# Parallel Scheduling of Data-Intensive Tasks

Xiao Meng and Lukasz Golab$^{(\boxtimes)}$

University of Waterloo, Waterloo, Canada
{x36meng,lgolab}@uwaterloo.ca

**Abstract.** Workloads with precedence constraints due to data dependencies are common in various applications. These workloads can be represented as directed acyclic graphs (DAG), and are often data-intensive, meaning that data loading cost is the dominant factor and thus cache misses should be minimized. We address the problem of parallel scheduling of a DAG of data-intensive tasks to minimize makespan. To do so, we propose greedy online scheduling algorithms that take load balancing, data dependencies, and data locality into account. Simulations and an experimental evaluation using an Apache Spark cluster demonstrate the advantages of our solutions.

**Keywords:** Parallel scheduling · Data-intensive tasks · Caching

## 1 Introduction

In the era of big data, many computational tasks are data-intensive: their data loading cost is higher than the subsequent computation cost. These tasks usually have precedence constraints due to data dependencies, represented as a directed acyclic graph (DAG). Examples include scientific workflows, *continuous queries* in streaming and publish-subscribe systems, and Extract-Transform-Load (ETL) pipelines in relational databases. Here, the DAG of tasks is periodically executed on a batch of new data. It is critical to finish these tasks as soon as possible (i.e., minimize the *makespan*) to accommodate the next batch of data. Otherwise, we will fall behind or will have to increase the batch size, thus increasing data latency, which is not desirable in real-time analytics. Sequencing data-intensive tasks then becomes a significant problem because some sequences may incur more cache misses than others, leading to a longer makespan[1].

Scheduling algorithms often assume that the execution times (or estimates) of tasks are known. However, the *cold* versus *hot* (with data already in memory/cache) runtimes of data-intensive tasks may be very different, by an order of magnitude or more. Furthermore, predicting the contents of the cache at any

---

[1] We assume a storage hierarchy with significant speed gaps between different levels, and use the term *cache* more generally, referring to SRAM cache memory, RAM memory, or distributed memory in a platform such as Spark, as appropriate.

point in time is difficult in modern data processing environments with multiple tenants, virtualization, and shared resources.

There has been previous work on a problem we call Serial Data-Intensive Scheduling (SDIS): given a DAG of tasks with data dependencies, SDIS finds an ordering of the tasks that obeys the precedence constraints given by the DAG and aims to minimize the likelihood of cache misses [6]. Knowing the contents of the cache at any time is not required; the only assumption was that the cache uses an LRU-based strategy, where the longer the wait, the slimmer the chance of unused data remaining in the cache.

However, this problem remains unsolved in distributed and parallel settings, such as a Spark [16] cluster or a multi-core database management system. That is the problem we address in this paper – the Multi-Processor Data-Intensive Scheduling (MPDIS) problem to minimize the makespan of a DAG of data-intensive tasks. The additional complexity of MPDIS over SDIS comes from two factors: 1) a larger search space of possible schedules and 2) a load balancing requirement missing from serial scheduling. Thus, a solution to the MPDIS problem must simultaneously ensure load balancing and data locality. We make the following contributions towards solving this problem:

1. We define the MPDIS problem of scheduling a DAG of data-intensive tasks on multiple processors, assuming a shared LRU cache, but without knowing the contents of the cache at any point in time.
2. We propose three greedy online algorithms to solve the MPDIS problem using cache metrics from the Programming Language and Compiler literature.
3. Using simulations and a Spark cluster, we experimentally show the effectiveness of our algorithms against existing techniques on real-world based DAGs.

The remainder of this paper is organized as follows. In Sect. 2, we review related work. We formulate our scheduling problem in Sect. 3 and propose solutions in Sect. 4. We present experimental results in Sect. 5 and we conclude in Sect. 6.

**Example 1:** Consider the DAG of tasks in Fig. 1, with edges showing data dependencies (e.g, the data output of task zero is the data input to task 3). Assume each task produces an output of unit size. Suppose for each of the six tasks, the computation cost (hot runtime) is one time unit while the loading cost of one data unit is ten time units. Assume the cache can hold up to two data items at the same time. Figure 2 shows two schedules, labelled S1 and S2, for two processing units, labelled PU1 and PU2. The task runtimes are coloured blue and plotted on a time axis. The figure also shows the contents of the cache at various points in time; e.g., "01" indicates that the cache currently holds the outputs of tasks zero and one.

For both schedules, there will be cache misses for items 0 and 1 since the cache is initially empty. This means that tasks 0 and 1 run cold, for a total of 11 time units (10 time units to load the data plus one time unit for the computation). For the first schedule, S1, at time 11, tasks 0 and 1 finish, and
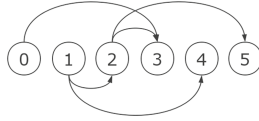
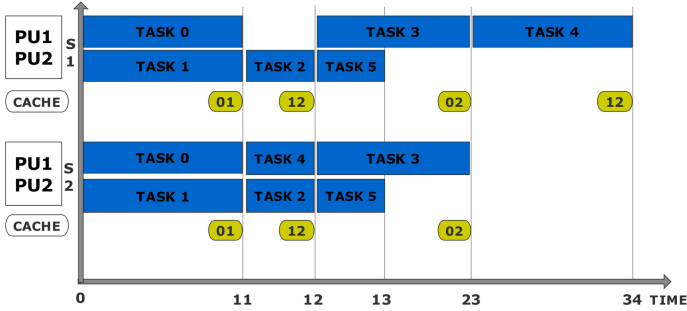**Fig. 1.** Example DAG of data-intensive tasks.



**Fig. 2.** Two schedules for example DAG from Fig. 1 on two processors.

both of their outputs are in the cache. Task 2 is started on processor 2. Task 3 waits for task 2 to finish because its input is the output of task 2. Task 3 takes one time unit because the data needed for task 2, namely the data output of task 1, is in the cache. At time 12, when task 2 is finished, task 3 and task 5 are started. The output of task 2 is in the cache, evicting the output of task 0 according to the LRU policy. Thus, now the cache holds the outputs of tasks 1 and 2. This means that task 3 causes a cache miss for item 0, and finishes at time 23, while task 5 finishes earlier at time 13 (because its input, which was the output of task 2, was in the cache). At this time, the cache holds the outputs of tasks 0 and 2. Thus, task 4 causes a cache miss for item 1, and therefore finishes at time 34.

In schedule S2, when tasks 0 and 1 terminate, tasks 2 and 4 run hot because their input (the output of task 1) is in the cache. When tasks 2 and 4 are done, the cache now contains the output of task 2 (which evicts the output of 0) and the output of task 1 (note that task 4 does not produce any output for use by subsequent tasks). This means that task 5 runs hot, but task 3 incurs a cache miss because it requires the output of task 0. Note that schedule S2 incurs fewer cache misses and has a shorter makespan, highlighting the need for a scheduling strategy for data-intensive workloads.

## 2    Related Work

Scheduling DAGs of tasks on multiple processors to minimize makespan is an NP-Complete problem with only a few exceptions [10]. Therefore, many heuristics have been proposed; however, data-intensive tasks were not considered. In

particular, [7] compared these heuristics empirically and found that Heterogeneous Earliest Finish Time (or HEFT) is among the best for multiprocessor DAG scheduling. We will use a modified HEFT as a baseline in our experimental comparison (Sect. 5).

In terms of data-intensive scheduling, there has been work on the SDIS problem [6] mentioned earlier. A solution was given that minimizes the distance in a serial schedule (in terms of the number of tasks) between the first and the last access of every data item. Heuristics were given to solve this minimization problem. However, the MPDIS problem, which is the focus of our work, was not considered.

We also point out related work on minimizing the (peak or total) memory footprint of parallel schedules; see, e.g., [5,11]. However, to the best of our knowledge, these methods do not consider data-intensive tasks; they schedule tasks to optimize memory usage but do not optimize the sequence in which data items are inserted into memory in order to avoid cache misses.

Scheduling with setup times is another related topic, in which there are different types of tasks, and scheduling a task of a different type than the previous task requires some setup time; in general, the execution time of a task may include a setup time that is dependent on the tasks that have been executed up to now [4]. However, the MPDIS problem is more complex because the sequence of tasks executed up to now may not be sufficient to determine the contents of the cache.

Since Spark [16] is one target system for our solutions, we briefly discuss data-intensive scheduling in Spark. The work on memory caching in Spark (e.g., [9,14, 15]) does not consider data dependencies among tasks, as we do. Furthermore, in a system such as Spark, there is a shared cache, but also local memory and disk. There has been work on the problem of *reducer placement* to schedule reducers (of a given task) on nodes that have much of the required data already in local memory [12]. MPDIS is an orthogonal problem of sequencing tasks, and reducer placement solutions may be applied independently to assign the reducers of a given task to the available machines, and further improve performance.

## 3   Problem Definition and Assumptions

We consider data-intensive (as defined earlier), non-preemptible tasks, with precedence constraints corresponding to data dependencies. Precedence constraints are expressed as a directed acyclic graph (DAG) $G = (V, E)$, where each node $v \in V$ represents a task and each directed edge $e = (u, v) \in E$ represents a precedence constraint. An edge in the DAG denotes that the data output of one task is the data input to another. Thus, an edge $(u, v)$ requires that task $u$ has to be completed before task $v$ starts. Optional input may include the size the data output of each task, represented as an edge weight in the DAG. Tasks are scheduled on $n$ *homogeneous* processing units that share a fast storage layer with an LRU-based replacement policy (which, as explained earlier, may be an SRAM cache, RAM memory, or distributed memory). However, we assume that the contents of the fast storage layer cannot be reliably predicted at any point

in time, as motivated earlier. This means that we cannot know with certainty whether a task will run cold or hot.

A precedence constraint $(u, v)$ indicates that the output of $u$ is the input to $v$. The intuition behind our scheduling objective is to schedule $v$ as soon as possible after $u$. The longer we wait, the more likely it is that other tasks will be scheduled, which may require other data inputs. Thus, the longer we wait, the more likely it is that the output of $u$ will be evicted from an LRU cache, causing a cache miss when $v$ runs. To formalize this intuition, we use the following data locality metrics from prior work (our solutions are independent of the data locality metric, and we will experiment with both of these metrics in Sect. 5):

– *Stack Distance* (SD) [8] is a metric from the programming languages and compiler literature. The stack distance between two accesses of a data item counts the *distinct* number of other data items that were accessed in between. The more data items accessed in between, the more likely it is that the original data item is no longer in the cache when it is accessed again[2]. We compute the stack distance of a schedule as the sum of the stack distances between every pair of consecutive references to the same data item, with "reference" denoting producing the item as output or consuming the item as input. If a task references more than one output, then we sequence these accesses in lexicographic order for computation (e.g., in Fig. 1, task 3 first accesses the output of task 0 and then the output of task 2).
– *Total Maximum Bandwidth* (TMB) was proposed in prior work on the SDIS problem [6]. TMB considers the first and the last access of a data item, and counts the distinct number of other data items that were accessed in between. (SD measures the same quantity, but for each pair of consecutive accesses of a data item.)

**Example 2:** Consider two data items, $A$ and $B$. Suppose they are accessed in the following sequence: $A, B, A, B, A$. The stack distance of this sequence is three: one distinct item $(B)$ is accessed between the first and the second access of $A$; $B$ is again accessed between the second and the third access of $A$; plus, one distinct item $(A)$ is accessed between the two accesses of $B$. The TMB of this sequence is two: one distinct item $(B)$ is accessed between the first and the last access of $A$ (not including $A$ itself); plus, one distinct item $(A)$ is accessed between the first and the last access of $B$.

**Example 3:** Recall the DAG in Fig. 1 and assume the following schedule: [0, 1, 2, 3, 4, 5]. That is, the tasks are sequenced as shown in the figure. The output of task 1 becomes the input to tasks 2 and 4. Thus, the output of task 1 is referenced three times: by task 1 at creation time, by task 2, and by task 4. The stack distance between the first and second reference is zero: no other tasks ran

---

[2] *Reference Distance* (RD) is a related metric that counts the total number of data accesses in between, not the distinct data accesses. SD was shown to be more accurate than RD in quantifying data locality [8], so we will not consider RD any further.

in between. The stack distance between the second and the third reference is two: task 3 ran in between and it accessed the outputs of task 0 and 2. Thus, it is more likely that the output of task 1 was evicted from the cache before it is needed by task 4. In total, we have:

- task 0 produces output that is referenced once by task 3. In between, task 1 produced output referenced by task 2, giving a stack distance of one.
- task 1 produces output that is referenced twice (becomes the input to two downstream tasks), giving stack distances of zero and two, respectively.
- task 2 also produces output that is referenced twice, with the corresponding stack distances of zero (nothing runs between tasks 2 and 3), and two (task 3 additionally references the output of task 0 and task 4 requires the output of task 1).

This gives a stack distance of $1 + 0 + 2 + 0 + 2 = 5$ for the entire schedule.

We now reiterate the two data-intensive scheduling problems mentioned earlier.

**Problem 1: Serial Data-Intensive Scheduling (SDIS).** Given a DAG of tasks with precedence constraints, produce a serial schedule that obeys the precedence constraints with the smallest SD or TMB.

A version of SDIS that minimized TMB was studied in [6]. In this paper, we solve the following problem:

**Problem 2: Multi-Processor Data-Intensive Scheduling (MPDIS).** Given a DAG of tasks with precedence constraints and $n$ processing units sharing a fast memory layer, produce a parallel schedule across the $n$ processors that obeys the precedence constraints, with the smallest SD or TMB over a serialized representation of the parallel schedule according to task start times (we compute stack distance over this serialized representation since all processing units access the same cache).

**Example 4:** We compute SD for the complete schedules in Fig. 2 below. S1 [0, 1, 2, 3, 5, 4] costs $1 + 2 + 1 = 4$ and S2 [0, 1, 4, 2, 3, 5] costs $1 + 0 + 1 = 2$. Note that S2 has a smaller stack distance and a shorter completion time.

We remark that there exists a weighted version of Problem 2, where instead of counting the number of other data items accessed between two references of some data item, we count the total size of the other data items accessed. Similarly, TMB can be extended to its weighted version, abbreviated WTMB [6]. Data item sizes can be given as edge weights in the precedence DAG.

## 4    Solutions

We present three solutions to the MPDIS problem in this section. Our solutions are *online*, meaning that tasks are scheduled on-the-fly rather than being statically assigned to processing units in a pre-defined order. We do not consider

offline algorithms that assemble a complete schedule apriori. Given our assumptions, even if we enumerated the possible schedules, we could not compute their makespans since task runtimes may be cold or hot, depending on the contents of the cache, which we cannot predict in advance.

### 4.1   Parallel SDIS (PS)

The first solution, Parallel SDIS, is an extension of the SDIS solution from [6] that produces a serial schedule to minimize TMB (and can be modified to minimize SD instead). First, we generate a single-threaded schedule $S$ using the existing SDIS solution. Then, whenever a processing unit is available, we schedule the next task from $S$, call it $t$, on this processing unit. Note that if $t$ is not schedulable at this time (i.e., all the tasks it depends on have not yet terminated), then the processing unit is idle until $t$ becomes schedulable.

**Example 5:** Fig. 3 shows an example of Parallel SDIS on two processing units (PU1 and PU2) using the DAG from Fig. 1, assuming the computation and loading times listed in Table 1, and assuming the cache can hold two data items. Assume we use SD as the data locality metric instead of TMB. Here, an optimal SDIS schedule using SD is [0, 1, 4, 2, 3, 5]. Given this SDIS schedule, our Parallel SDIS algorithm proceeds as follows. First, task 0 is scheduled on PU1 and runs cold for 51 time units. At the same time, task 1 is scheduled on PU2 and runs cold for 11 time units. When task 1 terminates, the next task in the SDIS schedule is task 4, which is now scheduled on PU2. Task 4 is now schedulable (because it only relies on task 1, which just terminated), and runs hot until time 12. At this time, the cache holds the outputs of task 1. Next in the SDIS schedule is task 2, which is scheduled on PU2 and runs hot until time 13. At this time, the cache holds the output of tasks 1 and 2. Next, task 3 is scheduled on PU2, but it must wait until task 0 terminates. Thus, task 3 begins running only at time 51 and terminates at time 52. When task 0 terminates at time 51, the last task is task 5, which is now scheduled on PU1. Task 5 runs hot for 10 time units, terminating at time 61. The makespan is thus 61.

**Time Complexity:** The complexity of PS depends on the complexity of the underlying SDIS solution. For example, the heuristic solution from [6] has a complexity of $O(|E||V|^2 + |V|^3 \log |V|)$, where $|E|$ and $|V|$ is the number of edges and vertices, respectively, in the DAG. After generating such a solution, we insert the serial schedule into a queue and pop the next schedulable task from the queue whenever a processing unit becomes available. Checking whether a task $t$ is schedulable has a complexity of $O(|E|)$ (assume we have a hash map of completed tasks; then it suffices to find the edges incident on $t$ and check if the predecessors of $t$ all exist in the set of completed tasks), and this is done for all $|V|$ tasks.
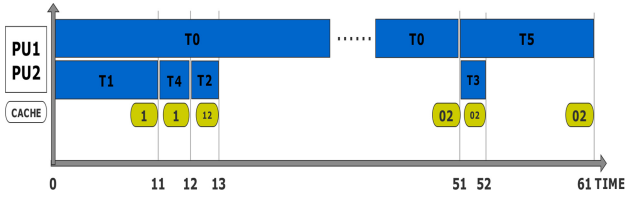
**Fig. 3.** Schedule for example DAG generated by Algorithm PS.

**Table 1.** Data loading and computation runtimes of tasks for example DAG from Fig. 1.

| Task | Loading (time units) | Computation (time units) |
|------|---------------------|--------------------------|
| 0    | 50                  | 1                        |
| 1    | 10                  | 1                        |
| 2    | 10                  | 1                        |
| 3    | 60                  | 1                        |
| 4    | 10                  | 1                        |
| 5    | 10                  | 10                       |

### 4.2   Online Greedy (OG)

Notice a potential problem with the Parallel SDIS algorithm: since it uses a single-threaded sequencing as a seed, the next task in the schedule may not yet be schedulable in parallel with another task that is currently running. This causes some processing units to be idle (e.g., PU2 in Fig. 3 is idle from time 13 to time 51). To address this problem, we propose an Online Greedy (OG) algorithm. OG does not compute a single-threaded schedule beforehand. Instead, whenever a processor becomes available, OG chooses the next *schedulable* task that yields the smallest SD or TMB when added to the current partial schedule (with ties broken arbitrarily). Thus, OG does not stall as long as there is at least one schedulable task.
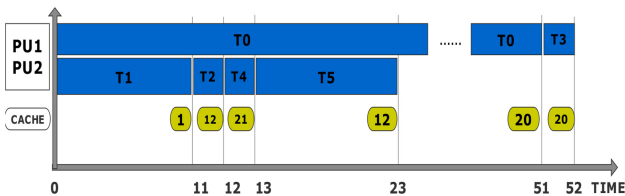


**Fig. 4.** Schedule for example DAG generated by Algorithm OG.

**Example 6:** Fig. 4 shows an example of OG on two processing units, again using the DAG from Fig. 1, the parameters listed in Table 1, and assuming the cache can hold two data items. Assume again that we use SD as the data locality metric. At the beginning, the only schedulable tasks are 0 and 1. Breaking ties randomly, we assign task 0 to PU1 and task 1 to PU2, and both tasks run cold. When task 1 finishes at time 11 and PU2 becomes free, there are two schedulable tasks: task 2 and task 4. To decide which task to schedule on PU2, we compute the SD of the following partial schedules and choose the task that gives the partial schedule with the lowest SD: [0, 1, 2] and [0, 1, 4]. Both are zero, so we break ties randomly. Let task 2 run on PU2.

Next, task 2 terminates at time 12 (it ran hot because the output of task 1 is in the cache). At this time, tasks 4 and 5 are schedulable. To decide which one to schedule next, we compute the SD of the following partial schedules and again choose the task that gives the partial schedule with the lowest SD: [0, 1, 2, 4] and [0, 1, 2, 5]. Both are again zero, so we break ties randomly. Let task 4 run on PU2 (it runs hot because the output of task 2 is in the cache), finishing at time 13. Now, task 0 is still running on PU1, so the only schedulable task is task 5. Thus, we run task 5 on PU2. It runs hot because the output of task 2 is in the cache, terminating at time 23. At this time, there are no schedulable tasks, so PU2 is idle. When task 0 terminates at time 51, the only remaining task is task 3, which runs hot until time 52. Note that the OG schedule terminates nine time units earlier than the PS schedule described in Example 6 (52 vs. 61).

**Time Complexity:** Computing the SD or TMB of a (partial) schedule requires $O(|E|)$ time: we loop over all the outgoing edges of the tasks already in the schedule, which gives $O(|E|)$. When making a scheduling decision, we compute the SD or TMB for $O(|V|)$ tasks that may be scheduled next, giving $O(|V||E|)$. The total number of scheduling decisions is $O(|V|)$, giving a complexity of $O(|V|^2|E|)$.

### 4.3   Greedy Complete Schedule (GCS)

At any point in time, algorithm OG greedily chooses a task whose addition to the current partial schedule yields the lowest total SD or TMB. We now observe that, in contrast to other on-line scheduling settings, we know the workload in advance: it is given in the form of a DAG. To leverage this observation, we propose a Greedy Complete Schedule (GCS) algorithm. The intuition is as follows: when a processing unit becomes available, we choose a schedulable task that leads to a *complete schedule* with the lowest SD or TMB. When deciding which task to schedule next, we "simulate" a complete schedule by greedily and repeatedly adding the next task that minimizes SD or TMB, until all the remaining tasks have been scheduled.

**Example 7:** Consider again the DAG from Fig. 1 and the task runtimes from Table 1. Assume again that there are two processing units, PU1 and PU2, and that we want to minimize SD. First, we schedule task 0 on PU1 and task 1 on PU2. At time 11, task 1 finishes and PU2 frees up. There are now two schedulable

tasks: 2 and 4. GCS now performs the following steps. If we were to schedule task 2 next, the partial schedule would be $[0, 1, 2]$. To simulate a complete schedule, we greedily keep adding tasks that minimize the SD of the partial schedule. Thus, we first compute the SD of the following schedules: $[0, 1, 2, 3]$ (with $SD = 5$), $[0, 1, 2, 4]$ (with $SD = 1$), and $[0, 1, 2, 5]$ (with $SD = 2$). Since adding task 4 leads to the lowest SD, the simulated schedule now becomes $[0, 1, 2, 4]$. Next, we consider adding tasks 3 and 5 to the simulated schedule, i.e., we compute the SD of $[0, 1, 2, 4, 3]$ and $[0, 1, 2, 4, 5]$. It turns out that $[0, 1, 2, 4, 5]$ has a lower SD. This gives a complete simulated schedule of $[0, 1, 2, 4, 5, 3]$. Similarly, returning to the initial partial schedule of $[0, 1]$, if we were to schedule task 4 next instead of task 2, the partial schedule would be $[0, 1, 4]$, and we simulate a complete schedule starting with this prefix. Finally, to decide between task 2 and task 4, we choose the task whose complete simulated schedule has a lower SD.

There is a potential problem with the above method: when simulating a greedy complete schedule, we need to know the set of schedulable tasks at every step. However, for this, we need to know which tasks have already terminated at any point in time, yet we do not know the task runtimes (because we do not know whether the tasks will run hot or cold). The simplest assumption to make is that tasks terminate in first-in-first-out (FIFO) order. For example, assuming a partial schedule $[0, 1, 2, 4]$ and two processing units, the FIFO assumption means that the next scheduling decision happens when task 0 terminates, and we update the set of schedulable tasks accordingly. However, in practice, task 1, task 2 or even task 4 could terminate before task 0, leading to a different set of schedulable tasks at this point in time. To sum up the challenge that must be addressed in our setting: *while we know the workload in advance, we do not know the order in which tasks will terminate.* As a result, we cannot predict the complete schedule throughout the execution of a workload, even if we assume a deterministic greedy heuristic at every scheduling step.

In algorithm GCS, we mitigate the above problem as follows. Instead of generating only one complete schedule assuming FIFO task termination order, we generate multiple possible complete schedules for each candidate task under consideration. One of these possible complete schedules assumes FIFO order. For the other schedules, we iterate through the partial schedule, and swap the completion order of every pair of consecutive tasks already started but not yet completed. For example, if the current partial schedule is $[0, 1, 2]$ and all three of these tasks are still running, we would consider completing the following swapped partial schedules: $[1, 0, 2]$, $[0, 2, 1]$. For each of these partial schedules, we generate a complete schedule as discussed above, and compute its SD. Finally, for each candidate task to be scheduled next, we record the lowest SD of any of its simulated complete schedules, and we select the task with the lowest SD of the best simulated schedule (we also experimented with choosing the task with the lowest average SD of all of its simulated full schedules, and obtained similar results).

**Time Complexity:** Algorithm GCS incurs an extra $O(|V|^3)$ factor compared to OG, for a complexity of $O(|V|^5|E|)$. This is because it no longer suffices to

compute the SD of the partial schedule of each schedulable task when making a scheduling decision. Instead, it takes $O(|V|^2|E|)$ time to assemble a complete schedule (specifically, $O(|V||E|)$ time to decide on the next task to add to the schedule, and there are $O(|V|)$ remaining tasks). Furthermore, there are $O(|V|)$ complete schedules to test, resulting from swapping the completion order of the tasks in the current partial schedule.

## 5   Experimental Evaluation

In this section, we present experimental results comparing our solutions (PS, OG, and GCS) with two baselines: B1 and B2. B1 is random scheduling, which chooses a random schedulable task whenever a processor is available. B2 is the HEFT algorithm, where the idea is to prioritize tasks based on the runtimes of the tasks depending on them. Since we do not know the actual runtimes, which can be cold or hot, we use the sum of the sizes of the data inputs of the dependent tasks (effectively assuming cold runtimes). We start with simulation results (Sect. 5.1) and then present results using a Spark cluster (Sect. 5.2).

### 5.1   Scheduling Simulation

We identified three DAGs used in real applications and concatenated them to create our first workload, referred to as DAG1 and illustrated in Fig. 5. The DAGs correspond to a network monitoring workflow [6], an image stitching workflow called Montage [1], and an earthquake analysis workflow called CyberShake [1]. Note that there are no dependencies across tasks from the three concatenated DAGs. In other words, DAG1 corresponds to a multi-tenant workload with three independent DAGs of tasks.
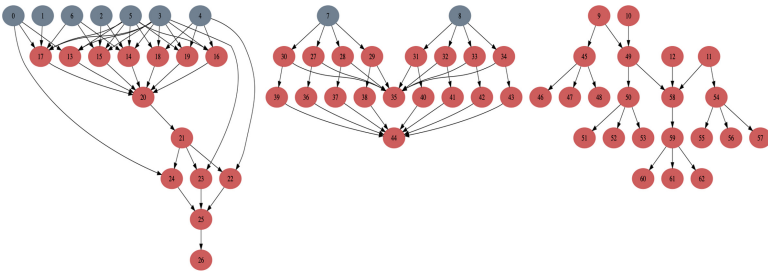


**Fig. 5.** DAG1 based on three real workloads.

In addition to DAG1, we created several larger versions of it, referred to as DAG1 v2 through DAG1 v6 (illustrations omitted due to space constraints). DAG1 v2 horizontally duplicates DAG1; DAG1 v3 vertically "grows" DAG1 by duplicating each level of tasks; DAG1 is similar to DAG1 v3 but adds more data

dependencies among the tasks; DAG1 v5 combines horizontal and vertical duplication; and DAG1 v6 adds more data dependencies to DAG1 v5. The number of tasks and edges in each DAG, as well as the average number of schedulable tasks at any point in the schedule (abbreviated $|SS|$), are presented in Table 2. Increasing the size and the complexity of the DAG correspondingly increases the number of schedulable tasks at any point in time. On the other hand, small DAGs may limit parallelism opportunities.

**Table 2.** Number of tasks, edges (precedence constraints), and the average size of the schedulable task set ($|SS|$) for each DAG.

| DAG | DAG1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| $|V|$ | 63 | 113 | 113 | 113 | 145 | 145 |
| $|E|$ | 96 | 192 | 205 | 240 | 224 | 237 |
| $|SS|$ | 7.2 | 12.4 | 11.6 | 11.2 | 16.1 | 15.2 |

Our simulation environment, implemented in Python, has two components: cache simulation and schedule simulation. For cache simulation, we used PyLRU [2]. By keeping the key of a data item in the LRU cache through a dictionary data structure, we simulate the cache contents at any given time. Schedule simulation uses a scheduler module, in which we implemented various scheduling algorithms for comparison. The input parameters include the DAG, with edge weights corresponding to data sizes (to compute weighted SD and weighted TMB), the cold and hot runtimes (defined as functions of the input data size), the cache size, and the number of cores/processing units. The simulator then schedules the tasks as prescribed by the given scheduling algorithm, and keeps track of statistics such as the simulated I/O, processor idle percentage, and makespan.

We consider different cache pressure points in our simulations. To do so, we set the cache size to 20 GB and we vary the input size to each task. Tasks at the first level of the DAG are set to be ten times slower than other tasks (i.e., they consume ten times more data) to simulate workloads that compute smaller data products over large raw data.

We start by setting the total data size in the simulation to 120 GB, which is the sum of the inputs to each task in the DAG. This gives a data to cache ratio of 6:1. In Table 3, we report the performance gap (in terms of makespan) between B1 and OG for different numbers of threads ranging from one to 8. We used weighted SD as the data locality metric for these initial experiments. We notice that increasing the level of parallelism does not lead to a bigger performance improvement of OG over B1. Therefore, for more cache pressure, we increase the size of the data from 120 GB to 240 GB, 480 GB, 600 GB, and 840 GB. This gives data to cache ratios between 6:1 and 42:1.

Figure 6 shows the results, with makespan on the y-axis (in seconds) and data size on the x-axis. The figure contains five sets of bars, one for each data size, as labelled. Each set contains seven bars, corresponding to the different

**Table 3.** Performance gap between B1 and OG in terms of makespan (120 GB data size).

| Number of threads | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Performance gap (B1/OG) | *1.95x* | *1.83x* | *1.84x* | *1.72x* |

scheduling algorithms and locality metrics we test, from right to left: B1, B2, PS using weighted SD, PS using weighted TMB (WTMB), OG using weighted SD, OG using WTMB, and GCS[3].

Additionally, for the largest, 840 GB, data size, we show I/O volume and the percentage of processing unit idle time for each scheduling algorithm in Table 4. Note that some idle time may be due to the workload: if there are only two tasks that can initially be scheduled, then regardless of the sequencing, only two processing units will initially be busy. Additionally, processing units may be idle if they are assigned a task that is currently not yet schedulable; this may happen in PS, but we designed OG and GCS to avoid these situations.

We make the following observations based on the simulation results:

**Table 4.** I/O transfer and CPU idle time percentage for DAG1 (840 GB data size).

| Method | I/O (GB) | Method | CPU idle % |
|---|---|---|---|
| B1 | *1741* | B1 | *12.5* |
| B2 | *1597* | B2 | *13.7* |
| PS(SD) | *1207* | PS(SD) | *12.8* |
| PS(WTMB) | *1210* | PS(WTMB) | *12.9* |
| OG(SD) | *921* | OG(SD) | *7.9* |
| OG(WTMB) | *940* | OG(WTMB) | *8.2* |
| GCS | *930* | GCS | *8.0* |

**Observation 1:** (weighted) SD gave schedules with smaller makespans than (weighted) TMB by up to 15%.

**Observation 2:** Both baselines, B1 and B2, give schedules with similar makespans.

**Observation 3:** For DAG1, as cache pressure (data size) increases, the improvement of OG and GCS over the baselines stabilizes at around 2.3x (as shown in Fig. 6). For DAG1-v2, the gap stabilizes at 3.1x; For DAG1-v3, the gap is at 2.9x; for DAG1-v4, the gap stabilizes around 3.2x (similar to v2);

---

[3] We only report GCS results using weighted SD; results using WTMB were worse and are omitted from the figures.
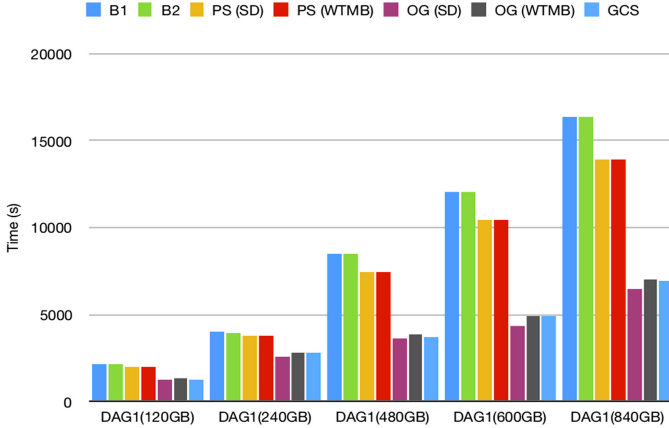
**Fig. 6.** DAG1 experiments (x-axis: data size; y-axis: makespan).

DAG1-v5 significantly increases the gap to 4.5x; and DAG1-v6 stabilizes at 4.3x (figures omitted due to space constraints). Thus, the more complex the DAG, the more potential for improvement over the baselines. The I/O volumes reported in Table 4 show a similar trend, helping to explain the reason for the improved workload runtimes due to our scheduling methods.

**Observation 4:** From the reported CPU idle times in Table 4, PS causes more CPU idle times than OG than GCS. This, along with the drop in I/O volumes also reported in Table 4, explains the trend shown in the figures: while PS outperforms the baselines in terms of the makespan of the workload, OG and GCS further outperform PS by a factor of two in many cases (especially for larger data sizes).

**Observation 5:** GCS rarely outperformed OG on the tested workloads and sometimes produced worse schedules. This suggests that simulating complete schedules without knowing the completion order of the tasks is not an effective strategy, even after allowing for some swaps as we do in GCS. Instead, a simple online greedy heuristic such as OG can work well in practice.

## 5.2   Data-Intensive Scheduling in Spark

For Spark experiments, we used a private cluster of 8 nodes (and a subset of 4 nodes from this cluster) running Ubuntu Linux 18.10. Each node is equipped with four Intel Xeon E5-2620 2.10 GHz 6-core CPUs, 64 GB of DDR3 RAM and 2.7 TB of local storage. The cluster runs Apache Spark 2.3.1 and Apache Hadoop 2.6 for HDFS.

We use the TPC-DS benchmark as the dataset generator [13], with scale factor 200. We again use the DAG from Fig. 5, with the tasks corresponding to data-intensive queries from the TPC-DS benchmark that were identified as such

by [6]. We use Spark standalone mode to simplify the setup and to avoid the impact of cluster managers such as YARN [3].

We use a default Spark configuration, and each executor is given all the cores available on a worker by default. We let Spark control its distributed memory cache using the default LRU strategy. We experimented with two setups. In the 4-node setup, we use four nodes in the cluster and we limit the number of concurrent tasks to 4. In the 8-node setup, we use all 8 nodes and we limit the number of concurrent tasks to 8.

To isolate the impact of task sequencing, we do not perform any dynamic resource allocation (e.g., giving some tasks more resources if some cores are idle). Additionally, since we focus on data-intensive tasks, data I/O is the bottleneck, not processing time. Thus, resource reallocation would be expected to have a limited impact on makespan. We implemented the workload as a packaged application, and included the scheduling algorithms as callable routines in the code.

We compared our algorithms, PS, OG and GCS, with the baselines, and the results are presented in Fig. 7, with makespan on the left and the number of cache misses on the right. We show results for both setups: 4 nodes with up to 4 concurrent tasks and 8 nodes with up to 8 concurrent tasks. We only use weighted SD in this experiment since previous experiments showed that weighted TMB produces worse schedules. We also monitor the total cache misses when running different schedules (we were unable to reliably compute the total I/O volume in Spark). To measure this, we collect the cache misses using Linux tools on each server first, and then we aggregate the statistics for a final total cache misses of the system.
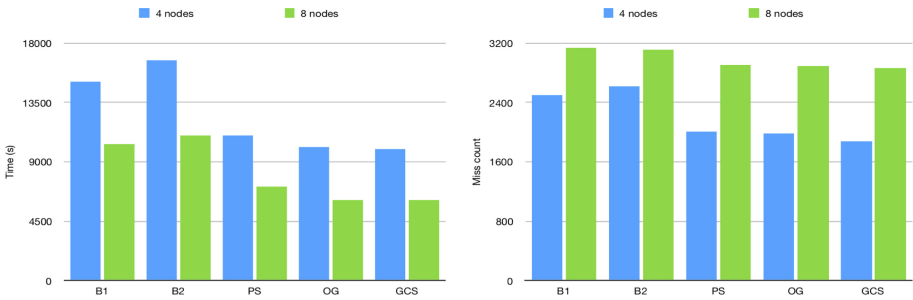


**Fig. 7.** Spark tests: makespan using 4 and 8 nodes (left figure; x-axis: algorithm; y-axis: runtime), and number of cache misses (right figure; x-axis: algorithm; y-axis: number of misses).

**Observation 6:** Increasing the available resources while at the same time increasing the maximum parallelism level leads to a decrease in schedule makespans.

**Observation 7:** In terms of makespan, PS outperforms the baselines by about 30%. OG further outperforms PS by about 20%, and the performance of OG and GCS was nearly identical. Figure 7 shows similar trends in terms of cache misses, suggesting a reason for the improved performance of our solutions compared to the baselines. However, the performance improvement of OG/GCS over PS is not as high as in the simulations, likely because Spark jobs incur some overhead regardless of sequencing.

## 6    Conclusions

We defined the MPDIS problem of scheduling a DAG of data-intensive tasks (with precedence constraints due to data dependencies) on multiple processing units. To measure data locality, we considered two cache metrics proposed in prior work: stack distance and total maximum bandwidth. We proposed online scheduling algorithms: Parallel SDIS (PS), a straightforward extension of serial scheduling of data-intensive tasks to a parallel setting; Online Greedy (OG), which addresses a problem with PS, namely the fact that it may leave some processing units idle while ensuring data locality; and Greedy Complete Schedule (GCS), which is based on OG but leverages the knowledge of the workload provided in the DAG. Experimental results showed that PS reduces makespan and I/O volume compared to the baselines, and OG further improves upon PS. GCS did not perform better than OG, showing that knowing the workload may not be sufficient in our setting as we do not know the completion order of tasks (and therefore cannot infer the set of schedulable tasks at any particular point in the future).

We assumed an architecture in which multiple processing units share a cache. In future work, we will study new versions of the MPDIS problem for shared-nothing settings. One version is to partition the workload across multiple shared-nothing clusters, and optimize the stack distance of the schedules within each partition. The additional complexity of this new problem is due to the interaction between partitioning and data locality of the partitioned schedules.

In this work, we assumed that once a processor finishes a task and becomes idle, it obtains a single new task. To reduce the scheduling overhead, we can instead schedule a batch of ready tasks at every scheduling round. In future work, we will evaluate the impact of such batch scheduling on data locality.

## References

1. Pegasus. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator
2. Pylru 1.2.0. https://pypi.org/project/pylru/
3. Spark standalone. https://spark.apache.org/docs/latest/spark-standalone.html
4. Allahverdi, A.: The third comprehensive survey on scheduling problems with setup times/costs. Eur. J. Oper. Res. **246**(2), 345–378 (2015)
5. Arras, P.A., Fuin, D., Jeannot, E., Stoutchinin, A., Thibault, S.: List scheduling in embedded systems under memory constraints. Int. J. Parallel Prog. **43**, 1103–1128 (2015)

6. Bär, A., Golab, L., Ruehrup, S., Schiavone, M., Casas, P.: Cache-oblivious scheduling of shared workloads. In: IEEE International Conference on Data Engineering, pp. 855–866 (2015)
7. Canon, L.C., Jeannot, E., Sakellariou, R., Zheng, W.: Comparative evaluation of the robustness of dag scheduling heuristics. In: Grid Computing, pp. 73–84 (2008)
8. Coffman, E.G., Denning, P.J.: Operating Systems Theory. Prentice-Hall, New Jersey (1973)
9. Deslauriers, F., McCormick, P., Amvrosiadis, G., Goel, A., Brown, A.D.: Quartet: harmonizing task scheduling and caching for cluster computing. In: USENIX Workshop on Hot Topics in Storage and File Systems (2016)
10. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv. (CSUR) **31**(4), 406–471 (1999)
11. Marchal, L., Simon, B., Vivien, F.: Limiting the memory footprint when dynamically scheduling dags on shared-memory platforms. J. Parallel Distrib. Comput. **128**, 30–42 (2019). https://doi.org/10.1016/j.jpdc.2019.01.009
12. Meng, X., Golab, L.: Optimal reducer placement to minimize data transfer in MapReduce-style processing. In: 2017 IEEE International Conference on Big Data, pp. 339–346 (2017)
13. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: International Conference on Very Large Data Bases, pp. 1049–1058 (2006)
14. Xu, E., Saxena, M., Chiu, L.: Neutrino: revisiting memory caching for iterative data analytics. In: USENIX Workshop on Hot Topics in Storage and File Systems (2016)
15. Yang, Z., Jia, D., Ioannidis, S., Mi, N., Sheng, B.: Intermediate data caching optimization for multi-stage and parallel big data frameworks. In: IEEE International Conference on Cloud Computing, pp. 277–284 (2018)
16. Zaharia, M., et al.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016). https://doi.org/10.1145/2934664