use a non-decreasing function $g_i(n_i)$ to evaluate the dynamic data transfer cost on server $i$, where $n_i$ is the number of data-remote tasks on server $i$. Compared to existing models, our model is more flexible, letting users define a personalized cost function $g_i(\cdot)$ for each server $i$ according to the server’s workload and network bandwidth. To quickly schedule tasks, we propose two efficient task-scheduling algorithms, called DynDLOff and DynDLOn, for the offline and online modes, respectively. DynDLOff first generates an initial task assignment that only contains

---

**Figure 1.** Data-locality-aware task scheduling (task $t_i$ processes block $i$). (a) Hadoop default scheduler’s assignment; (b) Optimal assignment.
data-local tasks, then refines the task assignment by gradually adding data-remote tasks. Based on the initial task assignment, DynDLOff can efficiently evaluate the dynamic data transfer cost for each server, so it has latencies of subseconds or seconds. DynDLOn is based on a delay-scheduling heuristic. Unlike the two-phase offline scheduling algorithm, the online scheduling algorithm assumes that the server’s free time is unknown. Once a server becomes idle, the scheduler assigns a task to the server. However, if there is no data-local task for the idle server, the scheduler delays assigning data-remote tasks to the server. By carefully controlling the delay’s duration, DynDLOn’s task assignments are as good as DynDLOff’s.

Real-world applications. DynDL can benefit a large number of big-data applications such as interactive query processing [16–18], social network analysis [19–21], and distributed machine learning [5,22] in reducing job completion time. We take the interactive query processing as an example to illustrate how the big-data applications are benefited. Interactive query processing is sensitive to the response time, because the time affects user experience, providers’ revenue, and the quality of service. However, lowering the response time is non-trivial in a big-data cluster due to the dynamic data transfer costs, servers’ free time, and scheduling latencies. DynDL reduces the response time by adaptively adjusting the data locality: if data-local servers quickly free up, it assigns more data-local tasks; otherwise, it assigns data-remote tasks to the idle servers that have enough network bandwidth. Moreover, it schedules tens of thousands of tasks within subseconds or seconds. As a result, DynDL reduces the overall response time of big-data query processing.

We summarize this paper’s key contributions as follows:

- We propose a novel data-locality-aware task-scheduling model for multicore servers, which considers the data placement, initial workloads, and dynamic data transfer costs. The model uses non-decreasing cost functions to evaluate the dynamic data transfer costs. Because the non-decreasing functions are not restricted to any specific forms, our model broadly applies to various environments.
- We also propose a two-phase offline scheduling algorithm. First, it generates an initial task assignment that only contains data-local tasks, then gradually adds data-remote tasks to reduce job completion time. We prove that the algorithm is optimal in terms of job completion time for specific uses.
- We present a delay-based online scheduling algorithm. The online algorithm controls the delay’s duration by computing the dynamic data transfer cost on the idle server. The online scheduling algorithm is faster than the offline algorithm, and the job completion time of its assignment is only 10% longer. This demonstrates the online algorithm’s effectiveness and efficiency.
- We conduct extensive experiments to evaluate DynDLOff and DynDLOn through simulations and real-world executions. Simulation results show that our algorithms offer approximately 30% improvement over algorithms that do not consider dynamic data transfer costs in terms of data-processing time. It has a latency of seconds for the scheduling instances containing 10,000 tasks and 100,000 processor cores. We build a real-life testbed on a real multicore-server-based computing cluster. Experiments illustrate how the dynamic data transfer costs affect job completion time in real executions.

The rest of this paper is organized as follows. Section 2 discusses related work. We explore the DynDL scheduling model in Section 3. Sections 4 and 5 detail the online algorithm, DynDLOn, and offline algorithm, DynDLOff. In Section 6, we evaluate our algorithms. We conclude the paper in Section 7.

2. Related Work

The data-aware task scheduler is one of the most important parts in a distributed system. Over the past decade, many data-aware task-scheduling algorithms [23] were proposed to handle data locality [7–12], heterogeneity [24–26], energy efficiency [27,28], resource provision [29,30], data-
emergency [31], workflow [32], and task preemption [33]. In this work, we focus on data-locality-aware task scheduling.

The data locality problem is fundamental to data-intensive distributed computing. Early research works proposed data-locality-aware scheduling algorithms for Data Grid, an architecture that gives people the ability to access, modify, and transfer vast amounts of geographically distributed data. Takefusa et al. [34] developed a simulator for Gfarm [35] (a data grid system), which used greedy scheduling algorithms to improve tasks’ data localities. Ranganathan et al. [36] developed four scheduling algorithms: JobRandom, JobLeastLoaded, JobDataPresent, and JobLocal for Data Grids, where JobDataPresent was a data-locality-aware scheduling algorithm that scheduled jobs close to input data. Raicu et al. [37] implemented task diffusion on Falkon [38], a fast and lightweight task-execution framework. The data diffusion architecture’s data-aware scheduler set the upper bound of the number of idle servers as a utility threshold. It skipped data-remote servers if the number of idle servers was below the utility threshold. However, the schedulers designed for Data Grids could not be applied readily to data-parallel frameworks, because the task execution models are different. Data Grid tasks are usually loosely coupled, so the scheduler’s focus is on individual tasks, and scheduling algorithms are usually greedy and simple. In data-parallel frameworks, the tasks are relatively tightly coupled, because a job cannot finish until all of its subtasks are completed. Thus, the problem of how to develop a simple yet effective data-locality-aware scheduler has garnered the big data research community’s attention.

Hadoop [2] is a classic data-parallel framework based on MapReduce abstraction. Hadoop’s default scheduling algorithm schedules jobs sequentially by a first-in, first-out policy. For the head-of-line job, the scheduler greedily searches for a data-local subtask for each idle server. If there is no data-local subtask, the scheduler randomly assigns a data-remote task. This task-scheduling algorithm is simple, so its data locality can be improved. Zaharia et al. [7] proposed delay scheduling to improve fairness and data locality in a shared cluster. Delay scheduling implemented max-min fairness [39]: according to fairness, if the job to be scheduled could not launch a data-local task, it would wait until other jobs had launched a task. It was worthwhile to wait, because delay scheduling assumed that the server became idle quickly enough. However, when servers freed up slowly, delay scheduling did not work well. Quincy [8] took an approach similar to delay scheduling. Quincy was a centralized scheduler that used min-cost max-flow (MCMF) to solve the scheduling problem, and achieved better fairness. Firmament [9] improved Quincy in terms of scheduling latency. Data-locality-aware scheduling was also applied to graph computation systems. For example, Persico et al. [21] compared the performance of two state-of-the-art big-data-analytic architectures (Kappa and Lambda) when deployed onto a public-cloud PaaS (Platform as a Service) running social network analysis applications. Our work is orthogonal to this work. Since Kappa and Lambda are Spark-based systems, our work can be applied to the systems to accelerate distributed social network analysis [20].

Some data-locality-aware approaches were designed to optimize throughputs and handle data skew. To optimize throughput, Wang et al. [10] designed a novel queueing architecture for data-parallel task scheduling by using a “join the shortest queue” (JSQ) policy and a MaxWeight policy. Xie et al. [11] found that the JSQ-MaxWeight algorithm was heavy-traffic optimal only in specific scenarios with two-level data locality. Thus, they proposed an algorithm that uses Weighted-Workload routing and priority services to optimize multilevel data locality. Data skew means that data blocks are unevenly distributed over servers, so some servers will become hot spots, which may decrease the data locality. To address this problem, Liu et al. [40] proposed an approach to mitigate data skew by adjusting tasks’ runtime resource allocation. The data skew problem is addressable by carefully placing data blocks. ActCap [41] used a Markov chain-based model to do node-capability-aware data placement for the continuously incoming data. Yu et al. [42] grouped data blocks in a few racks and assigned tasks onto these racks, which greatly decreased the number of off-switch exchanges, thereby shortening job completion time. Ma et al. [43] presented Dependency-Aware Locality for MapReduce (DALM) for processing the skewed and dependent input data. DALM accommodates data dependency in a data
We denote the core set as \( P \) and assume the cost of executing a data-local task is identical. For simplicity, let the data-local cost be \( a \) and the data-remote cost be \( b \). Given an assignment \( A \) that assigns tasks to cores. We measure an assignment’s quality by a makespan. To define the makespan, consider a processor core \( p \) and a data block on core \( s \). The task assignment strategy is a mapping function \( A : T \rightarrow P \) that assigns tasks to cores. We measure an assignment’s quality by a makespan. To define the makespan, we first define the task costs and core loads. Given an assignment \( A \), a task \( t \) is data-local if and only if \( A(t) \in DL^{loc}(t) \). Otherwise, task \( t \) is data-remote. Because data-parallel tasks are homogeneous, we assume the cost of executing a data-local task is identical. For simplicity, let the data-local cost be \( a \) and the data-remote cost be \( b \).

Some recent research works proposed sophisticated scheduling models to optimize communication costs. Selvitopi et al. [44] proposed an offline scheduling algorithm based on graph and hypergraph models, which correctly encoded the interactions between map and reduce tasks. Choi et al. [45] aimed at a problem where an input split consisted of multiple data blocks that were distributed and stored in different nodes. Beaumont et al. [46] proposed two data-locality-aware task scheduling algorithms that optimized makespan and communication, respectively, and theoretically studied their performance. Li et al. [47] proposed scheduling algorithms to optimize the locality-enhanced load balance and the map, local reduce, shuffle, and final reduce phases. Unlike the approaches that maximize data locality, our approach’s goal is to minimize the job completion time by adaptively adjusting data locality.

Overall, these task scheduling algorithms effectively minimize job completion time and maximize data locality, but few of them take the dynamic data transfer costs into account. We observe that if multiple data-remote tasks are assigned to the same server, the job completion time increases dramatically. The HTA (Hadoop Task Assignment) problem [13] considered dynamic data transfer costs by using a non-decreasing function, where the data transfer cost increases with the total number of data-remote tasks. Pandas [15] modeled data-remote tasks’ runtime as random variables. However, these approaches do not fit our scenario, where the data transfer costs on different servers are quite different. In this paper, we propose a novel data locality scheduling model with dynamic data transfer costs for multicore servers, and develop online and offline algorithms for the model. Although it seems similar to our previous work, BAR (Balance and Reduce) [14], it differs in terms of the task-scheduling model and algorithm. For instance, BAR’s scheduling model is based on the HTA problem, but DynDL and HTA are different. Additionally, BAR is an offline algorithm, and here we propose online and offline algorithms to solve DynDL. In Section 6, we compare the two approaches via extensive simulations.

3. The DynDL Scheduling Model

In this section, we focus on a scheduling problem that assigns data-parallel tasks on a multicore-server-based computing cluster. The cluster follows a shared-nothing architecture, consisting of many network-connected servers. Each server contains multiple homogenous processor cores that share the server’s storage space and network bandwidth. The data-parallel tasks are independent and data-intensive. Each task processes an input file block, and then it places the file blocks on the server’s local storage (disks or memories). Before performing data processing, tasks are assigned to an idle core, and then they read the data blocks from local storage or a remote server. The scheduling problem is to find a task assignment strategy that minimizes all the tasks’ completion time, taking into account the core’s initial loads, tasks’ running time, and data’s transfer time. We formally define the scheduling problem as follows.

The computing system is a 3-tuple \( (S, T, DL^{src}) \), where \( S \) and \( T \) are the server and task sets, respectively, and \( DL^{src} \) is a data placement function. Each server \( s \in S \) contains a set of cores \( core(s) \). We denote the core set as \( P \), such that \( P = \bigcup_{s \in S} core(s) \). Each core \( p \in P \) belongs to a unique server \( srv(p) \in S \). If task \( t \)'s data block stores on server \( s \)'s storage, we say that \( t \) prefers \( s \) (or the cores on \( s \)). The sets of servers and cores that task \( t \) prefers are denoted as \( DL^{src}(t) \) and \( DL^{core}(t) \), respectively. In real-life systems, each data block has a fixed number of replicas and each server has a fixed number of cores, so in our problem |\( DL^{src}(t) \)| and |\( core(s) \)| are constants.

**Task assignment and makespan.** The task assignment strategy is a mapping function \( A : T \rightarrow P \) that assigns tasks to cores. We measure an assignment’s quality by a makespan. To define the makespan, we first define the task costs and core loads. Given an assignment \( A \), a task \( t \) is data-local if and only if \( A(t) \in DL^{loc}(t) \). Otherwise, task \( t \) is data-remote. Because data-parallel tasks are homogeneous, we assume the cost of executing a data-local task is identical. For simplicity, let the data-local cost be \( a \) and the data-remote cost be \( b \).
be 1. Regarding data-remote tasks, because the cores on the same server share limited network bandwidth, the data-remote cost increases with the number of data-remote tasks. Let \( r_s^A \) be the number of data-remote tasks on server \( s \), then the task costs are

\[
C_A(t) = \begin{cases} 
1, & A(t) \in DLcore(t), \\
\delta_s(r_s^A), & \text{otherwise,}
\end{cases}
\]

where \( s = srv(A(t)) \) and \( \delta_s(\cdot) > 1 \) is the data-remote cost, which is a non-decreasing function of the number of remote tasks on server \( s \). The load of a core \( p \) is the time required to finish all the tasks assigned to \( p \). Given an assignment \( A \), the load of core \( p \), \( L_A(p) \), is

\[
L_A(p) = L^{init}(p) + \sum_{t: A(t) = p} C_A(t),
\]

where \( L^{init}(p) \) is the initial load on core \( p \) and \( \sum_{t: A(t) = p} C_A(t) \) is the time required to finish the tasks on \( p \). We use an initial load to model the core’s idle time. Once we determine the task costs and core loads, then we know the makespan of \( A \), the time when all the tasks are finished:

\[
makespan(A) = \max_{p \in \hat{P}_A} L_A(p).
\]

Here, \( \hat{P}_A \) is a set of active cores. A core is active if and only if at least one task is assigned to the core. Table 1 lists the frequently used notations.

**Table 1.** Frequently used notations.

| Notation | Description |
|----------|-------------|
| \( A \)  | task assignment |
| \( B \)  | balanced task assignment (see Section 4) |
| \( S, P, \) and \( T \) | server, core, and task sets |
| \( core(s) \) | set of cores that server \( s \) contains |
| \( srv(p) \) | server that contains core \( p \) |
| \( DL^s(t) \) and \( DL^p(t) \) | servers and cores that \( t \) prefers |
| \( \delta_s(n) \) | server \( s \)'s data-remote cost function; \( n \) is the number of data-remote tasks on \( s \) |
| \( L_A(p) \) | load of core \( p \) |
| \( L^{init}(p) \) | initial load of core \( p \) |
| \( f_A(t) \) | finish time of task \( t \) |
| \( r^s_A \) and \( r^p_A \) | numbers of data-remote tasks on server \( s \) (and core \( p \)) |
| \( l^s_A \) and \( l^p_A \) | numbers of data-local tasks on server \( s \) (and core \( p \)) |
| makespan(\( A \)) | makespan of task assignment \( A \) |

Based on these definitions, we define the scheduling problem.

**Scheduling problem.** Given a computing system, each server’s task cost function, and all the cores’ initial loads, the problem’s goal is to find an assignment that minimizes the makespan.

The task-scheduling problem is called an offline problem if each core’s initial load is completely known before scheduling. It is called an online problem if each core’s initial load is unknown until the core is free. The offline problem is NP-complete, because its restrict case (where all cores are idle at start time) is NP-complete [13]. Next, we present online and offline algorithms to address the task-scheduling problem.

4. DynDLOn: An Online Algorithm

Existing data-parallel frameworks such as Hadoop and Spark typically use greedy approaches to handle online scheduling. For example, Hadoop’s default scheduler tries its best to find a data-local task for each idle core. However, this heuristic and its variations such as delay scheduling [7] do not take the multicore servers into account, neglecting the fact that data-remote tasks on the same computer
compete for limited network bandwidth. In the following example, we illustrate how Hadoop’s default scheduling policy and delay scheduling work.

**Example 1.** Table 2 shows a scheduling instance that contains 4 servers, 8 cores, and 5 tasks. Each server $s$’s data-remote cost function is $g_s(n) = 1 + 0.5n$. Figure 2a,b respectively present examples of Hadoop’s and delay scheduling’s assignments. According to the initial loads, the first idle core is $p_{21}$. Both Hadoop and delay scheduling assign a data-local task to $p_{21}$. In this example, we assume the data-local task is $t_1$. The second idle core is $p_{12}$. Both Hadoop and delay scheduling assign a data-remote task to $p_{12}$. Unlike Hadoop, delay scheduling lets $p_{12}$ be idle for a short period, because there are no data-local tasks for $p_{12}$. Then, $p_{31}$ becomes the third idle core. Hadoop and delay scheduling assign a data-remote task to $t_3$ to $p_{11}$. Because $p_{12}$ and $p_{11}$ are on the same server, $t_2$’s and $t_3$’s running time increase from 1.5 to 2. The fourth idle core is $p_{21}$ again, because it has finished $t_1$. Then, Hadoop and delay scheduling assign a data-local task to $p_{21}$. The fifth idle core is $p_{22}$. Because $p_{22}$ has no data-local task, delay scheduling holds off on $p_{22}$ for a while. When $p_{22}$ is delayed, $p_{31}$ becomes idle. Then, because $t_5$ is a data-local task of $p_{31}$, delay scheduling assigns $t_5$ to $p_{31}$ and generates an assignment whose makespan is 3.25. However, Hadoop does not delay $p_{22}$, and assigns a data-remote task to $p_{22}$. It generates an assignment whose makespan is 3.5.

**Table 2.** A scheduling instance (data-remote cost function $g_s(n) = 1 + 0.5n$).

| Server | Data-Local Task | Core | Initial Load |
|--------|-----------------|------|--------------|
| $s_1$  | $t_1$           | $p_{11}$ | 0.75 |
|        |                 | $p_{12}$ | 0.5  |
| $s_2$  | $t_1, t_2, t_3, t_4$ | $p_{21}$ | 0.25 |
|        |                 | $p_{22}$ | 2    |
| $s_3$  | $t_3, t_4, t_5$ | $p_{31}$ | 2.25 |
|        |                 | $p_{32}$ | 14   |
| $s_4$  | $t_2, t_5$      | $p_{41}$ | 2.5  |
|        |                 | $p_{42}$ | 3.75 |

**Figure 2.** Assignment strategies of three online scheduling algorithms (DL indicates “data-local” and DR indicates “data-remote”). (a) Hadoop’s assignment, makespan $= 3.5$; (b) Delay scheduling’s assignment, makespan $= 3.25$; (c) DynDLOn’s assignment, makespan $= 3.25$.

From the example, we can see that the idea of delay scheduling is effective, increasing data locality and decreasing makespan. However, the original delay scheduling algorithm does not take the multicore servers into account. It assigns two data-remote tasks to server $s_1$, so the data remote costs on $s_1$ increase from 1.5 to 2, which may increase the overall makespan. To address this problem, our heuristic is to dynamically set the duration of delay according to the idle server’s data-remote cost. Figure 2c illustrates our heuristic. When $p_{11}$ is idle, the scheduler finds data-remote tasks existing on $s_1$, and sets a longer delay to $p_{11}$. Because $p_{22}$ and $p_{31}$ become idle when $p_{11}$ is delayed, the scheduler does not assign the second data-remote task to $s_1$. As a result, the heuristic reduces the number of data-remote tasks on $s_1$. 
We use this heuristic for our online algorithm DynDLOn in Algorithm 1. Each core $p$ is associated with two variables $p$.delay and $p$.wait, which indicate the time to delay and the time $p$ when waiting begins, respectively. When core $p$ is free, DynDLOn first tries to assign a data-local task to $p$. If not, DynDLOn compares $p$.wait and $p$.delay, and assigns a data-remote task if $p$.wait $\geq$ $p$.delay. To set $p$.delay, we consider two conditions. First, if server $s$ does not contain a data-remote task, $p$.delay is set to a predefined value $W$. In this case, DynDLOn rolls back to the original delay-scheduling algorithm. Second, if server $s$ contains at least one data-remote task, we prefer starting a new data-remote task after a running data-remote task completes. Because a data-remote task’s running time is $g_s(r^s)$, we set $p$.delay to $\max\{g_s(r^s), W\}$, where $r^s$ is the number of data-remote tasks on $s$.

Algorithm 1: DynDLOn.

**Input:** Server $s$, Core $p$, Unscheduled Tasks $T$, Data Placement $DL^{src}$

**Output:** Task assigned to $p$

// $p$ is an idle core on server $s$. $p$.wait is the time $p$ has waited.

1. foreach $t \in T$ do
2.   if $s$ is in $DL^{src}(t)$ then
3.     $p$.wait $\leftarrow$ currentTime
4.     return $t$
// $r^s$ is the number of data-remote tasks on server $s$.
5. if $r^s == 0$ then
6.   $p$.delay $\leftarrow W$
7. else
8.   $p$.delay $\leftarrow \max\{g_s(r^s), W\}$
9. wait $\leftarrow$ currentTime $- p$.wait
10. if $p$.wait $\geq$ $p$.delay then
11.   $p$.wait $\leftarrow$ currentTime
12. return randomtask($T$)
13. return NULL

The astute reader might feel that setting a longer duration of delay will decrease the system’s utility, because the cores are idle while waiting. This is rare, though—it only happens when the system runs a few jobs. When there are more jobs, the idle core launches a task from another job to avoid decreasing the system’s utility. Although we focus here on single-job systems, DynDLOn easily extends to multijob systems. In Section 6.2, we show DynDLOn’s performance on multijob systems in real-world executions.

5. DynDLOff: An Offline Algorithm

To complement our online algorithm, we also present an offline algorithm that knows the cores’ initial loads. The offline algorithm DynDLOff contains two phases. Phase I produces an initial assignment where all tasks are data-local and the cores are load balancing, and Phase II refines the initial assignment to lower the overall makespan by gradually increasing the number of data-remote tasks. By using the initial assignment, Phase II easily distinguishes data-remote and data-local tasks. It sets a few deadlines; generates a series of assignments, and selects the best assignment as the algorithm’s output. Because our algorithm computes the final assignment incrementally, it efficiently and effectively solves the scheduling problem. In the following, we present the phases in detail.
5.1. Phase I: Assign Data-Local Tasks

This phase generates a balanced assignment that only contains data-local tasks and evenly distributes data-local tasks across the cores. The balanced assignment, denoted as $B$, satisfies the following constraints.

- Constraint 1: all tasks are assigned to their preferred cores, such that
  $$\forall t \in T, B(t) \in DL_{\text{core}}(t).$$

  In the following, we refer to the assignment strategies that satisfy Constraint 1 as data-local assignments.

- Constraint 2: among the cores that task $t$ prefers, $B(t)$’s load is the smallest, such that
  $$\forall t \in T, L_B(B(t)) - 1 \leq \min_{p \in DL_{\text{core}}(t) \backslash B(t)} L_B(p).$$

**Property 1.** A balanced assignment $B$ has the following properties

1. For cores $p_i$ and $p_j$, if $L_B(p_i) - 1 > L_B(p_j)$, the data-local tasks on $p_i$ do not prefer $p_j$.
2. $B$ is optimal among all the data-local assignments in terms of the makespan.

**Proof.** Property 1 is correct because of Constraint 2. Property 2’s correctness is proved as follows. Assume data-local assignment $B'$ is optimal and does not satisfy Constraint 2. Without loss of generality, let task $t$ be the task that dissatisfies Constraint 2. Let $p^t$ be a core that holds $L_B(B'(t)) - 1 > L_B(p^t)$. By moving $t$ to core $p^t$, we obtain another data-local assignment $B''$ that holds $\text{makespan}(B'') \leq \text{makespan}(B')$. We continue the motions until Constraint 2 is satisfied, and then generate $B$. Because $B'$ is optimal and $\text{makespan}(B) \leq \cdots \leq \text{makespan}(B'') \leq \text{makespan}(B')$, it is easy to discern that $B$ is also optimal. □

To find a balanced assignment, we base Phase I on two steps. The first step (lines 1–2 of Algorithm 2) uses a greedy approach to assign all the tasks to their preferred cores. For each task $t$, the algorithm chooses the least-loaded server $s$ from $DL_{\text{srv}}(t)$, then assigns $t$ to the least-loaded core of $\text{core}(s)$. The data-local assignment $B_0$ generated by this step is referred to as an initial assignment. Figure 3a shows the initial assignment for Table 2’s scheduling instance. The second step (lines 3–17 of Algorithm 2) continuously moves tasks from high-load to low-load cores until the task assignment satisfies Constraint 2. Specifically, this step achieves the goal by finding cost-reducing paths [48] on a bipartite graph.
Algorithm 2: DynDLOff: Assign Data-Local Tasks (Phase 1).

**Input:** Servers $S$, Cores $P$, Tasks $T$, Data Placement $DL^{\text{src}}$

**Output:** Balanced Assignment $B$

// Generate a data-local assignment $B_0$

1. foreach $t \in T$ do
   2. $B_0(t) \leftarrow \text{argmin}_{p \in DL^{\text{src}}(t)} \{L^{\text{init}}(p) + t^p_s\}$

// Build a bipartite graph and eliminate cost-reducing paths

3. Build a bipartite graph $G$ according to $B_0$
4. Compute $MAX^{\text{src}}_B(s)$ and $MIN^{\text{src}}_B(s)$ for each server $s$
5. $DONE \leftarrow \emptyset$
6. $i \leftarrow 0$

7. while $|DONE| \neq |S|$ do
   8. $s_{\text{max}} \leftarrow \max_{s \in S \setminus DONE} MAX^{\text{src}}_B(s)$
   9. $S_{\text{visited}} \leftarrow \text{DepthFirstSearch}(G, s_{\text{max}})$
   10. $s_{\text{min}} \leftarrow$ the minimal load server in $S_{\text{visited}}$
   11. if $MAX^{\text{src}}_{B_i}(s_{\text{max}}) - MIN^{\text{src}}_{B_i}(s_{\text{min}}) > 1$ then
      12. Reverse the directions of $arc(s_{\text{max}}, t_1), \ldots, arc(t_i, s_{\text{min}})$
      13. Update $MAX^{\text{src}}_{B_{i+1}}(s)$ and $MIN^{\text{src}}_{B_{i+1}}(s)$ for each server $s$
      14. $i \leftarrow i + 1$
   15. else
   16. $DONE \leftarrow DONE \cup \{s_{\text{max}}\}$
17. Generate a balanced assignment $B$ based on the bipartite graph
18. return $B$

The bipartite graph contains a task vertex set $VT$ and a server vertex set $VS$, such that for each task $t$ there is a $v(t) \in VT$ in the bipartite graph. Moreover, for each server $s \in \bigcup_{t \in T} DL^{\text{src}}(t)$, there is a $v(s) \in VS$. The data placement strategy decides how the task vertices and server vertices are connected. For each $s \in DL^{\text{src}}(t)$, if $s = B_0(t)$, then there is a directional edge $arc(s, t)$ connecting two
vertices \( v(s) \) and \( v(t) \). Otherwise, edge \( arc(t, s) \) connects \( v(t) \) and \( v(s) \). Figure 3b shows a bipartite graph that corresponds to the initial assignment \( B_0 \).

On the bipartite graph, we define an alternating path to be a sequence of edges \( Path = \{ arc(s_1, t_1), arc(t_1, s_2), arc(s_2, t_2), \ldots, arc(t_{k-1}, s_k) \} \), with \( v(t_i) \in TV \) and \( v(s_i) \in SV \) for each \( i \). The cost-reducing path is a special case of an alternating path, holding that \( \text{MAX}_{B_i}^{\text{srv}}(s_1) - \text{MIN}_{B_i}^{\text{srv}}(s_k) > 1 \), where

\[
\text{MAX}_{B_i}^{\text{srv}}(s) = \max_{p \in \text{core}(s)} L_{B_i}(p), \\
\text{MIN}_{B_i}^{\text{srv}}(s) = \min_{p \in \text{core}(s)} L_{B_i}(p).
\]

In Figure 3b, the shadowed path from \( s_4 \) to \( s_1 \) is a cost-reducing path, because \( \text{MIN}_{B_0}^{\text{srv}}(s_1) = L_{B_0}(p_{12}) = 0.5 \) and \( \text{MAX}_{B_0}^{\text{srv}}(s_4) = L_{B_0}(p_{41}) = 3.5 \). Having a cost-reducing path, we flip the direction of each edge on the path to produce a new task assignment. Figure 3c shows a new bipartite graph after we flip the edge directions. After reversing edge directions, the algorithm produces a better task assignment \( B_{i+1} \) such that \( \text{makespan}(B_{i+1}) \geq \text{makespan}(B_i) \). The bipartite graph that Figure 3c shows corresponds to a task assignment, where \( t_5, t_3, \) and \( t_1 \) are assigned to \( s_3, s_2, \) and \( s_1 \), respectively. The new task assignment’s makespan is 3.25.

The second step continuously finds the cost-reducing paths to improve the task assignment until no cost-reducing path can be found. To detect a cost-reducing path, we first select a max-load server \( s_{\text{max}} \) and then perform a depth-first search starting from \( v(s_{\text{max}}) \) to traverse the bipartite graph. Among all the visited server vertices, the server vertex \( v(s_{\text{min}}) \) is selected as the path’s end, where \( s_{\text{min}} \) is the least-load server. If \( \text{MAX}_{B_i}^{\text{srv}}(s_{\text{max}}) - \text{MIN}_{B_i}^{\text{srv}}(s_{\text{min}}) > 1 \), we detect a cost-reducing path. Otherwise, we mark \( v(s_{\text{max}}) \) as DONE, and select the next max-load server as the start. The algorithm iteratively detects the cost-reducing paths until all the computer vertices are marked as DONE, and then it outputs the balanced assignment \( B \). Figure 3d shows the balanced assignment.

The task assignment \( B \) satisfies Constraints 1 and 2. Because the tasks are assigned to their preferred computers, \( B \) satisfies Constraint 1. For Constraint 2, we prove the correctness by contradiction. Assume \( B \) contains task \( t \) that dissatisfies Constraint 2. There must be a cost-reducing path passing through \( v(t) \), so \( B \) cannot be our algorithm’s output, which contradicts the assumption. Thus, \( B \) is a balanced assignment that satisfies Constraints 1 and 2.

### 5.2. Phase II: Assign Data-Remote Tasks

Phase II assigns data-remote tasks to reduce the balanced assignment’s makespan. As we mentioned, a data-remote task’s cost is dynamic, changing with the number of data-remote tasks on each server, so it is challenging to compute the cores’ loads. To address this challenge, Phase II utilizes the balanced assignment’s Property 1 to identify the data-remote tasks.

Figure 4 and Algorithm 3 show the basic idea of Phase II. Given several deadlines, the algorithm generates a series of assignments. For each deadline \( D \), it tests whether there exists a new assignment \( A \) with \( \text{makespan}(A) \leq D \). To perform the test, it divides the task set into two subsets, \( T_{\text{loc}} \) and \( T_{\text{rem}} \), according to the task finish time based on \( B \), such that \( T_{\text{loc}} = \{ t | t \in T \land F_B(t) \leq D \} \) and \( T_{\text{rem}} = \{ t | t \in T \land F_B(t) > D \} \), where \( F_B(t) \) is task \( t \)’s finish time (given the initial assignment \( B \) and a task \( t \)’s suppose \( t \) is the \( k \)th task on \( B(t) \), then \( F_B(t) = L_{\text{init}}(B(t)) + k \). To lower the makespan, each task in \( T_{\text{loc}} \) is assigned to \( B(t) \) and the tasks in \( T_{\text{rem}} \) are moved to cores whose loads are smaller than \( D \). According to the balanced assignment’s Property 1, we can conclude that the tasks in \( T_{\text{rem}} \) must be data-remote. Then, the algorithm checks \( \sum_{s \in S} t_{\text{max}}^s \geq |T_{\text{rem}}| \) to determine whether a task passes the test, where \( t_{\text{max}}^s \) is the maximum number of server \( s \)’s data-remote tasks that are completable before \( D \). The algorithm continuously checks different deadlines until a task fails the test. Then, it derives the final number of data-remote tasks and outputs a final assignment. Next, we illustrate how to compute deadlines and generate final assignments.
Algorithm 3: DynDLOff: Assign Data-Remote Tasks (Phase II).

**Input:** Balanced Assignment $\mathcal{B}$, Servers $S$, Cores $P$, Tasks $T$, Matrix $R$

**Output:** Final Assignment

1. $T \leftarrow \text{SortTasks}(\mathcal{B}, T, P)$
2. for $i \leftarrow 0$ to $|T|$ do
3. \hspace{1em} $D[i] \leftarrow F_B(T[i])$
   \hspace{1em} // Array tests stores the result of each test.
   \hspace{1em} // tests[i] contains the $r^s_{\text{max}}$ for each server $s$ for deadline $D[i]$.
4. $\text{index} \leftarrow \text{ComputeFinalDeadline}(0, |T|, D, \text{tests}, R)$
5. $\mathcal{A}_1 \leftarrow \text{ComputeFinalAssignment}(\mathcal{B}, T, D[\text{index}], \text{index}, \text{tests}[\text{index}])$
6. $\mathcal{A}_2 \leftarrow \text{ComputeFinalAssignment}(\mathcal{B}, T, D[\text{index} + 1], \text{index} + 1, \text{tests}[\text{index} + 1])$
7. return $\text{BestAssignment}(\mathcal{A}_1, \mathcal{A}_2)$

Computing proper deadline $D$. Given a balanced assignment $\mathcal{B}$ and initial load $L^{\text{init}}(p)$ for each core $p$, this step computes a proper deadline $D$. Based on $\mathcal{B}$ and $L^{\text{init}}$, we first sort the all the tasks by their finish time $F_B(t)$ in descending order such that $F_B(t_1) \geq F_B(t_2) \geq \cdots \geq F_B(t_{|T|})$, and then compute an array $D$ containing $|T|$ deadlines such that $D[i] = F_B(t_{i+1})$. Because we base the deadlines on the task finish time, deadline $D[i]$ implies there are $i$ data-remote tasks. Our goal is to find an index $k$ such that $\forall i \leq k$, deadline $D[i]$ passes the test and $\forall i > k$, deadline $D[i]$ fails the test. We find $k$ by performing a binary search on $D$, and then set $D[k]$.

To perform a test for $D[i]$, we compute $r^s_{\text{max}}$ for each server, then check whether $\sum_{s \in S} r^s \geq i$. Formally, given deadline $D$, balanced assignment $\mathcal{B}$, server $s$, and load $L_B(p)$ for each core $p \in \text{core}(s)$, this step is to compute the maximum number of data-remote tasks (i.e., $r^s_{\text{max}}$) that can be finished before $D$. To compute $r^s_{\text{max}}$, our algorithm checks all the possible values of $r^s_{\text{max}}$ and selects the largest value. Suppose that a possible value is $r^s_i$. Because the number of data-remote tasks is determined, we can compute the data-remote cost (i.e., $g(s_i)$). Let core’($s$) be computer s’s cores whose loads are smaller than $D$. The core can finish $r^s_i$ data-remote tasks before $D$ if and only if $\sum_{p \in \text{core’}(s)} \left\lfloor \frac{D - L_B(p)}{\bar{g}(r^s_i)} \right\rfloor > r^s_i$.

To check all the possible values, a naive way is to set the number of data-remote tasks to $1, 2, \cdots, |T_{\text{rem}}|$ in turn. However, it is time-consuming because it needs to perform $O(|T_{\text{rem}}|)$ tests. Our algorithm improves the complexity by exploring the upper and lower bounds of $r^s_{\text{max}}$, which only requires $O(\log |T_{\text{rem}}|)$ tests.

Theorem 1. Let $\Delta L_B(s, D) = \sum_{p \in \text{core’}(s)} \{D - L_B(p)\}$ be the remaining capacity of server $s$. If $\gamma$ satisfies $\gamma \cdot g(\gamma) \leq \Delta L_B(s, D) < (\gamma + 1) \cdot g(\gamma + 1)$, then $\gamma \geq r^s_{\text{max}} \geq \max\{\gamma - \text{core’}(c), 0\}$.

Proof. We prove $\gamma \geq r^s_{\text{max}}$ by contradiction. If $\gamma < r^s_{\text{max}}$, then $\Delta L_B(c, D) \geq r^s \cdot g(r^s_{\text{max}}) \geq (\gamma + 1) \cdot g(\gamma + 1)$. This contradicts the assumption. Thus, $\gamma$ is the upper bound of $r^s_{\text{max}}$. Figure 4. Key steps of Phase II (the assignment is the best when $|T_{\text{rem}}|$ is 1).
Next, we prove $r_{\text{max}}^* \geq \max\{\gamma - |\text{core}'(s)|, 0\}$. We derive that $r_{\text{max}}^* = \sum_{p \in \text{core}'(s)} \left\lfloor \frac{D - L_g(p)}{g(r_{\text{max}}^*)} \right\rfloor \geq \sum_{p \in \text{core}'(s)} \left\lfloor \frac{D - L_g(p)}{g(\gamma)} \right\rfloor \geq \sum_{p \in \text{core}'(s)} \left\lfloor \frac{D - L_g(s,D)}{g(\gamma)} - 1 \right\rfloor = \frac{\Delta L_g(s,D)}{g(\gamma)} - |\text{core}'(s)| \geq \gamma - |\text{core}'(s)|$. Because $r_{\text{max}}^* \geq 0$, we have $r_{\text{max}}^* \geq \max\{\gamma - |\text{core}'(s)|, 0\}$. □

To compute $\gamma$ (i.e., the upper bound of $r^*$), we precompute a matrix $R$ such that $R[s][i] = i \cdot g_s(i)$ for $i = 1, 2, \ldots, |T|$. We then use a binary search to find $R[s][\gamma] \leq \Delta L_g(s,D) < R[s][\gamma + 1]$. Once $\gamma$ is found, we test at most $|\text{core}'(p)|$ values. Overall, this step requires $O(\log |T_{\text{rem}}|)$ tests, and its time complexity is $O(\log |T|)$.

**Computing final assignment.** Let the assignment corresponding to $D[i]$ be $A_i$. We can conclude $\forall i \leq j \leq k$, $\text{makespan}(A_i) \geq \text{makespan}(A_j) \geq \text{makespan}(A_k)$ and $\forall k + 1 \leq i \leq j$, $\text{makespan}(A_{k+1}) \leq \text{makespan}(A_i) \leq \text{makespan}(A_k)$. Thus, there are either $k + 1$ data-remote tasks in the final assignment. Although it is easy to compute $\text{makespan}(A_k)$ (i.e., $D[k]$), computing $\text{makespan}(A_{k+1})$ is non-trivial, because $\text{makespan}(A_{k+1})$ does not equal $D[k + 1]$ and we need to find an assignment with the smallest makespan.

Our solution contains two steps. First, we compute the total number of data-remote tasks that are completable before $D[k + 1]$. Suppose server $s$ can finish $r^p$ data-remote tasks before $D[k + 1]$, then in total, $\sum_{s \in S} r^s$ data-remote tasks are completable before $D[k + 1]$. Second, we assign the remaining $(k + 1) - \sum_{s \in S} r^s$ data-remote tasks by using a max-min scheduling policy.

The max-min scheduling policy works iteratively. At each iteration, it assigns a data-remote task to a core whose future load is minimum. Let $r^p$ be the number of data-remote tasks that have been assigned to core $p$, and $A'$ be an assignment that assigns an additional data-remote task to $p$. Then, core $p$’s future load $L_{A'}^\text{core}(p)$ is as follows:

$$L_{A'}^\text{core}(p) = L_{A'}^\text{init}(p) + l_{A'}^p + (r^p + 1) \cdot g_s(r_{\text{future}}), \quad (4)$$

where $s = \text{srv}(p)$, $r_{\text{future}} = \sum_{p' \in S} r^{p'}$. For efficiency, we compute the future load of server $s$, that is, $L_{A'}^\text{core}(s) = \min_{p \in \text{core}(s)} L_{A'}^\text{core}(p)$, to instead $L_{A'}^\text{core}(p)$. By using a min heap, selecting a minimum future-load server takes $O(1)$, assigning a task to a minimum future-load core takes $O(1)$, and after updating the core’s and server’s loads, adjusting the min heap takes $O(\log |S|)$.

### 5.3. Time Complexity and Optimality

Here, we analyze DynDLOff’s time complexity and optimality. DynDLOff assigns tasks in polynomial running time. Although it is an approximate algorithm, DynDLOff can find optimal assignments in certain specific instances. We first show the time complexities of DynDLOff’s two phases.

**Theorem 2.** Phase I’s time complexity is $O(|T|^2)$.

**Proof.** The total time required to finish the first step is $O(|T|)$, because every task is examined exactly once, and $|D_{\text{srv}}(t)|$ and $|\text{core}(s)|$ are constants. The task assignment produced by the greedy approach satisfies Constraint 1 and roughly balances the cores’ loads. Hence, the second step is to adjust the assignment to reach the optimal makespan. The total time required to finish the second step is $O(|T|^2)$. First, detecting a cost-reducing path costs $O(|T|)$, because there are at most $k|T|$ edges, where $k$ is the maximum number of servers that a task prefers. Each cost-reducing path moves one unit of the load from the max-load to the min-load server. Because there are $O(|T|)$ units of loads, the number of iterations for detecting cost-reducing paths is $O(|T|)$. Overall, the second step’s time complexity is $O(|T|^2)$. Combined with the first step’s time, the time required to finish Phase I is $O(|T|^2)$. □

**Theorem 3.** Phase II’s time complexity is $O(|T| \log |T| + |S| \log^2 |T| + |T| \log |S|)$.
Proof. In Phase II, initializing \( T \) and \( D \) takes \( O(|T| \log |T|) \). To find a proper deadline, it tests \( O(\log |T|) \) deadlines. For each deadline, it takes \( O(|S| \log |T|) \) to determine whether the deadline passes the test, so it takes \( O(|S| \log^2 |T|) \). To compute a final assignment, it uses a min heap to maintain the servers’ loads. Building a min heap takes \( O(|S|) \), updating a min heap takes \( O(\log |S|) \), and there are \( O(|T|) \) updates, so it takes \( O(|S| + |T| \log |S|) \) to compute a final assignment. Overall, Phase II’s time complexity is \( O(|T| \log |T| + |S| \log^2 |T| + |T| \log |S|) \). □

Combining the two phases’ time complexities, DynDLOff’s time complexity is \( O(|S| \log^2 |T| + |T| \log |S| + |T|^2) \).

Next, we prove that DynDLOff is optimal when the maximum load core only contains data-local tasks. Let \( O \) be the optimal assignment, \( A_i \) be an assignment generated by DynDLOff that contains \( i \) data-remote tasks, and \( A_k \) be DynDLOff’s output. We conclude \( \forall i \leq j \leq k, makespan(A_i) \geq makespan(A_j) \), and \( \forall k \leq i \leq j, makespan(A_k) \leq makespan(A_i) \). Suppose \( O \) contains \( n \) data-remote tasks. We have \( makespan(A_n) \geq makespan(A_k) \geq makespan(O) \).

**Theorem 4.** For the maximum load core \( p \) of assignment \( A_n \) such that \( L_{A_n}^\text{core}(p) = makespan(A_n) \), if \( p \) only executes data-local tasks, then \( A_n \) is optimal.

Proof. Given a balanced assignment \( B \), let \( \langle p_1, p_2, \ldots, p_{|S|} \rangle \) be a sorted list such that \( L_B^\text{core}(p_i) > L_B^\text{core}(p_{i+1}) \). Suppose that the maximum load core of \( A_n \) is \( p_u \) (i.e., the \( u \)th core in the list).

For sake of contradiction, assume that \( \forall p_i \in P, L_{A_k}^\text{core}(p_i) > L_{A_n}^\text{core}(p_i) \). We divide the cores into two groups: \( P_u^- = \{ p_i | 1 \leq i \leq u \} \), \( P_u^+ = \{ p_i | u < i \leq |T| \} \). Let \( I_{u^-}(A) \) and \( I_{u^+}(A) \) denote the numbers of data-local tasks that \( A \) assigns to \( P_u^- \) and \( P_u^+ \), respectively. According to Phase II of DynDLOff, \( \forall p_i \in P_u^-, L_{A_k}^\text{core}(p_i) + 1 > L_{A_n}^\text{core}(p_u) \). Thus, we have \( \forall p_i \in P_u^-, L_{O}^\text{core}(p_i) \leq L_{A_n}^\text{core}(p_u) \). Then, \( I_{u^-}(B) - I_{u^-}(O) > I_{u^+}(B) - I_{u^-}(A_n) = n \). This means that \( O \) contains more than \( n \) data-remote tasks, which contradicts the assumption. Therefore, \( A_n \) is optimal. □

6. Evaluation

This section evaluates DynDLOn’s and DynDLOff’s effectiveness and efficiency by comparing them with state-of-the-art data-locality-aware task-scheduling algorithms. We used both simulation and real execution in our experiments (source codes are available online: https://github.com/jujuhoo/dyndl). The simulations studied DynDLOn’s and DynDLOff’s impact on makespan, data locality, and running time. The real executions studied the impact of dynamic data transfer costs on job completion time.

6.1. Simulations

In the simulations, we compared our algorithms with the state-of-the-art data-locality-aware scheduling algorithms in terms of job completion time, data locality, and algorithm running time.

6.1.1. Settings

We implemented the simulations using Java on a PC with an Intel Core i7-8700 CPU at 3.20-gigahertz (GHz) and 16-gigabyte (GB) memory (Intel Corporation, Santa Clara, CA, USA). In the following, we describe the algorithms, datasets, and benchmark measures used in the simulations.

**Algorithms.** We compared DynDLOn and DynDLOff with four state-of-the-art data-locality-aware task-scheduling algorithms. Among the algorithms, Hadoop, DELAY, and DynDLOn are online algorithms, which make scheduling decisions when a core is free, while list scheduling (LIST), Local-Tasks-First Priority Algorithm (LTFPA), HTA, BAR, and DynDLOff are offline algorithms, which assume that all cores’ initial loads are already known.

- Hadoop [2] is the default task-scheduling algorithm used by Hadoop. When a server is free, the algorithm chooses a data-local task, then assigns the task to the server. If there is no feasible task, then the algorithm selects a random data-remote task.
• DELAY [7] offers a variation on delay scheduling. The algorithm predefines a fixed delay threshold. If a server is free and there is no data-local task for the server, the algorithm skips the server. It will not assign data-remote tasks to the server until the delay exceeds the delay threshold. In the simulations, the delay threshold is set to 3.

• LIST is a variant of the classic list scheduling algorithm [49]. LIST first sorts the tasks by IDs, and then assigns them to the servers with the predicted earliest finish time so far. To compute the predicted finish time, LIST computes each server’s load based on the numbers of data-local tasks and data-remote tasks that have been assigned to the server. Besides, LIST considers the dynamic data transfer cost on each server.

• LTFPA [50] is a simple local-tasks-first priority algorithm used by Pandas [15]. The algorithm maintains a data-local queue $Q_m$ for each server $m$. When server $m$ becomes idle, the algorithm sends the head-of-line task from $Q_m$. When server $m$ becomes idle and $Q_m$ is empty, the scheduler sends a remote task to server $m$ from the longest queue in the system, if the length of the longest queue exceeds the threshold. Theoretically, the algorithm is throughput-optimal and heavy-traffic-optimal for all traffic scenarios. LTFPA considers the dynamic data transfer costs by modeling data-remote tasks’ runtime as random variables.

• HTA is the first algorithm designed for solving the Hadoop task-assignment problem [13]. It uses a non-decreasing function to model the dynamic data transfer costs. Unlike DynDL, its data transfer costs change with the total number of servers’ data-remote tasks.

• BAR [14] is a faster algorithm for solving the HTA problem, and also considers the dynamic data transfer costs.

Datasets and parameters. We implemented a workload generator to generate the initial loads and data placement for each simulation. The initial loads were generated as follows. For a core whose ID was $m (0 \leq m < |P|)$, we mapped it to the server whose ID was $\lfloor m/\text{core\_num} \rfloor$, where $\text{core\_num}$ is the number of cores on a server. The initial load of core $m$ was randomly chosen from the range $[0, f_{\text{init}}(m)]$, where

$$f_{\text{init}}(m) = \frac{\alpha \cdot m}{\text{core\_num}} + \beta$$

Here, $\alpha$ is a load skewness factor. When $\alpha = 0$, all the cores’ initial loads were chosen from $[0, \beta]$. When $\alpha > 0$, the cores with smaller IDs had lower initial loads. In the dataset, the initial loads are represented by a series of key–value pairs like “core1_load1; core2_load2; ...”.

For the data placements, the generator randomly selected $k$ servers from a discrete uniform distribution on the interval $[0, |S|)$ for each task, where $k$ is the number of data replicas. By default, $k$ was set to 3. We assumed the task was data-local if it was assigned to the selected servers. In the dataset, the data placements are represented by a series of key–value pairs like “task1_server1; task1_server12; task1_server13; task2_server21; ...”.

For simplicity, all the servers followed the same data-remote cost function, which was

$$g(n) = 1 + \theta \cdot \min\{n, \text{core\_num}\}.$$  

Here, $\theta$ is a network factor. A larger $\theta$ indicates that the network congestion is more severe. Because most data-parallel frameworks run at most $\text{core\_num}$ tasks concurrently on a server, we bound the number of concurrent data-remote tasks $n$ to $\text{core\_num}$.

In the simulations, the number of data replicas was set to 3. Other key parameters’ descriptions and default values are shown in Table 3.

Benchmark measures. In the simulations, we evaluated the scheduling algorithms’ effects on makespan and data locality. The makespan and data locality were measured by the tasks’ latest finish time and the number of data-remote tasks, respectively. We also computed the algorithms’ running times to measure scalability.
Table 3. Key parameters.

| Parameter | Description          | Default Value |
|-----------|----------------------|---------------|
| | Number of tasks | 100 |
| | Number of servers | 50 |
| core_num | Number of cores in a server | 40 |
| α | Load skewness factor | 0 |
| β | Range of initial load | 100 |
| θ | Network factor | 1 |

6.1.2. Simulation Results

In this section, we changed the parameters initial load, load skewness, network conditions, and the number of tasks in order to evaluate the scheduling algorithms’ effects on makespan and data locality. We also changed the number of tasks, servers, and cores to evaluate the offline algorithms’ scalability.

Effects of initial loads. In these simulations, we set $|T|$ to 100, $|S|$ to 50, $\text{core}_{\text{num}}$ to 40, $\alpha$ to 0, $\theta$ to 1, and changed the ranges of initial loads $[0, \beta]$. A larger $\beta$ indicates that fewer cores will be free soon, so the makespan will be longer. Figure 5a shows how the initial loads affect the makespans. When $\beta$ reached 100, 1000, and 10,000, the makespans computed by DynDLOn were 4, 15, and 78, respectively, and the makespans computed by DynDLOff were 3, 13, and 73, respectively. We observed that when $\beta$ was small ($\beta = 100$ or $\beta = 1000$), DynDLOff’s makespan was at most 20% lower than other offline algorithms'. However, when $\beta$ was large ($\beta = 10,000$), DynDLOff’s makespans were at least 30% lower than other offline algorithms'. For all $\beta$ settings, in terms of makespans, DynDLOn was slightly better than other online algorithms. Regarding data locality, a data-local task must wait for its preferred cores to be free. When $\beta$ is large, the waiting costs cannot be ignored, so for a larger $\beta$, the schedulers assign more data-remote tasks. Figure 5b shows that DynDLOn’s data locality was better than other online algorithms’. We also can see that BAR’s data locality was better than other offline algorithms’, because BAR’s data-remote costs increased with the total number of data-remote tasks. However, BAR’s makespan was twice as large as DynDLOff’s when $\beta = 10,000$. Because BAR’s data-remote cost function overestimated these costs, it had to assign more data-local tasks to high-load servers. Thus, although DynDLOff’s data locality was worse than BAR’s, its makespan was better.

![Figure 5a](image1.png)  
(a) Makespan  

![Figure 5b](image2.png)  
(b) Data locality  

**Figure 5.** Effects of initial loads. LIST: list scheduling; LTFPA: Local-Tasks-First Priority Algorithm; HTA: Algorithm for Hadoop Task Assignment Problem.

Effects of load skewness. In these simulations, we set $|T|$ to 100, $|S|$ to 50, $\text{core}_{\text{num}}$ to 40, $\beta$ to 100, $\theta$ to 1, and changed the load-skewness factor $\alpha$ to 10, 20, and 40. A larger $\alpha$ indicates that the servers’ initial loads are more imbalanced (i.e., some servers’ loads are much smaller than others), so data-remote tasks are more likely to be assigned to the same low-load server. Figure 6 shows how load skewness affected makespans and data localities. DynDLOn’s and DynDLOff’s makespans were far smaller than other algorithms’ makespans when the initial loads were skewed. Although DynDLOn’s and DELAY’s data localities were similar, DynDLOn’s makespan was 30% smaller.
than DELAY’s. This is because DELAY assigned more data-remote tasks to the same servers than DynDLOn did. When \( \alpha \) was larger, DynDLOff assigned more data-remote tasks. Because we set \( \beta \) to 100, a larger \( \alpha \) led to larger initial loads. DynDLOff does not wait for data-local cores to be free, so although it assigned more data-remote tasks, its makespan was better.

![Figure 6. Effects of load skewness.](image)

**Effects of network conditions.** In these simulations, we set \( |T| \) to 100, \( |S| \) to 50, \( core\_num \) to 40, \( \beta \) to 1000, and \( \alpha \) to 0, and changed the network factor \( \theta \) to 1, 2, and 4. A larger \( \theta \) indicates the network conditions are worse, so a data-remote task needs more time to transfer data. Figure 7 shows how the network conditions affected the makespans and data localities. Because Hadoop and DELAY are not sensitive to changes in network conditions, their makespans increased significantly when the network factor changed from 1 to 4. DynDLOn increases the delay time to decrease the chance of assigning multiple data-remote tasks to the same server, so its makespan increased slightly. From Figure 7b, DynDLOff’s data locality was worse than other offline algorithms’. This is because the initial loads were chosen from [0, 1000]. DynDLOff assigns more data-remote tasks to reduce the negative effects of the large initial loads. When we performed simulations that set \( \beta \) to 100, all the algorithms (except Hadoop) made most tasks data-local.

**Effects of network conditions.** In these simulations, we set \( |T| \) to 100, \( |S| \) to 50, \( core\_num \) to 40, \( \beta \) to 1000, and \( \alpha \) to 0, and changed the number of tasks \( |T| \) to 200, 2000, and 20,000. Because there were 2000

![Figure 7. Effects of network conditions.](image)
cores \((\text{core\_num} \cdot |S|)\), the ratios of the number of tasks to the number of cores were 1:10, 1:1, and 10:1. Figure 8 shows how the number of tasks affected the makespans and data localities. When \(|T|\) was 200 and 2000, DynDLOn and DynDLOff were 20% and 10% better than other algorithms in terms of the makespan, respectively. When \(|T|\) was 20,000, HTA took more than half an hour, so we did not mark HTA’s makespan and data locality as “timeout”. Except for LIST, we found that all the algorithms’ makespans were close, and most algorithms did not assign data-remote tasks. This is mostly because the number of tasks was 10 times larger than the number of cores. In this case, the algorithms more easily selected data-local tasks, so they performed similarly.

![Figure 8](image.png)

**Figure 8.** Effects of number of tasks.

**Algorithm running time.** In these simulations, the default settings were \(|S| = 1000, \text{core\_num} = 10, \ t = 10,000, \ β = 1000, \ α = 0, \) and \(θ = 1.\) We changed the number of tasks \(|T|\), the number of cores in each server \(c\), and the number of servers \(|S|\), and measured the offline algorithms’ running time, as can be seen in Figure 9. When \(|T|\) was changed from 500 to 3500, the running time of LTFPA, BAR, and DynDLOff changed from 15.11 to 18.03 milliseconds (ms), from 52.12 to 302.45 ms, and from 39.22 to 124.21 ms, respectively. Although the running time of DynDLOff was longer than that of LTFPA in Figure 9a, we show that DynDLOff was faster when the number of servers increased. When \(c\) was changed from 10 to 70, the running time of LTFPA and BAR was stable, but that of DynDLOff increased from 49.21 to 107.21 ms. This is because DynDLOff took the multicore-based server into account. However, because most servers had less than 100 cores, the upper bound of \(c\) was a constant and the running time of DynDLOff was bounded. When \(|S|\) was changed from 100 to 1000, the running time of LTFPA, BAR, and DynDLOff changed from 9.76 to 655.76 ms, from 229.11 to 1010.89 ms, and from 83.56 to 341.67 ms, respectively. From Figure 9c, we see that when \(|S|\) was larger than 400, the running time of DynDLOff was shorter than LTFPA’s. This is because the running time of LTFPA’s load-balancing phase increased with the number of servers.

To further evaluate DynDLOff’s scalability, next we set default settings for the simulations at \(|S| = 1000, \ c = 10, \ |T| = 10,000, \ β = 1000, \ α = 0, \) and \(θ = 1.\) We changed \(|S|\) and \(|T|\), and showed the running time of DynDLOff’s two phases. From Figure 9d, we see that the running time of Phase II was less than 100 ms in every instance, which illustrates the effectiveness of the binary-search-based optimization in Phase II. We also see that when there were 10,000 tasks and 100,000 cores, DynDLOff generated a task assignment in 4810 ms, which demonstrates that our algorithm can handle scheduling instances that contain tens of thousands of tasks and hundreds of thousands of tasks in a few seconds.
6.2. Real Execution

In the real-world executions, we compared the performance of DynDLOn and DELAY in multijob scenarios in terms of the total job completion time and data locality.

6.2.1. Environment and Settings

We implemented and executed our work in a real-life testbed that consisted of a master and multiple workers. On the master, we deployed two schedulers based on DynDLOn and DELAY, respectively. On the workers, we implemented task runners to emulate the existing MapReduce-like data-parallel frameworks. The testbed ran on a computing cluster with eight servers. Each server had 12 Intel Xeon X5650 cores, 24 GB main memory, and 1 gigabit per second (Gbps) Ethernet (Intel Corporation, Santa Clara, CA, USA). We generated 640, 1280, 2560, and 5120 MB synthetic data files and split each file into 10 data blocks. Each data block had two replicas. Regarding the initial loads, we ran background processes on servers. Each process’ running time was $t$ seconds, where $t$ was randomly chosen from $[2, 12]$. In the experiments, we ran multiple jobs concurrently and evaluated the scheduling algorithms’ effectiveness on multijob systems.

For the algorithms’ parameters, we implemented three delay scheduling algorithms, namely, DELAY3, DELAY6, and DELAY10, whose delay thresholds were set to 3, 6, and 10, respectively. The initial delay threshold of DynDLOn, $W$, was set to 3. In our implementation, if a server was running a data-remote task, DynDLOn did not assign more data-remote tasks to the server.

6.2.2. Real Execution Results

We performed two experiments on the testbed, which simultaneously ran 50 and 100 jobs, respectively. Figures 10 and 11 show the two experiments’ total job completion time and data locality. From Figure 10, we see that DELAY3’s job completion time increased significantly when the data block’s size increased due to data locality. Because DELAY3’s delay threshold (3 s) was much shorter than the average task running time (7 s), it assigned more than 200 data-remote tasks.
Compared to DELAY3, DELAY6’s and DELAY10’s delay thresholds were longer, assigning 8 and 0 data-remote tasks, respectively. Their job completion times were one-fourth of DELAY3’s. Regarding DynDLOn, its job completion time increased from 34.82 to 59.48 s when the data block’s size increased from 64 to 512 MB. Although its initial delay threshold was only 3 s, its data locality was much better than DELAY3’s. For example, when the data block’s size was 512 MB, it assigned 36 data-remote tasks, but DELAY3 assigned 230 data-remote tasks. DynDLOn assigned fewer data-remote tasks when the data block’s size was larger, which illustrates that DynDLOn can adaptively change data locality according to the network conditions. This is because DynDLOn will not assign more data-remote tasks to a server when the server is running a data-remote task. Although it assigned more data-remote tasks than DELAY6 and DELAY10, its job completion time was shorter when the data block’s size was smaller than 256 MB. This is because DynDLOn’s delay time is shorter and it does not concurrently run two data-remote tasks on the same server. When the size of the data block was 512 MB, DynDLOn’s job completion time was longer than that of DELAY6 and DELAY10, because transferring a data block takes 10 s, so DELAY6 and DELAY10 assigned fewer data-remote tasks. However, a longer delay threshold may introduce a larger delay cost, so DynDLOn is still competitive in job completion time.

![Figure 10](image_url)

**Figure 10.** Results of running 50 jobs.

![Figure 11](image_url)

**Figure 11.** Results of running 100 jobs.

DynDLOn is more adaptive than the delay scheduling algorithms, as can be seen in Figure 11. We can see that when the number of jobs increased from 50 to 100, DELAY6’s performance decreased significantly. When the number was 50, only 2% of the tasks were data-remote, but when the number of jobs was 100, nearly 50% of the tasks were data-remote. Moreover, for 512-MB data blocks, when the number of jobs was 50, DELAY6’s job completion time was close to DynDLOn’s, but when the number of jobs was 100, DELAY6’s job completion time was 6.5 times longer than DynDLOn’s. This is because a data-remote task’s running time was more than 10 s, larger than DELAY6’s delay threshold. When the number of jobs increased from 50 to 100, more tasks were data-remote. Then, it was likely that multiple tasks would run on the same server, so the running time increased. Furthermore, the tasks’ average running time was longer than DELAY6’s delay threshold, so the number of data-remote tasks...
also increased. DELAY10’s performance was not affected by the number of jobs. Because the tasks’ running time was randomly selected from [2, 12], the 10-s delay threshold was long enough to wait for data-local servers. However, this makes it difficult to choose a proper delay threshold that suits all the situations. As stated above, DynDLOn is more adaptive than the delay scheduling algorithms. So, we do not need to define the delay threshold precisely, because DynDLOn’s performance is stable for any given scenario.

7. Conclusions

This paper studies a fundamental problem for data-parallel frameworks: data-locality-aware task scheduling. Unlike existing research, our work focuses on a critical problem—data transfer costs that rise steeply with the number of concurrent data-remote tasks on multicore servers. To address this problem, we propose a novel and flexible task-scheduling model that employs a user-defined, non-decreasing function to evaluate the dynamic data transfer cost on each server. Although the cost function is not restricted to a specific form, we propose online and offline algorithms that generate near-optimal solutions. We theoretically prove the offline algorithm’s time complexity and optimality, and empirically study our algorithms’ efficiency and effectiveness through extensive experiments. Results from simulations and real executions show that our algorithms significantly reduce job completion time, adaptively adjust data locality, and process large-scale scheduling instances within subseconds or seconds.

Author Contributions: Conceptualization, J.J. and R.X.; Data curation, J.T.; Formal analysis, J.J.; Methodology, J.J.; Resources, R.X.; Software, J.J., Q.A. and W.Z.; Supervision, R.X.; Visualization, Q.A.; Writing—Review & Editing, J.J.

Funding: This research was funded by National Natural Science Foundation of China under Grants No. 61702096, No. 61602112, and No. 61702097; the Natural Science Foundation of Jiangsu Province under grant BK20170689 and BK20160695; the SCCC Science and Technology Program “research and application of key technologies on public components of the new generation grid dispatching control system platform”.

Acknowledgments: We would like to thank Feng Shan and the anonymous reviewers for their helpful comments and suggestions.

Conflicts of Interest: The authors declare no conflict of interest.

References
1. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI’04), San Francisco, CA, USA, 6–8 December 2004; pp. 137–150.
2. Apache Hadoop. Available online: http://hadoop.apache.org/ (accessed on 1 October 2018).
3. Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster Computing with Working Sets. In Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud’10), Boston, MA, USA, 22–25 June 2010.
4. Malewicz, G.; Austern, M.H.; Bik, A.J.C.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD’10), Indianapolis, IN, USA, 6–10 June 2010; pp. 135–146.
5. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. TensorFlow: A System for Large-Scale Machine Learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
6. Zhang, H.; Cho, B.; Seyfe, E.; Ching, A.; Friedman, M.J. Riffle: Optimized shuffle service for large-scale data analytics. In Proceedings of the Thirteenth EuroSys Conference (EuroSys’18), Porto, Portugal, 23–26 April 2018; pp. 43:1–43:15.
7. Zaharia, M.; Borthakur, D.; Sarma, J.S.; Elmeleegy, K.; Shenker, S.; Stoica, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In Proceedings of the 5th European Conference on Computer Systems (EuroSys’10), Paris, France, 13–16 April 2010; Morin, C., Muller, G., Eds.; ACM: New York, NY, USA, 2010; pp. 265–278.
8. Isard, M.; Prabhakaran, V.; Currey, J.; Wieder, U.; Talwar, K.; Goldberg, A.V. Quincy: fair scheduling for distributed computing clusters. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP’09), Big Sky, Montana, USA, 11–14 October 2009; pp. 261–276.

9. Gog, I.; Schwarzkopf, M.; Gleave, A.; Watson, R.N.M.; Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI’16), Savannah, GA, USA, 2–4 November 2016; Keeton, K., Roscoe, T., Eds.; USENIX: Berkeley, CA, USA, 2016; pp. 99–115.

10. Wang, W.; Zhu, K.; Ying, L.; Tan, J.; Zhang, L. Map task scheduling in MapReduce with data locality: Throughput and heavy-traffic optimality. In Proceedings of the 2013 IEEE Conference on Computer Communications (INFOCOM’13), Turin, Italy, 14–19 April 2013; pp. 1609–1617.

11. Xie, Q.; Yekkehkhany, A.; Lu, Y. Scheduling with multi-level data locality: Throughput and heavy-traffic optimality. In Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM’16), San Francisco, CA, USA, 10–14 April 2016; pp. 1–9.

12. Tan, J.; Meng, X.; Zhang, L. Coupling task progress for MapReduce resource-aware scheduling. In Proceedings of the 2013 IEEE Conference on Computer Communications (INFOCOM’13), Turin, Italy, 14–19 April 2013; pp. 1618–1626.

13. Fischer, M.J.; Su, X.; Yin, Y. Assigning tasks for efficiency in Hadoop: extended abstract. In Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA’10), New York, NY, USA, 13–15 June 2010; auf der Heide, F.M., Phillips, C.A., Eds.; ACM: New York, NY, USA, 2010; pp. 30–39.

14. Jin, J.; Luo, J.; Song, A.; Dong, F.; Xiong, R. BAR: An Efficient Data Locality Driven Task Scheduling Algorithm for Cloud Computing. In Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid’11), Newport Beach, CA, USA, 23–26 May 2011; pp. 295–304.

15. Xie, Q.; Pundir, M.; Lu, Y.; Abad, C.L.; Campbell, R.H. Pandas: Robust Locality-Aware Scheduling with Stochastic Delay Optimality. IEEE/ACM Trans. Netw. 2017, 25, 662–675. [CrossRef]

16. Xie, D.; Li, F.; Yao, B.; Li, G.; Zhou, L.; Guo, M. Simba: Efficient In-Memory Spatial Analytics. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD’16), San Francisco, CA, USA, 26 June–1 July 2016; pp. 1071–1085.

17. Armburst, M.; Xin, R.S.; Lian, C.; Huai, Y.; Liu, D.; Bradley, J.K.; Meng, X.; Kaftan, T.; Franklin, M.J.; Ghodsi, A.; et al. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD’15), Victoria, Australia, 31 May–4 June 2015; pp. 1383–1394.

18. Tigani, J.; Naidu, S. Google BigQuery Analytics; John Wiley & Sons: Hoboken, NJ, USA, 2014.

19. Ma, Z.; Cao, J.; Chen, X.; Xu, S.; Liu, B.; Yang, Y. GLPP: A Game-Based Location Privacy-Preserving Framework in Account Linked Mixed Location-Based Services. Secur. Commun. Netw. 2018, 2018, 9148768. [CrossRef]

20. Amato, F.; Moscato, V.; Picariello, A.; Piccialli, F.; Sperli, G. Centrality in heterogeneous social networks for lurkers detection: An approach based on hypergraphs. Concurr. Comput. Pract. Exp. 2018, 30, e4188. [CrossRef]

21. Persico, V.; Pescapè, A.; Picariello, A.; Sperli, G. Benchmarking big data architectures for social networks data processing using public cloud platforms. Future Gener. Comp. Syst. 2018, 89, 98–109. [CrossRef]

22. Bao, Y.; Peng, Y.; Wu, C.; Li, Z. Online Job Scheduling in Distributed Machine Learning Clusters. arXiv 2018, arXiv:1801.00936.

23. Tiwari, N.; Sarkar, S.; Bellur, U.; Indrawan, M. Classification Framework of MapReduce Scheduling Algorithms. ACM Comput. Surv. 2015, 47, 49:1–49:38. [CrossRef]

24. Zaharia, M.; Konwinski, A.; Joseph, A.D.; Katz, R.H.; Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI’08), San Diego, CA, USA, 8–10 December 2008; pp. 29–42.

25. Xu, H.; Lau, W.C. Task-Cloning Algorithms in a MapReduce Cluster with Competitive Performance Bounds. In Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS’15), Vienna, Austria, 2–5 July 2015; pp. 339–348.

26. Pham, X.; Huh, E. Towards task scheduling in a cloud-fog computing system. In Proceedings of the 18th Asia-Pacific Network Operations and Management Symposium (APNOMS’16), Kanazawa, Japan, 5–7 October 2016; pp. 1–4.
27. Mashayekhy, L.; Nejad, M.M.; Grosu, D.; Zhang, Q.; Shi, W. Energy-Aware Scheduling of MapReduce Jobs for Big Data Applications. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 2720–2733. [CrossRef]

28. Kaur, K.; Kumar, N.; Garg, S.; Rodrigues, J.J.P.C. EnLoc: Data Locality-Aware Energy-Efficient Scheduling Scheme for Cloud Data Centers. In Proceedings of the 2018 IEEE International Conference on Communications (ICC2018), Kansas City, MO, USA, 20–24 May 2018; pp. 1–6.

29. Palanisamy, B.; Singh, A.; Liu, L. Cost-Effective Resource Provisioning for MapReduce in a Cloud. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 1265–1279. [CrossRef]

30. Sandholm, T.; Lai, K. MapReduce optimization using regulated dynamic prioritization. In Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09), Seattle, WA, USA, 15–19 June 2009; pp. 299–310.

31. Qiu, T.; Zheng, K.; Han, M.; Chen, C.L.P.; Xu, M. A Data-Emergency-Aware Scheduling Scheme for Internet of Things in Smart Cities. *IEEE Trans. Ind. Inform.* **2018**, *14*, 2042–2051. [CrossRef]

32. Marozzo, F.; Duro, F.R.; Blas, F.J.G.; Carretero, J.; Talia, D.; Trunfio, P. A data-aware scheduling strategy for workflow execution in clouds. *Concurr. Comput. Pract. Exp.* **2017**, *29*, e4229. [CrossRef]

33. Zhu, Y.; Jiang, Y.; Wu, W.; Ding, L.; Teredesai, A.; Li, D.; Lee, W. Minimizing makespan and total completion time in MapReduce-like systems. In Proceedings of the 2014 IEEE Conference on Computer Communications (INFOCOM’14), Toronto, ON, Canada, 27 April–2 May 2014; pp. 2166–2174.

34. Takefusa, A.; Tatebe, O.; Matsuoka, S.; Morita, Y. Performance Analysis of Scheduling and Replication Algorithms on Grid Datafarm Architecture for High-Energy Physics Applications. In Proceedings of the 12th International Symposium on High-Performance Distributed Computing (HPDC’03), Seattle, WA, USA, 22–24 June 2003; pp. 34–47.

35. Tatebe, O.; Morita, Y.; Matsuoka, S.; Soda, N.; Sekiguchi, S. Grid Datafarm Architecture for Petascale Data Intensive Computing. In Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid’02), Berlin, Germany, 22–24 May 2002; pp. 102–110.

36. Ranganathan, K.; Foster, I.T. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. *IEEE Comput. Soc. Digit. Libr.* **2002**, *1*, 352.

37. Raicu, I.; Foster, I.T.; Zhao, Y.; Little, P.; Moretti, C.M.; Chaudhary, A.; Thain, D. The quest for scalable support of data-intensive workloads in distributed systems. In Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC’09), Garching, Germany, 11–13 June 2009; pp. 207–216.

38. Raicu, I.; Zhao, Y.; Dumitrescu, C.; Foster, I.T.; Wilde, M. Falkon: A Fast and Light-weight tasK executiON framework. In Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing (SC’07), Reno, NV, USA, 10–16 November 2007; p. 43.

39. Max-Min Fairness—Wikipedia. Available online: http://en.wikipedia.org/wiki/Max-minfairness (accessed on 1 October 2018).

40. Liu, Z.; Zhang, Q.; Ahmed, R.; Boutaba, R.; Liu, Y.; Gong, Z. Dynamic Resource Allocation for MapReduce with Partitioning Skew. *IEEE Trans. Comput.* **2016**, *65*, 3304–3317. [CrossRef]

41. Wang, B.; Jiang, J.; Yang, G. ActCap: Accelerating MapReduce on heterogeneous clusters with capability-aware data placement. In Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM’15), Hong Kong, China, 26 April–1 May 2015; pp. 1328–1336.

42. Yu, X.; Hong, B. Grouping Blocks for MapReduce Co-Locality. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS ’15), Hyderabad, India, 25–29 May 2015; pp. 271–280.

43. Ma, X.; Fan, X.; Liu, J.; Li, D. Dependency-Aware Data Locality for MapReduce. *IEEE Trans. Cloud Comput.* **2018**, *6*, 667–679. [CrossRef]

44. Selvitopi, R.O.; Demirci, G.V.; Turk, A.; Aykanat, C. Locality-aware and load-balanced static task scheduling for MapReduce. *Future Gener. Comput. Syst.* **2019**, *90*, 49–61. [CrossRef]

45. Choi, D.; Jeon, M.; Kim, N.; Lee, B. An Enhanced Data-Locality-Aware Task Scheduling Algorithm for Hadoop Applications. *IEEE Syst. J.* **2018**, *1*, 1–12. [CrossRef]

46. Beaumont, O.; Lambert, T.; Marchal, L.; Thomas, B. Data-Locality Aware Dynamic Schedulers for Independent Tasks with Replicated Inputs. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS’18), Vancouver, BC, Canada, 21–25 May 2018; pp. 1206–1213.
47. Li, J.; Wang, J.; Lyu, B.; Wu, J.; Yang, X. An improved algorithm for optimizing MapReduce based on locality and overlapping. *Tsinghua Sci. Technol.* 2018, 23, 744–753. [CrossRef]

48. Harvey, N.J.A.; Ladner, R.E.; Lovász, L.; Tamir, T. Semi-matchings for bipartite graphs and load balancing. *J. Algorithms* 2006, 59, 53–78. [CrossRef]

49. Graham, R.L. Bounds for certain multiprocessing anomalies. *Bell Syst. Tech. J.* 1966, 45, 1563–1581. [CrossRef]

50. Xie, Q.; Lu, Y. Priority algorithm for near-data scheduling: Throughput and heavy-traffic optimality. In Proceedings of the 2015 IEEE Conference on Computer Communications (INFOCOM’15), Hong Kong, China, 26 April–1 May 2015; pp. 963–972.