Data Races and the Discrete Resource-time Tradeoff Problem with Resource Reuse over Paths*

Rathish Das
Stony Brook University
radas@cs.stonybrook.edu

Shih-Yu Tsai
Stony Brook University
shitsai@cs.stonybrook.edu

Sharmila Duppala
Stony Brook University
sduppala@cs.stonybrook.edu

Jayson Lynch
MIT
jaysonl@mit.edu

Esther M. Arkin
Stony Brook University
esther.arkin@stonybrook.edu

Rezaul Chowdhury
Stony Brook University
rezaul@cs.stonybrook.edu

Joseph S. B. Mitchell
Stony Brook University
joseph.mitchell@stonybrook.edu

Steven Skiena
Stony Brook University
skiena@cs.stonybrook.edu

ABSTRACT

A determinacy race occurs if two or more logically parallel instructions access the same memory location and at least one of them tries to modify its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing atomic accesses to it. However, such solutions can reduce parallelism by serializing all accesses to that location. For associative and commutative updates to a memory cell, one can instead use a reducer, which allows parallel race-free updates at the expense of using some extra space. More extra space usually leads to more parallel updates, which in turn contributes to potentially lowering the overall execution time of the program.

We start by asking the following question. Given a fixed budget of extra space for mitigating the cost of races in a parallel program, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the overall running time? We argue that under reasonable conditions the races of a program can be captured by a directed acyclic graph (DAG), with nodes representing memory cells and arcs representing read-write dependencies between cells. We then formulate our original question as an optimization problem on this DAG. We concentrate on a variation of this problem where space reuse among reducers is allowed by routing every unit of extra space along a (possibly different) source to sink path of the DAG and using it in the construction of multiple (possibly zero) reducers along the path. We consider two different ways of constructing a reducer and the corresponding duration functions (i.e., reduction time as a function of space budget).

We generalize our race-avoiding space-time tradeoff problem to a discrete resource-time tradeoff problem with general non-increasing duration functions and resource reuse over paths of the given DAG.

For general DAGs, we show that even if the entire DAG is available offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs.

CCS CONCEPTS

• **Theory of computation → Parallel computing models**; **Distributed computing models**; **Complexity classes**; **Approximation algorithms analysis**; **Scheduling algorithms**; **Routing and network design problems**; **Shared memory algorithms**; • **Computing methodologies → Shared memory algorithms**.

KEYWORDS

Data races; Reducers; Space-time tradeoff; Discrete resource-time tradeoff; Resource reuse; Scheduling; Makespan

ACM Reference Format:
Rathish Das, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther M. Arkin, Rezaul Chowdhury, Joseph S. B. Mitchell, and Steven Skiena. 2019. Data Races and the Discrete Resource-time Tradeoff Problem with Resource Reuse over Paths. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3323165.3323209

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPAA '19, June 22–24, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323209

*This research was supported in part by NSF grants CCF-1439084, CCF-1526406, CNS-1408695, CNS-1755615, CCF-1439084, CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1553510, IIS-1546113, and US-Israel BSF grant #2016116. The authors would like to thank Riko Jacob and Francesco Silvestri for some very useful initial discussions on related problems at the "Hawaii Workshop on Parallel Algorithms and Data Structures" (AlgoPARC) in December 2017, and Nodari Sitchinava and others for organizing the workshop with support from NSF (grant CCF-1745331). Thanks to participants of the weekly Stony Brook Algorithms Reading Group meetings for fruitful discussions where this problem was posed in Spring 2018. The authors also thank anonymous referees for useful comments and suggestions which improved the presentation of this paper.

RIGHTSLINK

1 INTRODUCTION

A determinacy race (or a general race) [14, 27] occurs if two or more logically parallel instructions access the same memory location and at least one of them modifies its content. Races are often undesirable as they can lead to nondeterministic and incorrect program behavior. A data race is a special case of a determinacy race which can be eliminated by associating a mutual-exclusion lock with the memory location in question or allowing only atomic accesses to it. Such a solution, however, makes all accesses to that location serial and thus destroys all parallelism. Figure 1 shows an example.

One can use a reducer [8, 15, 30] to eliminate data races on a shared variable without destroying parallelism, provided the update operation is associative and commutative. Figure 2 shows the construction of a simple recursive binary reducer. For any integer $h > 0$ such a reducer is a full binary tree of height h and size $2^{h+1} - 1$ with the shared variable at the root. Each nonroot node is associated with a unit of extra space initialized to zero. All updates to the shared variable are equally distributed among the leaves of the tree. Each node has a lock and a waiting queue to avoid races by serializing the updates it receives, but updates to different nodes can be applied in parallel. As soon as a node undergoes its last update, it updates its parent using its final value. In fact, such a reducer can be constructed using only 2^h units of extra space because if a node completes before its sibling it can become its own parent (with ties broken arbitrarily) and the sibling then updates the new parent. Assume that the time needed to apply an update significantly dominates the execution time of every other operation the reducer performs and each update takes one unit of time to apply. Then a reducer of height h can correctly apply n parallel updates on a shared variable in $\lceil \frac{n}{2^h} \rceil + h + 1$ time provided at least 2^h processors are available. Hence, for large n, the speedup achieved by a reducer (w.r.t. serially and directly updating the shared variable) is almost linear in the amount of extra space used.

To see how extra space can speed up real parallel programs consider the iterative matrix multiplication code Parallel-MM shown in Figure 3 which multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$ and puts the results in another $n \times n$ matrix $Z[1..n][1..n]$; that is, it sets $Z[i][j] = \sum_{1 \le k \le n} X[i][k] \times Y[k][j]$ for $1 \le i, j \le n$. Since every $Z[i][j]$ value can be computed independently of others, all iterations of the loops in Lines 1 and 2 can be executed in parallel without compromising correctness of the computation. However, the same is not true for the loop in Line 4 because if parallelized, for fixed values of i and j, all iterations of that loop will update the same memory location $Z[i][j]$ giving rise to data races and thus producing potentially incorrect results. Use of a mutual-exclusion lock or atomic updates for each $Z[i][j]$ will ensure correctness but in that case even with an unbounded number of processors, the code will take $\Theta(n)$ time to multiply the two $n \times n$ matrices. Now if we put a reducer of height h (integer $h \in [1, \log_2 n]$) at the top of each $Z[i][j]$ the time to fully update each $Z[i][j]$ and thus the overall running time of the code will drop to $\Theta\left(\frac{n}{2^h} + h\right)$ at the cost of using $n^2 \times 2^h$ units of extra space. Observe that when $h = 1$, the running time of the code almost halves using $2n^2$ units of extra space, and when $h = \lfloor \log_2 n \rfloor$, the running time drops to $\Theta(\log n)$ using $\Theta\left(n^3\right)$ extra space.

In order to analyze a program P with data races, we capture those races in a directed acyclic graph (DAG) $D(P)$, assuming that there are no cyclic read-write dependencies among the memory locations accessed by P. Figure 4 shows an example. We restrict P to the set of programs that perform $O(1)$ other operations between two successive writes to the memory, e.g., Parallel-MM in Figure 3. We assume that an update operation is significantly more expensive than any other single operation performed by P and hence the costs of those operations can be safely ignored. Each node x of $D(P)$ represents a memory location, and a directed edge from node x to node y means that y is updated using the value stored at x. The in-degree $d_x^{(in)}$ of node x gives the number of times x is updated. With x we also associate a *work* value w_x and set $w_x = d_x^{(in)}$. Assuming that each update operation requires unit time to execute and each node has a lock and a wait queue to serialize the updates, the w_x value represents the time spent updating x (excluding all idle times). The w_x value also represents an upper bound on the time elapsed between the trigger time of any incoming edge of x and the time the edge completes updating x. We assume that updates along all outgoing edges of x trigger as soon as all incoming edges complete updating x. One can then make the following observation.

Observation 1.1. *The running time of P with an unbounded number of processors is upper bounded by the makespan of $D(P)$*[1].

Then one natural question to ask is the following.

Question 1.1. *Given a fixed budget of units of extra space to mitigate the cost of data races in P, which memory locations should be assigned reducers and how should the space be distributed among those reducers in order to minimize the makespan of $D(P)$?*

Figure 5 shows how to minimize the makespan of the DAG in Figure 4 using two units of extra space.

The question above ignores the possibility that space can be reused among reducers in $D(P)$. Indeed, after node x reaches its final value (i.e., updated $w_x = d_x^{(in)}$ times) it can release all (if any) space it used for its reducer which can then be reused by some other node y. A global memory manager can be used by the nodes to allocate/deallocate space for reducers. The following modified version of Question 1.1 now allows space reuse.

Question 1.2. *Repeat Question 1.1 but allow for space reuse among nodes of $D(P)$ by putting all extra space under the control of a global memory manager that each node calls to allocate space for its reducer right before its first update and to deallocate that space right after its last update.*

The problem with a single global memory manager is that it can easily become a performance bottleneck for highly parallel programs. Though better memory allocators have been developed for multi-core or multi-threaded systems [1–3, 5, 33], we can instead use an approach often used by recursive fork-join programs [7, 9, 16] which avoids repeated calls to an external memory manager altogether along with the overhead of repeated memory allocations/deallocations. A single large segment of memory is allocated

[1] To see why this is true start from the sink node and move backward toward the source by always moving to that predecessor y of the current node x that performed the last update on x and noting that after edge (y, x) was triggered it did not have to wait for more than $d_x^{(in)}$ time units to complete applying y's update to x.

RIGHTSLINK

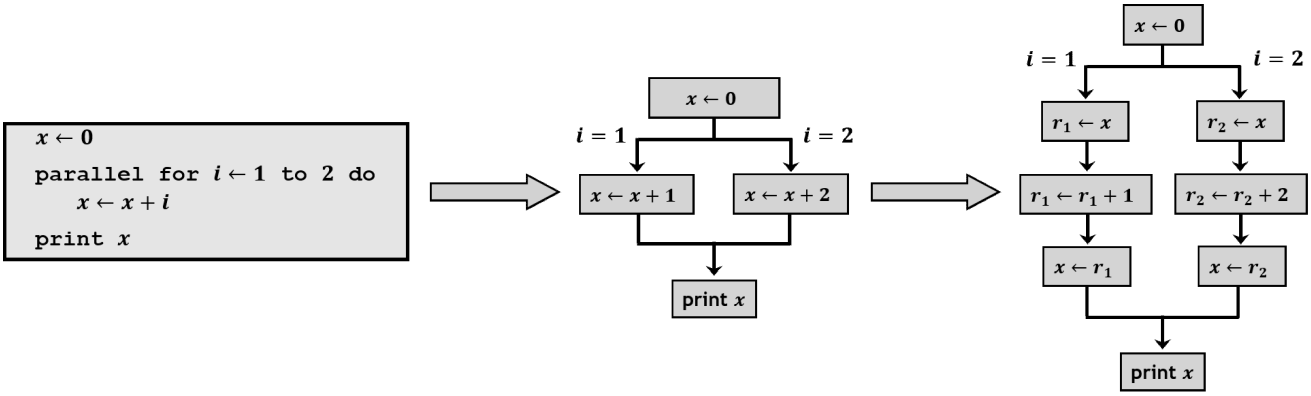

Figure 1: This figure shows a race on global variable x caused by two parallel threads trying to increment x, where r_1 and r_2 are local registers. The value printed by the 'print' statement depends on how the two threads are scheduled. Unless the two threads are executed sequentially, the print statement will print an incorrect result (either 1 or 2 depending on which thread updated x last).

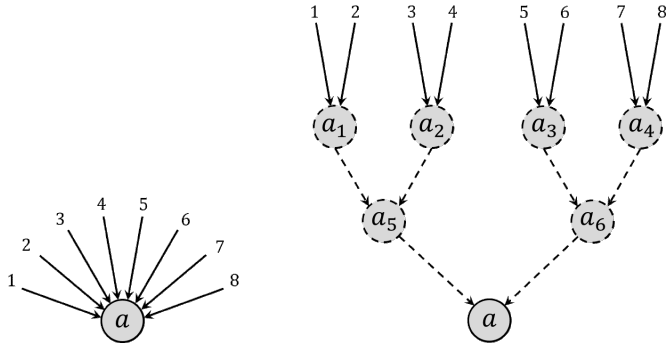

Figure 2: [Left] A memory location a with eight updates using an associative and commutative operator. [Right] The same location a with a recursive binary reducer of height two on top of it.

PARALLEL-MM(Z, X, Y, n)

(1) **parallel for** $i \leftarrow 1$ **to** n **do**
(2) **parallel for** $j \leftarrow 1$ **to** n **do**
(3) $Z[i][j] \leftarrow 0$
(4) **for** $k \leftarrow 1$ **to** n **do**
(5) $Z[i][j] \leftarrow Z[i][j] + X[i][k] \times Y[k][j]$

Figure 3: Parallel code that multiplies two $n \times n$ matrices $X[1..n][1..n]$ and $Y[1..n][1..n]$, and puts the result in $Z[1..n][1..n]$.

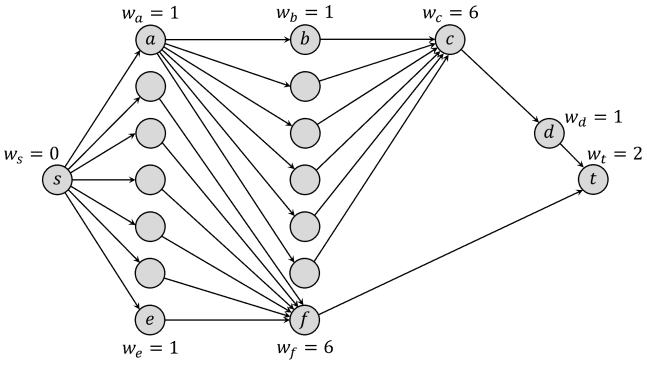

Figure 4: A DAG in which each node's work value is set to its in-degree. The makespan of this DAG is 11, and path $s \to a \to b \to c \to d \to t$ achieves it.

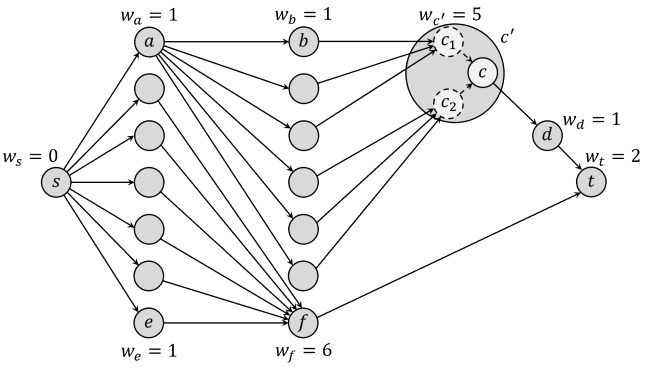

Figure 5: Node c from the DAG in Figure 4 has been replaced with a supernode c' in this figure which is nothing but node c with a reducer of height 1 on top. The makespan of this reduced DAG is 10, and path $s \to a \to b \to c_1 \to c \to d \to t$ achieves it.

RIGHTSLINK

before the initial recursive call is made and a pointer to that segment is passed to the recursive call. Each recursive call splits and distributes its segment among its child recursive calls and reclaims the space when the children complete execution. So, we will assume that all the given extra space initially reside at the source node (i.e., node with in-degree zero). Then they flow along the edges toward the sink (i.e., node with outdegree zero) possibly splitting along outgoing edges and merging at the tip of incoming edges as they flow. Each unit of space reaching node x moves out of x along some outgoing edge as soon as x becomes fully updated and those edges trigger. Every unit of space may participate in the construction of multiple reducers (possibly zero) along the path it takes.

Question 1.3. *Repeat Question 1.1 but now allow for space reuse among nodes of $D(P)$ by flowing each unit of space along a source to sink path and using it in the construction of zero or more reducers along that path.*

While several existing results [11, 12, 20, 35] can be extended to answer Questions 1.1 and 1.2, to the best of our knowledge, Question 1.3 had not been raised before. In this paper we investigate answers to Question 1.3 by extending it to a more general resource-time tradeoff question posed on a DAG in which nodes represent jobs (not necessarily of updating memory locations), resources (not necessarily space) flow along source to sink paths, and an general duration function (i.e., time needed to complete a job as a function of the amount of resources used) is specified for each node. We consider the following three duration functions: general non-increasing function for the general resource-time question, and recursive binary reduction and multiway (k-way) splitting for the space-time case.

For general DAGs, we show that even if the entire DAG is available to us offline the problem is strongly NP-hard under all three duration functions, and we give approximation algorithms for solving the corresponding optimization problems. We also prove hardness of approximation for the general resource-time tradeoff problem and give a pseudo-polynomial time algorithm for series-parallel DAGs. Our main results are summarized in Table 1.

Related Work

While several prior works either directly or indirectly address Questions 1.1 (nonreusable resources) and 1.2 (globally reusable resources), to the best of our knowledge, Question 1.3 (reusable along flow paths) has not been considered before.

The well-known time-cost tradeoff problem (TCTP) is closely related to our nonreusable resources question. In TCTP, some activities are expediated at additional cost so that the makespan can be shortened. Deadline and budget problems are two TCTP variants with different objectives. While the deadline problem seeks to minimize the total cost to satisfy a given deadline, the budget problem aims to minimize the project duration to meet the given budget constraint [4]. Most researchers consider the tradeoff functions to be either linear continuous or discrete giving rise to linear TCTP and discrete TCTP, respectively.

Linear TCTP was formulated by Kelley and Walker in 1959 [22]. They assumed affine linear and decreasing tradeoff functions. In 1961, linear TCTP was solved in polynomial time using network flow approaches independently by Fulkerson [17] and Kelley [21]. Phillips and Dessouky [29] later improved that result.

In 1997, De et al. [11] proved that discrete TCTP is NP-hard. For this problem, Skutella [35] proposed the first approximation algorithm under budget constraints which achieves an approximation ratio of $O(\log r)$, where r is the ratio of the maximum duration of any activity to the minimum one. Discrete TCTP can also be used to approximate the TCTP with general time-cost tradeoff functions, see, e.g., Panagiotakopoulos [28] and Robinson [31].

Our problem with globally reusable resources (Question 1.2) is very similar to the problem of scheduling precedence-constrained malleable tasks [36]. In 1978, Lenstra and Rinnooy Kan [23] showed that no polynomial time algorithm can solve it with approximation ratio less than $\frac{4}{3}$ unless P = NP. In 1989, Du and Leung [12] showed that the problem is strongly NP-hard even for two units of resources. In 2002, under the monotonous penalty assumptions of Blayo et al [6], Lepère et al. [24] first proposed the idea of two-step algorithms – computing an allocation first, and then scheduling tasks, and used this idea [25] to design a algorithm that achieve an approximation ratio of ≈ 5.236. In the first phase, they approximate an allocation using Skutella's algorithm [35]. Similarly, based on Skutella's approximation algorithm, Jansen and Zhang [20] devised a two-phase approximation algorithm with the best-known ratio of ≈ 4.730598 and showed that the ratio is tight when the problem size is large. For more details on the problems of scheduling malleable tasks with precedence constraints, please check Dutot et al. [13].

There are memory allocators based on global memory manager for multi-core or multi-threaded systems such as scalloc [3], Hoard [5], llalloc [2], Streamflow [33], and TCMalloc [1]. They use thread-local space for memory allocation and a global manager for memory deallocation/reuse. For the global manager, they use concurrent data structures. However, these data structures can not completely avoid synchronization [3, 19, 34] without compromising correctness.

Preliminaries, Problem Formulation

In general, the option to use reducers to trade off between extra space and the time to complete race-free writing operations leads to a *discrete resource-time tradeoff problem*, where, here, the valuable "resource" is the space that is added, in order to reduce the time necessary for the write operations. By investing in additional space, we can reduce the time it takes to do conflict-free write operations.

We formalize the discrete resource-time tradeoff problem. Consider a DAG, $D = (V, E)$, whose nodes V correspond to jobs, and whose edges represent precedence relations among jobs. Without loss of generality, we assume that the DAG has a single source and a single sink node. The duration of a job depends on how many resource it receives. For each job $v \in V$, there is a non-increasing duration function $t_v(r)$ that denotes the time required to complete job v using r units of resources. We call $\langle r, t_v(r)\rangle$ a *resource-time tuple* associated with job (node) v. We consider three classes of duration functions – general non-increasing step functions, k-way splitting functions, and recursive binary splitting functions.

General non-increasing step function. Let l_v be the number of resource-time tuples associated with job v. The i-th resource-time tuple is $\langle r_{v,i}, t_v(r_{v,i})\rangle$ where $1 \le i \le l_v$. Then, the duration function $t_v(r)$ is a step function with l_v steps described as follows:

RIGHTSLINK

Duration function	Hardness	Hardness of Approximation	Approximation Results
General non-increasing	strongly NP-hard	• makespan < 2 OPT with resources fixed • resource $< \frac{3}{2}$ OPT with makespan fixed	$\left(\frac{1}{\alpha}, \frac{1}{1-\alpha}\right)$ bi-criteria (resource, makespan), $0 < \alpha < 1$
Recursive binary	strongly NP-hard	–	• makespan ≤ 4 OPT with resources fixed • $\left(\frac{4}{3}, \frac{14}{5}\right)$ bi-criteria (resource, makespan)
Multiway splitting	strongly NP-hard	–	makespan ≤ 5 OPT with resources fixed

Table 1: Our main results on resource-time tradeoff problems in which resources are routed along source to sink paths (i.e., related to Question 1.3 and its generalization).

$$t_v(r) = \begin{cases} t_v(r_{v,i}), & \text{if } r_{v,i} \leq r < r_{v,i+1}, 1 \leq i < l_v, \\ t_v(r_{v,l_v}). & \text{if } r_{v,l_v} \leq r, \end{cases} \quad (1)$$

where $r_{v,1} = 0, r_{v,j} < r_{v,j+1}, t_v(r_{v,j}) \geq t_v(r_{v,j+1})$ for $1 \leq j < l_v$.

***k*-way splitting.** A *k-way split reducer* utilizes k units of extra space, $S_v = \{s_1, s_2, .., s_k\}$, associated with a node v, with $2 \leq k \leq d_v^{(in)}$, such that the write operations associated with incoming edges at v are distributed among the nodes in S_v, which then have edges linking each s_i to v. The duration function that results from k-way split reducers is given by

$$t_v(r) = \begin{cases} t_v(0), & \text{if } k \in \{0, 1\} \\ \lceil t_v(0)/k \rceil + k, & \text{if } 2 \leq k \leq \lfloor \sqrt{t_v(0)} \rfloor \\ t_v(\lfloor \sqrt{t_v(0)} \rfloor). & \text{if } \lfloor \sqrt{t_v(0)} \rfloor < k. \end{cases} \quad (2)$$

Recursive binary splitting. The duration function that results from a recursive binary split reducer is given by a step function, as follows. The resource-time tuples are defined for $r = 0$ and 2^i where $0 \leq i \leq k$ and $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$. The duration function $t_v(2^k) = \lceil t_v(0)/2^k \rceil + k + 1$ is minimized when $k = \lfloor \log_2 t_v(0) - \log_2 \log_2 e \rfloor$ (by differentiating $t_v(2^k)$ w.r.t. k).

$$t_v(r) = \begin{cases} t_v(0), & \text{if } r = 0, 1 \\ \lceil t_v(0)/2^i \rceil + i + 1, & \text{if } r = 2^i, 2 \leq i \leq k \\ t_v(2^i), & \text{if } 2^i \leq r < 2^{i+1}, 2 \leq i \leq k \\ t_v(2^k), & \text{if } i > k \end{cases} \quad (3)$$

When utilizing a reducer, extra space serves as the limited resource and the time taken for race-free writing at a node v is the duration of the job corresponding to v. Both the k-way splitting duration function and the recursive binary splitting duration function are special cases of general non-increasing function.

We distinguish between two optimization problems, depending on the objective function:

Minimum-Makespan Problem. Given a resource budget of B, assign the resources to nodes V such that the makespan of the project is minimized. Resources can be reused over a path.

Minimum-Resource Problem. Given a makespan target of T, minimize the amount of resources to achieve target makespan. Resources can be reused over a path.

Finally, we remark that instead of jobs corresponding to nodes of the DAG, we can transform the DAG D into another DAG D' in which jobs correspond to edges of D', and the precedence relations among jobs are enforced by introducing dummy edges, as follows: For each node v in D, we introduce an edge $e_v = (a_v, b_v)$ in D' (which then has the corresponding duration function, specified, e.g., by resource-time tuples). For each edge (u, v) of D, we introduce a dummy edge, $e = (b_u, a_v)$ in D', from the endpoint b_u of edge $e_u = (a_u, b_u)$ to the origin a_v of edge $e_v = (a_v, b_v)$, with resource-time function $t_e(r) = 0$ for all valid resource levels r.

2 APPROXIMATION ALGORITHMS

2.1 Bi-criteria Approximation for Non-increasing Duration Functions

We use linear programming in our approximation algorithms. First, we relax the discrete duration function to a linear one. We transform the DAG so that a relaxed linear non-increasing duration function can be used. The transformation happens in two steps.

Activity on edge reduction. We reduce the input DAG D into an equivalent DAG D' with activities on edges instead of nodes. This is a simple transformation described earlier in Section 1.

Activity with two tuples. Following [35], we create a DAG D'' from D' such that all activities in D'' are still on edges and each such activity has at most 2 resource-time tuples. Let j be a job with $l_j \geq 2$ resource-time tuples $\langle r_{j,i}, t_j(r_{j,i}) \rangle, 1 \leq i \leq l_j$ with $0 = r_{j,1} < r_{j,2} < \cdots < r_{j,l_j}$ and $t_j(r_{j,1}) \geq t_j(r_{j,2}) \geq \cdots \geq t_j(r_{j,l_j})$ (following Equation 1). Let edge (u, v) of D' represent job j. We add l_j parallel chains, each consisting of two edges in D''. For $1 \leq i \leq l_j$, we create a chain of two edges (u, u_i) and (u_i, v). We create a job j_i for edge (u, u_i) and associate two resource-time tuples with it. For $1 \leq i < l_j$, job j_i can be finished either using 0 resource in $t_j(r_{j,i})$ units of time or using $(r_{j,i+1} - r_{j,i})$ units of resource in 0 unit of time. The logic is that job j's duration can be reduced from $t_j(r_{j,i})$ to $t_j(r_{j,i+1})$ provided the resource difference $(r_{j,i+1} - r_{j,i})$ is allocated to j_i. Thus the duration function is $t_{j_i}(0) = t_j(r_{j,i})$ and $t_{j_i}(r_{j,i+1} - r_{j,i}) = 0$. Job j_{l_j}'s (bottommost edge in the l_j parallel edges for job j) duration cannot be further improved from $t_j(r_{j,l_j})$ units of time by using extra resources. The resource-time tuple at edge (u_i, v) is $\langle 0, 0 \rangle$ where $1 \leq i \leq l_j$.

There is a canonical mapping of resource usages and durations for jobs j_i to that of job j. Let x_i be the units of resource used for job j_i, then for job j, $\sum_{i=1}^{l_j} x_i$ units of resource are used. The time taken to finish job j is $\max\{t_{j_i}(x_i) | 1 \leq i \leq l_j\}$. Without loss of generality, if we use 0 unit of resource for job j_i if $t_{j,i}(0) \leq \max\{t_{j,1}(x_1), t_{j,2}(x_2), \cdots, t_{j,i-1}(x_{i-1})\}$, then this mapping is bijective. Thus we get the following lemma.

Lemma 2.1. *Any approximation algorithm $\mathcal{A}$ on DAG D'' (activity on edge and each edge has at most two resource-time tuples) with an approximation ratio α implies an approximation algorithm with the same approximation ratio α on general DAG D (activity on node and each job can have more than two resource-time tuples).*

RIGHTSLINK

From now on, we will only consider DAGs whose edges represent jobs, with each edge having at most two resource-time tuples.

Linear relaxation. In D'', any edge (u, v) can have either two resource-time tuples $\{\langle 0, t_{(u,v)}(0)\rangle, \langle r_{(u,v)}, 0\rangle\}$ or a single resource-time tuple $\{\langle 0, t_{(u,v)}(0)\rangle\}$. With linear relaxation, $r \in [0, r_{(u,v)}]$ units of resource can be used to reduce the completion time of the job corresponding to edge (u, v) that has two resource-time tuples. The corresponding duration function $t_{(u,v)}(r)$ is as follows:

$$t_{(u,v)}(r) = \left(t_{(u,v)}(0)/r_{(u,v)}\right) r \text{ for } r \in [0, r_{(u,v)}] \tag{4}$$

The linear duration function $t_{(u,v)}(r)$ for the job (u, v) with single resource-time tuple is as follows:

$$t_{(u,v)}(r) = t_{(u,v)}(0) \text{ for all } r \geq 0 \tag{5}$$

Linear programming formulation. Since we are allowed to reuse resources over a path we can model the problem as a network flow problem where resources are allowed to flow from the source to the sink in D''. Let E be the set of edges in D''. Let $f_{(u,v)}$ denote the amount of resources that flow through the edge (u, v). Using linear relaxation on edge (u, v), the time taken to finish the activity is $t_{(u,v)}(f_{(u,v)})$. Let the nodes in D'' denote events. From now onwards, we use a node and its corresponding event synonymously. Let $E_v = \{(x, v)\}$ be the set of edges that are incident on node v. Event v occurs if and only if all the jobs corresponding to the edges in E_v are finished. Let T_v denote the time when event v occurs. Let s and t denote the source node and the sink node, respectively. We assume $T_s = 0$. All variables are non-negative.

Constraints:

$$f_{(u,v)} \leq r_{(u,v)}\ ,\ \forall (u, v) \text{ with two resource-time tuples.} \tag{6}$$

$$T_u + t_{u,v}(f_{(u,v)}) \leq T_v\ ,\ \forall (u, v) \in E \tag{7}$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0\ ,\ \forall v \notin \{s, t\} \tag{8}$$

$$\sum_k f_{(s,k)} \leq B \tag{9}$$

Objective function:

$$\min T_t \tag{10}$$

Inequality 6 upper bounds the resource flow variable $f_{(u,v)}$ for edges with two tuples. This ensures that these variables remain in the range $[0, r_{(u,v)}]$ and the duration function is linear in this range. Note that there is no such upper bound on the edges with single resource-time tuple (except for the trivial upper bound B). This allows the flow of more resources over an edge that can be used later on a path. Equation 8 is a flow conservation constraint for all the nodes $v \notin \{s, t\}$. Inequality 9 constrains the flow of resources from source s to be upper bounded by the resource budget.

Solving the LP and rounding. We first solve the LP described above. This might give solution as fractional flow f_e^* and duration $t_e(f_e^*)$ at edge $e = (u, v)$. Let the resource-time tuples at edge e be $\{\langle 0, t_e(0)\rangle, \langle r_e, 0\rangle\}$. The range of feasible duration of activity e is $[0, t_e(0)]$. We divide this range into two parts $[0, \alpha t_e(0))$, $[\alpha t_e(0), t_e(0)]$ where $0 < \alpha < 1$. If $t_e(f_e^*) \in [0, \alpha t_e(0))$ we round it down to 0, otherwise, we round it up to $t_e(0)$. Observe that in the first case, the resource requirement at e can be increased by at most a factor of $1/(1 - \alpha)$. In the second case, the completion time can be increased at most by a factor of $1/\alpha$. Let f_e' denote the rounded integer resource requirement at edge e.

Computing min-flow. After rounding the LP solution, we get an integral resource requirement $f_e' \in \{0, r_e\}$ for every edge e. We now compute a min-flow through this DAG where f_e' serves as the lower bound on the flow through (or resource requirement at) edge e.

Constraints:

$$f_{(u,v)} \geq f'_{(u,v)}\ ,\ \forall (u, v) \in E \tag{11}$$

$$\sum_w f_{(v,w)} + \sum_u f_{(u,v)} = 0\ ,\ \forall v \notin \{s, t\} \tag{12}$$

Objective function:

$$\min \sum_k f_{(s,k)} \tag{13}$$

Let, f and f^* be the optimal solutions of LP 11–13 and LP 6–10, respectively.

Lemma 2.2. *$f^*/(1 - \alpha)$ is a feasible solution of min-flow LP 11–13.*

Proof. For every edge e, $f_e' \leq f_e^*/(1 - \alpha)$ holds in the optimal solution of LP 6–10. Hence, $f^*/(1 - \alpha)$ is a feasible solution of LP 11–13 as it meets the resource requirement f_e' at every edge e. □

Lemma 2.3. *f is an integral flow and $f \leq f^*/(1 - \alpha)$, where $0 < \alpha < 1$.*

Proof. The minflow problem has integral optimality. If f is the optimal solution then it is an integral flow. From Lemma 2.2 we know that $f^*/(1 - \alpha)$ is a feasible solution of LP 11–13. Since f is optimal and $f^*/(1 - \alpha)$ is feasible, we have, $f \leq f^*/(1 - \alpha)$. □

Bi-criteria approximation. We now summarize our bi-criteria approximation result for general non-increasing duration functions:

Theorem 2.4. *For any $\alpha \in (0, 1)$, there is a $(1/\alpha, 1/(1 - \alpha))$ bi-criteria approximation algorithm for the discrete resource-time tradeoff problem with a general non-increasing duration function which allows resource reuse over paths.*

Proof. First, we know from Lemma 2.3 that f is an integral flow and $f \leq f^*/(1 - \alpha)$, where $0 < \alpha < 1$.

Second, we claim that the makespan of the DAG used in the minflow LP 11–13 is at most a factor of $1/\alpha$ away from that of the LP 6–10 solution. Let us consider any $s - t$ path $\mathcal{P}$. The makespan is at least the sum of completion times of the edges in $\mathcal{P}$. Now, after rounding the LP 6–10 solution, the completion time of an edge may increase by at most a factor of α. Hence, the sum of duration of edges along any path is increased by a factor of at most α, and thus the makespan will not increase by more than an α factor. □

2.2 Improved Bi-criteria Approximation for Recursive Binary Splitting Functions

Putting $\alpha = 3/4$ in Theorem 2.4 we obtain a $(4/3, 4)$ bi-criteria approximation algorithm for general non-increasing duration functions. Hence, if we use 4/3 times more resources than OPT (i.e., the optimal solution), we are guaranteed to get a makespan within factor of 4 of OPT. In this section we show that the bound can be improved to $(4/3, 14/5)$ for recursive binary splitting functions.

RIGHTSLINK

For a node with in-degree x, the resource-time tuples based on the recursive binary splitting function are as follows: $\{\langle 0, x\rangle, \langle 1, x\rangle, \langle 2, t_1\rangle, \ldots, \langle 2^i, t_i\rangle, \langle 2^{i+1}, t_{i+1}\rangle \ldots, \langle 2^k, t_k\rangle\}$ where $t_j = \lceil x/2^j \rceil + j + 1$ for $j \geq 2$ and $k = \lfloor \log_2 x - \log_2 \log_2 e \rfloor$ is the largest value of j for which t_j decreases with the increase of j.

After solving LP 6–10 from Section 2.1, we sum up the (possibly fractional) resources allocated to all the l_j parallel edges corresponding to job j. Let r be that sum. Let t be the maximum among the time values given by the LP solution for the l_j parallel edges. Thus, the LP takes t units of time for job j.

We round r to an integer $\overline{r}$ based on the following criteria.

$$\overline{r} = \begin{cases} 0, & \text{if } r < 1 \\ 2^i & \text{if } 2^i \leq r < (2^i + 2^{i+1})/2, 0 \leq i \leq k \\ 2^{i+1}, & \text{if } (2^i + 2^{i+1})/2 \leq r < 2^{i+1}, 0 \leq i \leq k \end{cases}$$

We want to find a constant ρ, such that if $t = t_i/\rho$, then the LP must use at least $(2^i + 2^{i+1})/2 = 3(2^{i-1})$ units of resources.

In the full paper [10] we show that r can be expressed as follows.

$$r = 2\left(1 - \frac{1}{x}t\right) + \sum_{j=1}^{i+1}\left(2^j - \frac{2^j}{t_j}t\right) = 8 \cdot (2^{i-1}) - \frac{t_i}{\rho}\left(2/x + \sum_{j=1}^{i+1}\frac{2^j}{t_j}\right)$$

Since we want to have $r \geq 3(2^{i-1})$, we want to find the smallest value of ρ such that

$$\frac{t_i}{\rho}\left(\frac{2}{x} + \sum_{j=1}^{i+1}\frac{2^j}{t_j}\right) \leq 5 \cdot (2^{i-1}) \Rightarrow \quad \rho \geq 1/5\left(\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1}\frac{t_i}{2^{i-j-1}t_j}\right).$$

Our full paper [10] shows that $\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1}\frac{t_i}{2^{i-j-1}t_j} < 14$.

So, by setting $\rho = 14/5$, we get $\rho > 1/5\left(\frac{t_i}{2^{i-2}x} + \sum_{j=1}^{i+1}\frac{t_i}{2^{i-j-1}t_j}\right)$.

The computation above implies the following lemma.

Lemma 2.5. *If the LP uses $2^i \leq r < 3(2^{i-1})$ units of resources and we round r down to $\overline{r} = 2^i$ where $0 \leq i \leq k$, then $t_i \leq (14/5)t$ where t is the duration from the LP solution.*

We can also show the following (details in [10]).

Lemma 2.6. *If the LP uses $r < 1$ unit of resource and we round r down to 0, then $t_i \leq 2t$, where t is the duration from the LP solution.*

Lemma 2.7. *If r is rounded to $\overline{r}$ then $\overline{r} \leq (4/3)r$*

From Lemmas 2.5, 2.6 and 2.7, we get the following theorem.

Theorem 2.8. *There is a $(4/3, 14/5)$ bi-criteria approximation algorithm for the discrete resource-time tradeoff problem with resource reuse along paths when the recursive binary duration function is used.*

2.3 Single-criteria Approximation for k-Way and Recursive Binary Splitting

First, observe that Section 2.1 gives a bi-criterian approximation for both k-way and recursive binary splitting. Setting $\alpha = 1/2$ in Theorem 2.4, we obtain a $(2, 2)$ bi-criteria approximation. Now, after LP rounding, say a job j uses $\overline{r_j}$ units of resource and takes $\overline{t_j}$ units of time. Then the optimal solution uses $r_j^* \geq \overline{r_j}/2$ units of resource and takes $t_j^* \geq \overline{t_j}/2$ units of time for job j. Recall that job j consists of l_j parallel jobs j_i where $1 \leq i \leq l_j$. Hence, $\overline{r_j}$ is the sum of the resource (after rounding) used by l_j parallel jobs and $\overline{t_j}$ is the maximum time (after rounding) taken by l_j parallel jobs.

Approximation algorithm for k-way splitting. To obtain a single-criteria approximation, in the case of k-way splitting, we use at most r_j^* units of resource for job j. If $\overline{r_j} > r_j^*$, we reduce $\overline{r_j}$ to k (a nonnegative integer) units of resource such that $k \leq r_j^*$. Using k units of resource, job j takes $t_j(k)$ units of time to complete.

Lemma 2.9. $\lceil d/k \rceil + k \leq 2.5\overline{t_j}$ *for* $\overline{r_j} > 3$ *where* $d = t_j(0)$ *and* $k = \lfloor \overline{r_j}/2 \rfloor$.

Proof. Since $k = \lfloor \overline{r_j}/2 \rfloor \geq \overline{r_j}/2.5$ for $\overline{r_j} > 3$, we have $\lceil d/k \rceil \leq d/k + 1 \leq 2.5d/\overline{r_j} + 1 \leq 2.5\lceil d/\overline{r_j} \rceil + 1$. Also since $k = \lfloor \overline{r_j}/2 \rfloor \leq \overline{r_j} + 1$ and $2.5\overline{r_j} \geq \overline{r_j} + 2$ for $\overline{r_j} > 3$, we have $\lceil d/k \rceil + k \leq 2.5\lceil d/\overline{r_j} \rceil + 1 + \overline{r_j} + 1 \leq 2.5\left(\lceil d/\overline{r_j} \rceil + \overline{r_j}\right)$. Hence, $t_j(k) \leq 2.5\overline{t_j}$. □

Lemma 2.10. *If* $\overline{r_j} > 3$ *then* $t_j(k) \leq 5t_j^*$.

Proof. We know $t_j(k) = \lceil d/k \rceil + k$ as $k \geq 4$. Also in Lemma 2.9, we prove $t_j(k) \leq 2.5\overline{t_j}$. However, we show that $\overline{t_j} \leq 2t_j^*$. Hence, combining these two results we get $t_j(k) \leq 5t_j^*$. □

Lemma 2.11. *If* $t_j^* = d/4$ *then* $r_j^* \geq 2$.

Proof. Recall that in D'', job j is represented as l_j parallel jobs j_i where $1 \leq i \leq l_j$. The resource-time tuples of jobs j_1 and j_2 are $\{\langle 0, d\rangle, \langle 2, 0\rangle\}$ and $\{\langle 0, \lceil d/2 \rceil + 2\rangle, \langle 1, 0\rangle\}$, respectively. To attain $d/4$ duration, j_1 requires at least 3/2 units of resource and job j_2 requires 1/2 unit of resource (applying linear relaxation). Hence, $r_j^* \geq (3/2 + 1/2) = 2$ units of resource to achieve $t_j^* = d/4$. □

Lemma 2.12. *If* $\overline{r_j} \leq 3$ *then* $t_j(k) \leq 4t_j^*$.

Proof. If $\overline{r_j} \leq 3$ and $r_j^* < 2$, then we round down $\overline{r_j}$ to $k = 0$. So, from Lemma 2.11 it follows that after rounding down to 0 unit of resource, job j takes $d \leq 4t^*$ units of time.

If $\overline{r_j} \leq 3$ and $r_j^* \geq 2$, then we round $\overline{r_j}$ to $k = 2$. It is true that $t_j(2) \leq 2t_j(3)$ because $(\lceil d/2 \rceil + 2) \leq 2(\lceil d/3 \rceil + 3)$. Also, $t_j(3) \leq t_j(\overline{r_j}) \leq 2t_j^*$. Combining this two results we get $t_j(2) \leq 4t^*$. □

So, now we have the following result.

Theorem 2.13. *There is a 5-approximation algorithm for the minimum-makespan problem with k-way splitting duration function.*

Proof. Combining Lemmas 2.12 and 2.10 we get $t_j(k) \leq 5t_j^*$ for all valid $\overline{r_j}$. This proves that the makespan is at most 5 times the optimal solution. We now calculate the total amount of resource required to flow from the source of D'. We compute a min-flow in D' where k is the resource requirement for job j. Note that we are now working on D' that does not have l_j parallel chains for job j. Let f be the min flow from the source of D' such that all the resource requirements are met. The flow f^* from the LP solution before rounding is also a valid flow for the resource requirement k for job j as $k \leq r_j^*$. We know that min-flow gives an optimal integral solution. Hence, $f \leq f^*$. □

Approximation algorithm for recursive binary splitting. We have the following result.

RIGHTSLINK

Theorem 2.14. *There is a 4-approximation algorithm for the minimum-makespan problem with recursive binary splitting function.*

Proof. As in the case of k-way splitter, to get a single-criteria approximation, we use no more than r_j^* units of resource for job j. If $\overline{r_j} > r_j^*$, we reduce $\overline{r_j}$ to $\overline{r_j}/2$. We know that $t_j(\overline{r_j}/2) \leq 2t_j(\overline{r_j})$ from the properties of the recursive binary splitting function. Thus, $t_j(\overline{r_j}/2) \leq 2t_j(\overline{r_j}) \leq 4t_j(r_j^*) = 4t_j^*$. □

2.4 Exact Algorithm for Series-Parallel Graphs

We consider now the special case in which the underlying DAG D is a series-parallel graph. A series-parallel graph G can be transformed into (and represented as) a rooted binary tree T_G in polynomial time by decomposing it into its atomic parts according to its series and parallel compositions (see, e.g., [26]). In T_G, the leaves correspond to the vertices of G. Internal nodes of T_G are labeled as "s" or "p" based on series or parallel composition. We associate each internal node v of T_G with the series-parallel graph G_v, induced by the leaves of the subtree rooted at v.

Let $T(v, \lambda)$ denote the makespan of G_v using $0 \leq \lambda \leq B$ units of resources where B is the resource budget. We want to solve for $T(s, B)$, where s is the root of T_G. This can be done using dynamic programming, solving for the leaves first, and then progressing upward to the root of T_G. We compute $T(v, \lambda)$ as follows which assumes that node v corresponds to job j if it is a leaf, otherwise it has two children v_1 and v_2.

$$T(v,\lambda) = \begin{cases} t_j(\lambda) & \text{if } v \text{ is a leaf} \\ T(v_1,\lambda) + T(v_2,\lambda) & \text{if } v \text{ is an internal node with label "}s\text{"} \\ min_{0\leq i\leq\lambda}\left\{max\begin{Bmatrix} T(v_1,i), \\ T(v_2,\lambda-i) \end{Bmatrix}\right\} & \text{if } v \text{ is an internal node with label "}p\text{"} \end{cases}$$

There are $O(m)$ nodes in T_G if G has m edges. For each node v we compute $T(v, \lambda)$ for $0 \leq \lambda \leq B$. Computing $T(v, \lambda)$ for any particular value of λ takes $O(\lambda)$ time, since, if the node is a "p" node, then for $0 \leq i \leq \lambda$ we need to look up values $T(v_1, i)$. Thus, for any internal node v, it takes $\sum_{\lambda=0}^{B} O(\lambda) = O\left(B^2\right)$ time. As there are $O(m)$ nodes in T_G, the (pseudo-polynomial) time complexity of the algorithm is $O\left(mB^2\right)$.

3 NP-HARDNESS

In this section we give a variety of NP-hardness and inapproximability results related to the discrete time-resource tradeoff problem in the offline setting (i.e., when the entire DAG is available offline). All problems consider the version where there is resource reuse over paths, but they vary the cost-function, graph structure, and minimization goal. Section 3.1 gives several reductions from 1-in-3SAT. Theorem 3.1 gives a base reduction for the problem with general non-increasing duration function which will provide the ideas and structure for later more complex proofs. Theorems 3.3 and 3.4 adapt this proof to give constant factor inapproximability for the minimum-resource and minimum-makespan problems. Theorems 3.5 and 3.6 adapt the NP-hardness proof to apply when the duration functions are restricted to be the recursive binary splitting and the k-way splitting. Section 3.2 shows weak NP-hardness in bounded treewidth graphs by a reduction from Partition.

3.1 Reuse Over a Path with General Non-increasing Duration Function

Theorem 3.1. *It is (strongly) NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T.*

Our proof is based on a polynomial-time reduction from the strongly NP-hard problem 1-in-3SAT [32]: Given n variables (V_i, $1 \leq i \leq n$) and m clauses (C_j, $1 \leq j \leq m$), with each clause a disjunction of three literals, is there a truth assignment to the variables such that each clause has exactly one true literal?

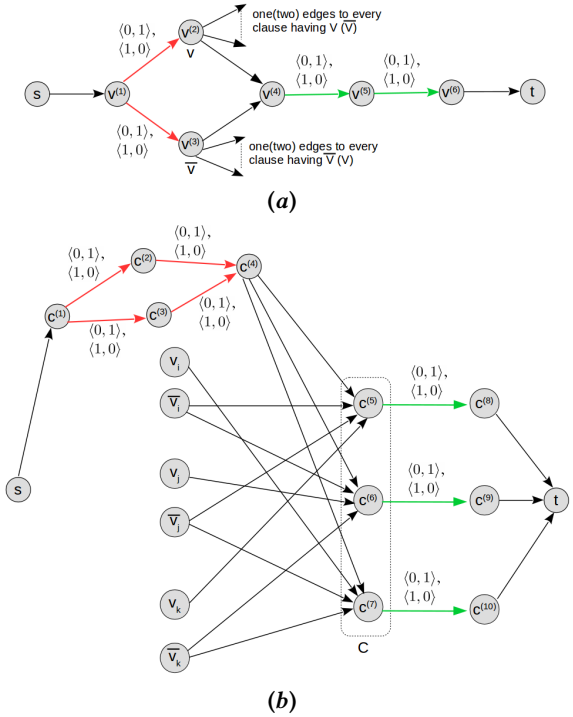

Figure 6: (*a*) Gadget for variable V, and (*b*) gadget for clause $C = (V_i \vee V_j \vee V_k)$ (Section 3.1).

Variable gadget. The gadget for variable V consists of nodes $V^{(1)}$, $V^{(2)}$, $V^{(3)}$, $V^{(4)}$, $V^{(5)}$, and $V^{(6)}$ as shown in Figure 6(a). We show in the hardness proof that a variable gadget will get exactly one unit of extra resource, otherwise the makespan will be greater than the target makespan of 1. Sending one unit of resource to node $V^{(2)}$ (Figure 6(a)) corresponds to setting the variable V to True and sending the unit of resource to $V^{(3)}$ corresponds to setting V to False. The remaining nodes ensure the extra resource is used in the variable and not transferred into one of the clauses.

Clause gadget. The gadget corresponding to clause C has 10 nodes $C^{(i)}$ ($1 \leq i \leq 10$) as shown in Figure 6(b). Edges $(C^{(1)}, C^{(2)})$, $(C^{(2)}, C^{(4)})$, $(C^{(1)}, C^{(3)})$ and $(C^{(3)}, C^{(4)})$ have resource-time pairs

RIGHTSLINK

as $\{\langle 0,1\rangle, \langle 1,0\rangle\}$. If clause C has three literals V_i, V_j and V_k, then node $C^{(5)}$ is connected to the nodes $V_i^{(3)}, V_j^{(3)}$ and $V_k^{(2)}$. These nodes correspond to $\neg V_i, \neg V_j$ and V_k, respectively. Node $C^{(6)}$ is connected to $V_i^{(3)}, V_j^{(2)}$ and $V_k^{(3)}$ which correspond to $\neg V_i, V_j$ and $\neg V_k$, respectively. Node $C^{(7)}$ is connected to $V_i^{(2)}, V_j^{(3)}$ and $V_k^{(3)}$. These nodes correspond to $V_i, \neg V_j$ and $\neg V_k$, respectively. Edges $(C^{(5)}, C^{(8)})$, $(C^{(6)}, C^{(9)})$, and $(C^{(7)}, C^{(10)})$ have resource-time pairs as $\{\langle 0,1\rangle, \langle 1,0\rangle\}$. The part of the clause gadget consisting of $C^{(1)}$, $C^{(2)}$, $C^{(3)}$ and $C^{(4)}$ demand at least two units of memory be allocated there and then these units of resource go to satisfy two of $C^{(5)}$, $C^{(6)}$ and $C^{(7)}$. There is still one of these lines that has no allocated resource so its cost is 1. Thus, to be satisfied, the corresponding variable must have had its path length reduced (by setting it to True).

See the full paper [10] for a complete construction from a 1-in-3SAT formula.

Lemma 3.2. *There exists a solution to the input instance of 1-in-3SAT iff there exists a valid flow of resources through the DAG achieving a makespan of* 1 *under a resource bound of* $B = n + 2m$.

Proof. **Forward direction.** We prove that if there is a solution to the 1-in-3SAT instance with n variables and m clauses, then the reduced DAG has a solution of makespan 1 with $(n + 2m)$ units of resource. If a variable V's truth assignment is True, then we allow one unit of resource to flow through node $V^{(2)}$ along the path $\langle S, V^{(1)}, V^{(2)}, V^{(4)}, V^{(5)}, V^{(6)}, T\rangle$, otherwise we allow one unit of resource to flow through node $V^{(3)}$ along the path $\langle S, V^{(1)}, V^{(3)}, V^{(4)}, V^{(5)}, V^{(6)}, T\rangle$. For every clause C, we allow one unit of resource to flow through the path $\langle S, C^{(1)}, C^{(2)}, C^{(4)}\rangle$ and another unit of resource through the path $\langle S, C^{(1)}, C^{(3)}, C^{(4)}\rangle$. Thus, 2 units of resource can flow from node C^4. In a valid assignment of 1-in-3SAT, for each clause C, exactly 2 nodes of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ will have the earliest start time of 1 and the other one will have 0 (Table 2).

Also, if only one literal is True in a clause, exactly two nodes among $C^{(5)}, C^{(6)}$ and $C^{(7)}$ need one unit of extra resource each to meet the makespan requirement (from Table 2). We are allowed to flow 2 units of resource from node $C^{(4)}$. Thus the project makespan is 1 using $(n + 2m)$ units of resource.

Backward direction. Now, we prove that if there exists a solution of makespan 1 using $(n + 2m)$ units of resource in the reduced DAG, then there also exists a solution to the 1-in-3SAT instance. To achieve a makespan of 1, every variable gadget needs 1 unit of resource and each clause gadget needs 2 units of resource, otherwise the makespan would be greater than 1. Also, any resource that is used in a variable gadget cannot be used further in any other variable or clause gadget because the resource can be reused over a path only. Similarly, any resource that is used in any clause gadget, cannot be reused in any other gadget. Only one node that is either $V^{(2)}$ or $V^{(3)}$, will have the earliest start time 0. Both cannot be 0, as there is only 1 unit of resource per variable gadget. Both cannot be 1 as in a clause C where the literal V or $\neg V$ is present, each of $C^{(5)}, C^{(6)}$ and $C^{(7)}$ would have earliest starting time of 1. This requires use of 3 units of resource in the clause gadget C to achieve a makespan of 1. However, each clause gadget can have exactly 2 units of resource. Thus, for every variable, it has to be a valid assignment (V is set to either True or False). From Table 2, if a clause has exactly one True literal, then the clause gadget requires 2 units of resource to achieve a makespan of 1. Otherwise, the clause gadget would have a makespan of 2 with the same amount of resource or would require more resource to achieve the target makespan of 1. Thus, each clause has exactly one True literal. This satisfies the 1-in-3SAT instance. □

V_i	V_j	V_k	$C^{(5)}$	$C^{(6)}$	$C^{(7)}$
True	True	True	$max(1,1,0) = 1$	$max(1,0,1) = 1$	$max(0,1,1) = 1$
False	True	True	$max(0,1,0) = 1$	$max(0,0,1) = 1$	$max(1,1,1) = 1$
True	False	True	$max(1,0,0) = 1$	$max(1,1,1) = 1$	$max(0,0,1) = 1$
True	True	False	$max(1,1,1) = 1$	$max(1,0,0) = 1$	$max(0,1,0) = 1$
False	False	True	$max(0,0,0) = 0$	$max(0,1,1) = 1$	$max(1,0,1) = 1$
False	True	False	$max(0,1,1) = 1$	$max(0,0,0) = 0$	$max(1,1,0) = 1$
True	False	False	$max(1,0,1) = 1$	$max(1,1,0) = 1$	$max(0,0,0) = 0$
False	False	False	$max(0,0,1) = 1$	$max(0,1,0) = 1$	$max(1,0,0) = 1$

Table 2: Makespan at nodes $C^{(5)}$, $C^{(6)}$ and $C^{(7)}$ for different truth value assignments to V_i, V_j and V_k in Figure 6(b).

We also prove hardness of approximation, both for the minimum-makespan problem and for the minimum-resource problem. We note that the minimum-makespan result follows from the prior construction since a satisfying answer to the reduction results in a makespan of 1 and any non-satisfying answer returns a makespan of at least 2.

Theorem 3.3. *The minimum-makespan discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than* 2 *unless* $P = NP$.

For the minimum-resource problem, we need substantial changes to the proof to get a hardness of approximation. The key idea is to put variable and clause gadgets in sequence, allowing the resources to be resued many times, a satifying solution needing only 2 units of resources and a non-satisfying solution requiring at least 3. A detailed proof is given in the full paper [10].

Theorem 3.4. *The minimum-resource discrete resource-time tradeoff problem that allows resources to be reused only over paths cannot have a polynomial-time approximation algorithm with approximation factor less than* 3/2 *unless* $P = NP$.

Finally, the full version of the paper [10] shows how to adapt the framework in this section to show NP-hardness for the recursive binary and k-way splitting costs. Adapting to these cost functions requires significantly more infrastructure and care in choosing values for the cost functions, but uses the same overall structure for the reduction.

Theorem 3.5. *It is (strongly) NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a recursive binary splitting function, satisfying a resource bound* B *and a makespan bound* T.

Theorem 3.6. *It is (strongly) NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a* k*-way splitting function, satisfying a resource bound* B *and a makespan bound* T.

RIGHTSLINK

3.2 Underlying Bounded Treewidth Graph

The proof of the following theorem is based on a reduction from Partition[18]. A complete proof appears in the full paper [10].

Theorem 3.7. *It is weakly NP-hard to decide if there exists a solution to the (offline) discrete resource-time tradeoff problem, with resource reuse over paths and a non-increasing duration function, satisfying a resource bound B and a makespan bound T, provided the undirected graph obtained by ignoring the directedness of the edges of the input DAG is of bounded treewidth.*

4 CONCLUSION

In this paper we introduce the discrete resource-time tradeoff problem with resource reuse in which each unit of resource is routed along a source to sink path and is possibly used and reused to expedite activities encountered along that path. We assume that a general duration function (i.e., time needed to complete an activity as a function of the amount of resources used) is associated with each activity. We consider two different objective functions: (1) optimize makespan given a limited resource budget and (2) optimize resource requirement given a target makespan.

Our original motivation came from a desire to mitigate the cost of data races in shared-memory parallel programs by using extra space to reduce the time it takes to perform conflict-free write operations to shared memory locations. We consider three duration functions: general non-increasing function for the general resource-time question, and recursive binary reduction and multiway (k-way) splitting for the space-time case.

We present the first hardness and approximation hardness results as well as the first approximation algorithms for our problems. We show that the makespan optimization problem is strongly NP-hard under all three duration functions. When the duration function is general non-increasing we also show that it is strongly NP-hard to achieve an approximation ratio less than 2 for the makespan optimization problem and less than $\frac{3}{2}$ for the resource optimization problem. We give a $\left(\frac{1}{\alpha}, \frac{1}{1-\alpha}\right)$ bi-criteria (resource, makespan) approximation algorithm for that same duration function, where $0 < \alpha < 1$. We present an improved $\left(\frac{4}{3}, \frac{14}{5}\right)$ bi-criteria approximation algorithm for the recursive binary reduction function. We also give 4- and 5-approximation algorithms for the makespan optimization problem under the recursive binary reduction function and the multiway (k-way) splitting function, respectively.

REFERENCES

[1] Google gperftools. Fast, multi-threaded malloc() and nifty performance analysis tools. http://code.google.com/p/gperftools/.

[2] llalloc. Lockless memory allocator. http://locklessinc.com/.

[3] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 451–469.

[4] Can Akkan, Andreas Drexl, and Alf Kimms. 2005. Network decomposition-based benchmark results for the discrete time–cost tradeoff problem. *European Journal of Operational Research* 165, 2 (2005), 339–358.

[5] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.

[6] Eric Blayo, Laurent Debreu, Gregory Mounie, and Denis Trystram. 1999. Dynamic load balancing for ocean circulation model with adaptive meshing. In *European Conference on Parallel Processing*. Springer, 303–312.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55–69.

[8] OpenMP Architecture Review Board. 1997. OpenMP: A proposed industry standard API for shared memory programming. *White Paper* (1997). url: http://www.openmp.org/specs/mp-documents/paper/paper.ps.

[9] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science & Engineering* 1 (1998), 46–55.

[10] Rathish Das, Shih-Yu Tsai, Sharmila Duppala, Jayson Lynch, Esther M. Arkin, Rezaul Chowdhury, Joseph S. B. Mitchell, and Steven Skiena. 2019. Data races and the discrete resource-time tradeoff problem with resource reuse over paths. *arXiv preprint arXiv:1904.09283* (2019).

[11] Prabuddha De, E. James Dunne, Jay B. Ghosh, and Charles E. Wells. 1997. Complexity of the discrete time-cost tradeoff problem for project networks. *Operations Research* 45, 2 (1997), 302–306.

[12] Jianzhong Du and Joseph Y-T Leung. 1989. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics* 2, 4 (1989), 473–487.

[13] Pierre-François Dutot, Grégory Mounié, and Denis Trystram. 2004. Scheduling parallel tasks: Approximation algorithms.

[14] Mingdong Feng and Charles E. Leiserson. 1999. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

[15] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 79–90.

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices* 33, 5 (1998), 212–223.

[17] Delbert R. Fulkerson. 1961. A network flow computation for project cost curves. *Management Science* 7, 2 (1961), 167–178.

[18] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., NY, USA.

[19] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 317–328.

[20] Klaus Jansen and Hu Zhang. 2006. An approximation algorithm for scheduling malleable tasks under general precedence constraints. *ACM Transactions on Algorithms (TALG)* 2, 3 (2006), 416–434.

[21] James E. Kelley Jr. 1961. Critical-path planning and scheduling: Mathematical basis. *Operations Research* 9, 3 (1961), 296–320.

[22] James E. Kelley Jr. and Morgan R. Walker. 1959. Critical-path planning and scheduling. In *papers presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*. ACM, 160–173.

[23] Jan Karel Lenstra and A. H. G. Rinnooy Kan. 1978. Complexity of scheduling under precedence constraints. *Operations Research* 26, 1 (1978), 22–35.

[24] Renaud Lepère, Grégory Mounié, and Denis Trystram. 2002. An approximation algorithm for scheduling trees of malleable tasks. *European Journal of Operational Research* 142, 2 (2002), 242–249.

[25] Renaud Lepere, Denis Trystram, and Gerhard J. Woeginger. 2002. Approximation algorithms for scheduling malleable tasks under precedence constraints. *International Journal of Foundations of Computer Science* 13, 04 (2002), 613–627.

[26] Rolf H. Möhring. 1989. Computationally tractable classes of ordered sets. In *Algorithms and Order*. Springer, 105–193.

[27] Robert H. B. Netzer and Barton P. Miller. 1992. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 1 (1992), 74–88.

[28] D. Panagiotakopoulos. 1977. A CPM time-cost computational algorithm for arbitrary activity cost functions. *INFOR: Information Systems and Operational Research* 15, 2 (1977), 183–195.

[29] Steve Phillips Jr. and Mohamed I. Dessouky. 1977. Solving the project time/cost tradeoff problem using the minimal cut concept. *Management Science* 24, 4 (1977), 393–400.

[30] James Reinders. 2007. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc.

[31] Don R. Robinson. 1975. A dynamic programming solution to cost-time tradeoff for CPM. *Management Science* 22, 2 (1975), 158–166.

[32] Thomas J. Schaefer. 1978. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*. ACM, 216–226.

[33] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management*. ACM, 84–94.

[34] Nir Shavit. 2011. Data structures in the multicore age. *Commun. ACM* 54, 3 (2011), 76–84.

[35] Martin Skutella. 1998. Approximation algorithms for the discrete time-cost tradeoff problem. *Mathematics of Operations Research* 23, 4 (1998), 909–929.

[36] John Turek, Joel L. Wolf, and Philip S. Yu. 1992. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 323–332.

RIGHTSLINK