Scalable Fine-Grained Parallel Cycle Enumeration Algorithms

Jovan Blanuša*
IBM Research Europe
Zurich, Switzerland
jov@zurich.ibm.com

Paolo Ienne
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland
paolo.ienne@epfl.ch

Kubilay Atasu
IBM Research Europe
Zurich, Switzerland
kat@zurich.ibm.com

ABSTRACT

Enumerating simple cycles has important applications in computational biology, network science, and financial crime analysis. In this work, we focus on parallelising the state-of-the-art simple cycle enumeration algorithms by Johnson and Read-Tarjan along with their applications to temporal graphs. To our knowledge, we are the first ones to parallelise these two algorithms in a fine-grained manner. We are also the first to demonstrate experimentally a linear performance scaling. Such a scaling is made possible by our decomposition of long sequential searches into fine-grained tasks, which are then dynamically scheduled across CPU cores, enabling an optimal load balancing. Furthermore, we show that coarse-grained parallel versions of the Johnson and the Read-Tarjan algorithms that exploit edge- or vertex-level parallelism are not scalable. On a cluster of four multi-core CPUs with 256 physical cores, our fine-grained parallel algorithms are, on average, an order of magnitude faster than their coarse-grained parallel counterparts. The performance gap between the fine-grained and the coarse-grained parallel algorithms widens as we use more CPU cores. When using all 256 CPU cores, our parallel algorithms enumerate temporal cycles, on average, 260× faster than the serial algorithm of Kumar and Calders.

Code repository: https://github.com/IBM/parallel-cycle-enumeration

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**.

KEYWORDS

Cycle enumeration; Parallel graph algorithms; Graph mining

ACM Reference Format:
Jovan Blanuša, Paolo Ienne, and Kubilay Atasu. 2022. Scalable Fine-Grained Parallel Cycle Enumeration Algorithms. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490148.3538585

*Also with Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences.

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9146-7/22/07…$15.00
https://doi.org/10.1145/3490148.3538585

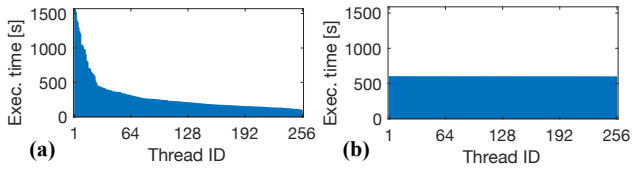


Figure 1: Per-thread execution time of (a) the coarse-grained parallel Johnson algorithm vs. (b) our fine-grained parallel Johnson algorithm using the *wiki-talk* graph and a 12h time window. Thanks to a perfect load balancing, our fine-grained method is 3× faster on a 64-core CPU executing 256 threads.

1 INTRODUCTION

A graph-based data representation is desirable when analyzing large and complex datasets because it exposes the connectivity of the underlying data objects and enables the discovery of complex relationships between them [42]. Analysing graph-structured data has important applications in many domains, such as finance [39], healthcare [64], cybersecurity [44], and advertising [65]. The existence of certain patterns, such as cycles, cliques, and motifs, in a graph can reveal nontrivial relationships between different graph objects [2]. As the volume of graph data continues to grow, the discovery of such relationships becomes computationally challenging, necessitating more efficient algorithms and scalable parallel implementations that can exploit modern multi-core processors.

Simple cycles and temporal cycles. This paper introduces efficient parallel algorithms for enumerating simple cycles of directed graphs. A simple cycle is a sequence of edges that starts and ends with the same vertex and visits other vertices of the graph at most once. Enumerating simple cycles has important applications in computational biology [30, 34], network science [19, 68], software bug tracking [56], and electronic design automation [21, 43, 49].

Furthermore, some graphs have their edges annotated with timestamps, which we refer to as temporal graphs. In such graphs one can also look for temporal cycles [32], which are special cases of simple cycles, in which the edges are ordered in time. For instance, in financial transaction graphs, a temporal cycle represents a series of transactions in which the money initially sent from one bank account returns back to the same account; the existence of such cycles is a strong indicator of financial fraud such as money laundering, tax avoidance [23, 58], and credit card fraud [51]. Finding temporal cycles in temporal graphs also enables detecting circular trading, which can be used for manipulating stock prices [24, 27, 45].

Parallelisation challenges. We focus on parallelising the algorithms by Johnson [28] and Read-Tarjan [53] for finding cycles because these algorithms achieve the lowest time complexity bounds

Table 1: Our fine-grained parallel Read-Tarjan algorithm is the only solution that is both work efficient and scalable.

Parallel algorithm	Work efficient	Scalable
Coarse-grained parallel algorithms	✓	
Our fine-grained parallel Johnson		✓
Our fine-grained parallel Read-Tarjan	✓	✓

reported for directed graphs [38]. Both algorithms are recursively formulated and construct a recursion tree in a depth-first fashion. However, these algorithms employ different pruning techniques to limit the size of their recursion trees. In practice, the Johnson algorithm is faster than the Read-Tarjan algorithm because it uses more aggressive recursion-tree pruning techniques [20, 38].

The naïve way of parallelising the aforementioned algorithms involves searching for cycles starting from different vertices or edges in parallel, which we refer to as the coarse-grained parallel methods. Such coarse-grained parallel approaches are straightforward to implement using the popular vertex-centric [37, 40] and edge-centric [55] graph processing frameworks. However, real-world graphs often exhibit a power-law or a log-normal distribution of vertex degrees [4, 11]. In such graphs, the execution time of coarse-grained parallel approaches is dominated by searches that start from a small set of vertices or edges as illustrated in Figure 1a. This behaviour leads to a workload imbalance and limits scalability.

The shortcomings of coarse-grained parallel approaches can be addressed by decomposing the search for cycles starting from a given edge or vertex into finer-grained tasks. Fine-grained parallelism has been exploited by other graph mining algorithms [1, 6, 15]. However, to our knowledge, it has not been applied to asymptotically-optimal simple cycle enumeration algorithms, such as the Johnson algorithm and the Read-Tarjan algorithm. In particular, the pruning efficiency of the Johnson algorithm depends on a strictly sequential depth-first-search-based recursion tree exploration. As such, decomposing the Johnson algorithm into fine-grained tasks is not possible without giving up some of its pruning efficiency. In contrast, the Read-Tarjan algorithm does not require a strictly sequential depth-first-search-based recursion tree exploration, hence, it is easier to decompose into fine-grained tasks.

Contributions. In this paper, we contribute scalable parallel versions of the Johnson and the Read-Tarjan algorithms. To our knowledge, we are the first ones to parallelise these two algorithms in a fine-grained manner. We are also the first to demonstrate an almost linear performance scaling on a system that can execute up to a thousand concurrent software threads. Such a scalability is enabled by our decomposition of long sequential searches into fine-grained tasks, which are then dynamically scheduled across CPU cores, leading to an ideal load balancing as shown in Figure 1b.

To decompose the Johnson algorithm into fine-grained tasks, we have relaxed its strictly depth-first-search-based exploration. In this way, we have enabled it to perform multiple independent depth-first searches in parallel. However, this additional flexibility is at the expense of some pruning efficiency. Because of the reduced pruning efficiency, our fine-grained parallel Johnson algorithm performs more work than its serial version— i.e., it is not work-efficient. In contrast, our fine-grained parallel Read-Tarjan algorithm does not perform more work than its sequential version and is work efficient. Nevertheless, both fine-grained algorithms are scalable both in theory and in practice. Table 1 shows the key results of our theoretical analysis. Interestingly, despite not being work efficient, our fine-grained Johnson algorithm outperforms our fine-grained parallel Read-Tarjan algorithm in most of our experiments.

Table 2: Capabilities of the related work versus our own. Competing algorithms either fail to exploit fine-grained parallelism or do it on top of asymptotically inferior algorithms.

Related work	[32]	[51]	[47]	[50]	[22]	Ours
Fine-grained parallelism				✓		✓
Asymptotic optimality	✓		✓		✓	✓
Temporal cycles	✓					✓
Time-window constraints	✓	✓				✓
Cycle-length constraints		✓	✓	✓	✓	

Paper structure. The remainder of this paper is organised as follows. Section 2 discusses the related work. Section 3 introduces the notation used, formally defines the concepts of work efficiency and scalability, and covers state-of-the-art sequential algorithms for finding simple cycles. Section 4 covers coarse-grained parallel versions of the Johnson and the Read-Tarjan algorithms. Section 5 and Section 6 introduce our fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms, respectively. Section 7 discusses adaptations of our algorithms to compute temporal cycles. Section 8 provides an experimental evaluation of all the parallel algorithms covered in this work. Section 9 presents our conclusions.

2 RELATED WORK

Simple cycle enumeration algorithms. Enumeration of simple cycles of graphs is a classical computer science problem [5, 20, 28, 36, 38, 53, 59, 61, 62, 66]. The backtracking-based algorithms by Johnson [28], Read and Tarjan [53], and Szwarcfiter and Lauer [59] achieve the lowest time complexity bounds reported in the literature for enumerating simple cycles in directed graphs. These algorithms implement advanced recursion tree pruning techniques to improve on the brute-force Tiernan algorithm [62]. Section 3.4 covers such pruning techniques in further detail. A cycle enumeration algorithm that is asymptotically faster than the aforementioned algorithms [28, 53, 59] has been proposed in Birmelé et al. [5], however, it is applicable only to undirected graphs. The algorithms for simple cycle enumeration can be specialised to find temporal cycles, such as in Kumar and Calders [32], and our parallel algorithms lend themselves to the same specialisation. Simple cycles can also be enumerated by computing the powers of the adjacency matrix [14, 29, 48] or by using circuit vector space algorithms [18, 38, 67], but the complexity of such approaches grows exponentially with the length of the cycles or the size of the graphs.

Cycle-length and time-window constraints. To make cycle enumeration problem tractable, it is common to search for cycles under some constraints. For instance, the length of the cycles—i.e., the maximum number of edges in the cycle, can be constrained, such as in Gupta and Suzumura [22], Peng et al. [47], and Qiu et al [51]. In temporal graphs, it is also common to search for cycles within a sliding time window, such as in Kumar and Calders [32] and Qiu et al [51]. Constraining the length of the cycles or the size of the time windows effectively narrows down the search to a spatial or temporal neighbourhood, respectively. In this work,

Table 3: Summary of the notation used in the paper.

Symbol	Description
$u \rightarrow v$	A directed edge connecting vertex u with v.
n, e	Number of vertices and edges in a graph.
δ	Size of a time window.
c	Number of simple cycles in a graph.
s	Number of maximal simple paths in a graph.
Π	The current simple path explored by an algorithm.
Blk	The set of blocked vertices.
Blist	The unblock list data structure.
p	Number of threads used for parallel algorithms.
$T_p(n)$	Execution time of a parallel algorithm.
$W_p(n)$	Amount of work a parallel algorithm performs.

we focus on temporal graphs, therefore, we use time window constraints when enumerating both simple and temporal cycles. Note that length-constrained simple cycles can also be enumerated using incremental algorithms, such as in Qiu et al. [51]. However, this algorithm is based on the brute-force Tiernan algorithm [62], which makes it slower than nonincremental algorithms that use recursion tree pruning techniques [47]. In addition, because incremental algorithms maintain auxiliary data structures, such as paths, to be able to construct cycles incrementally, they are not as memory-efficient as nonincremental algorithms [47]. Table 2 offers comparisons between the capabilities of these methods and ours.

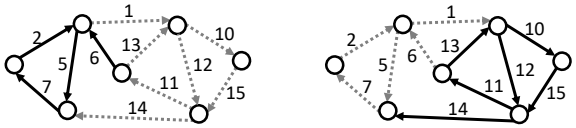
Parallel and distributed algorithms for cycle enumeration. Cui et al. [13] proposed a multi-threaded algorithm for detecting and removing simple cycles of a directed graph. The algorithm divides the graph into its strongly-connected components and each thread performs a depth-first search on a different component to find cycles. However, sizes of the strongly-connected components in real-world graphs can vary significantly [41], which leads to a workload imbalance. Rocha and Thatte [54] proposed a distributed algorithm for simple cycle enumeration based on the bulk-synchronous parallel model [63], but it searches for cycles in a brute-force manner. Qing et al. [50] introduced a parallel algorithm for finding length-constrained simple cycles. It is the only other fine-grained parallel algorithm we are aware of in the sense that it can search for cycles starting from the same vertex in parallel. However, the way this algorithm searches for cycles is similar to the way the brute-force Tiernan algorithm [62] works. To our knowledge, we are the first ones to introduce fine-grained parallel versions of asymptotically-optimal simple cycle enumeration algorithms, which do not rely on a brute-force search, as we show in Table 2.

3 BACKGROUND

This section introduces the main theoretical concepts used in this paper and provides an overview of the most prominent simple cycle enumeration algorithms. The notation used is given in Table 3.

3.1 Preliminaries

We consider a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ having a set of vertices $\mathcal{V}$ and a set of directed edges $\mathcal{E} = \{u \rightarrow v \mid u, v \in \mathcal{V}\}$. The set of neighbors of a given vertex v is defined as $\mathcal{N}(v) = \{w \mid \forall\, v \rightarrow w \in \mathcal{E}\}$. An outgoing edge of a given vertex v is defined as $v \rightarrow w$ and an incoming edge is defined as $u \rightarrow v$, where $v \rightarrow w, u \rightarrow v \in \mathcal{E}$. A



(a) Time window $[2:7]$ **(b) Time window** $[10:15]$

Figure 2: Two snapshots of a temporal graph associated with two different time windows of size $\delta = 5$. The solid arrows indicate the edges that belong to the respective time windows.

path between the vertices v_0 and v_k, denoted as $v_0 \rightarrow v_1 \ldots \rightarrow v_k$, is a sequence of vertices such that there exists an edge between two consecutive vertices of the sequence. A *simple path* is a path with no repeated vertices. A simple path is *maximal* if the last vertex of the path has no neighbors or all of its neighbors are already in the path [16]. A **cycle** is a path of non-zero length from a vertex v to the same vertex v. A **simple cycle** is a cycle with no repeated vertices except for the first and the last vertex. The number of maximal simple paths and the number of simple cycles in a graph are denoted as s and c, respectively (see Table 3). Note that s can be exponentially larger than c [61]. The goal of **simple cycle enumeration** is to compute all simple cycles of a directed graph $\mathcal{G}$, ideally without computing all maximal simple paths of it.

A **temporal graph** is a graph that has its edges annotated with timestamps. [46]. In temporal graphs, a **temporal cycle** is a simple cycle, in which the edges appear in the increasing order of their timestamps. A simple cycle or a temporal cycle of a temporal graph occurs within a **time window** $[t_{w1} : t_{w2}]$ if every edge of that cycle has a timestamp t_s such that $t_{w1} \leq t_s \leq t_{w2}$. Figure 2 shows the simple cycles of a temporal graph that occur within two different time windows of size $\delta = 5$. This graph contains one simple cycle in the time window $[2:7]$ (Figure 2a), which is also a temporal cycle, and two simple cycles in the time window $[10:15]$ (Figure 2b).

3.2 Task-level parallelism

The parallel algorithms described in this paper can be implemented using shared-memory parallel processing frameworks, such as TBB [31], Cilk [8], and OpenMP [52]. These frameworks enable decomposition of a program into tasks that can be independently executed by different software threads. In our setup, tasks are dynamically created and scheduled. A *parent* task can create several *child* tasks. A dynamic task management system assigns the tasks created to the work queues of available threads. Furthermore, a work-stealing scheduler [8, 9, 31] enables a thread that is not executing a task to *steal* a task from the work queue of another thread. Stealing tasks enables dynamic load balancing and ensures full utilisation of the threads when there are sufficiently many tasks.

3.3 Work efficiency and scalability

We use the notions of *work efficiency* and *scalability* to analyse parallel algorithms [7]. We refer to the time to execute a parallel algorithm on a problem of size n using p threads as $T_p(n)$. The size of a graph is determined by the number of vertices n as well as the number of edges e, but we will refer only to n for simplicity. The *depth* of an algorithm is the length of the longest sequence of dependent operations in the algorithm. The time it takes to execute such a sequence is equal to the execution time of the parallel algorithm

using an infinite number of threads, denoted by T_∞. In addition, the *work* performed by a parallel algorithm that uses p threads is the sum of the execution times of the individual threads. The *work efficiency* and the *scalability* are formally defined as follows.

Definition 3.1. (*Work efficiency*) A parallel algorithm is work efficient if and only if $W_p(n) \in O(T_1(n))$.

Definition 3.2. (*Scalability*) A parallel algorithm is scalable if and only if $\lim_{n\to\infty}\left(\lim_{p\to\infty}\frac{T_p(n)}{T_1(n)}\right) = 0$.

Informally, a work efficient parallel algorithm performs no more work than its serial version. Moreover, scalability implies that, for large enough inputs, increasing the number of threads increases the speedup of the parallel algorithm with respect to its serial version.

We also define the notion of *strong scalability* as follows [25].

Definition 3.3. (*Strong scalability*) A parallel algorithm is strongly scalable if and only if $\frac{T_1(n)}{T_p(n)} = \Theta(p)$ for large enough n.

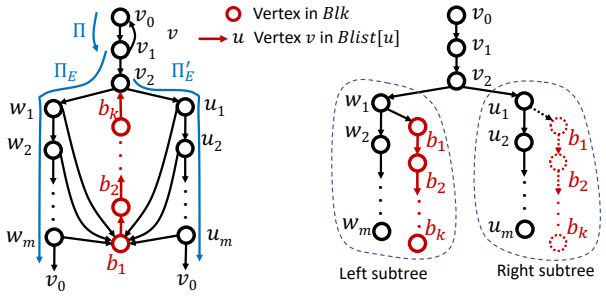
Whereas Definition 3.2 implies that the speedup $T_1(n)/T_p(n)$ achieved by a parallel algorithm with respect to its serial execution is infinite when the number of threads p is infinite, Definition 3.3 implies that the speedup is always in the order of p. Another related concept is weak scalability, which requires the speedup to be in the order of p when the input size per thread is constant. Note that both strong scalability and weak scalability guarantee scalability.

3.4 Simple cycle enumeration algorithms

The following algorithms for simple cycle enumeration perform recursive searches to incrementally update simple paths that can lead to cycles. Each algorithm iterates the vertices or edges of the graph and independently constructs a recursion tree to enumerate all the cycles starting from that vertex or edge. The difference between these algorithms is to what extent they reduce the redundant work performed during the recursive search, which we discuss next.

The Tiernan algorithm [62] enumerates simple cycles using a brute-force search. It recursively extends a simple path Π by appending a neighbor u of the last vertex v of Π provided that u is not already in Π. A clear downside of this algorithm is that it can repeatedly visit vertices that can never lead to a cycle. When searching for cycles in the graph shown in Figure 3a starting from vertex v_0, this algorithm would explore the path $b_1, \ldots, b_k$ $2m$ times. From each vertex w_i and u_i, with $i \in \{1, \ldots, m\}$, the Tiernan algorithm would explore this path only to discover that it cannot lead to a simple cycle. As noted by Tarjan [61], the Tiernan algorithm explores every simple path and, consequently, all maximal simple paths of a graph. Exploring a maximal simple path takes $O(n+e)$ time because a path can contain up to n vertices, and the Tiernan algorithm explores every outgoing edge of every vertex in that path. Given a graph with s maximal simple paths (see Table 3), the worst-case time complexity of the Tiernan algorithm is $O(s(n+e))$.

The Johnson algorithm [28] improves upon the Tiernan algorithm by avoiding the vertices that cannot lead to simple cycles. For this purpose, the Johnson algorithm maintains a set of blocked vertices *Blk* that are avoided during the search. In addition, a list of vertices $Blist[w]$ is stored for each vertex w. Whenever a vertex w is unblocked (i.e., removed from *Blk*) by the Johnson algorithm, the vertices in $Blist[w]$ are also unblocked. This unblocking process is performed recursively until no more vertices can be unblocked, which we refer to as the *recursive unblocking* procedure.



(a) Example graph **(b) Recursion tree**

Figure 3: (a) An example graph, and (b) the recursion tree constructed when searching for cycles in (a) starting from vertex v_0. The nodes of the recursion tree represent the recursive calls of the depth-first search. The dotted path of the right subtree is explored only by the Read-Tarjan algorithm.

A vertex v is blocked (i.e. added to *Blk*) when visited by the algorithm. If a cycle is found after recursively exploring every neighbor of v that is not blocked, the vertex v is unblocked upon backtracking. Otherwise, if no cycles are found by exploring the neighbors of v, v is not unblocked immediately upon backtracking. The *Blist* data structure is updated to enable unblocking of v in a later step by adding v to the list $Blist[w]$ of every neighbor w of v. This delayed unblocking of vertices enables the Johnson algorithm to discover each cycle in $O(n+e)$ time in the worst case [28]. Because this algorithm also requires $O(n+e)$ time to determine that there are no cycles, its worst-case time complexity is $O((n+e)(c+1))$. Note that because s can be exponentially larger than c [61], the Johnson algorithm is asymptotically faster than the Tiernan algorithm.

In the example shown in Figure 3a, every simple path Π that starts from v_0 and goes through $b_1, \ldots, b_k$ vertices is a maximal simple path, and thus, it cannot lead to a simple cycle. The Johnson algorithm would block $b_1, \ldots, b_k$ immediately after visiting this sequence once and then keep these vertices blocked until it backtracks to v_1, at which point, the algorithm would have finished exploring both subtrees shown in Figure 3b. As a result, the Johnson algorithm visits $b_1, \ldots, b_k$ vertices only once, rather than $2m$ times the Tiernan algorithm would visit them. Note that because these vertices get blocked during the exploration of the left subtree of the recursion tree, they are not going to be visited again during the exploration of the right subtree. Effectively, a portion of the right subtree is pruned (see the dotted path in Figure 3b) based on the updates made on *Blist* during the exploration of the left subtree. This strictly sequential depth-first exploration of the recursion tree is critically important for achieving a high pruning efficiency, but it also makes scalable parallelisation of the Johnson algorithm extremely challenging, which we are going to cover in Section 5.

The Read-Tarjan algorithm [53] also has a worst-case time complexity of $O((n+e)(c+1))$. This algorithm maintains a current path Π between a starting vertex and a frontier vertex. A recursive call of this algorithm iterates the neighbors of the current frontier vertex and performs a depth-first search (DFS). Assume

that v_0 is the starting vertex and v_1 is the frontier vertex of Π (see Figure 3a). From each neighbor $y \in \{v_0, v_2\}$ of v_1, a DFS tries to find a path extension Π_E back to v_0 that would form a simple cycle when appended to Π. In the example shown in Figure 3a, the algorithm finds two path extensions, one indicated as Π_E and one that consists of the edge $v_1 \rightarrow v_0$. The algorithm then explores each path extension by iteratively appending the vertices from it to the path Π. For each vertex x of a path extension added to Π, the algorithm also searches for an alternate path extension from that vertex x to v_0 using a DFS. In the example given in Figure 3a, the algorithm iterates through the vertices of the path extension Π_E and finds an alternate path extension Π'_E from the neighbor u_1 of v_2. If an alternate path extension is found, a child recursive call is invoked with the updated current path Π, which is $v_0 \rightarrow v_1 \rightarrow v_2$ in our example. Otherwise, if all the vertices in Π_E have already been added to the current path Π, Π is reported as a simple cycle. In our example, the Read-Tarjan algorithm explores both Π_E and Π'_E path extensions, and each one leads to the discovery of a cycle.

The Read-Tarjan algorithm also maintains a set of blocked vertices *Blk* for recursion-tree pruning. However, differently from the Johnson algorithm, *Blk* only keeps track of the vertices that cannot lead to new cycles when exploring the current path extension. The vertices in *Blk* are avoided while searching for additional path extensions that branch from the current path extension. For instance, the left subtree of the recursion tree shown in Figure 3b demonstrates the exploration of the path extension Π_E shown in Figure 3a. During the exploration of Π_E, vertices $b_1, \ldots, b_k$ are added to *Blk* immediately after visiting w_1, and they are not visited again while exploring Π_E. However, when exploring another path extension Π'_E in the right subtree, the vertices $b_1, \ldots, b_k$ are visited once again (see the dotted path of the right subtree). As a result, the Read-Tarjan algorithm visits $b_1, \ldots, b_k$ twice instead of just once. As we are going to show in Section 6, this drawback becomes an advantage when parallelising the Read-Tarjan algorithm because it enables independent exploration of different subtrees of the recursion tree.

Time window constraints can be supported trivially by all three algorithms covered. Such constraints restrict the search for simple cycles to those that occur within a time window of a given size δ. When a search for cycles starting from an edge with a timestamp t is invoked, these algorithms consider only the edges with timestamps that belong to the time window $[t : t + \delta]$. In consequence, fewer vertices are visited during the search for cycles.

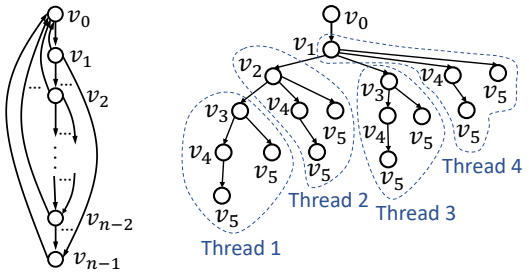
4 COARSE-GRAINED PARALLEL METHODS

The most straightforward way of parallelising the Johnson and the Read-Tarjan algorithms is to search for cycles that start from different vertices or edges in parallel. Each such search can then execute on a different thread and construct its own recursion tree. Such a coarse-grained approach to parallelising the cycle enumeration algorithms is work efficient. However, it is not scalable, which we prove in this section.

Proposition 4.1. *The coarse-grained parallel Johnson and Read-Tarjan algorithms are work efficient.*

The proof of Proposition 4.1 is trivial, and we omit it for brevity.

Theorem 4.2. *The coarse-grained parallel Johnson and Read-Tarjan algorithms are not scalable.*



(a) Example graph **(b) Recursion tree**

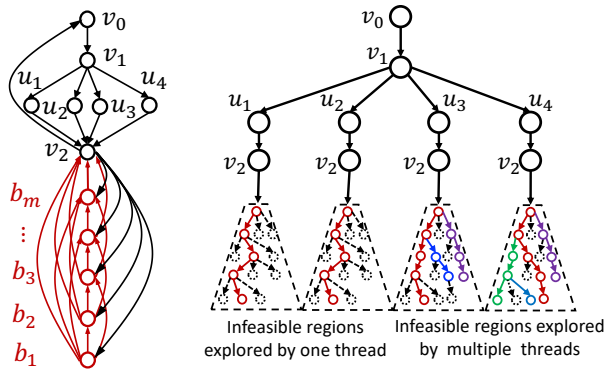
Figure 4: (a) A graph with an exponential number of simple cycles, all of which can be found by starting from the edge $v_0 \rightarrow v_1$. (b) The recursion tree of the Johnson algorithm for n = 6 constructed when the algorithm starts from the edge $v_0 \rightarrow v_1$. Whereas a coarse-grained parallel algorithm explores the complete recursion tree using a single thread, our fine-grained parallel algorithms can explore different regions of the search tree in parallel using several threads.

Proof. In this case, $T_\infty(n)$ represents the worst-case execution time of a search for cycles that starts from a single vertex or edge, and it depends on the number of cycles discovered during this search. In the worst case, a single recursive search can discover all cycles of a graph. An example of such graph is given in Figure 4a, where each vertex v_i, with $i \in \{1, \ldots, n-1\}$, is connected to v_0 and to every vertex v_j such that $j > i$. In that graph, any subset of vertices $v_2, \ldots, v_{n-1}$ defines a different cycle. Therefore, the total number of cycles in this graph is equal to the number of all such subsets $c = 2^{n-2}$. Before the search for cycles, both the Johnson and the Read-Tarjan algorithm find all vertices that start a cycle, which is only v_0 in this case. Therefore, the search for cycles will be performed only by one thread. Because all cycles of the graph are discovered by a single thread, this thread performs all the work the sequential algorithm would perform, which leads to $T_\infty(n) = T_1(n)$. Because it holds that $\lim_{n\to\infty} T_\infty(n)/T_1(n) = 1$, the coarse-grained algorithms are not scalable based on Definition 3.2. □

Summary. Theorem 4.2 shows that the main drawback of the coarse-grained parallel algorithms is their limited scalability. This limitation is apparent for the graph shown in Figure 4a, which has an exponential number of cycles in n. When using a coarse-grained parallel algorithm on this graph, all the cycles will be discovered by a single thread. Because only one thread can be effectively utilised, increasing the number of threads will not result in a reduction of the overall execution time of the coarse-grained parallel algorithm. Figure 1 shows the load imbalance exhibited by the coarse-grained parallel algorithms in practice. Section 8 demonstrates the limited scalability of coarse-grained parallel algorithms in further detail.

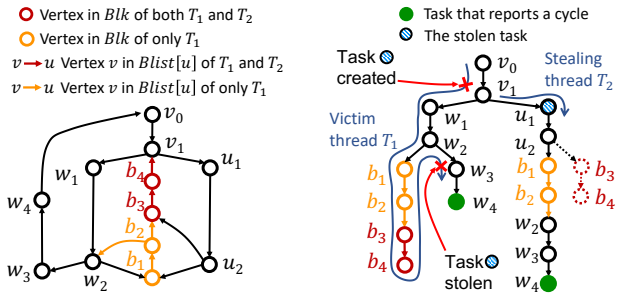
5 FINE-GRAINED PARALLEL JOHNSON

To address the load imbalance issues that manifest themselves in the coarse-grained parallel Johnson algorithm, we introduce the fine-grained parallel Johnson algorithm. The main goal of our fine-grained algorithm is to enable several threads to explore a recursion tree concurrently as shown in Figure 4b, where each thread executes a subset of the recursive calls of this tree. However, enabling several threads to explore the recursion tree concurrently



(a) Example graph (b) Recursion tree

Figure 5: (a) An example graph and (b) the recursion tree of our fine-grained parallel Johnson algorithm when searching for cycles that start from vertex v_0, which can redundantly traverse the vertices $b_1, \ldots, b_m$ several times. The serial Johnson algorithm would traverse these vertices only once.



(a) Example graph (b) Recursion tree

Figure 6: (a) An example graph and (b) the recursion tree of our fine-grained parallel Johnson algorithm when searching for cycles that start from v_0. The thread T_2 can prune the dotted part of the tree by avoiding b_3 and b_4 that the thread T_1 has blocked after creating the task stolen by T_2.

is in conflict with the sequential depth-first exploration required by the Johnson algorithm to achieve a high pruning efficiency.

The naïve approach. A straightforward way of enabling concurrent exploration of the recursion tree is to break the dependencies between different paths of the recursion tree by creating new copies of the *Blk* and *Blist* data structures when invoking child recursive calls. In such a case, there would be no data sharing between different recursive calls, and different calls would store inconsistent copies of the data structures. As a result, a recursive call would be unaware of the vertices visited and blocked by other calls that precede it in the depth-first order except for its direct ancestors in the recursion tree. Hence, this approach exhaustively explores all maximal simple paths in the graph, and is identical to the brute-force solution of Tiernan (see Section 3.4). When enumerating the simple cycles of the graph shown in Figure 5a starting from v_0, this approach would explore all $4 \times 2^{m-1}$ maximal simple paths instead of just four that would be visited by the Johnson algorithm.

Our approach. To enable different threads to concurrently explore the recursion tree in a depth-first fashion while also taking advantage of the powerful pruning capabilities of the Johnson algorithm, each thread executing our fine-grained parallel Johnson algorithm maintains its own copy of the Π, *Blk*, and *Blist* data structures. Because a thread maintains a copy of the blocked vertex set *Blk*, it will not fruitlessly visit the vertices that it has already blocked. Yet, different threads will store inconsistent copies of the *Blk* and *Blist* data structures, which could lead to some redundant work. This redundant work could happen when different threads are exploring the same infeasible region as depicted in Figure 5b. However, because the threads executing our fine-grained parallel algorithm still take advantage of the powerful pruning methods of the Johnson algorithm, the amount of work performed will be significantly lower than that of the brute-force Tiernan algorithm.

Copy-on-steal. Our fine-grained parallel Johnson algorithm implements each recursive call as a separate task. If a child task and its parent task are executed by the same thread, the child task reuses the Π, *Blk*, and *Blist* data structures of the parent task. However, if a child task has been stolen—i.e., it is executed by a thread other than the thread that created it, a new copy of these data structures are allocated by the child task. In this way, each thread of our fine-grained parallel algorithm maintains its own copy of the Π, *Blk*, and *Blist* data structures. We refer to this mechanism as *copy-on-steal*.

The problem with copying data structures between different threads upon task stealing is that the thread that has created the stolen task can modify its data structures before another thread steals this task. This inconsistency has to be somehow managed. A straightforward solution to this problem is to execute the stolen task after restoring the data structures to the state they were in when the stolen task was created. Even though such an approach would guarantee correct execution, the stealing thread would not be able to reuse the blocked vertices that have been discovered between the time the task was created and the time the task was stolen. For example, in Figure 6, assume that we are searching for simple cycles that start from v_0. While visiting v_1, the thread T_1 creates two new tasks, continues its depth-first search for simple cycles from vertex w_1 using the first task it has created, and pushes the second task it has created into its work queue to be continued from the vertex u_1. Suppose that the thread T_2 steals this second task from T_1 while T_1 is visiting w_3. At this point, T_1 has blocked the vertices b_1, b_2, b_3, and b_4 because it has discovered that it is not possible to construct a simple cycle that ends in v_0 while going through b_1, b_2, b_3, and b_4, once v_1 and w_1 have been visited. If T_2 discards the changes T_1 has made to its data structures, it would still be able to discover the simple cycle that goes through b_1 and b_2 all the way to v_0. However, T_2 would have to visit vertices b_3 and b_4 even though T_1 has already concluded that these vertices cannot lead to a simple cycle that ends in v_0, once v_1 has been visited. Therefore, a naïve state restoration will lead to unnecessary work.

We have developed an alternative solution that makes it possible for the stealing thread to capitalise on the information already discovered by the thread from which it stole the task, henceforth referred to as the *victim thread*. The stealing thread first determines its initial path Π_2 by removing the vertices the victim thread has added to its path Π_1 since the stolen task was created. The stealing thread then invokes the recursive unblocking procedure for each vertex $v \in \Pi_1 \setminus \Pi_2$, which removes these vertices from the set of blocked vertices *Blk* and recursively unblocks the vertices from

the blocked list $Blist[v]$ of each such v. For instance, in Figure 6, the stealing thread T_2 invokes recursive unblocking for vertices w_3, w_2, and w_1 because the victim thread T_1 has added them to its current path after the stolen task was created. As a result of the recursive unblocking procedure, T_2 unblocks the vertices b_1 and b_2, but the vertices b_3 and b_4 remain blocked. The vertices that remain blocked are the ones that cannot take part in a cycle that has Π_2 as a prefix because these vertices can only be unblocked by invoking the recursive unblocking procedure for the vertices in Π_2. Therefore, some of the blocked vertices discovered by the victim thread can be avoided by the stealing thread, which would not have been possible if the stealing thread simply discarded the changes the victim thread has made to its *Blk* and *Blist* data structures.

Synchronization. Without countermeasures, the copy-on-steal method can suffer from race conditions because the Π, *Blk*, and *Blist* data structures can be concurrently accessed by a victim thread and several stealing threads. For instance, if a stealing thread copies the data structures of a victim thread while the victim thread performs a recursive unblocking, the stealing thread could end up copying data that is not in a stable state. To prevent such conditions, we define critical sections and implement coarse-grained locking by maintaining a mutex per thread. The lock is acquired when the victim thread enters the recursive unblocking procedure or when the stealing threads attempt to copy the Π, *Blk*, and *Blist* data structures from the victim thread. The lock is released when the recursive unblocking or the copy operation is complete. In this way, the race conditions are eliminated and a correct execution is guaranteed.

Theoretical analysis. We now show that the fine-grained parallel Johnson algorithm is not work efficient, but scalable.

Theorem 5.1. *The fine-grained parallel Johnson algorithm is not work efficient.*

Proof. According to Lemma 3 presented by Johnson [28], a vertex cannot be unblocked more than once unless a cycle is found, and once a vertex is visited, it can be visited again only after being unblocked. As a result, a vertex is visited and unblocked at most c times by the Johnson algorithm. In the fine-grained parallel Johnson algorithm executed using p threads, each thread maintains a separate set of the data structures used for managing the blocked vertices (*Blk* and *Blist*). Because the threads are unaware of each-others blocked vertices, each vertex is visited at most pc times, c times by each thread. Additionally, a vertex cannot be visited more than s times because each maximal simple path of a graph is explored by a different thread in the worst case, and during each simple path exploration, a vertex is visited at most once. Therefore, the maximum number of times a vertex can be visited by the fine-grained parallel Johnson algorithm is $\min\{s, pc\}$. When the Johnson algorithm visits a vertex, it also iterates through its outgoing edges, thus visiting all n vertices executes in $O(n+e)$ time. Prior to executing the recursive search, this algorithm checks if the input graph contains at least one cycle using a single thread. This check can be performed in $O(n+e)$ time. As a result, the work performed by the fine-grained parallel Johnson algorithm is

$$W_p(n) \in \begin{cases} O\left(n+e\right), & \text{if } c = 0, \\ O\left(pc(n+e)\right), & \text{if } p < s/c \text{ and } c \neq 0, \\ O\left(s(n+e)\right), & \text{otherwise.} \end{cases} \quad (1)$$

When $c > 0$, $p > 1$, and $s > c$, the work performed by the fine-grained parallel Johnson algorithm $W_p(n)$ is greater than the execution time $T_1(n)$ of the sequential Johnson algorithm. Thus, the fine-grained parallel Johnson algorithm is not work efficient. □

The work inefficiency of the fine-grained parallel Johnson algorithm occurs if more than one thread performs the work the sequential Johnson algorithm would perform between the discovery of two cycles. We illustrate this behaviour using the graph from Figure 5a, which contains $c = 4$ cycles and $s = c \times 2^{m-1}$ maximal simple paths, each starting from vertex v_0. When discovering each cycle, the fine-grained parallel Johnson algorithm explores an infeasible region of the recursion tree, as shown in Figure 5b, in which vertices $b_1, \ldots, b_m$ are visited. If this infeasible region is explored using a single thread, each vertex b_i, with $i \in \{1, \ldots, m\}$, will be visited exactly once. However, if p threads are exploring the same infeasible region of the recursion tree, vertices $b_1, \ldots, b_m$ will be visited up to p times because the threads are unaware of each-others blocked vertices. In this case, the fine-grained parallel Johnson algorithm performs more work than necessary, and, thus, it is not work efficient. Additionally, each infeasible region of the recursion tree that visits vertices $b_1, \ldots, b_k$ can be executed by at most $s/c = 2^{m-1}$ threads because there are 2^{m-1} maximal simple paths that can be explored in each infeasible region. In this case, each vertex b_i, with $i \in \{1, \ldots, m\}$, is visited up to s times, and, thus, the fine-grained parallel Johnson algorithm behaves as the Tiernan algorithm (see Section 3.4).

Lemma 1. *The depth $T_\infty(n)$ of the fine-grained parallel Johnson algorithm is in $O(n+e)$.*

Proof. The worst-case depth of this algorithm occurs when a thread performs copy-on-steal and explores a path of length n. Performing copy-on-steal requires $O(n+e)$ operations because at most n vertices in Π and *Blk*, and at most e pairs of vertices in *Blist* are accessed during the copy-on-steal. Exploring the path of length n requires $O(n+e)$ operations because a recursive call that visits a vertex v of this path also iterates through every outgoing edge of v. As a result, the depth of this algorithm is $T_\infty(n) \in O(n+e)$. □

Theorem 5.2. *The fine-grained parallel Johnson algorithm is scalable when $\lim_{n\to\infty} c = \infty$.*

Proof. For this algorithm, $T_1(n) \in O((n+e)(c+1))$ and $T_\infty(n) \in O(n+e)$ (see Lemma 1). Given our assumption that $\lim_{n\to\infty} c = \infty$, we have $\lim_{n\to\infty} \frac{T_\infty(n)}{T_1(n)} = \lim_{n\to\infty} \frac{n+e}{(n+e)(c+1)} = 0$. Therefore, this algorithm is scalable based on Definition 3.2. Note that it is sufficient for c to increase sublinearly in n for this proof to hold. □

Even though the fine-grained parallel Johnson algorithm is scalable, a strong or weak scalability is not guaranteed due to the work inefficiency of this algorithm. Nevertheless, our experiments show that this algorithm is strongly scalable in practice (see Figure 9).

Summary. Our relaxation of the strictly depth-first-search-based recursion-tree exploration reduces the pruning efficiency of the Johnson algorithm. In the worst case, the fine-grained parallel Johnson algorithm could perform as much work as the brute-force Tiernan algorithm does—i.e., $O(s(n+e))$. However, in practice this worst-case scenario does not happen (see Section 8). In addition,

our fine-grained parallel Johnson algorithm can suffer from synchronisation issues in some rare cases (see Section 8) because our copy-on-steal mechanism can lead to long critical sections. In the next section, we introduce a fine-grained parallel algorithm that is scalable, work efficient, and less prone to synchronisation issues.

6 FINE-GRAINED PARALLEL READ-TARJAN

In this section, we show that the Read-Tarjan algorithm is straightforward to parallelise in a scalable and work efficient way. Because the Read-Tarjan algorithm allocates a new copy of the *Blk* set during each path extension computation, a recursive call can compute different path extensions in an arbitrary order. Additionally, discovery of a new path extension results in the invocation of a single recursive call, and these calls can be executed in an arbitrary order. Consequently, several threads can concurrently explore different paths of the same recursion tree constructed by the Read-Tarjan algorithm for a given starting edge. There are no data dependencies or ordering requirements between different calls apart from those that exist between a parent and a child. To exploit the parallelism available during the recursion tree exploration, we execute each recursive call and each path extension computation as a separate task that can be independently scheduled. We refer to the resulting algorithm as the fine-grained parallel Read-Tarjan algorithm.

To prevent different threads from concurrently modifying the current path Π being explored, each task receives a copy of Π from its parent task. To improve the pruning efficiency, each task also receives a copy of the blocked vertex set *Blk* from its parent task. However, unlike the Johnson algorithm, the fine-grained parallel Read-Tarjan algorithm does not communicate the *Blk* sets from child tasks back to their parent tasks. This parallel algorithm also takes advantage of the copy-on-steal mechanism (see Section 5) to eliminate unnecessary copy operations between tasks executed by the same thread. However, because the Read-Tarjan algorithm does not use the *Blist* data structures and the recursive unblocking procedure used by the Johnson algorithm, the critical sections of the fine-grained parallel Read-Tarjan algorithm is much shorter than those of the fine-grained parallel Johnson algorithm.

Theoretical analysis. We now show that the fine-grained parallel Read-Tarjan algorithm is both work efficient and scalable.

Theorem 6.1. *The fine-grained parallel Read-Tarjan algorithm is work efficient.*

Proof. Because the Read-Tarjan algorithm executes $O(c)$ recursive calls [53], and each path extension exploration invokes at most one recursive call, our fine-grained parallel Read-Tarjan algorithm is executed using $O(c)$ tasks. Each task executes a DFS that explores at most n vertices and e edges. Additionally, each task receives a copy of Π and *Blk*, and because these data structures contain at most n vertices, the overhead of copying them is $O(n)$. Therefore, this algorithm performs $O(n+e)$ work per task. Because the same amount of work is performed even if there are no cycles in the graph, the total amount of work this algorithm performs is $W_p(n) = O\left((n+e)(c+1)\right)$. As a result, based on Definition 3.1, the fine-grained parallel Read-Tarjan algorithm is work efficient. □

Using the graph from Figure 5a, threads of the fine-grained parallel Read-Tarjan algorithm that start from v_0 independently explore four different path extensions $\Pi_E = v_1 \rightarrow u_i \rightarrow v_2 \rightarrow v_0$, with $i \in \{1 \dots 4\}$. When exploring a path extension Π_E, each thread invokes a DFS starting from v_2 to explore a different infeasible region of the search tree as shown in Figure 5b. Because the DFS would fail to find any other path extensions, the same infeasible region will not be explored more than once. Therefore, the amount of work the fine-grained parallel Read-Tarjan algorithm performs does not increase compared to its single-threaded execution.

Lemma 2. *The depth $T_\infty(n)$ of the fine-grained parallel Read-Tarjan algorithm is in $O(n(n+e))$.*

Proof. In the worst case, a thread executing this algorithm invokes a recursive call for each vertex of its longest simple cycle, which has a length of at most n. Each recursive call executes a DFS that can visit n vertices and e edges of the graph. Therefore, the depth of this algorithm is $T_\infty(n) \in O\left(n(n+e)\right)$. □

The worst-case depth of the fine-grained parallel Read-Tarjan algorithm can be observed when this algorithm is executed on the graph given in Figure 4a. This graph has $c = 2^{n-2}$ cycles and the length of its longest cycle $v_0 \rightarrow v_1 \dots \rightarrow v_{n-1} \rightarrow v_0$ is n. The algorithm invokes a recursive call for each vertex of the cycle and performs a DFS in each such call, which leads to $T_\infty \in O(n(n+e))$.

Theorem 6.2. *The fine-grained parallel Read-Tarjan algorithm is strongly scalable when $\lim_{n\to\infty} c/n = \infty$.*

Proof. Because the fine-grained parallel Read-Tarjan algorithm is work-efficient, we can apply Brent's rule [10] as follows:

$$\frac{T_1(n)}{p} \leq T_p(n) \leq \frac{T_1(n)}{p} + T_\infty(n).$$

Substituting $T_1(n)$ with $O((c+1)(n+e))$ and $T_\infty(n)$ with $O(n(n+e))$ (see Lemma 2), for a positive constant C_0, it holds that

$$1\Big/\left(\frac{1}{p} + \frac{T_\infty(n)}{T_1(n)}\right) = 1\Big/\left(\frac{1}{p} + C_0\frac{n}{c+1}\right) \leq \frac{T_1(n)}{T_\infty(n)} \leq p.$$

Given that $\lim_{n\to\infty} c/n = \infty$, there exist $n_0 > 0, C_1 > 0$ such that if $n > n_0$, then $(c+1)/n > C_1 p$. Thus, for every $n > n_0$, it holds that $kp \leq \frac{T_1(n)}{T_\infty(n)} \leq p$, where $k = C_1/(C_0 + C_1) < 1$. As a result, $\frac{T_1(n)}{T_\infty(n)} = \Theta(p)$, which, based on Definition 3.3, completes the proof. Note that this proof requires c to grow superlinearly with n. □

Summary. The pruning efficiency of the Read-Tarjan algorithm is not affected by the fine-grained parallelisation. The fine-grained parallel Read-Tarjan algorithm performs $O((n+e)(c+1))$ work: the same as the work performed by its serial version. In addition, the synchronization overheads of the fine-grained parallel Read-Tarjan algorithm are not as significant as those of the fine-grained Johnson algorithm because of its shorter critical sections. Furthermore, this algorithm is the only asymptotically-optimal parallel algorithm for cycle enumeration for which we have proved strong scalability.

7 EXTENSIONS TO TEMPORAL CYCLES

To efficiently enumerate temporal cycles, we take advantage of the 2SCENT algorithm contributed by Kumar and Calders [32], which is based on the Johnson algorithm. The 2SCENT algorithm introduces two highly effective optimisations, called *closing times* and *path bundling*. The closing times optimisation extends the concept of the blocked vertex set *Blk* to keep track of the blocked

temporal edges. The path bundles optimisation enables a single search to simultaneously explore several temporal paths that share a common sequence of vertices. To enable efficient temporal cycle enumeration, we have incorporated both optimisations into our coarse- and fine-grained parallel algorithms introduced in the prior sections. Given that our parallel algorithms for temporal cycle enumeration are based on our parallel formulations of the Johnson and the Read-Tarjan algorithms, our conclusions regarding the work efficiency and scalability, summarised in Table 1, remain valid. However, we omit the respective proofs due to space constraints.

The 2SCENT algorithm also uses a preprocessing step that reduces the number of vertices visited during its search for cycles. However, this preprocessing step has a strictly sequential formulation because it processes the edges in the increasing order of their timestamps. Moreover, the time complexity of the preprocessing step of the 2SCENT algorithm is in the order of the time complexity of its recursive search for cycles. In our own implementation, we use a lighter-weight linear-time preprocessing algorithm, inspired by the algorithms for computing strongly-connected components [17, 60], which can be parallelised in a scalable manner.

Our scalable preprocessing method computes a *cycle-union* for each starting edge, which is the set of vertices that take part in one or more temporal cycles starting from that edge. The cycle-union of a given starting edge $v_0 \rightarrow v_1$ is computed as the intersection between the set of vertices reachable from vertex v_1 and the set of vertices from which vertex v_0 is reachable. When computing temporal cycles, we say that a vertex u is reachable from a vertex v if there exists a simple path from v to u, in which the edges appear in the increasing order of their timestamps. When performing the reachability analysis under time window constraints, we only consider the paths that belong to a time window of a given size δ. This preprocessing method is lightweight because each cycle-union can be computed in $O(n + e)$ time, similarly to the computation of a strongly-connected component [17] of a graph, and it is also straightforward to parallelise because the cycle-unions can be computed independently for each starting edge or starting vertex.

8 EXPERIMENTAL EVALUATION

This section evaluates the performance of our coarse- and fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms on temporal graphs. As Table 2 shows, we are the only ones to offer fine-grained parallel versions of the state-of-the-art algorithms by Johnson and Read-Tarjan. However, all the methods covered in Table 2 can be parallelised using the coarse-grained approach we described in Section 4, which we use as our main comparison point. Furthermore, we provide direct comparisons with 2SCENT [32] because it is the only other work that supports time window constraints and can perform temporal cycle enumeration.

Because exhaustive enumeration of simple cycles is not tractable in general, it is common to search for cycles under some constraints (see Table 2). In the experiments, we use time-window constraints when searching for both simple and temporal cycles. Our experiments are performed using the temporal graphs listed in Table 4. The TR, FR, and MS graphs are from *Harvard Dataverse* [26], the NL graph is from *Konect* [33], the AML graph is from the *AML-Data* repository [3], and the rest are from *SNAP* [35]. We control the complexity of cycle enumeration by selecting the time-window size δ appropriately for each graph. The window sizes used in our experiments are given in Table 4. We do not report simple cycle enumeration results for the MS graph because, in this case, our algorithms did not finish in $12h$ even if we set $\delta = 1s$. Note that we use larger time windows when enumerating temporal cycles because the complexity of enumerating temporal cycles is lower.

Table 4: Temporal graphs. The time span T is in days. Figure 7a and 7b use the time window sizes δ_s and δ_t, respectively.

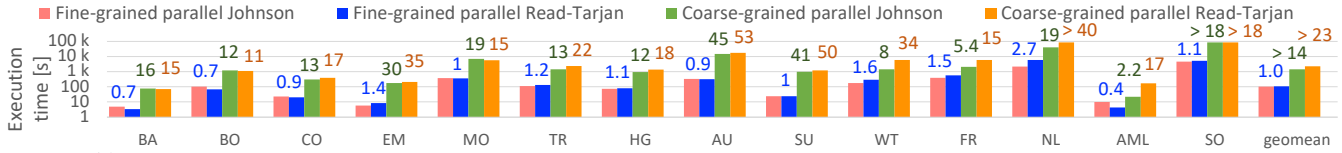
Graph	n	e	T	δ_s	δ_t
bitcoinalpha (BA)	3.3 k	24 k	1901	71h	3000h
bitcoinotc (BO)	4.8 k	36 k	1903	75h	1000h
CollegeMsg (CO)	1.3 k	60 k	193	3h	96h
email-Eu-core (EM)	824	332 k	803	4h	144h
mathoverflow (MO)	16 k	390 k	2350	30h	288h
transactions (TR)	83 k	530 k	1803	72h	800h
higgs-activity (HG)	278 k	555 k	6	3000s	72h
askubuntu (AU)	102 k	727 k	2613	20h	336h
superuser (SU)	138 k	1.1 M	2773	5h	168h
wiki-talk (WT)	140 k	6.1 M	2277	12h	144h
friends2008 (FR)	481 k	12 M	1826	1300s	5h
wiki-dynamic (NL)	1 M	20 M	3602	29s	1000s
messages (MS)	313 k	26 M	1880	/	4h
AML-Data (AML)	10 M	34 M	30	48h	720h
stackoverflow (SO)	2.0 M	48 M	2774	3h	66h

The experiments are performed on a cluster of four Intel Xeon Phi 7210 - Knights Landing (KNL) processors [57].[1] An Intel KNL CPU has 64 physical cores and supports 256 simultaneous threads, which makes it an ideal platform for evaluating the scalability of parallel algorithms. This cluster enables execution of 1024 simultaneous threads on 256 physical CPU cores. We use the *Threading Building Blocks* (TBB) [31] library for parallelising the algorithms on a single processor, and we distribute the execution of the algorithms across multiple processors using the Message Passing Interface (MPI) [12]. When using distributed execution, each processor stores a copy of the input graph in its main memory and searches for cycles starting from a different set of graph edges. The starting edges are divided among the processors such that when the edges are ordered in the ascending order of their timestamps, k consecutive edges in that order are assigned to k different processors. Each processor then uses its own dynamic scheduler to balance the workload of the recursive searches that start from its given set of starting edges.

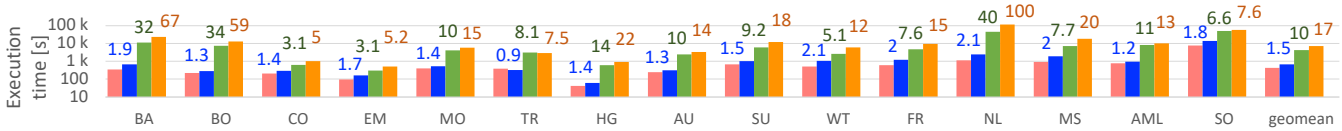
The granularity of the tasks has a significant impact on the performance of the parallel cycle enumeration algorithms. Figure 7 compares the coarse- and fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms. These comparisons are provided for both simple cycle enumeration and temporal cycle enumeration, respectively in Figures 7a and 7b. We observe that our fine-grained parallel algorithms outperform the coarse-grained parallel algorithms by an order of magnitude both for simple cycle enumeration and for temporal cycle enumeration. This behavior is a clear outcome of the scalability of our fine-grained parallelisation.

Figure 8 shows the impact of the time window size on the fine-grained and coarse-grained parallel Johnson algorithms when performing temporal cycle enumeration. Note that enumerating cycles

[1] Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.



(a) Performance of the parallel algorithms for finding all simple cycles within a time window of size δ_s given in Table 4.



(b) Performance of the parallel algorithms for finding all temporal cycles within a time window of size δ_t given in Table 4.

Figure 7: Comparisons between the fine-grained and the coarse-grained parallel versions of the Johnson and the Read-Tarjan algorithms for (a) simple and (b) temporal cycle enumeration using 1024 threads. The numbers above the bars show the execution time of each algorithm relative to that of the fine-grained parallel Johnson algorithm for the same benchmark.

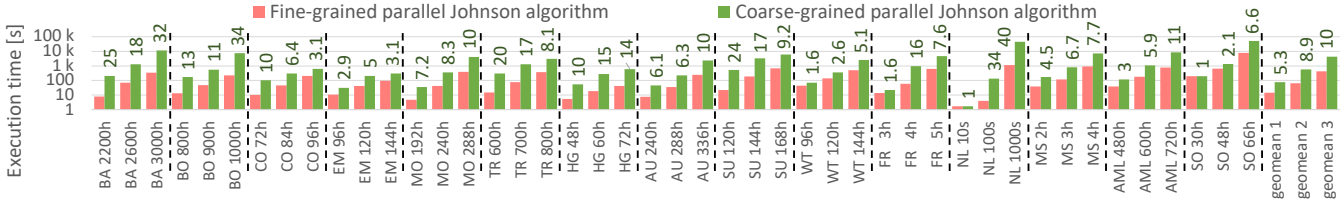


Figure 8: The speed-up of the fine-grained parallel Johnson algorithm with respect to the coarse-grained parallel Johnson algorithm for different time window sizes. Larger time windows increase the performance gap between the two algorithms.

in larger time windows is more challenging because larger time windows contain a larger number of cycles. Interestingly, increasing the size of the time window increases the performance gap between the fine-grained and the coarse-grained Johnson algorithms.

The work performed by the algorithms evaluated can be quantified based on the number of edges visited during their execution. Based on this metric, our fine-grained parallel Johnson algorithm on average performs 6.1% more work than the work-efficient coarse-grained parallel Johnson algorithm does when enumerating simple cycles. The maximum difference observed is around 14%. The difference is always less than 1% when enumerating temporal cycles.

The evaluation of the scalability of the coarse-grained parallel and fine-grained parallel algorithms is performed in Figure 9. All three algorithms evaluated perform temporal cycle enumeration and use up to 1024 software threads. We also report the performance of the 2SCENT algorithm [32]. In this setting, the only difference between our single-threaded Johnson algorithm and 2SCENT is the scalable pre-processing method we introduced in Section 7.

The performance of the fine-grained parallel Johnson and Read-Tarjan algorithms improves linearly until 256 threads. When we use more than 256 threads, the CPU cores start performing simultaneous multithreading, which leads to a sublinear performance scaling. In case of the WT graph, our fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms are respectively 435× and 470× faster than their serial versions when we use 1024 threads. In addition, our fine-grained parallel Johnson algorithm is on average 260× faster than 2SCENT when 2SCENT completes in 24 hours.

The Johnson and the Read-Tarjan algorithms have comparable performances as shown in Figure 7. However, our fine-grained parallel Read-Tarjan algorithm is slightly slower than our fine-grained parallel Johnson algorithm. This behaviour is expected given that the Read-Tarjan algorithm performs more edge visits than the Johnson algorithm despite having the same worst-case time complexity (see Section 3.4). In our experiments, the fine-grained parallel Read-Tarjan algorithm on average performs 47% more edge visits than the fine-grained parallel Johnson algorithm.

The simple cycle enumeration results for AML are clear outliers. The coarse-grained parallel Read-Tarjan algorithm performs 2.4× more edge visits than the coarse-grained parallel Johnson algorithm, yet it is 7.7× slower due to a more severe load imbalance. However, our fine-grained parallel Johnson algorithm is 2.3× slower than our fine-grained parallel Read-Tarjan algorithm. In this case, the fine-grained parallel Johnson algorithm is only 37× faster than its serial version. However, the fine-grained parallel Read-Tarjan algorithm exhibits a good scaling and is 214× faster than its serial version.

Our analysis has shown that such a role reversal is not caused by the work inefficiency of our fine-grained parallel Johnson algorithm, which performs only 10% more edge visits than its serial version when enumerating the simple cycles of AML. The reason is the synchronization overheads exerted on our fine-grained parallel Johnson algorithm by recursive unblocking (see Section 5). In fact, the synchronization overheads of our fine-grained parallel Johnson algorithm are visible only when enumerating the simple cycles of AML, which can be explained by a very low cycle-to-vertex ratio observed in this case. Because a vertex is blocked if it cannot take part in a cycle, the probability of a vertex being blocked is higher when the cycle-to-vertex ratio is lower. In consequence, more vertices are unblocked during the recursive unblocking of the fine-grained parallel Johnson algorithm, which leads to longer critical sections and more contention on the locks. Nevertheless, our fine-grained parallel Johnson algorithm achieves a good trade-off between pruning efficiency and lock contention in most cases.

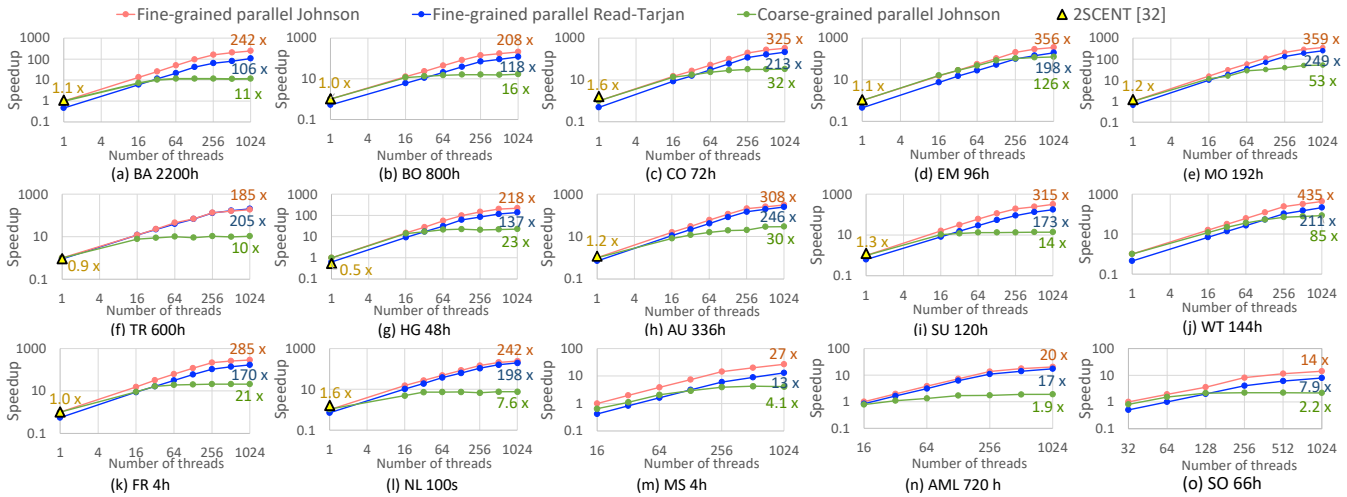


Figure 9: Effect of the number of threads on the performance of temporal cycle enumeration algorithms. The baseline is our fine-grained parallel Johnson algorithm. The relative performance of 2SCENT [32] is shown when it completes in 24h. Note that the 2SCENT implementation is single-threaded and the single-threaded execution results are not available for all graphs.

9 CONCLUSIONS

This work has made three contributions to the area of parallel cycle enumeration. First, we have introduced fine-grained parallel versions of the Johnson and the Read-Tarjan algorithms for enumerating simple and temporal cycles. We have shown that our fine-grained parallel algorithms are scalable both in theory and in practice. We have evaluated our algorithms on 15 temporal graph datasets, and demonstrated a near-linear performance scaling on a compute cluster with a total number of 256 CPU cores that can execute 1024 simultaneous threads, where our parallel algorithms achieved an up to 470× speedup with respect to their serial versions.

Secondly, we have shown that the coarse-grained parallel versions of the Johnson and the Read-Tarjan algorithms are not scalable. When using 1024 simultaneous software threads, our fine-grained parallel algorithms are on average an order of magnitude faster than the coarse-grained parallel algorithms. In addition, the performance gap between the fine-grained and coarse-grained parallel algorithms increases as we use more physical cores. The performance gap increases as we increase the time window size as well.

Thirdly, we have shown that our fine-grained parallel Johnson algorithm is not work efficient. Yet, it outperforms our fine-grained parallel Read-Tarjan algorithm in most of our experiments. In some rare cases, our fine-grained parallel Johnson algorithm can suffer from synchronisation overheads. In such cases, our fine-grained parallel Read-Tarjan algorithm offers a more scalable alternative.

ACKNOWLEDGMENTS

The support of Swiss National Science Foundation (project number 172610) for this work is gratefully acknowledged. The authors would like to thank Haris Pozidis and Radu Stoica from IBM Research Europe - Zurich for their valuable comments on this work.

REFERENCES

[1] Ehab Abdelhamid, Ibrahim Abdelaziz, Panos Kalnis, Zuhair Khayyat, and Fuad Jamour. 2016. ScaleMine: Scalable Parallel Frequent Subgraph Mining in a Single Large Graph. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Salt Lake City, UT, USA, 716–727. doi: 10.1109/SC.2016.60.

[2] Charu C. Aggarwal and Haixun Wang (Eds.). 2010. *Managing and Mining Graph Data*. Advances in Database Systems, Vol. 40. Springer US, Boston, MA. doi: 10.1007/978-1-4419-6045-0.

[3] Erik Altman. 2021. AML-Data. Available online: https://github.com/IBM/AML-Data. Accessed: 2022-05-30.

[4] Albert-László Barabási and Márton Pósfai. 2016. *Network science*. Cambridge University Press, Cambridge, United Kingdom, Chapter The scale-free property, 1–57.

[5] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. 2013. Optimal Listing of Cycles and st-Paths in Undirected Graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1884–1896. doi: 10.1137/1.9781611973105.134.

[6] Jovan Blanuša, Radu Stoica, Paolo Ienne, and Kubilay Atasu. 2020. Manycore clique enumeration with fast set intersections. *PVLDB* 13, 12 (Aug. 2020), 2676–2690. doi: 10.14778/3407790.3407853.

[7] Guy E. Blelloch and Bruce M. Maggs. 2010. *Parallel Algorithms*. CRC Press, London, England, Chapter 25, 25.1–25.40.

[8] 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 (Aug. 1996), 55–69. doi: 10.1006/jpdc.1996.0107.

[9] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi: 10.1145/324133.324234.

[10] Richard P. Brent. 1974. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM* 21, 2 (April 1974), 201–206. doi: 10.1145/321812.321815.

[11] Anna D. Broido and Aaron Clauset. 2019. Scale-free networks are rare. *Nat Commun* 10, 1 (Dec. 2019), 1017. doi: 10.1038/s41467-019-08746-5.

[12] CORPORATE The MPI Forum. 1993. MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing - Supercomputing '93*. ACM Press, Portland, Oregon, United States, 878–883. doi: 10.1145/169627.169855.

[13] Huanqing Cui, Jian Niu, Chuanai Zhou, and Minglei Shu. 2017. A Multi-Threading Algorithm to Detect and Remove Cycles in Vertex- and Arc-Weighted Digraph. *Algorithms* 10, 4 (Oct. 2017), 115. doi: 10.3390/a10040115.

[14] G. Danielson. 1968. On finding the simple paths and circuits in a graph. *IEEE Trans. Circuit Theory* 15, 3 (Sept. 1968), 294–295. doi: 10.1109/TCT.1968.1082837.

[15] Apurba Das and Srikanta Tirthapura. 2019. Shared-Memory Parallel Maximal Biclique Enumeration. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, Hyderabad, India, 34–43. doi: 10.1109/HiPC.2019.00016.

[16] P. Erdős and T. Gallai. 1959. On maximal paths and circuits of graphs. *Acta Mathematica Academiae Scientiarum Hungaricae* 10, 3-4 (Sept. 1959), 337–356. doi: 10.1007/BF02024498.

[17] Lisa K. Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On Identifying Strongly Connected Components in Parallel. In *Parallel and Distributed Processing*. Vol. 1800. Springer Berlin Heidelberg, Berlin, Heidelberg, 505–511. doi: 10.1007/3-540-45591-4_68.

[18] Norman E. Gibbs. 1969. A Cycle Generation Algorithm for Finite Undirected Linear Graphs. *J. ACM* 16, 4 (Oct. 1969), 564–568. doi: 10.1145/321541.321545.

[19] Pierre-Louis Giscard, Paul Rochet, and Richard C. Wilson. 2017. Evaluating balance on social networks from their simple cycles. *Journal of Complex Networks* 5 (May 2017), 750–775. doi: 10.1093/comnet/cnx005.

[20] Roberto Grossi. 2016. Enumeration of Paths, Cycles, and Spanning Trees. In *Encyclopedia of Algorithms*. Springer New York, New York, NY, 640–645. doi: 10.1007/978-1-4939-2864-4_728.

[21] A. Gupta and C. Selvidge. 2005. Acyclic modeling of combinational loops. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.* IEEE, San Jose, CA, 343–348. doi: 10.1109/ICCAD.2005.1560091.

[22] Anshul Gupta and Toyotaro Suzumura. 2021. Finding All Bounded-Length Simple Cycles in a Directed Graph. arXiv:2105.10094 [cs.DS]

[23] László Hajdu and Miklós Krész. 2020. Temporal Network Analytics for Fraud Detection in the Banking Sector. In *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*. Springer International Publishing, Cham, Switzerland, 145–157. doi: 10.1007/978-3-030-55814-7_12.

[24] Md. Nazrul Islam, S. M. Rafizul Haque, Kaji Masudul Alam, and Md. Tarikuzzaman. 2009. An approach to improve collusion set detection using MCL algorithm. In *2009 12th International Conference on Computers and Information Technology*. IEEE, Dhaka, Bangladesh, 237–242. doi: 10.1109/ICCIT.2009.5407133.

[25] Joseph JaJa. 1992. *Introduction to parallel algorithms*. Addison Wesley, Boston, MA.

[26] Jaroslaw Jankowski, Radosław Michalski, and Piotr Bródka. 2017. Spreading processes in multilayer complex network within virtual world. doi: 10.7910/DVN/V6AJRV.

[27] Zhi-Qiang Jiang, Wen-Jie Xie, Xiong Xiong, Wei Zhang, Yong-Jie Zhang, and Wei-Xing Zhou. 2013. Trading networks, abnormal motifs and stock manipulation. *Quantitative Finance Letters* 1, 1 (Dec. 2013), 1–8. doi: 10.1080/21649502.2013.802877.

[28] Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (March 1975), 77–84. doi: 10.1137/0204007.

[29] T. Kamae. 1967. A Systematic Method of Finding All Directed Circuits and Enumerating All Directed Paths. *IEEE Trans. Circuit Theory* 14, 2 (June 1967), 166–171. doi: 10.1109/TCT.1967.1082699.

[30] Steffen Klamt and Axel von Kamp. 2009. Computing paths and cycles in biological interaction graphs. *BMC Bioinformatics* 10, 1 (Dec. 2009), 181. doi: 10.1186/1471-2105-10-181.

[31] Alexey Kukanov. 2007. The Foundations for Scalable Multicore Software in Intel Threading Building Blocks. *ITJ* 11, 04 (Nov. 2007), 309–322. doi: 10.1535/itj.1104.05.

[32] Rohit Kumar and Toon Calders. 2018. 2SCENT: an efficient algorithm for enumerating all simple temporal cycles. *PVLDB* 11, 11 (July 2018), 1441–1453. doi: 10.14778/3236187.3236197.

[33] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web - WWW '13 Companion*. ACM Press, Rio de Janeiro, Brazil, 1343–1350. doi: 10.1145/2487788.2488173.

[34] Yung-Keun Kwon and Kwang-Hyun Cho. 2007. Analysis of feedback loops and robustness in network evolution based on Boolean models. *BMC Bioinformatics* 8, 1 (Dec. 2007), 430. doi: 10.1186/1471-2105-8-430.

[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. Available online: https://snap.stanford.edu/data. Accessed: 2022-05-30.

[36] G. Loizou and P. Thanisch. 1982. Enumerating the cycles of a digraph: A new preprocessing strategy. *Information Sciences* 27, 3 (Aug. 1982), 163–182. doi: 10.1016/0020-0255(82)90023-8.

[37] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, Indianapolis Indiana USA, 135–146. doi: 10.1145/1807167.1807184.

[38] Prabhaker Mateti and Narsingh Deo. 1976. On Algorithms for Enumerating All Circuits of a Graph. *SIAM J. Comput.* 5, 1 (March 1976), 90–99. doi: 10.1137/0205007.

[39] Nav Mathur. 2017. *Graph Technology for Financial Services*. Technical Report. Neo4J. 1–14 pages. https://neo4j.com/use-cases/financial-services Accessed: 2022-05-30.

[40] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2 (Nov. 2015), 1–39. doi: 10.1145/2818185.

[41] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2014. Graph structure in the web — revisited: a trick of the heavy tail. In *Proceedings of the 23rd International Conference on World Wide Web - WWW '14 Companion*. ACM Press, Seoul, Korea, 427–432. doi: 10.1145/2567948.2576928.

[42] Mark Needham. 2019. *Graph algorithms : practical examples in Apache Spark and Neo4j*. O'Reilly, Beijing. isbn: 978-1492047681.

[43] O. Neiroukh, S.A. Edwards, and Xiaoyu Song. 2008. Transforming Cyclic Circuits Into Acyclic Equivalents. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 27, 10 (Oct. 2008), 1775–1787. doi: 10.1109/TCAD.2008.2003305.

[44] S. Noel, E. Harley, K.H. Tam, M. Limiero, and M. Share. 2016. CyGraph: Graph-Based Analytics and Visualization for Cybersecurity. In *Handbook of Statistics*. Vol. 35. Elsevier, Oxford, England, 117–167. doi: 10.1016/bs.host.2016.07.001.

[45] Girish Keshav Palshikar and Manoj M. Apte. 2008. Collusion set detection using graph clustering. *Data Min Knowl Disc* 16, 2 (April 2008), 135–164. doi: 10.1007/s10618-007-0076-8.

[46] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, Cambridge, United Kingdom, 601–610. doi: 10.1145/3018661.3018731.

[47] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards bridging theory and practice: hop-constrained s-t simple path enumeration. *PVLDB* 13, 4 (Dec. 2019), 463–476. doi: 10.14778/3372716.3372720.

[48] J. Ponstein. 1966. Self-Avoiding Paths and the Adjacency Matrix of a Graph. *SIAM J. Appl. Math.* 14, 3 (March 1966), 600–609. doi: 10.1137/0114051.

[49] Sri Harsha Pothukuchi and Amit Dhuria. 2021. Deterministic loop breaking in multi-mode multi-corner static timing analysis of integrated circuits. Patent No. 11003821, Filed Feb 21, 2020, Issued May 11, 2021.

[50] Zhu Qing, Long Yuan, Zi Chen, Jingjing Lin, and Guojie Ma. 2020. Efficient Parallel Cycle Search in Large Graphs. In *Database Systems for Advanced Applications*. Vol. 12113. Springer International Publishing, Cham, Switzerland, 349–367. doi: 10.1007/978-3-030-59416-9_21.

[51] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *PVLDB* 11, 12 (Aug. 2018), 1876–1888. doi: 10.14778/3229863.3229874.

[52] Michael J. Quinn. 2004. *Parallel programming in C with MPI and openMP*. McGraw-Hill, Dubuque, Iowa.

[53] R. C. Read and R. E. Tarjan. 1975. Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees. *Networks* 5, 3 (July 1975), 237–252. doi: 10.1002/net.1975.5.3.237.

[54] Rodrigo Caetano Rocha and Bhalchandra D. Thatte. 2015. Distributed cycle detection in large-scale sparse graphs. In *2015 Simpósio Brasileiro de Pesquisa Operacional (SBPO)*. SOBRAPO, Porto de Galinhas, Pernambuco, Brasil, 1–12. doi: 10.13140/RG.2.1.1233.8640.

[55] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Farminton Pennsylvania, 472–488. doi: 10.1145/2517349.2522740.

[56] SAS. 2021. SAS OPTGRAPH Procedure: Graph Algorithms and Network Analysis. Available online: https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/procgralg/procgralg_optgraph_examples.htm. Accessed: 2022-05-30.

[57] Avinash Sodani. 2015. Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, Cupertino, CA, USA, 1–24. doi: 10.1109/HOTCHIPS.2015.7477467.

[58] Toyotaro Suzumura and Hiroki Kanezashi. 2021. Anti-Money Laundering Datasets: InPlusLab Anti-Money Laundering Datasets. Available online: https://github.com/IBM/AMLSim. Accessed: 2022-05-30.

[59] J. Szwarcfiter and P. Lauer. 1976. A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics* 16 (1976), 192–204.

[60] R. Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1 (1972), 146–160.

[61] Robert Tarjan. 1973. Enumeration of the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 2, 3 (Sept. 1973), 211–216. doi: 10.1137/0202017.

[62] James C. Tiernan. 1970. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM* 13, 12 (Dec. 1970), 722–726. doi: 10.1145/362814.362819.

[63] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. doi: 10.1145/79173.79181.

[64] Fei Wang, Peng Cui, Jian Pei, Yangqiu Song, and Chengxi Zang. 2020. Recent Advances on Graph Analytics and Its Applications in Healthcare. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Virtual Event CA USA, 3545–3546. doi: 10.1145/3394486.3406469.

[65] Jim Webber. 2021. *Powering Real-Time Recommendations with Graph Database Technology*. Technical Report. Neo4J. 1–7 pages. https://neo4j.com/use-cases/real-time-recommendation-engine Accessed: 2022-05-30.

[66] Herbert Weinblatt. 1972. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. *J. ACM* 19, 1 (Jan. 1972), 43–56.

[67] J. T. Welch. 1965. Numerical applications: Cycle algorithms for undirected linear graphs and some immediate applications. In *Proceedings of the 1965 20th National Conference*. ACM Press, Cleveland, Ohio, United States, 296–301. doi: 10.1145/800197.806053.

[68] Xiaoping Zhou, Xun Liang, Jichao Zhao, and Shusen Zhang. 2018. Cycle Based Network Centrality. *Sci Rep* 8, 1 (Dec. 2018), 11749. doi: 10.1038/s41598-018-30249-4.