Analysis of Work-Stealing and Parallel Cache Complexity

Yan Gu
UC Riverside
ygu@cs.ucr.edu

Zachary Napier
UC Riverside
znapi001@ucr.edu

Yihan Sun
UC Riverside
yihans@cs.ucr.edu

ABSTRACT

Parallelism has become extremely popular over the past decade, and there have been a lot of new parallel algorithms and software. The randomized work-stealing (RWS) scheduler plays a crucial role in this ecosystem. In this paper, we study two important topics related to the randomized work-stealing scheduler.

Our first contribution is a simplified, classroom-ready version of analysis for the RWS scheduler. The theoretical efficiency of the RWS scheduler has been analyzed for a variety of settings, but most of them are quite complicated. In this paper, we show a new analysis, which we believe is easy to understand, and can be especially useful in education. We avoid using the potential function in the analysis, and we assume a highly asynchronous setting, which is more realistic for today's parallel machines.

Our second and main contribution is some new parallel cache complexity for algorithms using the RWS scheduler. Although the sequential I/O model has been well-studied over the past decades, so far very few results have extended it to the parallel setting. The parallel cache bounds of many existing algorithms are affected by a polynomial of the span, which causes a significant overhead for high-span algorithms. Our new analysis decouples the span from the analysis of the parallel cache complexity. This allows us to show new parallel cache bounds for a list of classic algorithms. Our results are only a polylogarithmic factor off the lower bounds, and significantly improve previous results.

APoCS'22, January 12, 2022 , Alexandria, Virginia

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Hardware advances in the last decade have brought multicore parallel machines to the mainstream. While there are multiple programming paradigms and tools to enable parallelism in multicore machines, the one based on ***nested parallelism*** with ***randomized work-stealing (RWS) scheduler*** is with no doubt the most popular and widely used. The nested parallelism model and its variants have been supported by most parallel programming languages (e.g., Cilk, TBB, TPL, X10, Java Fork-join, and OpenMP), introduced in textbooks (e.g., Cormen, Leiserson, Rivest and Stein [43]), and employed in a variety of research papers (to list a few: [6, 12, 13, 18, 22, 24, 25, 27–31, 35, 42, 44, 45, 55, 72**? ?** –74]). At a high level, this model allows an algorithm to recursively and dynamically create (*fork*) parallel tasks, which will be executed on P processors by a dynamic scheduler. This nested (binary) fork-join provides a good abstraction for shared-memory parallelism. On the user (algorithm designer or programmer) side, it is a simple extension to the classic programming model with additional keywords for creating new tasks (e.g., `fork`) and synchronization (e.g., `join`) between tasks.

The ***randomized work-stealing (RWS) scheduler*** plays a crucial role in this ecosystem. It automatically and dynamically maps a nested parallel algorithm to the hardware efficiently both in theory and in practice. In this paper, we study two important topics related to the RWS scheduler. First, we show a ***new, simplified, and classroom-ready version of analysis for the RWS scheduler***, which we believe is easier to understand than existing ones. Second, we provide some ***new analyses for parallel cache complexity*** for nested-parallel algorithms based on the RWS scheduler. We provide a list of almost optimal bounds listed in Table 1, and more discussions will be given later.

Our first contribution is a simplified analysis for the RWS scheduler. The theoretical efficiency of the RWS scheduler was first given by Blumofe and Leiserson [32], and is later analyzed for a variety of settings (to list a few: [1–3, 10, 12, 19, 63, 69, 70]). Although these analyses essentially consider more complex settings (e.g., to also consider external I/Os), the analyses themselves are quite complicated. To the best of our knowledge, the details of the proofs are covered in very

few courses related to parallelism, and in most cases, RWS is just treated as a black box. Hence, we simply consider the goal of bounding the number of steals of the RWS scheduler, and we want to answer the question of what the simplest analysis for the RWS scheduler can be, or what is the most comprehensible version in education.

This paper presents a simplified analysis in Section 3. Unlike most of the existing analyses, our version does not rely on defining the potential function for a substructure of the computation, which we believe is easy to understand. Our analysis is inspired by a recent analysis [19] that is similar to [7, 71]. Our analysis differs from [19] in two aspects. First, unlike [19], we do not assume all processors run in lock-steps (the PRAM setting). Instead, we assume a highly asynchronous setting, which is more realistic for today's parallel machines. Second, we separate the math calculation from the details of the RWS algorithm, which may be helpful for classroom teaching.

Our second and main contribution of this paper is on the parallel cache complexity for algorithms using the RWS scheduler. On today's machines, the memory access cost usually dominates the running time of most combinatorial algorithms. To capture this, sequentially, Aggarwal and Vitter [5] first formalized the external-memory model to capture the I/O cost of an algorithm, which was refined by Frigo et al. [48] as the ideal-cache model. The cost measure is called *I/O complexity* [5] (noted as Q_1) or *cache complexity* [48] when specifying the communication cost between the cache and the main memory. While this model has received great success in the algorithm and database communities, so far, few results have extended it to the parallel (distributed cache) setting, which we summarize in Section 2.4. Among them, Acar, Blelloch and Blumofe [2] first defined and showed that the parallel cache complexity Q_P is at most $Q_1 + O(PDM/B)$, where P, D, M and B are the number of processors, span (aka. depth, the longest critical path of dependences), cache size, and cache block size, respectively (definitions in Section 2). However, this bound is pessimistic and usually too loose. Frigo and Strumpen [50] and later work by Cole and Ramachandran [39, 40] showed tighter parallel cache complexity for many cache-oblivious algorithms summarized Table 1. However, the bounds have a polynomial overhead[1] on the input size when the span is polynomial to the input size. Such overhead can easily dominate the cache bound when plugging in the real-world input size, as discussed in the caption of Table 1. It remained an open question on how to close this gap.

In this paper, we significantly close this gap for a variety of algorithms. The polynomial overhead in the previous analysis [2, 39, 40, 50] is due to the high span of these algorithms (a polynomial of the algorithm's span shows up in the bound). The key insight in our analysis is to break such polynomial correlation between the algorithm's span and the overhead for parallel cache complexity. Our new analysis is inspired by the abstraction of k-d grid proposed recently by Blelloch and Gu [26], which was previously used to show sequential cache bounds. We extend the idea for analyzing parallel cache complexity. Our analysis directly studies the "real" dependencies in the computation structure of these algorithms, instead of those caused by parallelism. In particular, the core of our analysis is just the *recurrences* of these algorithms. As a result, the parallel dependency of the computation (the algorithm's span) does not show up in the analysis. This helps us avoid the complication of digging into the details of the scheduling algorithms in the analysis, and the analysis is no more complicated than just solving recurrences. To do this, we define the (α, β, k, l, m)-recurrence, which effectively models the pattern of recurrences of many classic algorithms, and show a general theorem, Theorem 4.2, to solve these recurrences. By applying these results, we can achieve the new bounds in Table 1. The parallel overheads are polylogarithmic instead of polynomial to the input size, compared to the lower bounds by [11, 26]. We believe the methodology is of independent interest, and could be useful in analyzing other parallel or sequential algorithms. We leave this as future work.

The contributions of this paper are as follows.

(1) We show a new analysis for the randomized work-stealing scheduler, which avoids the use of potential functions and is very simple.
(2) We propose the (α, β, k, l, m)-recurrence and a general theorem (Theorem 4.2) for solving it, which applies to solving the parallel cache complexity for many classic parallel algorithms.
(3) We show new cache complexity bounds for a variety of algorithms, shown in Table 1, which significantly improves existing results, and is very close to the lower bounds (only a polylogarithmic overhead).

2 PRELIMINARIES

2.1 Nested Parallelism

The nested parallelism model is a programming model for shared-memory parallel algorithms. This model allows algorithms to recursively and dynamically create new parallel tasks (threads). The computation will be simulated (scheduled) on P loosely synchronized processors, and explicit synchronization can be used to let threads reach consensus. Some commonly-used examples include the (binary) fork-join model and binary forking model. More precisely,

[1] Such an overhead is usually n^a, where a is a constant and $1/2 \le a < 1$.

Algorithm	Seq. Bound	Parallel Overhead: New in this paper	Parallel Overhead: Previous best [26, 39, 50]	Parallel Overhead: Lower Bound [11, 26]
Gaussian Elimination Kleene's algorithm for APSP	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{2/3} n \cdot \frac{n^2}{B} + Pn$	$P^{1/3}n^{1/3} \cdot \frac{n^2}{B} + Pn\log B$	$P^{1/3} \cdot \frac{n^2}{B}$
Triangular System Solver	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn$	$P^{1/3}n^{1/3} \cdot \frac{n^2}{B} + Pn\log B$	$P^{1/3} \cdot \frac{n^2}{B}$
Cholesky Factorization LU Decomposition	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn\log n$	$P^{1/3}n^{1/3}\log^{1/3} n \cdot \frac{n^2}{B} + Pn\log n$	$P^{1/3} \cdot \frac{n^2}{B}$
LWS Recurrence	$\frac{n^2}{BM} + \frac{n}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n}{B} + Pn$	$P^{1/2} \cdot n^{1/2} \cdot \frac{n}{B} + Pn$	$P^{1/2} \cdot \frac{n}{B}$
GAP Recurrence	$\frac{n^3}{BM} + \frac{n^2\log M}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2}n^{\kappa/2}\log M \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} \cdot \frac{n^2}{B}$
Parenthesis Recurrence	$\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1$	$P^{1/3}\log^{5/3} n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/3}n^{\kappa/3} \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/3} \cdot \frac{n^2}{B}$
RNA Recurrence	$\frac{n^4}{BM} + \frac{n^2}{B} + 1$	$P^{1/2}\log^2 n \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2}n^{\kappa/2} \cdot \frac{n^2}{B} + Pn^\kappa$	$P^{1/2} \cdot \frac{n^2}{B}$
Protein Accordion Folding	$\frac{n^3}{BM} + \frac{n^2}{B} + 1$	$P^{1/2}\log n \cdot \frac{n^2}{B} + Pn\log^2 n$	$P^{1/2}n^{1/2}\log n \cdot \frac{n^2}{B} + Pn\log^2 n$	$P^{1/2} \cdot \frac{n^2}{B}$

Table 1: Sequential and parallel cache complexity for a list of algorithms. Here M, B, and P are the cache size, the block size, and the number of processors, respectively, and $\kappa = \log_2 3 \approx 1.58$. The sequential bounds are from various existing papers [26, 36, 45, 48]. The parallel cache complexity is the sequential cache complexity plus the parallel overhead. The best previous parallel bounds are either from [39, 50], or what we computed based on the algorithms from [26] and the analysis from [39, 50]. The new bounds are analyzed in Sec. 4. For all algorithms/problems, the new parallel bounds are much tighter. Using LWS recurrence as an example, for the best previous bound, the parallel overhead always dominates unless $P^{1/2}n^{3/2}/B < n^2/BM$, which gives $n = \omega(PM^2)$ (at least 2^{45} when plugging in real-world values for M and P, which is impossible to store and compute). Our new bound improve the dominating term of $O(P^{1/2}n^{3/2}/B)$ by $\sqrt{n}/\log^2 n$. Similarly, for other algorithms, our new bounds replace the n^a in the dominating terms for $a = 1/2$, $1/3$, $\kappa/2$, or $\kappa/3$, with the $\log^b n$ for $b \le 2$. In all cases, the gaps between new parallel upper bounds and the lower bounds on the specific computations are $\log^b n$ for $b \le 2$, which were previously a polynomial factor n^a for $a = 1/2$, $1/3$, $\kappa/2$, or $\kappa/3$.

in this model, we assume a set of threads that have access to a shared memory. A computation starts with a single root thread and finishes when all threads finish. Each thread supports the same operations as in the sequential RAM model, but also has a `fork` instruction that forks a new child thread that can be run in parallel. Threads can synchronize with each other using some primitives. The most common synchronization is `join`, where every `fork` corresponds to a later `join`, and the `fork` and corresponding `join` are properly nested. In more general models (e.g., the binary-forking model [23]) threads are also allowed to be synchronized with any other threads, probably by using atomic primitives such as `test_and_set` or `compare_and_swap`. This model is the most widely-used model for multicore programming, and is supported by many parallel languages including NESL [17], Cilk [49], the Java fork-join framework [58], OpenMP [65], X10 [34], Habanero [33], Intel Threading Building Blocks [57], the Task Parallel Library [76], and many others. In this model, we usually require the computation to be either ***race-free*** [46] (i.e., no logically parallel instructions access the same memory location and at least one is a write), or to only use atomic operations (e.g., `test_and_set` or `compare_and_swap`) to deal with concurrent writes (e.g., [21]).

2.2 Work-Span Measure

For a computation using nested parallelism, we can measure its *work* and *span* by evaluating its series-parallel computational DAG (i.e., a DAG modeling the dependence between operations in the computation). The ***work*** W is the number of operations in this computation, or the costs of all tasks in the computation DAG (the time complexity on the RAM model). The ***span*** (or ***depth***) D is the maximum number of operations over all directed paths in the computation DAG.

2.3 Randomized Work-Stealing (RWS) Scheduler

In practice, a nested-parallel computation can be scheduled on multicore machines using the ***randomized work-stealing (RWS)*** algorithm. More details of the RWS scheduler can be found in [32], and here we overview the high-level ideas that will be used in our analysis. The RWS scheduler assigns one double-ended queue (***deque***) for each processor that can

execute a thread at a time. One processor starts with taking the root thread. Each processor then proceeds as follows:

- If the current thread runs a `fork`, the processor enqueues one thread at the front of its queue (spawned child), and executes the other thread (continuation).
- If the current thread completes, the processor pulls a thread from the front of its own queue.
- If a processor's queue is empty, it randomly selects one of the other processors, and steals a thread from the end of that processor's queue (victim queue). If that fails, the processor retries until succeeds.

Since a steal can be pretty costly (involves complicated interprocessor communication) in practice, a common practice is to wait for at least the time for a successful steal before retrying [2].

The overhead of executing the computation based on the RWS scheduler is mainly on the steal attempts (both successful ones and failed ones), plus maintaining the deque. Hence, bounding the number of steals is of great interest for multicore parallelism.

Theorem 2.1. *Executing a series-parallel computation DAG with work W and span D on an RWS scheduler uses $O(PD)$ steals* whp *to W,[2] where P is the number of processors.*

Theorem 2.1 was first shown by Blumofe and Leiserson [32], and was later proved in different papers (e.g.,[1–3, 10, 12, 19, 63, 69, 70]). Since RWS is an asynchronous algorithm, certain synchronization assumptions are required. Early work [32] assumed all processors are fully synchronized and all operations have unit cost. Later work [2] relaxed it (and thus made it more realistic) that a steal attempt takes at least s and at most ks time steps where s is the cost for a steal and $k \geq 1$ is a constant. In this paper, we further relax the assumption of synchronization—we assume that, between a failed steal and the next steal attempt of each processor, every other processor can try at most k steal attempts for some constant $k \geq 1$. Also, between two steal attempts from the same processor, another processor that has work to do will execute at least one instruction. We believe such a relaxation is crucial since, in practice, processors are highly asynchronous due to various reasons, including cache misses, processor pipelines, branch prediction, hyper-threading, changing clock speeds, interrupts, or the operating system scheduler. Hence, it is hard to define what time steps mean for different processors. However, it is reasonable to assume that processors run in similar speeds within a constant factor, and the steal attempts are not too often (in practice the gap between two steal attempts is usually set to be at least hundreds of to thousands of cycles). In the analysis, we consider the simpler case for $k = 1$, but it is easy to see that a larger k will not asymptotically affect the scheduling result (Theorem 2.1) as long as k is a constant.

[2]We use the term $O(f(n))$ with high probability (*whp*) in n to indicate the bound $O(kf(n))$ holds with probability at least $1 - 1/n^k$ for any $k \geq 1$. With clear context we drop "in n".

2.4 Cache Complexity

Cache complexity (aka. I/O complexity) measures the memory access cost of an algorithm, which in many cases can be the bottleneck of the execution time, especially for parallel combinatorial algorithms. The idea was first introduced by Aggarwal and Vitter [5], and has been widely studied since then. Here we use definition by Frigo et al. [48] that is the most adopted now. Here we assume a two-level memory hierarchy. The CPU is connected to a small-memory (cache) of size M, and this small-memory is connected to a large-memory (main memory) of effectively infinite size. Both small-memory and large-memory are divided into blocks of size B (cachelines), so there are M/B cachelines in the cache. The CPU can only access the memory on blocks resident in the cache and it is free of charge. Finally, we assume an optimal offline cache replacement policy, which is automatic, to transfer the data between the cache and the main memory, and a unit cost for each cacheline load and evict. The practical policy such as LRU or FIFO, are $O(1)$-competitive with the optimal offline algorithm if they have a cache with twice the size. The cache complexity of an algorithm, Q_1, is the total cost to execute this algorithm on such a model.

The above measure is sequential. One way to extend it to the parallel setting is the "Parallel External Memory" (PEM) model [9] that analogs the PRAM model [67]. However, since modern processors are highly asynchronous [23] instead of running in lock-steps as assumed in PRAM, the PEM model cannot measure the communication between cache and main memory well. An alternative solution is to assume multiple processors work independently, and they either share a common cache or own their individual caches. For individual caches (aka. distributed caches), the parallel cache complexity Q_P based on RWS is upper bounded by $Q_1 + O(SM/B)$ *whp* where S is the number of steals [2]. This is easy to see since each steal can lead to, at worst, an entire reload of the cache, with cost $O(M/B)$. Applying Theorem 2.1, we can get $Q_P \leq Q_1 + O(PDM/B)$.

There have also been studies on other cache configurations, such as shared caches [24], multi-level hierarchical caches [18, 22, 68], and varying cache sizes [14, 15]. Many parallel cache-efficient algorithms have been designed inspired by these measurements (e.g., [9, 18, 20, 25, 26, 36–38, 75]).

3 SIMPLIFIED RWS ANALYSIS

The randomized work-stealing (RWS) scheduler plays a crucial role in the ecosystem of nested-parallel algorithms and

multicore platforms, and it dynamically maps the algorithms to the hardware. RWS is efficient both theoretically and practically. The theoretical efficiency of the RWS scheduler has been first given by Blumofe and Leiserson [32]. A lot of later work analyzed RWS in different settings [1–3, 10, 12, 19, 63, 69, 70]. As mentioned, most of these analyses are quite involved, especially when they also consider some more complicated settings (e.g., external I/O costs). This makes the analysis very hard to cover in undergraduate or graduate courses, and RWS is usually treated as a black box. Given the importance of the RWS scheduler, it is crucial to make the analysis more comprehensible so that it can be taught in classes. In the following, we will show a simplified analysis for RWS, which proves Theorem 2.1.

Unlike most of the existing analyses, our version does not rely on defining the potential function for a substructure of the computation. We understand that this will limit the applicability of this analysis in some settings (e.g., we do not extend the results to consider external I/O costs), but our goal is to provide a simple proof for a reasonably general setting (the binary-forking model [23]), and make it easy to understand in most parallel algorithm courses.

Our analysis is inspired by a recent analysis in [19] that is similar to those in [7, 71]. Our analysis differs from [19] in two aspects. First, the analysis in [19] assumes all processors run in lock-steps (PRAM setting). However, on today's machines, the processors are loosely synchronized with different relative processing rates changing over time. Hence, the processors can run in different speeds and do not and should not be synchronized. In our analysis, we assume, between a failed steal attempt and the next steal attempt on each processor, every other processor can make at most k steal attempts for some constant $k \geq 1$. We believe this is a reasonably realistic assumption that all today's machines and RWS implementations satisfy. For simplicity, in our analysis we use $k = 1$, and it is easy to see that this does not affect the asymptotical bounds.

Secondly, instead of showing a single long proof, we tried our best to improve its understandability, and separate the math and calculation from the main idea of the proof. We start from the most optimistic case as a motivating example in Lemma 3.1, and then generalize it to Lemma 3.2 and Lemma 3.3. The only mathematics tools we use are Chernoff bound and union bound. These lemmas pave the path to, and decouple the mathematics from, the proof of Theorem 2.1, which involves more details and the core idea about RWS. We believe this can be helpful for classroom teaching.

We say two steal attempts ***overlap*** with each other when they choose the same victim processor, and happen concurrently. To start with, we first show the simplest case where none of the steal attempts overlap with each other. This is the optimal case because in fact multiple attempts may happen simultaneously and only one can succeed. We show the lemma below as a starting point of our analysis.

Lemma 3.1. *Given a victim processor* Π *and* $(P-1)(D+\log(1/\epsilon))$ *non-overlapping steal attempts, the probability that at least* D *tasks from the deque of the processor* Π *are stolen is at least* $1-\epsilon$*, where* P *is the number of processors.*

Here we overload the notation D in the lemma that was previously defined as the span of the computation. We do so because later in the proof of Theorem 2.1, we plug in D as the span of the algorithm, so we just use D for convenience. We use the classic version of Chernoff bound that considers independent random variables $X_1, \ldots, X_t$ taking values in $\{0, 1\}$. Let X be the sum of these random variables, and let $\mu = \mathbb{E}[X]$ be the expected value of X, then for any "offset" $0 < \delta < 1$, $\Pr(X \leq (1-\delta)\mu) \leq e^{-\delta^2\mu/2}$.

Proof of Lemma 3.1. Recall that in RWS, each steal attempt independently chooses a victim processor and tries to steal a task. Hence, it has probability $1/(P-1)$ to choose processor Π (and steal one task if so). We now consider each of the steal attempts as a random variable X_i. $X_i = 1$ if it chooses processor Π, and 0 otherwise. In Chernoff Bound, let the number of random variables $t = 2(P-1)(D+\log(1/\epsilon))$. Then for t steals, the expected number of hits is $\mu = 2(D+\log(1/\epsilon))$. We are interested in the probability p that fewer than D steal attempts hit processor Π. In this case, $\delta = (\mu - D)/\mu$, so $(1-\delta)\mu = D$. Applying Chernoff bound, we can get that the probability p is no more than $e^{-\delta^2\mu/2}$. Here $\delta^2\mu/2 = (\mu-D)^2/2\mu > (\mu-2D)/2 = \ln(1/\epsilon)$. The ">" step is because we discard the $D^2/2\mu$ term that is always positive. Hence, $e^{-\delta^2\mu/2} < e^{-\ln(1/\epsilon)} = \epsilon$, which proves the lemma. □

Next, we consider the general case that multiple processors try to steal concurrently, so the steal attempts can overlap. In this case, although unlikely, many processors may make the steal attempts "almost" at the same time, where only one processor wins (using arbitrary tie-break) and the others fail. As mentioned, we assume that between a failed steal attempt and the next steal attempts on one processor, every other processor can have at most one steal attempt.

Lemma 3.2. *Given a specific victim processor* Π *and* $(P-1)e(D+\log(1/\epsilon))/(e-1)$ *steal attempts from* $P-1$ *other processors, the probability that* D *tasks from the deque of the processor* Π *are stolen is at least* $1-\epsilon$.

Proof. Although multiple processors can attempt to steal concurrently, two steals from the same processor will never be concurrent. Hence, based on our assumption, the most pessimistic situation is that the $P-1$ processors always have $P-1$ steals at the same time, which maximizes the chance that

a steal hits the queue but fails to get the task. The probability that at least one of the $P-1$ concurrent steals chooses this victim processor Π (so that at least one of the tasks is stolen) is at least $1-(1-1/(P-1))^{P-1} > 1-1/e$.

We can similarly use Chernoff bound to show that the probability that fewer than D tasks are stolen after $S' = 2(D+\log(1/\epsilon))/(1-1/e)$ steps is small. We consider each random variable as a group of $P-1$ steal attempts, with probability of at least $1-1/e$ to choose the deque of the specific processor Π. In this case, the expected value of the sum $\mu = 2(D+\log(1/\epsilon))$ and the offset $\delta = (\mu - D)/\mu$ remain the same as in Lemma 3.1, so the probability is also the same. □

Here note that in the analysis, we do not need the assumption that the D tasks are in the same deque of a certain processor. In fact, the analysis easily extends to when the D tasks are from different processors as long as there is always one task available to be stolen. Therefore, we show the relaxed form of Lemma 3.2.

Lemma 3.3. *Given D tasks and $(P-1)e(D+\log(1/\epsilon))/(e-1)$ steal attempts, the probability that these D tasks are stolen is at least $1-\epsilon$, as long as at least one task is available at the time of any steal attempt.*

As discussed in Section 2.1, any nested-parallel computation can be viewed as a DAG, and each (non-termination) node is an instruction and has either two successors (for a `fork`) or one successor (otherwise). The RWS scheduler dynamically maps each node to a processor. For each specific path, the length is no more than the span D of the algorithm, based on the definition. Based on the RWS algorithm, a node will be mapped to the same processor that executes the predecessor node, except for the spawned children (definition in Section 2.3). The spawned child of a processor Π is ready to be stolen during the process when Π is executing the other branch (continuation), and will be executed by Π if it is not stolen during this process.

We now prove Theorem 2.1 by showing that the computation must have been terminated after $O(PD)$ steals. We will use Lemma 3.3 and apply union bound.

Theorem 2.1. We consider a path in the DAG and show that all instructions on this path will be executed with no more than $O(PD)$ steals. Each node on this path is either a spawned child (that can be stolen) or executed directly after the previous node by the same processor. Now let's consider a processor that is not working on the instructions on this path. When the next steal is attempted, the processor working on this path either has added one more node v on this path that is ready to be stolen, or has executed the node v. This is because we assume a processor executes at least one instruction between two steal attempts from another processor. The only case that the next node is not executed is when it is a spawned child. It will not be executed immediately, and needs to wait until to be stolen for execution, or for the continuation branches to finish and execute these nodes.

Hence, let's consider the worst case that all nodes on the path are spawned children. Lemma 3.3 upper bounds the number of steals to finish the execution of this path. Namely, after $(P-1)e(D+\log(1/\epsilon))/(e-1)$ steal attempts, all nodes are stolen and executed with probability at least $1-\epsilon$. For a DAG with the longest path length D, there are at most 2^D paths in the DAG. Now we set $\epsilon = 1/(2^D \cdot W^c)$ where W is the work of the computation (the number of nodes in the DAG). For any constant $c \geq 1$, $(P-1)e(D+\log(2^D \cdot W^c))/(e-1) = O(P(D+\log W))$ steals are sufficient for executing all existing paths. Now we take the union bound on the probability that all 2^D paths will finish, which is $1-2^D \cdot \epsilon = 1-W^{-c}$. Since each node in the DAG can have at most two successors, the DAG needs to have $D = \Omega(\log W)$ longest path length to contains W nodes. Hence, the $\log W$ term will not dominate, which simplifies the number of steals to be $O(PD)$. □

We have attempted to include this analysis in a few lectures of parallel algorithm courses, and we also would like to include the answer to a frequently asked question. The question is, in the analysis, we apply union bound on 2^D paths, but apparently, the $O(PD)$ steals cannot cover all paths since PD is a much lower-order term than 2^D in practice. Theoretically, the answer is that $\epsilon = 2^D \cdot n^c$ is a sufficiently small term for us to apply union bound, which can give us the desired bound in Theorem 2.1. The more practical and easy-to-understand answer is that we are assuming the worst case, and in practice we do not need all spawn children to be stolen in the execution. In fact, it is likely that most of them are executed by the same processor that spawns this child. Take a parallel-for-loop as an example, which can be viewed $\log n$ level of binary-forks. For most of the paths in this DAG, the spawn children are executed by the same processor that executes the parent node. Because of the design of the RWS algorithm, most of the successful steals will involve a large chunk of work, so steal attempts are infrequent. The analysis shows that, once $O(PD)$ steals are made, the path must have finished, but it is more likely that the computation has finished even before this number of attempts are made.

4 ANALYSIS FOR PARALLEL CACHE COMPLEXITY

Studying parallel cache complexity for nested-parallel algorithms scheduled by RWS is a crucial topic for parallel computing and has been studied in many existing papers

(e.g., [2, 39, 40, 50]). The goal in these analyses is to show the parallel overhead when scheduling using RWS, in addition to the sequential cache complexity. We show the best existing parallel bounds for a list of widely used algorithms and problems in Table 1. While the results in these papers are reasonably good for algorithms with low (polylogarithmic) span, the bounds for parallel overhead can be significant for algorithms with linear or super-linear span. Compared to the lower bounds for the parallel overhead, the upper bounds given in these papers incur polynomial (usually $n^{1/2}$ or $n^{1/3}$) overheads. Such parallel overhead will dominate most of the input range when compared to the sequential cache bounds. Meanwhile, it is known that the practical performance of many of these algorithms is almost as good as low-span algorithms [37, 66]. Hence, it remains an open problem for decades to tightly bound the parallel overhead of such algorithms.

In this section, we show a new analysis to give almost tight parallel cache bounds for the list of problems in Table 1, which are only a polylogarithmic factor off the lower bounds for the main term. Unlike the previous approaches that analyze the scheduler, we directly study the recurrence relations of such computations and find it surprisingly simple. This new analysis is inspired by the concept of *kd-grid* [26] (see more details in Section 4.2).

In the rest of this section, we first review the existing work on this topic in Section 4.1. Section 4.2 presents the high-level idea and the main theorem (Theorem 4.2) of our analysis, which provides a general approach to solve the cache complexity based on recurrences. In Section 4.3, we use a simple example of Kleene's algorithm to show how to use the newly introduced main theorem. Finally, we show the new results for more complicated algorithms in Section 4.4, and discuss the applicability and open problems in Section 4.5.

4.1 Related Work

Given the importance of I/O efficiency and the RWS scheduler, parallel cache bounds have been studied for over 20 years. The definition on distributed cache was given by Acar et al. [2], and they also showed a trivial parallel upper bound on P processors: $Q_P \leq Q_1 + O(PDM/B)$. To achieve this bound, one just needs to pessimistically assume that in each of the $O(PD)$ steals, the stealing thread accesses the entire cache from the original processor, which is $O(M/B)$ additional cache misses. This bound is easy to understand and good for algorithms with polylogarithmic span, but is too loose for linear and super-linear span algorithms. Hence, the following later works showed tighter parallel cache bounds for algorithms with certain structures.

Frigo and Strumpen [50] first analyzed the parallel cache complexity of a class of divide-and-conquer computations, such as matrix multiplication and 1D Stencil, where the problems have subproblem cache complexity as a "concave" function of the computation cost (see more details in [50]). Actually, the idea from [50] is general and can be applied to a variety of algorithms as shown in Table 1. Later work by Cole and Ramachandran [39] pointed out a missing part of the analysis in [50]—the additional cache misses by accessing the execution stacks after a successful steal. They carefully studied this problem, and showed that in most cases, this additional cost is asymptotically bounded by other terms (so the bounds are the same as [50]). In other cases, this can lead to a small overhead (e.g., matrix multiply in row-major format). The authors of [39] also extended the set of applicable algorithms and showed tighter parallel cache bounds for problems such as FFT and list ranking. For the algorithms in this paper, we assume the matrices are in bit-interleaved format [48], so algorithms incur no asymptotic cost for accessing the execution stacks after steals. Even not, we note that all algorithms in Table 1 do not require accesses to the cactus stack anyway (many later RWS implementations chose not to support that for better practicality).

In this paper, we do not consider the additional cost of false sharing [40] or other schedulers [41, 79], but it seems possible to extend the analysis in this paper to the other settings. We leave this as future work.

4.2 Our Approach

As opposed to directly analyzing the algorithms on the scheduler in previous work [39, 40, 50], our key observation is to directly study the computation structure of these algorithms. Interestingly, our analysis is mostly independent with the RWS scheduler, and only plugs in some results from [50] for some basic primitives such as matrix multiplication. By doing so, our analysis can bound the parallel cache complexity much better than the previous results.

The idea of our analysis is motivated by the recent work by Blelloch and Gu [26]. This work studies several parallel dynamic programming and algebra problems, and defines a structure called kd-grid to reveal the computational structure of these problems. It uses kd-grid with $k = 2$ or 3 to model a list of classic problems, such as matrix multiplication, to capture the memory access pattern of these problems. By using kd-grid, their analysis decouples the parallel dependency (and, effectively, the span) from the sequential cache complexity in many parallel algorithms (see details in [26]). Although they only applied the kd-grid analysis on sequential cache complexity, the high-level idea motivates us to also revisit the analysis of parallel cache complexity, and inspired us to directly analyze the essence of the computation structure (the recurrences) of the algorithms. This effectively avoids the crux in previous analysis [2, 39, 40, 50],

which incurs a polynomial overhead in the cache complexity charged by the span of the algorithm. In all of our analyses, the span of these algorithms, no matter linear or super-linear, do not show up in the analysis, which is very different from previous work. Of course, larger span does lead to more parallel cache overhead since it increases the number of steal attempts and each successful one incurs at least one additional cache miss. However, in all applications in this paper, this term is bounded by either the sequential bound or the main term for parallel overhead. Combining all together, we summarize all cache bounds in Table 1, and our new parallel cache bounds for linear and super-linear span algorithms are almost as good as those of the low-span algorithms (e.g., matrix multiplication).

As mentioned, our analysis will use previous results to derive the parallel cache bounds for kd-grids, and use the recurrence relation to bound the entire algorithm. We formalize the recurrences we study for these algorithms and problems, which we refer to as the (α, β, k, l, m)-recurrence.

Definition 4.1 ((α, β, k, l, m)-recurrence). An (α, β, k, l, m)-recurrence is a recurrence in the following form:

$$Q(n) = \alpha \cdot Q(n/\beta) + \sum k_i \cdot n^{l_i} \log^{m_i} n$$

where l_i and m_i are non-negative numbers, and k_i is a function of P, M and B.

In the next section, we will use Kleene's algorithm as an example to show an instantiation of this recurrence relation. The (α, β, k, l, m)-recurrence is easy to solve using the master method [16]:

Theorem 4.2 (Main Theorem). *The solution to $Q(n)$, an (α, β, k, l, m)-recurrence, is:*

$$O\Big(\sum k_i \cdot n^{l_i} \log^{m_i} n + \sum k_j \cdot n^{l_j} \log^{m_j+1} n + \sum k_r n^{\log_\beta \alpha}\Big)$$

for $l_i > \log_\beta \alpha$, $l_j = \log_\beta \alpha$, and $l_r < \log_\beta \alpha$.

As shown here and in the next section, the parallel dependencies of the computation (the algorithm's span) do not show up in the analysis and the solution, which is different from the previous analyses [2, 39, 40, 50].

In the rest of this section, we will first use Kleene's algorithm as an example to show how to use our approach to derive tighter parallel cache bound. Then we show a list of cache-oblivious algorithms that we can apply Theorem 4.2 to and get improved bounds.

It is worth mentioning that the cache bound contains the term for the call stack of the (recursive) subproblems. This term is a constant in the sequential bound, and in many cases the parallel term is the same as the number of steals (e.g., for matrix multiplication and Kleene's algorithm). In other cases, directly applying Theorem 4.2 leads to a $O(Pn^{\log_\beta \alpha})$ term, which is suboptimal since Cole and Ramanchadran [39] showed that this term can be $O(PD)$. Hence, when $n^{\log_\beta \alpha} > D$, instead of using $n^{\log_\beta \alpha} > D$ from Theorem 4.2, we plug in the $O(PD)$ term from [39], which gives a tighter result.

4.3 Kleene's Algorithm as an Example

To start with, we use Kleene's algorithm for all-pair shortest-paths (APSP) as an example to explain the analysis. Kleene's algorithm solves the all-pair shortest-paths (APSP) problem that takes a graph $G = (V, E)$ (with no negative cycles) as input. The Kleene's algorithm was first mentioned in [47, 51, 59, 64], and later discussed in full details in [8]. It is a divide-and-conquer algorithm that is I/O-efficient, cache-oblivious and highly parallelized. The pseudocode of Kleene's algorithm is in Algorithm 1.

Algorithm 1: Kleene(A)

Input: Distance matrix A initialized based on the input graph $G = (V, E)$
Output: Computed Distance matrix A

if $|A| = 1$ **then return** A
$A_{00} \leftarrow$ Kleene(A_{00})
$A_{01} \leftarrow A_{01} + A_{00}A_{01}$
$A_{10} \leftarrow A_{10} + A_{10}A_{00}$
$A_{11} \leftarrow A_{11} + A_{10}A_{01}$
$A_{11} \leftarrow$ Kleene(A_{11})
$A_{01} \leftarrow A_{01} + A_{01}A_{11}$
$A_{10} \leftarrow A_{10} + A_{11}A_{10}$
$A_{00} \leftarrow A_{00} + A_{10}A_{01}$
return A

In Kleene's algorithm, the graph G is represented as the matrix A, where $A[i][j]$ is the weight of the edge between vertices i and j (the weight is $+\infty$ if the edge does not exist). A is partitioned into 4 submatrices indexed as $\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$. The matrix multiplication is defined in a closed semi-ring with $(+, \min)$. The high-level idea is first to compute the APSP between the first half of the vertices only using the paths between these vertices. Then by applying some matrix multiplication, we update the shortest paths between the second half of the vertices using the computed distances from the first half. We then apply another recursive subtask on the second half vertices. The computed distances are finalized, and then we use them to update the shortest paths from the first-half vertices.

The cache complexity $Q(n)$ and span $D(n)$ of this algorithm follow the recurrence relations:

$$Q(n) = 2Q(n/2) + 6Q_{MM}(n/2) \quad (1)$$

$$D(n) = 2D(n/2) + 2D_{MM}(n/2) \quad (2)$$

where $Q_{MM}(n)$ is the I/O cost of a matrix multiplication of input size n. Note that the recurrence relation for the cache complexity is true no matter if we are considering the sequential case (e.g., Q_1 and $Q_{MM,1}$) or the parallel case (e.g., Q_P and $Q_{MM,P}$). For the parallel matrix multiplication algorithm from [26], we have $D_{MM}(n) = O(\log^2 n)$, $D(n) = O(n)$, $Q_{MM,1} = \Theta\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + 1\right)$, and $Q_1 = \Theta\left(\frac{n^3}{B\sqrt{M}} + \frac{n^2}{B} + n\right)$.

If we directly use the result from [2], then we get the parallel cache bound $Q_P = Q_1 + O\left(\frac{PnM}{B}\right)$. As the significant growth of processor count and cache size, the $O(PnM/B)$ term dominates unless n is very large. The tighter bound from [50] shows $Q_P = Q_1 + O\left(\frac{P^{1/3}n^{7/3}}{B} + Pn\right)$. This bound is tighter than the previous one from [2], but the $O(P^{1/3}n^{7/3}/B)$ term still dominates unless $n = \omega(P^{1/2}M^{3/4})$, which is unlikely in practice. The parallel lower bound for this computation [11, 26] is $Q_P = Q_1 + \Omega\left(\frac{P^{1/3}n^2}{B}\right)$, so a polynomial gap remains between the lower and upper cache bounds. Our analysis significantly closes this gap to polylogarithmic.

Now we use Theorem 4.2 to directly solve this (α, β, k, l, m)-recurrence. Equation (1) includes the cache complexity of matrix multiplication. The parallel bounds on P processors based on the algorithm from [26] is:

$$Q_{MM,P} = O\left(\frac{n^3}{B\sqrt{M}} + \frac{P^{1/3}n^2\log^{2/3} n}{B} + P\log^2 n\right). \quad (3)$$

which can be shown by the analysis from [39, 50]. Now we can plug in Equation (3) to Equation (1), and get an (α, β, k, l, m)-recurrence for $Q_P(n)$. In this case, we have $\alpha = \beta = 2$, and $\{(k_i, l_i, m_i)\} = \{(1/B\sqrt{M}, 3, 0), (P^{1/3}/B, 2, 2/3), (P, 0, 2)\}$. Plugging in Theorem 4.2 directly gives the solution of:

$$Q_P(n) = O\left(\frac{n^3}{B\sqrt{M}} + \frac{P^{1/3}n^2\log^{2/3} n}{B} + Pn\right).$$

In this case, the input size is $O(n^2)$ so the corresponding term $l_i = 2 > \log_\beta \alpha = 1$. Hence, even though Kleene's algorithm has linear span as opposed to the polylogarithmic span for matrix multiplication, the additional steals caused by the span will not affect the input term (the $l_i = 2$ term in this case for Kleene's algorithm and matrix multiplication). In fact, as one can see, either the span bound, or the span recurrence (Equation (2)), does not show up in the entire analysis.

Since Kleene's algorithm is very simple, we can also show how to directly solve the recurrence by plugging Equation (3) in Equation (1). We believe this can illustrate a more intuitive idea of our analysis.

$$\begin{aligned}
Q(n) =& 6Q_{MM}(n/2) + 12Q_{MM}(n/4) + \cdots + 3n \cdot Q_{MM}(1) \\
=& O\left(\frac{6n^3}{8B\sqrt{M}} + \frac{12n^3}{64B\sqrt{M}} + \cdots\right) + \\
& O\left(\frac{6P^{1/3}n^2\log^{2/3} n}{4B} + \frac{12P^{1/3}n^2\log^{2/3} n}{16B} + \cdots\right) + \\
& O\left(P\log^2 n + 2P\log^2(n/2) + \cdots + Pn\right) \\
=& O\left(\frac{n^3}{B\sqrt{M}} + \frac{P^{1/3}n^2\log^{2/3} n}{B} + Pn\right).
\end{aligned}$$

Here the terms in the first two big-Os are decreasing geometrically, while the last term increases geometrically. The main term for parallel overhead is only a polylogarithmic factor ($O(\log^{2/3} n)$) more than the lower bound, as opposed to a polynomial factor ($O(n^{1/3})$) in the previous terms.

Although Kleene's algorithm can also be directly analyzed as shown above, using Theorem 4.2 enables a simpler way to show a tighter bound of Kleene's algorithm than previous analysis. More importantly, for many algorithms that are more complicated than Kleene's algorithm, it is nearly impossible to show new bounds by directly plugging in the recurrences, in which case using Theorem 4.2 easily enables simple analysis and tighter bounds. We will then present these algorithms and our new analysis and bound in Section 4.4.

4.4 Other Applications

We have shown the main theorem (Theorem 4.2) and the intuition why it leads to better parallel cache complexity for Kleene's algorithm. We now apply the theorem to a variety of classic cache-oblivious algorithms, which leads to better parallel cache bound. The details of the algorithms can be found in [26, 45]. Some of these algorithms are complicated, here we only show the recurrences and the parallel cache bounds since those are all we need.

4.4.1 Building Blocks. Before we go over the applications, we first show the parallel cache complexity of some basic primitives (kd-grid and matrix transpose) that the applications use.

Matrix Multiplication (MM). Matrix multiplication is modeled as a 3d-grid in [26]. The sequential cache bound

$Q_{MM,1}$ is $\Theta\left(\frac{n^3}{B\sqrt{M}}+\frac{n^2}{B}+1\right)$, and the parallel bound on p processors $Q_{MM,P}$ is $O\left(\frac{n^3}{B\sqrt{M}}+\frac{P^{1/3}n^2\log^{2/3}n}{B}+P\log^2 n\right)$. Here we assume the matrix is stored in the bit-interleaved (BI) format [48], which can be easily converted from other formats such as the row-major format with the same cost as matrix transpose.

Matrix Transpose (MT). Matrix transpose is another widely used primitives in cache-oblivious algorithms. The sequential cache bound $Q_{MT,1}$ is $\Theta\left(\frac{n^2}{B}+1\right)$, and the parallel bound on p processors [39, 50] is:

$$Q_{MT,P}=O\left(\frac{n^2}{B}+P\log^2 n\right). \tag{4}$$

The 2d-grid. The 2d-grid can be viewed as an analog of matrix multiplication, but is a 2 dimensional computation instead of 3 dimensional as in MM ($O(n^3)$ arithmetic operations and memory accesses). It can also be viewed as a matrix-vector multiplication but the matrix is implicit. The 2d-grid is a commonly used primitive in dynamic programming algorithms [26]. The sequential cache bound $Q_{2D,1}$ is $\Theta\left(\frac{n^2}{BM}+\frac{n}{B}+1\right)$, and the parallel bound on p processors [39, 50] is

$$Q_{2D,P}=O\left(\frac{n^2}{BM}+\frac{P^{1/2}n\log n}{B}+P\log^2 n\right). \tag{5}$$

4.4.2 Gaussian Elimination. Here we consider the parallel divide-and-conquer Gaussian elimination algorithm shown in [26], with the recurrence of the cache bound as $Q(n)=2Q(n/2)+4Q_{MM}(n/2)$. Compared to Kleene's algorithm (Equation (1)), this recurrence only differs by a constant. Hence, the parallel cache bound is asymptotically the same as Kleene's algorithm.

4.4.3 Triangular System Solver. The triangular system solver (TRS) solves the linear system that takes the output of Gaussian elimination (i.e., $Ax=b$ where A is an upper triangular matrix). We consider the parallel divide-and-conquer algorithm for a triangular system solver from [26] with cubic work and linear span. The cache bound is:

$$Q_{TRS}(n)=4Q_{TRS}(n/2)+2Q_{MM}(n/2).$$

To analyze the parallel cache complexity, we can plug in Equation (3) and get the (α,β,k,l,m)-recurrence with $\alpha=4$, $\beta=2$, and $\{(k_i,l_i,m_i)\}=\{\ (1/B\sqrt{M},3,0),\ (P^{1/3}/B,2,2/3),\ (P,0,2)\}$. Applying and Theorem 4.2 leads to the parallel cache complexity as

$$Q_{TRS,P}=O\left(\frac{n^3}{B\sqrt{M}}+\frac{P^{1/3}n^2\log^{5/3}n}{B}+Pn\right). \tag{6}$$

4.4.4 Cholesky Factorization and LU Decomposition. Both Cholesky factorization and LU decomposition are widely used linear algebraic tools to decompose a matrix to the product of a lower triangular matrix and an upper triangular matrix. The divide-and-conquer algorithms for Cholesky factorization and LU decomposition [45] are quite similar in the way that they are designed on top of triangular system solver and matrix multiplication. The cache bounds for both algorithms are:

$$Q(n)=2Q(n/2)+Q_{TRS}(n/2)+O(1)\cdot Q_{MM}(n/2).$$

We can plug in Equation (3) and Equation (6) to get the (α,β,k,l,m)-recurrence with $\alpha=2$, $\beta=2$, and $\{(k_i,l_i,m_i)\}=\{(1/B\sqrt{M},3,0),\ (P^{1/3}/B,2,5/3),\ (P,0,2)\}$. Since $\log_\beta\alpha=2$, the parallel bound is almost the same as Q_{TRS}, except that the span for these algorithms is $O(n\log n)$, which increases the last term by a logarithmic factor. Hence for these two problems, we have:

$$Q_p=O\left(\frac{n^3}{B\sqrt{M}}+\frac{P^{1/3}n^2\log^{5/3}n}{B}+Pn\log n\right).$$

4.4.5 LWS Recurrence. The LWS (least-weighted subsequence) recurrence [56] is one of the most commonly-used DP recurrences in practice. Given a real-valued function $w(i,j)$ for integers $0\le i<j\le n$ and D_0, for $1\le j\le n$,

$$D_j=\min_{0\le i<j}\{D_i+w(i,j)\}.$$

This recurrence is widely used in real-world applications [4, 53, 54, 60–62]. Here we assume that $w(i,j)$ can be computed in constant work based on a constant size of input associated to i and j, which is true for all these applications.

Here we consider the parallel divide-and-conquer algorithm to solve LWS recurrence from [26] with quadratic work and linear span. This algorithm partitions the problems into two halves, solves the first one, applies a 2d-grid computation, and solves the second one. The cache bound is $Q(n)=2Q(n/2)+Q_{2D}(n/2)$.

Here, by using Equation (5), the (α,β,k,l,m)-recurrence has $\alpha=2,\beta=2$, and $\{(k_i,l_i,m_i)\}=\{(1/BM,2,0),(P^{1/2}/B,1,1),(P,0,2)\}$. Since, $\log_\beta\alpha=1$, so the parallel cache bound is:

$$Q_P(n)=O\left(\frac{n^2}{BM}+\frac{P^{1/2}n\log^2 n}{B}+Pn\right).$$

4.4.6 GAP Recurrence. The GAP problem [52, 54] is a generalization of the edit distance problem that has many applications in molecular biology, geology, and speech recognition. Given a source string X and a target string Y, an "edit" can be a sequence of consecutive deletes corresponding to a gap in X, and a sequence of consecutive inserts corresponding to a gap in Y. Let $w(p,q)$ $(0\le p<q\le n)$ be the cost of deleting the substring of X from $(p+1)$-th to q-th character, $w'(p,q)$

be inserting the substring of Y accordingly, and $r(i, j)$ be the cost to change the i-th character in X to j-th character in Y.

Let $D_{i,j}$ be the minimum cost for such transformation from the prefix of X with i characters to the prefix of Y with j characters, the recurrence for $i, j > 0$ is:

$$D_{i,j} = \min \begin{cases} \min_{0 \le q < j} \{D_{i,q} + w'(q, j)\} \\ \min_{0 \le p < i} \{D_{p,j} + w(p, i)\} \\ D_{i-1,j-1} + r(i, j) \end{cases}$$

corresponding to either replacing a character, inserting or deleting a substring. The best parallel divide-and-conquer algorithm to compute the GAP recurrence is proposed by Blelloch and Gu [26]. The cache bound recurrence of the algorithm in [26] is $Q(n) = 4Q(n/2) + 4(n/2) \cdot Q_{2D}(n/2) + 2Q_{MT}(n/2)$, which includes 4 subproblems with half size, a linear number of 2d-grid (see more details in [26]), and 2 matrix transpose calls.

To derive parallel cache complexity, we can apply Equation (4) and Equation (5) and get the (α, β, k, l, m)-recurrence with $\alpha = 4, \beta = 2$, and $\{(k_i, l_i, m_i)\} = \{(1/BM, 3, 0), (P^{1/2}/B, 2, 1), (P, 0, 2)\}$. Then using Theorem 4.2 gives $\log_\beta \alpha = 2$ and

$$Q_P(n) = O\left(\frac{n^3}{BM} + \frac{P^{1/2} n^2 \log^2 n}{B} + P n^{\log_2 3}\right).$$

4.4.7 RNA recurrence. The RNA problem [54] is a generalization of the GAP problem. In this problem, a weight function $w(p, q, i, j)$ is given, which is the cost to delete the substring of X from $(p+1)$-th to i-th character and insert the substring of Y from $(q+1)$-th to j-th character. Similar to GAP, let $D_{i,j}$ be the minimum cost for such transformation from the prefix of X with i characters to the prefix of Y with j characters, the recurrence for $i, j > 0$ is:

$$D_{i,j} = \min_{0 \le p < i, 0 \le q < j} \{D_{p,q} + w(p, q, i, j)\}.$$

This recurrence is widely used in computational biology, like to compute the secondary structure of RNA [78]. The RNA recurrence can be viewed as a 2d version of the LWS recurrence, and the latest algorithm from [26] has the cache bound of $Q(n) = 4Q(n/2) + Q_{2D}(n^2)$.

For parallel cache complexity, the (α, β, k, l, m)-recurrence by plugging in Equation (5) is $\alpha = 4, \beta = 2$, and $\{(k_i, l_i, m_i)\} = \{(1/BM, 4, 0), (P^{1/2}/B, 2, 1), (P, 0, 2)\}$. The parallel cache bound can be solved as:

$$Q_P(n) = O\left(\frac{n^4}{BM} + \frac{P^{1/2} n^2 \log^2 n}{B} + P n^{\log_2 3}\right).$$

4.4.8 Protein Accordion Folding. The recurrence for protein accordion folding [77] is:

$$D_{i,j} = \max_{1 \le k < j-1} \{D_{j-1,k} + w(i, j, k)\}$$

for $1 \le j < i \le n$, with $O(n^2/B)$ cost to precompute $w(i, j, k)$. In [26], a parallel divide-and-conquer algorithm for protein accordion folding is given, with cubic work and linear span. The recurrence for the cache bound is $Q(n) = 2Q(n/2) + Q_{MT}(n/2) + (n/2) \cdot Q_{2D}(n/2)$, which includes 2 subproblems with half size, a linear number of 2d-grid, and 1 matrix transpose call.

For parallel cache complexity, we can apply Equation (4) and Equation (5) and get $\alpha = 2$, $\beta = 2$, and $\{(k_i, l_i, m_i)\} = \{(1/BM, 3, 0), (P^{1/2}/B, 2, 1), (P, 0, 2)\}$. Since $\log_\beta \alpha = 1$, we can get:

$$Q_P(n) = O\left(\frac{n^3}{BM} + \frac{P^{1/2} n^2 \log n}{B} + P n \log^2 n\right).$$

4.4.9 Parenthesis Recurrence. The Parenthesis recurrence solves the following problem in many applications [43, 53, 54, 80]: for a linear sequence of objects, an associative binary operation, and the cost of performing that operation on any two objects, the goal is to compute the min-cost way to group the objects by repeatedly applying the binary operations. Let $D_{i,j}$ be the minimum cost to merge the objects indexed from $i + 1$ to j (1-based), the recurrence for $0 \le i < j \le n$ is:

$$D_{i,j} = \min_{i < k < j} \{D_{i,k} + D_{k,j} + w(i, k, j)\}$$

where $w(i, k, j)$ is the cost to merge the two partial results of objects indexed from $i + 1$ to k and those from $k + 1$ to j.

The parallel cache-oblivious algorithm for the parenthesis recurrence is introduced in [36]. The original problem and some subproblems are triangle ones ($Q_\triangle$), but when computing them, we need other square subproblems ($Q_\square$). The recurrence relations for the cache bounds are $Q_\triangle(n) = 2Q_\triangle(n/2) + Q_\square(n/2)$, and $Q_\square(n) = 4Q_\square(n/2) + 4Q_{MM}(n/2)$.

To compute the parallel cache complexity, we apply Equation (3) and Theorem 4.2 to $Q_\square$ first to obtain a parallel cache complexity of $Q_{\square,P}(n) = O\left(\frac{n^3}{B\sqrt{M}} + \frac{P^{1/3} n^2 \log^{5/3} n}{B} + P n^{\log_2 3}\right)$. Then we substitute this into $Q_\triangle$ to apply Theorem 4.2 again and get:

$$Q_P(n) = Q_{\triangle,P}(n) = O\left(\frac{n^3}{B\sqrt{M}} + \frac{P^{1/3} n^2 \log^{5/3} n}{B} + P n^{\log_2 3}\right).$$

4.5 Discussions

A common theme in our analysis is to apply the parallel cache bounds of the basic primitives from [50] in a certain step, and then use the recursive structure to derive parallel cache bound for the entire algorithm. We note that in many cases this is better than directly using the result in [50] for the entire algorithm. However, we note that this is not always true, and it relies on the recursive structure. More precisely, it depends on whether the number of base-case subproblems

is asymptotically no more than the input elements (true for all algorithms discussed above). A counterexample is the edit distance problem (or longest common subsequence). The recurrence for the divide-and-conquer algorithm is $Q(n) = 4Q(n/2)+O(1)$. In this case, using Theorem 4.2 gives a looser bound than using the analysis from [50]. Hence, how to tightly bound the parallel cache complexity for edit distance remains an open problem.

REFERENCES

[1] U. A. Acar. *Parallel computing theory and practice.* 2016.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theoretical Computer Science (TCS)*, 35(3), 2002.

[3] U. A. Acar, A. Charguéraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2013.

[4] A. Aggarwal and M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1-2), 1990.

[5] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9), 1988.

[6] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, and R. Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[7] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–32, 2008.

[8] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

[9] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

[10] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[11] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23:1–155, 2014.

[12] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[13] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Implicit decomposition for write-efficient connectivity algorithms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[14] M. A. Bender, E. D. Demaine, R. Ebrahimi, J. T. Fineman, R. Johnson, A. Lincoln, J. Lynch, and S. McCauley. Cache-adaptive analysis. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[15] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley. Cache-adaptive algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 958–971, 2014.

[16] J. L. Bentley, D. Haken, and J. B. Saxe. A general method for solving divide-and-conquer recurrences. *ACM SIGACT News*, 12(3):36–44, 1980.

[17] G. E. Blelloch. Nesl: A nested data-parallel language. Technical report, Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[18] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2008.

[19] G. E. Blelloch, L. Dhulipala, and Y. Sun. Introduction to parallel algorithms (draft), 2018.

[20] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[21] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2012.

[22] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[23] G. E. Blelloch, J. T. Fineman, Y. Gu, and Y. Sun. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[24] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2004.

[25] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low depth cache-oblivious algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[26] G. E. Blelloch and Y. Gu. Improved parallel cache-oblivious algorithms for dynamic programming. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2020.

[27] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[28] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[29] G. E. Blelloch and M. Reid-Miller. Fast set operations using treaps. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 1998.

[30] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems (TOCS)*, 32(3), 1999.

[31] G. E. Blelloch, H. V. Simhadri, and K. Tangwongsan. Parallel and I/O efficient set covering algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2012.

[32] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[33] Z. Budimlić, V. Cavé, R. Raman, J. Shirako, S. Taşırlar, J. Zhao, and V. Sarkar. The design and implementation of the habanero-java parallel programming language. In *Symposium on Object-oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 185–186, 2011.

[34] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[35] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi. Provably efficient scheduling of cache-oblivious wavefront algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 339–350, 2017.

[36] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2008.

[37] R. A. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. *Theory of Computing Systems (TOCS)*, 47(4):878–919, 2010.

[38] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. In *International Colloquium on Automata, Languages, and Programming*, pages 226–237. Springer, 2010.

[39] R. Cole and V. Ramachandran. Revisiting the cache miss analysis of multithreaded algorithms. In *Latin American Symposium on Theoretical Informatics (LATIN)*, pages 172–183. Springer, 2012.

[40] R. Cole and V. Ramachandran. Analysis of randomized work stealing with false sharing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 985–998. IEEE, 2013.

[41] R. Cole and V. Ramachandran. Bounding cache miss costs of multithreaded computations under general schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[42] R. Cole and V. Ramachandran. Resource oblivious sorting on multicores. *ACM Transactions on Parallel Computing (TOPC)*, 3(4), 2017.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[44] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Semi-asymmetric parallel graph algorithms for nvrams. *Proceedings of the VLDB Endowment (PVLDB)*, 13(9), 2020.

[45] D. Dinh, H. V. Simhadri, and Y. Tang. Extending the nested parallel model to the nested dataflow model with provably efficient schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[46] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[47] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *IEEE Symposium on Switching and Automata Theory*, 1971.

[48] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1999.

[49] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 33(5), 1998.

[50] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. *Theory of Computing Systems*, 45(2):203–233, 2009.

[51] M. Furman. Application of a method of fast multiplication of matrices to problem of finding graph transitive closure. *Doklady Akademii Nauk SSSR*, 194(3), 1970.

[52] Z. Galil and R. Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science (TCS)*, 64(1), 1989.

[53] Z. Galil and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science (TCS)*, 92(1), 1992.

[54] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than O(1) dependency. *J. Parallel Distrib. Comput.*, 21(2), 1994.

[55] Y. Gu, O. Obeya, and J. Shun. Parallel in-place algorithms: Theory and practice. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 114–128, 2021.

[56] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM J. on Computing*, 16(4), 1987.

[57] https://www.threadingbuildingblocks.org.

[58] http://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html.

[59] S. C. Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[60] J. Kleinberg and E. Tardos. *Algorithm design*. Pearson Education India, 2006.

[61] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11), 1981.

[62] M. Künnemann, R. Paturi, and S. Schneider. On the fine-grained complexity of one-dimensional dynamic programming. *arXiv preprint arXiv:1703.00941*, 2017.

[63] S. K. Muller and U. A. Acar. Latency-hiding work stealing: Scheduling interacting parallel computations with work stealing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[64] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2), 1971.

[65] http://www.openmp.org.

[66] T. Schardl. *Performance engineering of multicore software: Developing a science of fast code for the post-Moore era*. PhD thesis, 2016.

[67] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms*, 2(1):88–102, 1981.

[68] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 30–41, 2014.

[69] K. Singer, K. Agrawal, and I.-T. A. Lee. Scheduling i/o latency-hiding futures in task-parallel platforms. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, pages 147–161. SIAM, 2020.

[70] K. Singer, Y. Xu, and I.-T. A. Lee. Proactive work stealing for futures. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 257–271, 2019.

[71] W. Suksompong. Bounds on multithreaded computations by work stealing. Master's thesis, Massachusetts Institute of Technology, 2014.

[72] Y. Sun and G. E. Blelloch. Parallel range, segment and rectangle queries with augmented maps. In *SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 159–173, 2019.

[73] Y. Sun, G. E. Blelloch, W. S. Lim, and A. Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proceedings of the VLDB Endowment (PVLDB)*, 13(2):211–225, 2019.

[74] Y. Sun, D. Ferizovic, and G. E. Blelloch. Pam: Parallel augmented maps. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, 2018.

[75] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.

[76] https://msdn.microsoft.com/en-us/library/dd460717%28v=vs.110%29.aspx.

[77] J. J. Tithi, P. Ganapathi, A. Talati, S. Aggarwal, and R. Chowdhury. High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[78] M. S. Waterman and T. F. Smith. Rna secondary structure: A complete mathematical analysis. *Mathematical Biosciences*, 42(3-4), 1978.

[79] J. Yang and Q. He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.

[80] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *ACM Symposium on Theory of Computing (STOC)*, 1980.