Noname manuscript No.
(will be inserted by the editor)

Efficient Parallel Graph Trimming by Arc-Consistency

Bin Guo · Emil Sekerinski

Received: date / Accepted: date

arXiv:2107.12720v2 [cs.DC] 7 Apr 2022

Abstract Given a large data graph, trimming techniques can reduce the search space by removing vertices without outgoing edges. One application is to speed up the parallel decomposition of graphs into strongly connected components (SCC decomposition), which is a fundamental step for analyzing graphs. We observe that graph trimming is essentially a kind of arc-consistency problem, and AC-3, AC-4, and AC-6 are the most relevant arc-consistency algorithms for application to graph trimming. The existing parallel graph trimming methods require worst-case $\mathcal{O}(nm)$ time and worst-case $\mathcal{O}(n)$ space for graphs with n vertices and m edges. We call these parallel AC-3-based as they are much like the AC-3 algorithm. In this work, we propose AC-4-based and AC-6-based trimming methods. That is, AC-4-based trimming has an improved worst-case time of $\mathcal{O}(n+m)$ but requires worst-case space of $\mathcal{O}(n+m)$; compared with AC-4-based trimming, AC-6-based has the same worst-case time of $\mathcal{O}(n+m)$ but an improved worst-case space of $\mathcal{O}(n)$. We parallelize the AC-4-based and AC-6-based algorithms to be suitable for shared-memory multi-core machines. The algorithms are designed to minimize synchronization overhead. For these algorithms, we also prove the correctness and analyze time complexities with the work-depth model.

In experiments, we compare these three parallel trimming algorithms over a variety of real and synthetic graphs on a multi-core machine, where each core corresponding to a worker. Specifically, for the maximum number of traversed edges per worker by using 16 workers, AC-3-based traverses up to 58.3 and 36.5 times more edges than AC-6-based trimming and AC-4-based trimming, respectively. That is, AC-6-based trimming traverses much fewer edges than other methods, which is meaningful especially for implicit graphs. In particular, for the practical

Bin Guo
Department of Computing and Software, McMaster University, Hamilton, Ontario
Tel.: +1204-9529589
E-mail: guob15@mcmaster.ca

Emil Sekerinski
Department of Computing and Software, McMaster University, Hamilton, Ontario
E-mail: emil@mcmaster.ca

running time, AC-6-based trimming achieves high speedups over graphs with a large portion of trimable vertices.

Keywords graph, trimming, parallel, constraint satisfaction problem (CSP), arc-consistency (AC)

Acknowledgment

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

1 Introduction

In numerous applications, like social networks [56], pattern matching [12], communication networks [34], knowledge graphs [60], and model verification [27], data is organized into directed graphs with vertices for objects and edges for their relationships. The large size of such graphs motivates graph *trimming*, i.e. removing vertices without outgoing edges to speed up subsequent processing, such as cycle detection [39], k-core decomposition [4], and in particular graph decomposition [29]. For instance, for the communication network *wiki-talk* [34] with 2.4 million vertices, surprisingly 94.5% of the vertices can be trimmed, which greatly reduces the graph size for subsequent processing.

One issue is that trimming such unqualified vertices may cause other vertices to become useless. Naively repeating the trimming process may lead to a quadratic worst-case time complexity. Thus, linear time bounded graph trimming methods are desired. Additionally, the availability of multi-core processors motivates efficient parallelization of such graph trimming methods. Here, a *worker* is a working process corresponding to a physical core for a multi-core processor.

To the best of our knowledge, there exists little work on parallel trimming over large data graphs, except for [30, 29, 54, 32, 11]. In these studies, the graph trimming is adopted to quickly remove the vertices without out-going edges so that can speed up the strongly connected component (SCC) decomposition. In [30], McLendon et al. first apply a linear time graph trimming method to remove *size*-1 SCCs; however, a parallel version is not provided. In [29], Hong et al. propose a quadratic time graph trimming technique by "peeling" *size*-1 and *size*-2 SCCs, i.e. SCCs with only 1 or 2 vertices. The "peeling" step is straightforward: (1) all vertices are checked in parallel and the *trimmable* ones are removed, which may cause other vertices to become trimmable; (2) this process is repeated until no vertex can be removed from the graph. The advantage of this graph trimming technique is that it can be highly parallelized without difficulties. However, it has a quadratic worst-case time complexity of $\mathcal{O}(nm/\mathcal{P}+\alpha)$, where n is the number of vertices, m is the number of edges, $\mathcal{P}$ is the number of workers, and α is the depth of the algorithm (explained in the next section). This parallel trimming technique is widely used in later SCC decomposition methods [54, 32, 11].

In this work, we apply the well-known *arc-consistency* (AC) algorithms to graph trimming. Based on that, we not only classify existing graph trimming algorithms but also propose a new graph trimming algorithm that improves the time and space complexities by an order of magnitude. Before discussing these contributions, we first show an application of graph trimming, the SCC decomposition in large graphs [30, 29, 54, 32, 11].

1.1 An Application of Graph Trimming

Detecting the strongly connected components in directed graphs, the so-called SCC decomposition, is one of the fundamental analysis steps in many applications such as social networks [34], communication networks [55], knowledge networks [2], and model checking graphs [27]. Given a directed graph $G = (V, E)$, a *strongly connected component* of G is a maximal set of vertices $C \subseteq V$ such that every two vertices u and v in C are reachable from each other. The early SCC algorithms

are based on depth-first search (DFS) [57,15]. However, lexicographical-first DFS is P-complete and even the random DFS is hard to parallelize [49,1]. The breadth-first search (BFS) based Forward-Backward (FW-BW) algorithm has been proposed. Unlike DFS, BFS can be parallelized without difficulty. Starting from a selected pivot vertex, FW-BW performs a forward BFS to identify the vertex set FW that the pivot can reach, followed by a backward BFS to identify the set BW that can reach the pivot. The intersection between FW and BW is an SCC that contains the pivot [21]. In the worst case, each vertex can be selected as a pivot to travel the whole graph in $\mathcal{O}(m)$, which yields a quadratic time complexity of $\mathcal{O}(mn)$ [21]. In [14,22], the worst-case time complexity is improved to $\mathcal{O}(m \log n)$ by using a divide-and-conquer approach.

Interestingly, real-world graphs demonstrate SCC features that follow the *power-law property* [29], that is, several large SCCs take the majority of vertices and the rest of them are trivial SCCs. More importantly, most of the trivial SCCs are size-1 SCCs. The key observation is that a size-1 SCC is easy to identify: it has zero incoming edges or zero outgoing edges. Therefore, graph trimming can be used to remove such size-1 SCCs in parallel with less computational effort than FW-BW and thus in practice can speed up FW-BW. Analogously to size-1 SCCs, size-2 [29] and size-3 [32] SCCs also can be trimmed but with more computational effort.

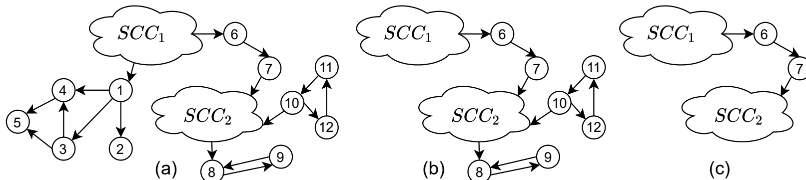

Fig. 1: A graph that can use graph trimming to remove size-1, size-2 and size-3 SCCs.

Figure 1 illustrates the FW-BW algorithm with graph trimming. Figure 1(a) shows that there are altogether two large SCCs, SCC_1 and SCC_2 (whose sizes are greatly larger than 3) and the other trivial SCCs. It is easy to see that vertices v_1 to v_7 are size-1 SCCs, vertices v_8 and v_9 compose one size-2 SCC, and vertices v_{10} to v_{12} compose one size-3 SCC. In Figure 1(b), we first try to trim all size-1 SCCs: (1) in the first repetition, vertices v_5 and v_2 are removed since they have no outgoing edge, which causes vertex v_4 to have no outgoing edges; (2) in the second repetition, vertex v_4 is removed, which causes vertex v_3 to have no outgoing edges; (3) in the third repetition, vertex v_3 is removed, which causes vertex v_1 to have no outgoing edges; (4) in the final repetition, vertex v_1 is removed. Similarly, in Figure 1(c), size-2 and size-3 SCCs can also be removed. Note that vertices v_6 and v_7, located between two large SCCs, are size-1 SCCs, but they can not be directly trimmed. After the first round of graph trimming, the FW-BW algorithm can identify two large SCCs, SCC_1 and SCC_2, which also can be deleted from the graph. After removing the two large SCCs, the second round of trimming can remove vertices v_6 and v_7 with two iterations.

The naive trimming method as used in FW-BW [22] has a quadratic time complexity of $\mathcal{O}(mn)$ in the worst case. The drawback of such trimming is that it sacrifices the better worst-case time complexity of FW-BW, $\mathcal{O}(m \log n)$ [22]. From our experiments, we noticed that the running time of such trimming will dramatically increase with the number of peeling steps α because of the increasing number of repetitions. This is why FW-BW with trimming as in [29] is only efficient for *small-word graphs*. The small-world property states that the *diameters* (greatest shortest-path distance between any pair of vertices) of graphs are very small even for very large graph instances [59], which always implies a small number of peeling steps. The focus of this paper is to improve traditional graph trimming so that algorithms based on FW-BW with trimming [30, 29, 54, 32, 11] can be more efficient, especially for non-small-world graphs.

For instance, in [32], the parallel SCC decomposition algorithm ISPAN is proposed. It combines the power of graph trimming and FW-BW, both of which can be efficiently parallelized. In particular, graph trimming is used at two places; before large SCC detection, trimming is used to remove the size-1 SCCs; after the large SCC is detected, trimming is again used to remove size-1, size-2, and size-3 SCCs. The evaluation uses 56 workers over 16 graphs and shows that ISPAN achieves a significant speedup of 171 - 6591 times over the sequential DFS-based Tarjan's algorithm [15] and of 85 - 1475 times over the parallel DFS-based UFSCC algorithm [8].

1.2 The New Method

Essentially, graph trimming is a kind of *Constrain Satisfaction Problem* (CSP), that is, a set of vertices must satisfy a number of constraints or limitations, e.g. each vertex needs at least one outgoing edge or it will be removed as a size-1 SCC. Many filtering algorithms [19] have been proposed to remove values that obviously do not belong to the solution of a CSP and thus reduce the search space. The closest related filtering algorithms to graph trimming are *Arc-Consistency* (AC) algorithms for binary CSPs, in particular AC-3 [41], AC-4 [45], and AC-6 [23].

Table 1 summarizes the time and space complexities of these three algorithms. AC-6 has the best worst-case time and space complexities. AC-4 and AC-6 have the same time complexity. However, in reality, AC-3 and AC-6 perform sometimes better than AC-4 due to AC-4 always running close to its worst-case time. The details are explained in the next section.

Algorithm	Time ($\mathcal{O}$)	Space ($\mathcal{O}$)
AC-3	ed^3	$e + kd$
AC-4	ed^2	ed^2
AC-6	ed^2	ed

Table 1: The worst-case time and space complexities of three arc-consistency algorithms, where e is the number of arcs, k is the number of variables, d is the size of the largest variable's domain.

The key observation is that the graph trimming technique in [30] is like AC-4 (*AC-4-based*). Also, the other widely used graph trimming technique in [29,54, 32,11] is like AC-3 (*AC-3-based*). Stimulated by AC-6, we design a novel graph trimming algorithm (*AC-6-based*). Compared to AC-3-based and AC-4-based trimming, our new AC-6-based trimming is more complicated and not easy to parallelize. To the best of our knowledge, there exists little work on parallel AC algorithms [13,33]. In this work, we design efficient sequential and parallel versions of AC-6-based trimming for a multi-threaded shared memory architecture.

Table 2 summarizes the complexities of different parallel graph trimming algorithms in the *work-depth* model, where the *work* is the number of operations used by the algorithm and the *depth* is the length of the longest sequential dependence in the computation. We can see that all three trimming algorithms have the different parallel depth, and AC-3-based trimming has a smallest depth. AC-3-based trimming has larger worst-case work and time complexities than the other two algorithms. The AC-6-based and AC-4-based algorithms have the same worst-case work and time complexities. We show that AC-6-based trimming traverses fewer edges and uses less space. For example, over all tested graphs in our first experiments, AC-6-based trimming reduces the number of traversed edges 3.3 - 192.5 times compared with AC-4-based trimming and 1.5 - 44 times compared with AC-3-based trimming.

Trimming	On-The-Fly	Worst-Case ($\mathcal{O}$) Work	Depth	Space
AC-3-based	✓	$\alpha(n+m)$	αDeg_{out}	n
AC-4-based	✗	$n+m$	$\lvert Q_p\rvert Deg_{in} Deg_{out}$	$n+m$
AC-6-based	✓	$n+m$	$\lvert Q_p\rvert {Deg_{in}}^2$	n

Table 2: The worst-case work, depth, and space complexities of parallel graph trimming algorithms, where n is the number of vertices, m is the number of edges, $\mathcal{P}$ is the number of total workers, α is the number of peeling steps, Deg_{out} is the maximal out-degree for all vertices, Deg_{in} is the maximal in-degree for all vertices, $|Q_p|$ is the upper-bound size of waiting sets among $\mathcal{P}$ workers such that sometimes $|Q_p| \geq \alpha$.

To parallelize the AC-4-based and AC-6-based algorithms, the conventional way is with mutual exclusion by using `Lock` and `Unlock` operations that guarantee exclusive access to data structures shared by multiple workers. In this work, however, we use atomic primitives to minimize the synchronization overhead.

1.3 On-the-fly Property

The *on-the-fly* property [8] means an algorithm can run on an implicit graph defined as $G = (v_0, \texttt{POST})$, where v_0 is the *initial vertex* and $\texttt{POST}(v)$ is a function that returns all of the *successors* of vertex v. One drawback of the FW-BW method is that the backward search requires reverse edges, which means all edges have to be loaded into memory; storing the graph as an adjacent list with only outgoing

edges is not sufficient. The on-the-fly property is necessary when handling large graphs that occur in e.g. verification [42], as it may allow the algorithm to terminate early after processing only a fraction of the graph without needing memory space for loading the whole graph. It also benefits algorithms that rely on implicit graphs [48], in which the edges are calculated online by function $\texttt{POST}(v)$.

The on-the-fly properties of three graph trimming algorithms are summarized in Table 2. It is easy to see that the AC-4-based trimming cannot run on-the-fly as it requires reverse edges and thus the whole graph must be loaded into the memory. AC-3-based and AC-6-based trimming can run on-the-fly as they only rely on the post of vertex v when traversing each vertex $v \in V$ and their space usage is bounded by $\mathcal{O}(n)$. However, compared with AC-3-based trimming, AC-6-based trimming needs much less work. Note that on implicit graphs, all the edges are computed online by the function $\texttt{POST}(v)$, which typically costs more running time than directly loading edges from memory like with explicit graphs. The proposed AC-6-based trimming traverses fewer edges than AC-3-based trimming and thus performs better on implicit graphs.

1.4 Contribution

The contributions of this work are summarized below:

- We provide a formal definition of graph trimming based on the Constraint Satisfaction Problem (CSP) and Arc-Consistency (AC). Following three well-known arc-consistency algorithms, that is, AC-3, AC-4, and AC-6, we categorize the existing graph trimming algorithms as AC-3-based [29, 54, 32, 11] and AC-4-based algorithms [30].
- We revisit the existing parallel AC-3-based algorithm. We give the detailed steps of the AC-4-based algorithm and parallelize it using atomic primitives.
- We propose a novel AC-6-based algorithm that has optimized time and space complexities. We further parallelize the AC-6-based algorithm using atomic primitives. These are the main contributions of this work.
- For all three graph trimming algorithms, we formally discuss their correctness, time complexity, and space complexity. The time complexities for parallel algorithms are analyzed in the work-depth model.
- Finally, for all three parallel trimming algorithms, our experiments compare the number of traversed edges and practical running time with 1 to 16 workers over a variety of real and synthetic graphs.

This paper is organized as follows. Section 2 provides the background. Section 3 discusses the relation between graph trimming and Arc Consistency. Section 4 revisits the AC-3-based graph trimming algorithm. Section 5 provides the AC-4-based graph trimming algorithm. Section 6 proposes the AC-6-based graph trimming algorithm. The related work is discussed in Section 7. In Section 8, we discuss the implementations. In Section 9, we provide the experimental evaluation over a variety of data graphs. Section 10 concludes three trimming algorithms and discusses the future work.

2 Preliminaries

Given a directed graph $G = (V, E)$, let $n = |V|$ and $m = |E|$ be the numbers of vertices and edges, respectively. A vertex v in graph G is also denoted as $v(G)$. As opposed to an undirected graph, $(v, w) \in E$ does not imply that $(w, v) \in E$. The *post* of vertex v in G is the set of all the successors (outgoing edges) of v, defined by $v.post = \{w \mid (v, w) \in E\}$; when the context is clear, we use $v.post$ instead of $v(G).post$. The *pre* of vertex v is the set of all the predecessors (ingoing edges) of v, defined by $v.pre = \{w \mid (w, v) \in E\}$. For each vertex $v \in V$, its *out-degree* is the number of successors $|v.post|$ and its *in-degree* is the number of predecessors $|v.pre|$. To analyze the time complexity of trimming algorithms, we use Deg_{out} and Deg_{in} to denote the maximum out-degree and in-degree among all vertices in a graph G, respectively.

A transposed graph $G^T = (V, E^T)$ is equivalent to the graph $G = (V, E)$ with all its edges reversed, $E^T = \{(w, v) \mid (v, w) \in E\}$. It is easy to see that $v(G).post = v(G^T).pre$ and $v(G).pre = v(G^T).post$ for each $v \in V$. A transposed graph G^T can be generated in order to efficiently obtain $v(G).pre$ without traversing the whole original graph G.

Definition 1 (Trimmed Graph) Given a directed graph $G = (V, E)$, the trimmed graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is a maximal subgraph of G, where each vertex has at least one outgoing edge, formally $\forall v \in V' : v.post \neq \emptyset$.

This work focuses on the graph trimming algorithms that can obtain trimmed graphs according to Definition 1. Without changing the original graph $G = (V, E)$, each vertex $v \in V$ is assigned a status, denoted as $v.status$, with values `LIVE` and `DEAD`, which indicates if vertex v is located in the graph (live) or removed (dead), respectively.

2.1 Graph Storage

In this work, explicit graphs and implicit graphs are discussed. Explicit graphs are typically stored in the *compressed sparse row* (CSR) format [28, 29]. This format uses two arrays to represent the graph: an $\mathcal{O}(n)$-sized array stores an index to the beginning of each vertex's adjacency list and an $\mathcal{O}(m)$-sized array stores each vertex's adjacency list. The CSR representation is compact, memory bandwidth-friendly, and thus suitable for efficient graph traversals. It is easy to see that successors of each vertex $v \in V$ are ordered and thus can be traversed one by one in order.

On modern computers, registers can move data around in single clock cycles. However, registers are very expensive. The dynamic random access memory is very cheap but takes hundreds of cycles after a request to receive the data. To bridge this gap between them are the cache memories, named L1, L2, L3 in decreasing speed and cost. If the data is stored in memory sequentially, the CPU can prefetch the data into the cache for fast accessing, which is cache-friendly. For a graph stored in CSR format, we can see that sequentially traversing all edges is cache-friendly as the cache hit rate is high, but randomly traversing all edges is not cache-friendly as the cache hit rate is low.

Implicit graphs are defined as $G = (v_0, \texttt{POST})$ assuming that all the vertices in G are reachable from vertex v_0, where v_0 is the *initial vertex* and $\texttt{POST}(v)$ is a function that returns all of the *successors* of vertex v, that is, $\texttt{POST}(v) = v.post$. One kind of implicit graphs are model checking graphs [48] such that for each vertex v in a graph G, all the edges are calculated online by $\texttt{POST}(v)$. Another kind of implicit graphs are external graphs such that all the edges are stored on disks sequentially; once a vertex v is traversed, the edges of v are loaded into memory. The advantage of implicit graphs is that they allow handling large graphs with limited memory usage. However, much running time is spent on generating the edges via $\texttt{POST}(v)$. If an algorithm can run on implicit graphs without loading the whole graphs into memory, we say this algorithm has the *on-the-fly property*

2.2 Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) [52,19] can be defined as a triple $P = (X, D, C)$, where $X = \{X_1, ..., X_n\}$ is a set of n variables, $D = \{D(X_1), ..., D(X_n)\}$ is the set of n domains such that $D(X_i)$ is a set of possible values of variable X_i, and C is a set of constraints that specify allowable combinations of values. A *solution* of a constraint set C is an instantiation of the variables such that all constraints are satisfied. Here, we restrict to *binary constraints* C_{ij} between pairs (X_i, X_j) of variables, i.e. $C = \{C_{ij} \mid i, j \in 1...n\}$.

2.3 Arc-Consistency

A value $v_i \in D_i$ is *binary consistent* with a constraint C_{ij} if there exists $v_j \in D_j$ such that (v_i, v_j) satisfies C_{ij}. Then $v_j \in D_j$ is called a *support* of $v_i \in D_j$ over C_{ij}. A value $v_i \in D_i$ is *viable* if it has supports for every D_j such that each $C_{ij} \in C$ is satisfied. A variable in a CSP is *arc-consistent* (AC) if every value in its domain satisfies each binary constraint $C_{ij} \in C$.

Several AC algorithms have been proposed for removing values that are not viable. AC-1 [40] revisits all the binary arcs that have to be revisited once some domains are reduced. Improving on AC-1, algorithm AC-2 [40] only revisits the arcs that are affected by reducing some domains. Algorithm AC-3 [40,41] generalizes and simplifies AC-2.

Algorithm 1 shows the detailed steps of AC-3. Initially, the global set Q includes all constraint arcs $C_{ij} \in C$ (line 1). Each constraint arc C_{ij} is picked and then removed from the set Q (line 3), and each pair of values in the domains $D(X_i)$ and $D(X_j)$ are checked by the procedure `Revise` (line 4), that is, for each value $v_i \in D(X_i)$, if $D(X_j)$ does not contain a value v_j such that (v_i, v_j) satisfies the constraint C_{ij}, the value v_i is repeatedly removed from $D(X_i)$ (lines 7 - 13). If $D(X_i)$ is changed, the associated constraints C_{ij} are placed into Q again (lines 5 and 6). This process is repeated until the set Q becomes empty (line 2). It is easy to see that AC-3 is not efficient since the revision of any domain will force neighbor constraints to be revisited again.

AC-4 [45] improves the worst-case time complexity of AC-3 by using auxiliary data structures, *supports* and *counters*, but its average running time is close to the worst-case time complexity. However, AC-3 has better average running time and

Algorithm 1: AC-3

```
   input  : An arc-consistency problem P = (X, D, C)
   output: A filtered domain set D
 1 Q := C
 2 while Q ≠ ∅ do
 3 │ remove a constraint C_ij from Q
 4 │ if Revise(X_i, X_j, D) then
 5 │ │ for X_k ∈ {X_k' : C_k'i ∈ C} \ {X_j} do
 6 │ │ │ Q := Q ∪ {C_ki}

 7 procedure Revise(X_i, X_j, D)
 8 │ revised := FALSE
 9 │ for v_i ∈ D(X_i) do
10 │ │ if no value v_j in D(X_j) satisfies C_ij then
11 │ │ │ delete v_i from D(X_i)
12 │ │ │ revised := TRUE
13 │ return revised
```

space usage than AC-4 and thus AC-3 is always preferred to AC-4 [52] in practice. Algorithm AC-6 [5] combines AC-3 and AC-4. It only records one support for each value, unlike AC-4 which records all supports, since a single support is enough to prove that a value is viable. Because of this, AC-6 has the same worst-case time complexity as AC-4 but averagely performs much better than AC-4 in many applications. Further, AC-6 needs less space than AC-4 since for each value only a single support is recorded. The corresponding time, work, and space complexities in the worst case are summarized in Table 1.

2.4 Parallel Complexity Analysis

We assume parallel programs run on shared-memory multi-core machines, where different cores access a shared global memory simultaneously. Shared-memory parallelism has many advantages, but writing correct, efficient, and scalable shared-memory multi-core programs is difficult. In this paper, the parallel graph trimming algorithms are designed for nested *fork-join* parallelism, in which a *fork* specifies workers that can execute in parallel, and a *join* specifies a synchronization point among multiple workers. In practice, our parallel algorithms can be implemented by OpenMP [16].

We analyze our parallel algorithms in the *work-depth* model [15, 53], where the *work*, denoted as $\mathcal{W}$, is the total number of operations that are used by the algorithm and the *depth*, denoted as $\mathcal{D}$, is the longest length of sequential operations [31]. This model is particularly convenient for analyzing nested parallel algorithms. Assuming that a scheduler dynamically load-balances a parallel computation across all available workers, the expected running time is $\mathcal{O}(\mathcal{W}/\mathcal{P} + \mathcal{D})$ when using $\mathcal{P}$ workers. For the multi-core architecture, a worker is a working process corresponding to a physical core. In particular, for sequential algorithms, the work and the depth terms are equivalent. A parallel algorithm is *work-efficient* if

its work is asymptotically equal to the work of the fastest sequential algorithm for the same problem [6].

Definition 2 (Number of Peeling Steps α) Given a directed graph $G = (V, E)$, integer α is defined as the number of peeling steps: in the peeling process, a step removes all vertices with zero out-degrees and thus decrements the out-degrees of adjacent neighbors. Neighbors whose out-degrees becomes zero must be removed in the next step. This is repeated until no vertices have an out-degree of zero.

To analyze the depth of our trimming algorithms, the number of peeling step α is introduced in Definition 2, which is analogous to the *peeling-complexity* proposed to analyze the depth of parallel k-core decomposition algorithms [18]. Essentially, α is a property of graphs, which indicates the longest chain size of trimmable vertices. It is easy to see that α can be as large as n in the worst case, e.g. in a chain graph. However, α is significantly smaller than n in practice. For example, for the graphs in our experiments, α ranges from 3 to $11,686$, which is small compared to their millions of vertices.

2.5 Atomic Primitives

All algorithms are implemented for shared-memory parallel machines; that is, multiple workers access the same memory [53]. The conventional way is with mutual exclusion by using `Lock` and `Unlock` operations that guarantee exclusive access to data structures shared by multiple workers. Compared with using locks, a implementation by using atomic primitives has much less synchronization overhead and the unexpected delay while workers are within critical sections can be highly reduced. The compare&swap (`CAS`) and fetch&add (`FAA`) operations are universal atomic primitives that are supported on the majority of current parallel architectures [58,43,44].

As shown in Algorithm 2, the `CAS` atomic primitive takes three arguments, a variable (location) x, an old value a and a new value b. It checks the value of the variable x, and if it equals to the old value a, it updates the pointer to the new value b and then returns *true*; otherwise, it returns *false* to indicate that the updating fails. Here, we use a pair of *angular brackets*, $\langle ... \rangle$, to indicate that the operations in between are executed atomically.

Algorithm 2: CAS(x, a, b)

```
1 ⟨ if x = a then
2 |   x := b; return TRUE
3 else return FALSE ⟩                          /* ⟨...⟩ atomic */
```

The `FAA` atomic primitive is shown in Algorithm 3. The old value of x is fetched and added by a. The new value of x is returned. For instance, there is a race condition when one worker is executing "$x := x + a$" and the other worker is executing "$x := x + b$" concurrently. Using `FAA` can efficiently get the correct result without workers affecting each other.

Algorithm 3: FAA(x, a)

```
1 ⟨x := x + a⟩                                   /* ⟨...⟩ atomic */
```

3 Graph Trimming as Arc Consistency

Intuitively, we can regard graph trimming as a graph with a constraint that each vertex has at least one outgoing edge. Based on this observation, we define graph trimming as an arc-consistency problem with one single variable, viz. the set of all vertices, and a single binary constraint, viz. each vertex must have at least one outgoing edge as one support. Then, trimming a graph means determining the domain of available vertices.

More formally, given a directed graph $G = (V, E)$, graph trimming can be defined as an arc-consistency problem (X, D, C) with variables $X = \{X_1\}$, domains $D = \{D(X_1)\}$ and constraint $C = \{C_{11}\}$. Here, we assume that $X_1 = V$ and $C_{11} = E$, that is, each vertex $v_1 \in D(V)$ must has at least one support vertex $v_1' \in D(V)$ in the same domain, where $(v_1, v_1') \in E$.

Consequently, three important AC algorithms, AC-3, AC-4, and AC-6, can be applied to graph trimming. Interestingly, we find that one widely used graph trimming method [29,54,32,11] is analogous to AC-3 (we call it AC-3-based). The other [30] is analogous to AC-4 (we call it AC-4 based); however, the detailed steps are not discussed, and a parallel version is not provided. As a contribution, we design a novel graph trimming algorithm based on AC-6 (we call it AC-6-based).

In Table 3, we summarize the notations that will be frequently used when discussing the graph trimming algorithms.

Notation	Description
$G = (V, E)$	a directed graph with n vertices and m edges
$G^T = (V, E^T)$	a transposed graph of $G = (V, E)$
$u(G).deg_{in}$	the in-degree of u in G
$u(G).deg_{out}$	the out-degree of u in G
$u(G).post$	the successors of u in G
$u(G).pre$	the predecessor of u in G
$u(G).S$	the supporting set used by AC-6-Trimming
$u(G).status$	the status (`LIVE` or `DEAD`) of u in G
Deg_{in}	the maximal in-degree among all vertices in G
Deg_{out}	the maximal out-degree among all vertices in G
α	the number of peeling steps in G
Q	the waiting set used by AC-4-Trimming and AC-6-Trimming
$\mathcal{P}$	the total number of workers
Q_p	a private waiting set used by workers p
$\lvert Q_p \rvert$	the upper-bound size of waiting sets among $\mathcal{P}$ workers

Table 3: The notations that frequent used when discussing the graph trimming algorithms.

4 AC-3-Based Graph Trimming

In the graph trimming problem, there exists only a single variable and a single constraint. Therefore, AC-3, as shown in Algorithm 1, can be simplified when applied to graph trimming. The idea is straightforward: (1) for all vertices in a graph, the vertices with zero out-degrees are removed; (2) this process is repeated until the graph does not change. This naive trimming method is widely used [29, 54, 32, 11] for quickly removing the size-1 SCCs, but the correctness and complexities are not formally discussed. In this section, we revisit the existing parallel AC-3-based algorithm for graph trimming and formally discuss the correctness and complexities. The sequential AC-3-based algorithm is immediate and not discussed further.

4.1 The Parallel AC-3-Based Algorithm

Algorithm 4 shows the detailed steps of the parallel AC-3-based algorithm for graph trimming. The procedure `ZeroOutDegree`(v) (lines 11 - 14) returns `TRUE` if vertex v has at least one available outgoing edge and `FALSE` otherwise. All vertices in V are initialized as `LIVE`. After partitioning V into $V_1 \dots V_{\mathcal{P}}$, we have $\mathcal{P}$ workers execute the procedure $\texttt{Trim}_p(V_p)$ in parallel (lines 4 and 5). The main procedure $\texttt{Trim}_p(V_p)$ (lines 7 - 10) removes the vertices in V_p that have an out-degree of zero. The removing process repeats until the graph does not change (lines 2 - 6). One advantage of this algorithm is that it is easy to parallelized: each copy of procedure $\texttt{Trim}_p(V_p)$ for a worker p (line 5) can run in parallel with only *change* as the sole shared variable.

Algorithm 4: Parallel AC-3-based Graph Trimming

input : Graph $G = (V, E)$
output: Trimmed Graph G

```
1  for v ∈ V do v.status := LIVE
2  repeat
3  |  change := FALSE
4  |  partition V into V_1 ... V_P
5  |  Trim_1(V_1) || ... || Trim_P(V_P)
6  until ¬ change
7  procedure Trim_p(V_p)
8  |  for v ∈ V_p : v.status = LIVE do
9  |  |  if ZeroOutDegree(v) then
10 |  |  |  v.status := DEAD; change := TRUE
11 procedure ZeroOutDegree(v)
12 |  for w ∈ v.post with w.status = LIVE do
13 |  |  return FALSE
14 |  return TRUE
```

For the specific implementations in [29, 32, 11], there are two strategies to improve the AC-3-based algorithm for graph trimming.

- If the transposed graph G^T is loaded in memory, another constraint can be considered, that each vertex $v \in V$ must have at least one available incoming edge. That means the in-degree also be checked (line 9). In this case, more size-1 SCCs can be quickly trimmed. The problem is that the transposed graph is required, which costs $\mathcal{O}(n + m)$ memory space.
- The number of repetitions can be limited to a constant number like 3 or the repetitions stop when the number of removed vertices is less than a threshold like 100 (line 6). The problem is that some of the trimable vertices may not be removed. This strategy is sometimes effective at reducing the computational time but sometimes not, since the worst-case time complexity is not improved.

Correctness. For the correctness, trimming has to be sound and complete. Soundness means that all removed vertices, which are assigned a status of `DEAD`, must have no outgoing edges or only edges to removed vertices:

$$\begin{aligned} \mathit{sound}(V) \;\equiv\; \forall v \in V :\; & v.\mathit{status} = \texttt{DEAD} \implies \\ & (\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD}) \end{aligned} \tag{1}$$

Completeness means that all vertices that have no outgoing edges or have only outgoing edges to removed vertices are removed:

$$\begin{aligned} \mathit{complete}(V) \;\equiv\; \forall v \in V :\; & (\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD}) \implies \\ & v.\mathit{status} = \texttt{DEAD} \end{aligned} \tag{2}$$

The algorithm has to ensure both soundness and completeness for all vertices in the graph:

$$\mathit{sound}(V) \;\wedge\; \mathit{complete}(V) \tag{3}$$

which is equivalent to:

$$\forall v \in V : v.\mathit{status} = \texttt{DEAD} \;\equiv\; (\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD}) \tag{4}$$

For arguing about the correctness of for-loops, we use following rule: consider the loop **for** $x \in X$ **do** S and let $P(X')$ be a predicate. If (1) initially $P(\emptyset)$ holds and (2) under precondition $P(X')$ the body S establishes postcondition $P(X' \cup \{x\})$ for any $X' \subset X$ and $x \in X \setminus X'$, then finally $P(X)$ holds; X' is the set of visited elements and $P(X')$ is the loop invariant.

Theorem 1 (Soundness) *For any $G = (V, E)$ Algorithm 4 terminates with $\mathit{sound}(V)$.*

Proof The invariant of the for-loop of $\texttt{Trim}_p$ (lines 8-10) is $\mathit{sound}(V')$: initially that holds as the universal quantification in $\mathit{sound}(V')$ is empty. The invariant is preserved as $v.\mathit{status}$ is only set to `DEAD` if the status of all $w \in v.\mathit{post}$ is `DEAD` (lines 9 and 10). We use the fact that $\texttt{ZeroOutDegree}(v)$ returns $(\forall w \in v.\mathit{post} : w.\mathit{status} = \texttt{DEAD})$. The postcondition of $\texttt{Trim}_p(V_p)$ is therefore $\mathit{sound}(V_p)$. The postcondition (line 5) is then $\mathit{sound}(V_1) \wedge \ldots \wedge \mathit{sound}(V_{\mathcal{P}})$, which is equivalent to $\mathit{sound}(V)$. Thus $\mathit{sound}(V)$ is the invariant of the repeat-until loop (lines 2 - 6) and therefore holds on termination. □

Theorem 2 (Completeness) *For any $G = (V, E)$ Algorithm 4 terminates with $complete(V)$.*

Proof The invariant of the for-loop of $\texttt{Trim}_p$ (lines 8-10) is $\neg change \implies complete(V')$, where V' is the set of visited vertices. If *change* is `TRUE`, the invariant is obviously preserved as *change* is not set to `FALSE` in this or any other parallel copy of $\texttt{Trim}_p$. Suppose *change* is `FALSE` and $complete(V')$ holds. For $v \in V \setminus V'$ that remains `LIVE`, the procedure `ZeroOutDegree`(v) (line 9), which computes $(\forall w \in v.post : w.status =$ `DEAD`$)$, must return false. Since setting $v.status$ to `DEAD` may invalidate $complete(V' \cup \{v\})$ for this or some other parallel copy of $\texttt{Trim}_p$, variable *change* is set to `TRUE`, which re-establishes the invariant for this and all other parallel copies of $\texttt{Trim}_p$. □

Complexities. The complexity of parallel AC-3-based graph trimming has been discussed in existing work [29,54,32,11], but not with the work-depth model. We adopt the work-depth model to analyze the time complexity.

Theorem 3 *Algorithm 4 requires $\mathcal{O}(\alpha(n+m))$ expected work, $\mathcal{O}(\alpha Deg_{out})$ depth, and thus $\mathcal{O}(\alpha(n+m)/\mathcal{P} + \alpha Deg_{out})$ time complexity.*

Proof For the inner for-loop (lines 8 - 10), checking the out-degree of each vertex $v \in V$ requires $\mathcal{O}(n+m)$ work in the worst case since all edges may need to be traversed in case of some vertices are removed. For the outer repeat-loop (lines 2 - 6), all vertices must be checked again once at least one vertex is removed. The repetition is carried out α times. Therefore, the expected work is $\mathcal{O}(\alpha(n+m))$.

We analyze the working depth. For the procedure $\texttt{Trim}_p$, the inner for-loop (lines 12 and 13) in procedure `ZeroOutDegree` run in sequential with depth $\mathcal{O}(Deg_{out})$. The repetition (lines 2 - 6) is carried out α times. Therefore, the theoretical working depth is $\mathcal{O}(\alpha Deg_{out})$, and thus the theoretical time complexity is $\mathcal{O}(\alpha(n+m)/\mathcal{P} + \alpha Deg_{out})$. □

Theorem 4 *The space complexity of Algorithm 4 is $\mathcal{O}(n)$.*

Proof Each vertex $v \in V$ requires *status* in memory to record if vertex v is `LIVE` or `DEAD`. Besides *status*, no other auxiliary data structures are utilized. Therefore, the space complexity is $\mathcal{O}(n)$. □

5 AC-4-Based Graph Trimming

AC-4 improves the worst-case time complexity of AC-3 by using auxiliary data structures, *supports* and *counters*. Specifically, for each value in its domain, its supports are recorded, and its total number of supports is recorded with a counter. When removing one value, the corresponding counters located by supports are decreased by one. The values whose counters are reduced to zero must be removed, which may cause other values to be removed. In a word, the supports and counters are used for efficient propagation after unqualified values are removed.

The AC-4 algorithm can be applied to graph trimming, which we call AC-4-based trimming. For a directed graph $G = (V, E)$, the supports can be simplified as the transposed graph $G^T = (V, E^T)$, and the counters can be implemented

by out-degree counters for all vertices $v \in V$, denoted as $v.deg_{out}$. AC-4-based graph trimming is used in [30] for quickly removing size-1 SCCs to speed up SCC decomposition; nevertheless, the details of this algorithm are not discussed, and its parallel version is not given. In this section, we provide both sequential and parallel AC-4-based algorithms.

5.1 The Sequential AC-4-Based Algorithm

Algorithm 5 shows the detailed steps of the sequential AC-4-based algorithm. Compared to the AC-3-based algorithm, there are two new data structures: 1) the transposed graph $G^T = (V, E^T)$ is required for accessing the predecessors of a vertex $v \in V$ (line 6); 2) a waiting set Q is required for propagation when processing removed vertices (line 3). The procedure `DoDegree(`v, Q`)` (lines 9 - 11) removes the vertex v and puts it into Q for propagation if v is `LIVE` and its out-degree counter $v.deg_{out}$ is zero. That means all vertices in the waiting set Q are dead.

Now we explain Algorithm 5. Initially, for all vertices, their status and out-degree counters are correctly initialized (line 1). For each vertex $v \in V$, the out-degree counter is checked by calling procedure `DoDegree(`v, Q`)` (line 3). The removed vertices are added into the wait set Q and then propagated to update the out-degree counters of other vertices (lines 4 - 8). That is, for each vertex $w \in Q$, all its predecessors' out-degree counters are off by 1 and then checked by the procedure `DoDegree(`v, Q`)` (lines 6 - 8). During this process, new vertices may be removed and added into the waiting set Q so that the algorithm does not terminate until Q becomes empty (line 4).

Algorithm 5: Sequential AC-4-based Graph Trimming

input : Graph $G = (V, E)$ and its transposed graph $G^T = (V, E^T)$
output: Trimmed graph G

1 **for** $v \in V$ **do** $v.status, v.deg_{out} :=$ LIVE, $|v(G).post|$
2 **for** $v \in V$ **with** $v.status =$ LIVE **do**
3 | $Q := \emptyset$; `DoDegree(`v, Q`)`
4 | **while** $Q \neq \emptyset$ **do**
5 | | remove a vertex w from Q
6 | | **for** $v' \in w(G^T).post$ **do**
7 | | | $v'.deg_{out} := v'.deg_{out} - 1$
8 | | | `DoDegree(`v', Q`)`

9 **procedure** `DoDegree(`v, Q`)`
10 | **if** $v.deg_{out} = 0 \wedge v.status =$ LIVE **then**
11 | | $v.status :=$ DEAD; $Q := Q \cup \{v\}$

Correctness. We show soundness and completeness together.

Theorem 5 (Soundness and Completeness) *For any $G = (V, E)$ Algorithm 5 terminates with $sound(V)$ and $complete(V)$.*

Proof Let V' be the set of vertices visited by the outer for-loop (lines 2 - 8). The invariant of the outer for-loop (lines 2 - 8) is that all vertices are sound, all visited vertices are complete, and that for each vertex $v \in V'$ the counter $v.deg_{out}$ is the number of live vertices of outgoing edges:

$$sound(V) \ \wedge \ complete(V') \ \wedge \ (\forall v \in V : v.deg_{out} = |\{w \in v.post \mid w.status = \texttt{LIVE}\}|)$$

The invariant holds initially as setting all vertices to LIVE makes them sound and V' is initially empty.

The invariant of the while-loop (lines 4 - 8) is that all states are sound, but setting a vertex to DEAD may lead to its predecessors to be incomplete; also, all vertices in Q have been set to DEAD and all $v'.deg_{out}$ are off by one where v' are all vertices with a successor in Q,

$$\begin{aligned} & sound(V) \ \wedge \ complete(V' \setminus Q.pre) \ \wedge \ (\forall v \in Q : v.status = \texttt{DEAD}) \\ & \wedge \ (\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in Q\}|) \end{aligned}$$

where $Q.pre = (\cup\, q \in Q : q.pre)$. Since the while-loop terminates only when $Q = \emptyset$, it follows that the invariant of the outer for-loop is preserved. We now argue that the while-loop preserves this invariant:

- $sound(V)$ is preserved as $v \in V$ is set to DEAD only if $v.deg_{out} = 0$, which implies that there cannot be LIVE vertices in $v.post$.
- $complete(V' \setminus Q.pre))$ is preserved as $v \in V' \setminus Q.pre$ is completed vertices and v is indeed set to DEAD if $v.deg_{out} = 0$, which implies that there cannot be LIVE vertices in $v.post$.
- $\forall v \in Q : v.status = \texttt{DEAD}$ is preserved as v is added to Q only after v is set to DEAD.
- $\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in Q\}|$ is preserved as $v.deg_{out}$ is initialized as the number of available out-going edges and decremented only when removed successors propagated.

At termination of outer for-loop (lines 2 - 8), we get $Q = \emptyset$ and $V' = V$. The postcondition of outer for-loop is $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 6 *The worst-case time complexity of Algorithm 5 is $\mathcal{O}(n + m)$.*

Proof The out-degree counter $v.deg_{out}$ for all vertices can be initially calculated within $\mathcal{O}(n + m)$ time (line 1) as each edge is traversed once. Each vertex $v \in V$ can be removed and then added into the waiting set Q at most once (lines 10 and 11); each reversed edge in $v(G^T).post$ is traversed at most once (lines 6 - 8). In this case, in lines 2 - 8, we get a running time of $\mathcal{O}(n + m)$. Therefore, the total worst-case running time is $\mathcal{O}(n + m)$. □

Theorem 7 *The space complexity of Algorithm 5 is $\mathcal{O}(n + m)$.*

Proof In line 7, the transposed graph $G^T = (V, E^T)$ is used. In this case, in order to generate G^T, the whole graph $G = (V, E)$ must be stored in memory, which requires $\mathcal{O}(n + m)$ space. For all vertices $v \in V$, storing $v.deg_{out}$ and $v.status$ uses $\mathcal{O}(n)$ space. Therefore, the total used space is $\mathcal{O}(n + m)$. □

5.2 The Parallel AC-4-Based Algorithm

Algorithm 6 shows the detailed steps of the parallel AC-4-based algorithm. All vertices in V are partitioned into $V_1 \dots V_{\mathcal{P}}$ (line 2) so that $\mathcal{P}$ workers can execute the procedure $\texttt{Trim}_p(V_p)$ in parallel (line 3). Compared with Algorithm 5, there are three refinements. First, each worker $p \in [1...\mathcal{P}]$ has its private waiting set Q_p for propagation (line 6) so that the operations on Q_p do not require to be synchronized. Secondly, the out-degree counter deg_{out} has to be updated by the atomic primitive fetch&add FAA since multiple workers may decrease such a counter (line 11). Thirdly, it is possible that $v.deg_{out} = 0$ (in line 13) is detected by multiple workers; we use the atomic primitive CAS to set the $v.status$ from LIVE to DEAD (line 13) and return TRUE if successful, which ensures that v is added into a single one waiting set Q_p (line 14).

Algorithm 6: Parallel AC-4-based Graph Trimming

```
   input  : Graph G = (V, E) and its transposed graph G^T = (V, E^T)
   output: Trimmed graph G
 1 for v ∈ V do v.status, v.deg_out := LIVE, |v.post|
 2 partition V into V_1, ..., V_P
 3 Trim_1(V_1) || ... || Trim_P(V_P)

 4 procedure Trim_p(V_p)
 5   for v ∈ V_p with v.status = LIVE do
 6     Q_p := ∅; DoDegree_p(v)
 7     while Q_p ≠ ∅ do
 8       remove a vertex w from Q_p
 9       for v' ∈ w(G^T).post do
10         FAA(v'.deg_out, −1)
11         DoDegree_p(v', Q_p)

12 procedure DoDegree_p(v, Q_p)
13   if v.deg_out = 0 ∧ CAS(v.status, LIVE, DEAD) then
14     Q_p := Q_p ∪ {v}
```

Correctness. We show the soundness and completeness together.

Theorem 8 (Soundness and Completeness) *For any $G = (V, E)$ Algorithm 6 terminates with sound(V) and complete(V).*

Proof The invariant of the while-loop (lines 4 - 8) in procedure $\texttt{Trim}_p(V_p)$ is the same as that in Algorithm 5 except that it adds one more conjunct. That is, a removed vertex can only be added into a single one Q_p for propagation.

$$\begin{aligned}&sound(V_p) \wedge complete(V_p' \setminus Q_p.pre) \wedge (\forall v \in Q_p : v.status = \texttt{DEAD})\\ &\wedge (\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in \cup Q_{1..\mathcal{P}}\}|)\\ &\wedge (\forall i, j \in \{1..\mathcal{P}\} : i \neq j \implies Q_i \cap Q_j = \emptyset)\end{aligned}$$

We now argue that the while-loop preserves this invariant:

- $(\forall v \in V : v.deg_{out} = |\{u \in v.post \mid u.status = \texttt{LIVE} \vee u \in \cup Q_{1..\mathcal{P}}\}|)$ is preserved as $v.deg_{out}$ is off by one atomically when a worker is decreasing.
- $(\forall i, j \in \{1..\mathcal{P}\} : i \neq j \wedge Q_i \cap Q_j = \emptyset)$ is preserved as $v.status$ is set from LIVE to DEAD by the atomic primitive CAS and only when successful, v is added to one Q_p.

The postcondition of line 3 is then $sound(V_1) \wedge complete(V_1) \ldots sound(V_{\mathcal{P}}) \wedge complete(V_{\mathcal{P}})$, which is equivalent to $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 9 *Algorithm 6 requires $\mathcal{O}(n+m)$ expected work, $\mathcal{O}(|Q_p| Deg_{in} Deg_{out})$ depth, and thus $\mathcal{O}((n+m)/\mathcal{P} + |Q_p| Deg_{in} Deg_{out})$ time complexity.*

Proof This algorithm has the same framework as Algorithm 5, so the total expected work equals the running time of Algorithm 5, that is $\mathcal{O}(n+m)$. The initial for-loop (lines 1) can easily run in parallel within expected depth $\mathcal{O}(Deg_{out})$.

We analyze the working depth for the procedure $\texttt{Trim}_p$. For each round of the outer while-loop (lines 7 - 11), it runs with depth $|Q_p|$ which is the upper-bound size of waiting sets among $\mathcal{P}$ workers. As Q_p is private for worker p without synchronization, it is possible that $|Q_p| \geq \alpha$. The most inner for-loop (line 9) runs sequentially with depth $\mathcal{O}(Deg_{in})$, and the out-degree counters have to concurrently update with depth Deg_{out}. Therefore, the total working depth is $\mathcal{O}(\alpha Deg_{in} Deg_{out})$ and thus the worst-case time complexity is $\mathcal{O}((n+m)/\mathcal{P} + \alpha Deg_{in} Deg_{out})$. □

Theorem 10 *The space complexity of Algorithm 6 is $\mathcal{O}(n+m)$.*

Proof By using the atomic primitive CAS in line 12, each vertex $v \in V$ may be removed at most once and then put into at most single one waiting set Q_p, so all waiting set for $\mathcal{P}$ workers require $\mathcal{O}(n)$ space. Similar to Algorithm 5, storing deg_{out} and $status$ requires $\mathcal{O}(n)$ space and the reverse edges require $\mathcal{O}(n+m)$ space (line 8). Therefore, the total used space is $\mathcal{O}(n+m)$. □

6 AC-6-Based Graph Trimming

As mentioned, AC-4 has better worst-case time complexity than AC-3, but AC-4 always has a worse average running time than AC-3. Additionally, AC-4 does not have the on-the-fly property. AC-6 improves AC-4 by only recording one support for each value since one support is enough to guarantee that a value is viable. In this case, compared with AC-4, AC-6 performs better in many applications, requires less space usage, and has the on-the-fly property.

To the best of our knowledge, we are the first to introduce AC-6 to graph trimming and call it the AC-6-based algorithm. The idea is novel: 1) each vertex v maintains a set of vertices $v.S$ that choose v as an available outgoing edge; 2) when removing v as it has no outgoing edges, each vertex $w \in v.S$ has to find another available outgoing edge to replace v; otherwise, w has to be removed; 3) this process repeats until no vertices can be removed. In this section, we propose new sequential and parallel AC-6-based algorithms for graph trimming, which is the main contribution of this work.

6.1 The Sequential AC-6-Based Algorithm

Analogous to AC-6, the AC-6-based trimming algorithm is based on the concept of *support*. That is, for each vertex v in a given directed graph G, the support of v is one of v's available outgoing edges and v cannot be removed if v's support exists. One auxiliary data structure, the supporting set, is needed to store all the supports for propagation, which is formally defined below.

Definition 3 (Supporting Set) Given a directed graph $G = (V, E)$, for a vertex $v \in V$, the supporting set $v.S$ of v is the set of predecessors that choose v as their single one support: $(\forall v \in V : v.S \subseteq \{u \in v.pre \mid u.status = \texttt{LIVE}\}) \ \wedge \ (\forall v \in V : v.S \neq \emptyset \implies v.status = \texttt{LIVE}) \ \wedge \ (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset)$.

In other words, $v.S$ records all `LIVE` vertices that have v as their support. Absolutely, v must be `LIVE` if the vertices in $v.S$ choose v as a support as an existing support has to be an available outgoing edge; a vertex can be added into at most one supporting set as each vertex only needs to maintain single one support.

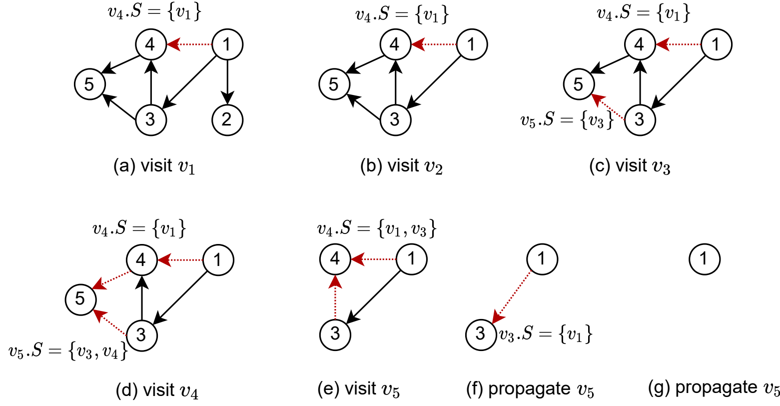

Fig. 2: Steps of the sequential AC-6-based trimming algorithm based on part of the graph in Figure 1.

Figure 2 illustrates the AC-6-based algorithm based on the part of the example graph in Figure 1. The dashed red arrows are the edges visited to find available supports and then added to the corresponding supporting set $v.S$. Each vertex is successively visited from v_1 to v_5 as shown in Figure 2 (a) to (e). In Figure 2 (a), v_1 is visited and its first support v_4 is found with vertex v_1 added into $v_4.S$. In Figure 2 (b), v_2 is removed since v_2 has no outgoing edges; no propagation happens as $v_2.S$ is empty. In Figure 2 (c), v_3 is visited and its first support v_5 is found with v_3 added into $v_5.S$. In Figure 2 (d), v_4 is visited and its first support v_5 is found with v_3 added into $v_4.S$. In Figure 2 (e), v_5 is removed as it cannot find any support; since $v_5.S$ includes v_3 and v_4 the propagation happens as follow: v_3 finds a next available support v_4 with v_3 added into $v_4.S$, and the v_4 is failed

to find a next available support so that v_4 should be removed in the next step. In Figure 2 (f), v_4 is removed; since the supporting set $v_4.S$ includes vertices v_1 and v_3 the propagation happens as follow: v_1 finds a next available support, v_3, which is added into $v_4.S$, and v_3 fails to find a next available support so that v_3 should be removed in the next step. In Figure 2 (g), the vertex v_3 is removed and $v_1 \in v_3.S$ should be further propagated. Finally, v_1 should also be removed as it has no outgoing edges. As we can see, AC-6-based trimming can remove some of the vertices without propagation, e.g. v_2.

Algorithm 7: Sequential AC-6-based Graph Trimming

```
   input  : Graph G = (V, E)
   output: Trimmed graph G
 1 for v ∈ V do v.status, v.S := LIVE, ∅
 2 for v ∈ V do
 3 |  Q := ∅; DoPost(v)
 4 |  while Q ≠ ∅ do
 5 |  |  remove a vertex w from Q
 6 |  |  for v' ∈ w.S do
 7 |  |  |  w.S := w.S \ {v'}
 8 |  |  |  DoPost(v')

 9 procedure DoPost(v)
10 |  for w ∈ v.post with w.status = LIVE do
11 |  |  w.S := w.S ∪ {v}; v.post := v.post \ {w}; return
12 |  v.status := DEAD; Q := Q ∪ {v}
```

Algorithm 7 shows the detailed steps of the sequential AC-6-based algorithm. For each vertex v in the graph, a supporting set $v.S$ is required for recording the vertices that choose v as an available support. We first consider the procedure `DoPost`(v) (lines 9 - 12). If v successfully finds a live successor w, then w is added to $v.S$ and the procedure finishes (lines 10 and 11). Otherwise, v has to be removed from the graph as v has no available outgoing edges, and v is set to `DEAD` and then put into the waiting set Q (line 12). Note that, the visited vertex w is removed from $v.post$ to avoid redundant checking (line 11), which can ensure that each edge is visited at most once. Now we explain the main algorithm (lines 1 - 8). Initially, all vertices are `LIVE` and their supporting sets are empty (line 1). For each vertex $v \in V$, the support $v.s$ is checked by the procedure `DoPost`(v) (line 3) and the removed vertices are added into Q for propagation (lines 4 - 8). That is, a vertex $w \in Q$ is removed from Q (line 5) and for all the vertices in $w.S$ are checked by the procedure `DoPost`(v') (lines 6 - 8). This propagation is repeated until Q is empty (line 4), as vertices may be removed and added into Q by the procedure `DoPost`(v') (line 8).

Correctness. We show the soundness and completeness together.

Theorem 11 (Soundness and Completeness) *For any $G = (V, E)$ Algorithm 7 terminates with $sound(V)$ and $complete(V)$.*

Proof Let V' be the set of vertices visited by the outer for-loop (lines 2 - 8). The invariant of the outer for-loop is that all vertices are sound, all visited vertices are complete, all visited LIVE vertices must have a support, and that for each vertex $v \in V$ the supporting set $v.S$ includes all visited vertices that choose v as their single one support:

$$\begin{aligned} &sound(V) \ \wedge \ complete(V') \ \wedge \ (\forall v \in V' : v.status = \texttt{LIVE} \implies v \in S) \\ &\quad \wedge \ (\forall v \in V : v.S \subseteq \{u \in v.pre \mid u.status = \texttt{LIVE} \wedge u \in V'\}) \\ &\quad \wedge \ (\forall v \in V : v.S \neq \emptyset \implies v.status = \texttt{LIVE}) \\ &\quad \wedge \ (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset) \end{aligned}$$

where $S = (\cup\, v \in V : v.S)$. The invariant holds initially as setting all vertices to LIVE and V' is empty.

The invariant of the while-loop (lines 4 - 8) is that all states are sound but setting a vertex to DEAD may lead to its predecessors to be incomplete; also, all vertices $w \in Q$ are set to DEAD and all vertices in $w.S$ have to update their support.

$$\begin{aligned} &sound(V) \ \wedge \ complete(V' \setminus Q.S) \ \wedge \ (\forall v \in Q : v.status = \texttt{DEAD}) \\ &\quad \wedge \ (\forall v \in V' : v.status = \texttt{LIVE} \implies v \in S) \\ &\quad \wedge \ (\forall v \in V : v.S \subseteq \{u \in v.pre \mid u.status = \texttt{LIVE} \wedge u \in V'\}) \\ &\quad \wedge \ (\forall v \in V : v.S \neq \emptyset \implies v.status = \texttt{LIVE} \vee v \in Q) \\ &\quad \wedge \ (\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset) \end{aligned}$$

where $S = (\cup\, v \in V : v.S)$ and $Q.S = (\cup\, q \in Q : q.S)$. Since the while-loop terminates only when $Q = \emptyset$, it follows that the invariant of the outer for-loop is preserved.

We now argue that the while-loop preserves this invariant:

- $sound(v)$ is preserved as $v \in V$ is set to DEAD if w cannot find a support in $v.post$, which implies that there cannot be LIVE vertices in $v.post$.
- $complete(V' \setminus Q.S)$ is preserved as $v \in V' \setminus Q.S$ is indeed set to DEAD if the support of v not exists, which implies that there cannot be LIVE vertices in $v'.post$.
- $\forall v \in Q : v.status = \texttt{DEAD}$ is preserved as v is added to Q only after v is set to DEAD.
- $\forall v \in V' : v.status = \texttt{LIVE} \implies v \in S$ is preserved as v has to find a support after being visited if v is LIVE.
- $\forall v \in V : v.S \subseteq \{u \in v.pre \mid u.status = \texttt{LIVE} \wedge u \in V'\}$ is preserved as the visited vertices $u \in V'$ is LIVE when choosing v as a support.
- $\forall v \in V : v.S \neq \emptyset \implies v.status = \texttt{LIVE} \vee \wedge v \in Q$ is preserved as new vertices can be added into $v.S$ if v is LIVE, and after setting v to DEAD and adding into Q all vertices in $v.S$ will find next available support.
- $\forall u, v \in V : u \neq v \implies u.S \cap v.S = \emptyset$ is preserved as each vertex maintains at most single one support.

At termination of the outer for-loop (lines 2 - 8), we get $Q = \emptyset$ and $V' = V$. The postcondition of the outer for-loop is $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 12 *The time complexity of the Algorithm 7 is $\mathcal{O}(n+m)$.*

Proof Each vertex $v \in V$ can be removed and then added into the waiting set Q at most once. In the procedure `DoPost(`v`)`, each outgoing edge of v is traversed at most once to find the support (lines 10 and 11) as the visited vertex w is removed from $v.post$ to avoid repetitive visiting. In this case, The most inner for-loop (lines 6 - 8) calls procedure `DoPost(`v`)` to find a support for vertex v. Therefore, with this assumption, the worst-case time complexity is $\mathcal{O}(n+m)$. □

Theorem 13 *The space complexity of the Algorithm 7 is $\mathcal{O}(n)$.*

Proof The global waiting set Q has a maximum size of $\mathcal{O}(n)$ as each vertex $v \in V$ can be set to `DEAD` and added into Q at most once. The supporting sets have the total size at most $\mathcal{O}(n)$ as each vertex $v \in V$ has at most one support recorded in a corresponding supporting set. Obviously, *status* requires $\mathcal{O}(n)$ space. Therefore, the worst-case space complexity is $\mathcal{O}(n)$. □

6.2 The Parallel AC-6-Based Algorithm

Algorithm 8 shows the detailed steps of the parallel AC-6-based trimming algorithm. Compared with the sequential AC-6-based trimming in Algorithm 7, there are two refinements. First, each worker $p \in [1 \ldots \mathcal{P}]$ has its private waiting set Q_p for propagation so that the synchronization on Q_p is unnecessary (lines 6, 8, and 19). Secondly, the supporting set $w.S$ for each vertex $w \in V$ can concurrently have new vertices added by multiple workers synchronized by locking (lines 14 - 17). When adding vertices to $w.S$, vertex w has to be `LIVE` (line 15) as no vertices can be added to $w.S$ after setting w to `DEAD`. When setting vertex v to `DEAD`, we have to lock v to ensure that no other workers are adding vertices to $v.S$; otherwise, after v is added into Q and propagated (lines 19 and 8 - 11), other workers still have possibility to add vertices into $v.S$ which can never be propagated. In other words, we lock $v.S$ when setting v from `LIVE` to `DEAD` to ensure that all vertices in $v.S$ are propagated together. Note that, when removing vertices from $w.S$ (line 10), it is unnecessary to lock $w.S$ as currently w is `DEAD` so that no workers can add vertices into $w.S$ and w is only accessed by a single worker, p.

We implement the lock by the `CAS` primitive with busy waiting (lines 20 and 21). Here, the busy waiting is suitable as there are at most two operations within locking (lines 15 and 16) so that the expected waiting time is really short.

Correctness.

Theorem 14 (Soundness and Completeness of Parallel AC-6-based Trimming) *For any $G = (V, E)$ Algorithm 8 terminates with sound(V) and complete(V).*

Proof The invariant of the while-loop (lines 4 - 8) in procedure $\texttt{Trim}_p(V_p)$ is the same as that in Algorithm 5 except that it adds one more conjunct. That is, a

Algorithm 8: Parallel AC-6-based Graph Trimming

```
   input  : G = (V, E)
   output: Trimmed graph G
 1 for v ∈ V do  v.S, v.status := ∅, LIVE; Unlock(v)
 2 partition V into V_1, ..., V_𝒫
 3 Trim_1(V_1) ‖ ... ‖ Trim_𝒫(V_𝒫)

 4 procedure Trim_p(V_p)
 5 |  for v ∈ V_p do
 6 |  |  Q_p = ∅; DoPost_p(v, Q_p)
 7 |  |  while Q_p ≠ ∅ do
 8 |  |  |  remove a vertex w from Q_p
 9 |  |  |  for v' ∈ w.S do
10 |  |  |  |  w.S := w.S \ {v'}
11 |  |  |  |  DoPost_p(v', Q_p)

12 procedure DoPost_p(v, Q_p)
13 |  for w ∈ v.post with w.status = LIVE do
14 |  |  Lock(w)
15 |  |  if w.status = LIVE then
16 |  |  |  w.S := w.S ∪ {v}; Unlock(w); return
17 |  |  Unlock(w)
18 |  Lock(v); v.status := DEAD; Unlock(v)
19 |  Q_p := Q_p ∪ {v}

20 procedure Lock(w)
21 |  while ¬ CAS(w.lock, FALSE, TRUE) do  skip

22 procedure Unlock(w)
23 |  w.lock := FALSE
```

removed vertex can only be added into a single one Q_p for propagation.

$$
\begin{aligned}
& \mathit{sound}(V_p) \;\wedge\; \mathit{complete}(V'_p \setminus Q_p.S) \;\wedge\; (\forall v \in Q_p : v.\mathit{status} = \texttt{DEAD}) \\
& \wedge\;\; (\forall v \in V'_p : v.\mathit{status} = \texttt{LIVE} \implies v \in S) \\
& \wedge\;\; (\forall v \in V_p : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}) \\
& \wedge\;\; (\forall v \in V_p : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee v \in Q_p) \\
& \wedge\;\; (\forall u, v \in V_p : u \neq v \implies u.S \cap v.S = \emptyset) \\
& \wedge\;\; (\forall i, j \in \{1..\mathcal{P}\} : i \neq j \implies Q_i \cap Q_j = \emptyset)
\end{aligned}
$$

where $S = (\cup\, v \in V : v.S)$, $Q_p.S = (\cup\, q \in Q_p : q.S)$, and $V' = \cup V'_{1..\mathcal{P}}$. In the algorithm, for each vertex v, multiple workers add new vertices into $v.S$ concurrently, and during this time v cannot be set to DEAD. We now argue that the while-loop preserves this invariant:

- $\forall v \in V_p : v.S \subseteq \{u \in v.\mathit{pre} \mid u.\mathit{status} = \texttt{LIVE} \wedge u \in V'\}$ is preserved as v is locked when a worker is adding new vertices to $v.S$.
- $\forall v \in V_p : v.S \neq \emptyset \implies v.\mathit{status} = \texttt{LIVE} \vee (v.\mathit{status} = \texttt{DEAD} \wedge v \in Q_p)$ is preserved as 1) v is locked when the worker p setting v to DEAD to ensure that

after setting v to DEAD no vertices can be added into $v.S$ by other workers, and 2) only the current worker p can set v to DEAD and add v to the private set Q_p.
- $\forall i, j \in \{1..\mathcal{P}\} : i \neq j \;\wedge\; Q_i \cap Q_j = \emptyset$ is preserved as only single one worker p can add v to Q_p after setting v to DEAD.

At the termination of outer for-loop (lines 2 - 8), we get $Q_p = \emptyset$ and $V'_p = V_p$. The postcondition of line 3 is then $sound(V_1) \wedge complete(V_1) \wedge \ldots \wedge sound(V_{\mathcal{P}}) \wedge complete(V_{\mathcal{P}})$, which is equivalent to $sound(V) \wedge complete(V)$. □

Complexities.

Theorem 15 *The Algorithm 8 requires $\mathcal{O}(n+m)$ expected work, $\mathcal{O}(|Q_p|{Deg_{in}}^2)$ depth, and $\mathcal{O}((n+m)/\mathcal{P} + |Q_p|{Deg_{in}}^2)$ time complexity.*

Proof This algorithm has the same framework as Algorithm 7, so the total expected work equals the running time of Algorithm 7, that is $\mathcal{O}(n+m)$. The initial for-loop (lines 1) can run in parallel within expected depth $\mathcal{O}(1)$.

We analyze the working depth for procedure $\texttt{Trim}_p$. For each round of the outer while-loop (lines 7 - 11), it runs with depth $|Q_p|$ which is the upper-bound size of waiting sets among $\mathcal{P}$ workers. As Q_p is private for a worker p without synchronization, it is possible that $|Q_p| \geq \alpha$. The most-inner for-loop (line 9) runs sequentially with depth Deg_{in} as Deg_{in} is the upper-bound size for a supporting set. The supporting sets concurrently add new vertices with depth Deg_{in} (line 16). The locking operation (lines 14 and 18) needs to busy-check by the CAS primitive only a few times with high probability as there are at most two operations within the lock. Therefore, the total working depth is $\mathcal{O}(|Q|{Deg_{in}}^2)$ with high probability and the worst-case running time is $\mathcal{O}((n+m)/\mathcal{P}+|Q|{Deg_{in}}^2)$ with high probability. □

Theorem 16 *The space complexity of Algorithm 8 is $\mathcal{O}(n)$, which equals to its sequential version Algorithm 7.*

Proof Each vertex $v \in V$ may be removed at most once and then put into at most one waiting set Q_p, so all waiting sets require $\mathcal{O}(n)$ space. Each vertex has at most one support which is stored into the corresponding supporting set and thus the total size of all supporting sets is $\mathcal{O}(n)$. The status for each vertex $v \in V$ has the size of $\mathcal{O}(n)$. Therefore, the total space complexity is $\mathcal{O}(n)$. □

7 Related Work

7.1 Parallel DFS-based SCC Decomposition

In Section 1, several methods for BFS-based SCC decomposition were introduced. Although DFS is inherently sequential [49], there is a lot of work based on Tarjan's algorithm. In [39], Lowe proposed a synchronized Tarjan's algorithm, that is, multiple instances of Tarjan's algorithm run without overlapping stacks. To do this, a worker is suspended on a vertex which is located in another worker's stack and then both workers' stacks can be merged if necessary. The drawback is that this stack merging leads to a worst-case quadratic time complexity of $\mathcal{O}(n^2)$. Lowe's experiments show decent speedups on model checking graphs with trivial SCCs, but

not for graphs with large SCCs. In [50], Renault et al. present a novel algorithm without sacrificing the linear time complexity, $\mathcal{O}(n+m)$, and the on-the-fly property. Multiple instances of Tarjan's algorithm run and communicate completely explored SCCs via a shared union-find structure. Bolomen et al. [8, 7] proposed an improved UFSCC algorithm which communicates partially found SCCs by using a modified union-find data structure. In their experiments, UFSCC shows a significant speedup compared to Renault's algorithm [50] on implicit model checking inputs and synthetic graphs. One notable property of these algorithms is that they can run on-the-fly on implicit graphs.

However, above DFS-based SCC algorithms do not utilize graph trimming techniques to remove trivial SCCs. A possible reason is that the traditional graph trimming technique has quadratic worst-case time complexity and, more importantly, it is hard to run on-the-fly. The proposed parallel AC-6-based graph trimming algorithm has linear running time and has the on-the-fly property, so it can be used to quickly trimming a high ratio of size-1 SCCs for above DFS-based SCC algorithms.

7.2 Graph Trimming

Generally, the term "graph trimming" is widely used in graph algorithms to minimizing the search space and the trimming rules may be different for different problems. For example, in [26], "graph trimming" computes a smaller and smaller unsatisfiable core for a propositional formula; in [24], "graph trimming" is used to minimize the number of vertices as monitors to identify all interesting links; in [20], given a graph in which each vertex has a nonnegative weight, "graph trimming" deletes vertices with a small total weight such that the remaining graph does not contain any long simple paths. Note that, in the current work, given a directed graph, the terminology "graph trimming" is specifically confined as each vertex has at least one outgoing edge.

Another related term is "graph pruning". For example, in [25], given a geographical graph, "graph pruning" can dynamically jump over some searching branches by some simple rules for finding the path between two nodes.

8 Implementation

All graph trimming algorithms are implemented in C++ with OpenMP as treading library. In this section, we explore the details of implementation, especially the parallelism.

Graph Storage. All tested graphs are stored in the *compressed sparse row* CSR format [28] for efficient traversals. With this format, all edges are linear in the memory. In other words, for each vertex the edges can be traversed in sequential order. In the current work, our experiments focus on graphs with edges linearly stored in memory.

Parallelism. OpenMP (Open Multi-Processing) [16] is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems. In this paper, OpenMP (version 4.5) is used as the threading library to implement the parallel algorithms. The task-level parallelism is implemented by using the clause "`#pragma omp parallel for`" (C++ code). Given a input graph, this implementation statically assigns the same number of vertices to each worker p. For data-level parallelism, however, it is critical to handle a potential workload imbalance problem. Note that real-world graphs can be highly irregular because of their scale-free property, e.g., a few vertices can have a huge number of successors while many vertices have only several successors. Therefore, statically assigning the same number of vertices to each worker naturally induces workload imbalance since the work of each vertex involves immediate propagation. There is a better strategy. All of the vertices in the graph can be dynamically assigned to each worker p by the clause "`#pragma omp for schedule(dynamic, s)`". That means each worker executes a chunk of iterations with size s and then requests another chunk until no chunks remain to distribute. If one of the workers finishes processing a chunk of vertices early, it applies to the next chunk of vertices at once without waiting for other workers. In this way, we realize a relatively balanced load for each worker without difficulties. Note that the chunk size cannot be either much large or small; the too large chunk size may cause work-load imbalance for multiple workers; the too small chunk size may cause much running time spent on scheduling.

For simplicity, our parallel trimming algorithms sacrifice some parallelism. That is, the most inner for-loop can run in parallel (lines 12 - 13 in Algorithm 1, lines 9 - 11 in Algorithm 6, and lines 9 -11 in Algorithm 8); the private waiting set Q_p for a worker p can be balanced in Algorithm 6 and Algorithm 8) so that Q_p has at most α vertices. As shown in Table 4, the working depth can be improved if we achieve full parallelism. However, the scheduler will be challenged to parallel inside each worker p efficiently. One solution is to maintain a *frontier* (subset) of all affected vertices, and in each step all vertices in a frontier can be processed in parallel [17].

Trimming	Worst-Case ($\mathcal{O}$) Work	Depth	Space
AC-3-based	$\alpha(n+m)$	α	n
AC-4-based	$n+m$	αDeg_{out}	$n+m$
AC-6-based	$n+m$	αDeg_{in}	n

Table 4: The worst-case work, depth, time, and space complexities of full parallelized graph trimming algorithms.

In practice, our algorithms can be highly parallelized. There are two reasons. First, most real graphs always have millions of edges, and $|Q|$, Deg_{in}, and Deg_{out} are relatively much smaller than $n+m$. Secondly, multi-core machines always have a limited number of workers, e.g. $\mathcal{P} = 32$. Therefore, our trimming algorithms can achieve a load balance among multiple workers with high probability.

Traverse Edges. Since the edges are linearly stored in memory, we can optimize the implementation of trimming algorithms. In the procedure `ZeroOutDegree` of Algorithm 4, for vertex v only the first `LIVE` edge needs to be found. In this case, each vertex can maintain an index *edge_index* to record the position of visited edges. In the next round, we can "jump" over the edges that have already visited. By doing this, we can reduce the number of traversed edges to a certain degree. Similarly, we can apply this strategy to the procedure `DoPost` of Algorithm 7 and Algorithm 8; by doing this, each edge can be traversed at most once.

Cache-Friendliness. For multi-core architectures, contiguous memory accessing is much faster than random memory accessing because of the possibility of pre-fetching by L1, L2, and L3 caches. Of course, accessing cache is faster than accessing the memory by an order of magnitude. For explicit graphs stored in memory, the cache can affect the running time by an order of magnitude. A *cache-friendly* program has a large portion of contiguous memory accessing that can fully utilize the cache to obtain speedup. In contrast, a *cache-unfriendly* program has a large portion of random memory accessing that can not efficiently utilize the cache.

Since all edges are stored in contiguous memory as CSR format for a tested graph, we compare the cache property of three different graph trimming methods together as follow:

- AC-3-based Graph Trimming is cache-friendly as all edges are stored in an array and can be traversed sequentially with a high cache hit rate.
- AC-4-based Graph Trimming is less cache-friendly as each vertex v are traversed almost randomly, but v's edges are traversed sequentially with a medium cache hit rate.
- AC-6-based Graph Trimming is least cache-friendly as for each vertex v, both v and v's edges are traversed almost randomly with low cache hit rate.

Memory Usage. We compare the practical memory usage in Table 5. Assume that storing a vertex or an integer takes H bits. All three algorithms require 1 bit for the status of each vertex, in total n bits. For both *AC4Trim* and *AC6Trim*, there are $\mathcal{P}$ waiting sets $Q_1 \dots Q_{\mathcal{P}}$, in total nH bits, since each vertex can be put into Q_p at most once. For *AC4Trim*, a reversed graph has to be loaded into memory, in total $(n+m)H$ bits; each vertex maintains an out-degree counter deg_{out}, in total nH bits. For *AC6Trim*, each vertex has a supporting set S, in total nH bits, since each vertex can be put into a set S at most once. For *AC3Trim* and *AC6Trim*, each vertex maintains an index *edge_index* to "jump" over the visited edges, in total nH bits.

9 Experiments

In this section, we evaluate three different parallel algorithms for graph trimming:

- the AC-3-based trimming algorithm [29,32] (*AC3Trim* for short),
- the AC-4-based trimming algorithm [30] (*AC4Trim* for short),
- the AC-6-based trimming algorithm (*AC6Trim* for short).

AC3-based	*AC4-based*	*AC6-based*
bit$[n]$: $\forall v.status$	bit$[n]$: $\forall v.status$	bit$[n]$: $\forall v.status$
bit$[nH]$:$\forall v.edge_index$	bit$[nH]$: $\forall v.deg_{out}$	bit$[n]$: $\forall v.lock$
	bit$[nH]$: $Q_1 \dots Q_{\mathcal{P}}$	bit$[nH]$:$\forall v.edge_index$
	bit$[(n+m)H]$: G^T	bit$[nH]$: $\forall v.S$
		bit$[nH]$: $Q_1 \dots Q_{\mathcal{P}}$

Table 5: Compare the memory usage for *AC3Trim*, *AC4Trim* and *AC6Trim*, where storing a vertex takes H bits.

The experiments are performed on a server with an AMD CPU (16 cores, 32 hyper-threads, 32 MB of last-level cache) and 96 GB main memory. The server runs the Ubuntu Linux (18.04) operating system. All tested algorithms are implemented in C++ and compiled with g++ version 7.3.0 with the -O3 option [1]. OpenMP [16] version 4.5 is used as the threading library. We perform every experiment at least 50 times (at least 10 times for time-consuming experiments) and calculate their means with 95% confidence intervals.

We first give in total 15 real and synthetic benchmark graphs. Before the evaluation, we discuss the workload balance. Then, over these tested graphs, we evaluate the number of traversed edges and then compare the real running times by varying the workers from 1 to 32. We also evaluate the stability and the scalability by using 16 workers.

9.1 Graph Benchmarks

We evaluate the performance of our method on a variety of model checking, real-world, and synthetic graphs shown in Table 6.

- The *cambridge.6*, *bakery.6* and *leader-filters.7* graphs come from the model checking problems in the BEEM database [48], which are implicit and can be generated on-the-fly. For convenience, these graphs are converted to explicit graphs [8] and stored in files.
- The *livej*, *patent*, and *wikitalk* graphs are obtained from SNAP [38] [2]; they represent the Live-Journal social network [3], the U.S patent dataset is maintained by the National Bureau of Economic Research [37], and Wikipedia Talk (communication) network [36], respectively. The *dbpedia*, *baidu*, and *wiki-talk-en* graphs are collected from the University of Koblenz-Landau [35]; they represent the DBpedia network [2], the hyperlink network between the articles of the Baidu [47] encyclopedia, and the communication network of the English Wikipedia [55], respectively.
- The *com-friendster*, *twitter* and *twitter-mpi* are three super large graphs with billions of edges obtained from the Network Repository [51] [3]; they represent

[1] All our implementations, benchmarks, and results are available at `https://github.com/Itisben/graph-trimming.git`

[2] `https://snap.stanford.edu`

[3] `http://networkrepository.com`

the online gaming social network [46], the follower network from Twitter [9], the twitter follow data collected in 2009 [10], respectively.

– The *ER*, *BA*, and *RMAT* graphs are synthetic graphs; they are generated by the SNAP [38] system using the Erdös-Rényi graph model (which generates a random graph), the Barabasi-Albert graph model (which generates a graph with power-law degree distribution), and the R-MAT graph model (which generates large-scale realistic graph similar to social networks), respectively; for these generated graphs, the average degree is fixed to 8 by choosing $1,000,000$ vertices and $8,000,000$ edges.

All these graphs are stored in the Compressed Sparse Row (CSR) binary format [28, 29], which is compact and memory bandwidth-friendly. Taking the super large graph *twitter* for example, the text file that includes all edges requires 30 GB while the CSR binary format only requires 6 GB.

Name	$\|V\|$	$\|E\|$	Deg_{in}	Deg_{out}	α	%Trim
cambridge.6	3.3M	9.5M	15	6	65	0.25%
bakery.6	11.8M	40.4M	24	4	47	22.26%
leader-filters.7	26.3M	91.7M	12	6	73	100.00%
dbpedia	4.0M	13.8M	473.0K	1.0K	116	36.23%
baidu	2.1M	17.8M	98.0K	2.6	9	27.97%
livej	4.8M	69.0M	14.0K	20.3K	8	12.23%
patent	6.0M	16.5M	779	770	5	100.00%
wiki-talk-en	3.0M	25.0M	121.3K	488.2K	7	87.42%
wikitalk	2.4M	5.0M	3.3K	100.0K	5	94.49%
com-friendster	125M	1.8B	4.2K	3.6K	11.7K	100.00%
twitter	41.4B	1.4B	770.2K	3.9M	6	10.05%
twitter-mpi	52.6B	2.0B	3.5M	780.0K	7	17.52%
ER	1.0M	8.0M	25	24	3	0.03%
BA	1.0M	8.0M	8	5.2K	122	100%
RMAT	1.0M	8.0M	335	1.9K	7.0K	99.98%

Table 6: The characteristics for model checking, real-world, and synthetic graphs. Here, columns denote the number of vertices n, the number of edges m, the maximum in-degree, the maximum out-degree, the number of peeling steps, and the percentage of trimmable vertices, respectively.

Table 6 provides an overview of the 15 tested graphs. For some graphs, e.g. *cambridege.6* and *ER*, less than 1% of vertices can be trimmed. However, for most of the other graphs, a high ratio of vertices can be trimmed, especially for *leader-filters.7*, *BA*, and *com-friendster*, whose vertices can be trimmed altogether. More importantly, for most graphs, the trimming steps α, maximum in-degree Deg_{in}, and maximum out-degree Deg_{out} are always small. When analyzing the parallel time complexity, these three values are associated with the parallel depths. The small values of the depths indicate that the execution can be highly parallelized [6].

9.2 Workload Balance

Given a tested graph, all vertices are partitioned into multiple chunks, which can be dynamically assigned to workers for workload balance. One issue is how to determine the size of chunks. A large size of chunks may lead to workload imbalance, while a small size of chunks may lead to a high cost of scheduling. Since the trend is similar for all tested graphs, we select three typical graphs with a variety of Deg_{in}, Deg_{out}, and α for the evaluation. In Figure 3, we test three trimming algorithms over three selected graphs, *leader-filters.7*, *livej*, and *wiki-talk-en*, that have millions of vertices, by using 16 workers and varying the chunk size from 1 to 2^{20}. All three trimming algorithms tend to be efficient when choosing a chunk size between 2^{10} and 2^{16}. Therefore, in our experiments, we fix the chunk size to $2^{12} = 4096$ for both workload balance and efficient scheduling.

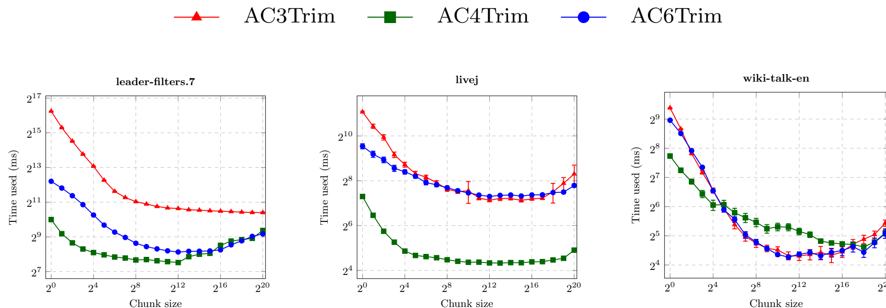

Fig. 3: The practical running time for *AC3Trim, AC4Trim* and *AC6Trim* with 16 workers by varying the chunk size.

The other issue is the upper-bound size of the waiting set Q_p for each worker p in AC4Trim and *AC6Trim*. Here, Q_p is private to worker p, and thus the vertices in Q_p are processed sequentially by worker p without synchronization. A large size of Q_p may lead to workload imbalance. In Table 7, over all tested graphs we show the upper-bound size of Q_p, denoted as $|Q_p|$, for *AC4Trim* and *AC6Trim* by using 16 workers. We can see that $|Q_p|$ is relatively small compared with millions of vertices. Further, over all tested graphs *AC6Trim* has $|Q_p|$ bounded by 900, while *AC4Trim* has $|Q_p|$ up to 20761. Especially, *AC6Trim* has much smaller values of $|Q_p|$ than *AC4Trim* for graphs like *dbpedia* and *twitter-mpi*. That means that *AC6Trim* on average has better workload balance than *AC4Trim*.

9.3 Evaluating the Number of Traversed Edges

To evaluate the Arc-Consistency algorithms, the traditional approach is to count the total number of checked constraints. For each constraint check, a pair of values in the domain $D(X_i)$ and $D(X_j)$ is checked. Such an evaluation is reasonable because 1) most of the running time is spent on checking numerous constraints, 2) the time used for checking each constraint significantly varies for different kinds

Name	*AC4Trim* $\|Q_p\|$	*AC6Trim* $\|Q_p\|$
cambridge.6	17	6
bakery.6	21	52
leader-filters.7	16	103
dbpedia	**20761**	**852**
baidu	439	108
livej	274	8
patent	95	55
wiki-talk-en	84	5
wikitalk	33	7
com-friendster	646	677
twitter	293	287
twitter-mpi	15217	327
ER	**1**	**1**
BA	66	27
RMAT	299	411

Table 7: The upper-bound size of Q_p for *AC4Trim* and *AC6Trim* by using 16 workers. The best and worst cases are in **bold** for each column.

of arc-consistency problems. Analogous to evaluating Arc-Consistency algorithms, we compare the total number of traversed edges of three trimming algorithms. This is especially meaningful for the implicit graphs since their edges are generated on-the-fly, costing most of the running time.

In this experiment, we exponentially increase the number of workers from 1 to 32 and count the largest number of traversed edges per worker over graphs in Table 6. The plots in Figure 4 depict the maximum number of traversed edges per worker for the three compared methods. The x-axis is the number of workers and the y-axis is the number of traversed edges. Also, we choose the total number of edges in a graph as a baseline (denoted as m). Note that, for the AC-4-based trimming algorithm, the out-degree counter of each vertex v in graph is initialized as $v.deg_{out} = |v.post|$. To calculate $v.deg_{out}$, there are two cases: 1) we can traverse all v's successors one by one to count the total numbers of successors (denoted as *AC4Trim*), which means all v's edges of are traversed once; 2) if v' successors are stored successively, we can take the difference between the index of v's first successor and v's last successor without traversing the edges (denoted as *AC4Trim**), which means only v is traversed once and all v's edges are not traversed. Absolutely, *AC4Trim* traverses a higher number of edges than *AC4Trim**.

In Figure 4, a first look over nearly all testing graphs reveals that the number of traversed edges of all three algorithms is linearly decreasing with an increasing number of workers, which achieves a good load balance. Over most of the testing graphs, *AC6Trim* traverses fewer edges compared with *AC4Trim* and *AC3Trim*. *AC3Trim* sometimes traverses more edges than the baseline m for some graphs with large α. Specifically, we make four observations:

- Over graphs with a higher value of α, e.g. *cambridge.6, bakery.6, leader-filter.7, dbpedia, com-friendster* and *RMAT*, *AC3Trim* always traverses much more edges than both *AC4Trim* and *AC6Trim* and even more than the baseline m. This is because *AC3Trim* has the worst work complexity $\mathcal{O}(\alpha(n+m))$ which requires α number of repetitions, but *AC4Trim* and *AC6Trim* have a linear work complexity of $\mathcal{O}(n+m)$.

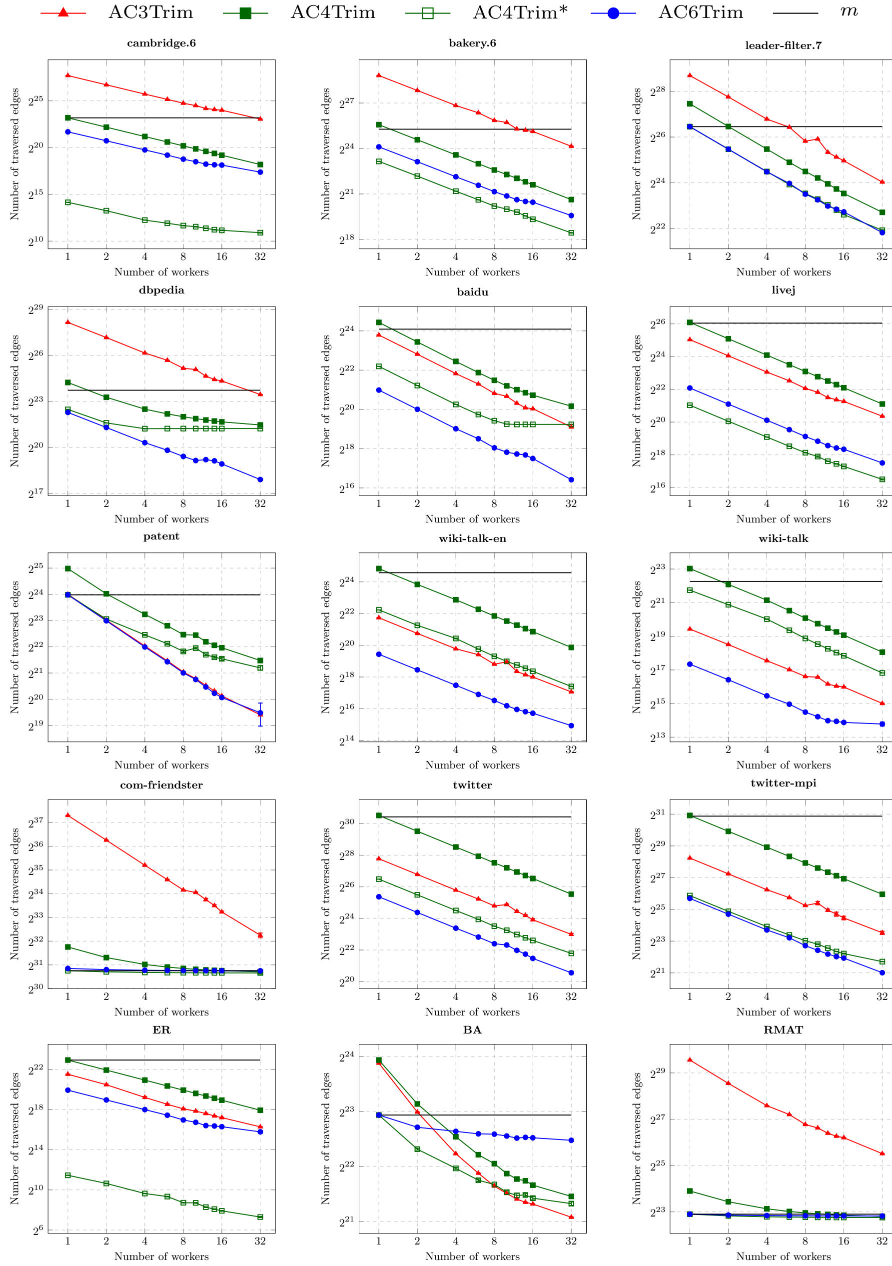

Fig. 4: The maximum number of traversed edges per worker for *AC3Trim*, *AC4Trim*, *AC4Trim** and *AC6Trim* by varying the number of workers. The number of edges m in a graph is chosen as a baseline.

- Over the graphs with a lower value of α, e.g. *wiki-talk-en* and *wikitalk*, *AC3Trim* traverses fewer edges than *AC4Trim*. This is because *AC3Trim* exe-

cutes always close to the best-case time complexity, but *AC4* executes always close to the worst-case time complexity. This is why *AC3Trim* is sometimes more powerful than *AC4Trim* in real-world graphs with a relatively low value of α.

- Over all graphs, *AC6Trim* always traverses much fewer edges than *AC4Trim* even if they have nearly the same time complexity. The reason is that *AC6Trim* can traverse only part of the edges of removed vertices, which is close to the best-case time complexity. However, *AC4Trim* has to traverse all edges to initialize the counters $\forall v \in V : v.deg_{out}$ and all ingoing edges of removed vertices, which is close to the worst-case time complexities. Therefore, *AC6Trim* certainly traverses fewer edges than *AC4Trim*.

- Over all graphs, for all three methods, the number of traversed edges is well bounded without obvious variation even these three methods are non-deterministic. That is, for the number of traversed edges, the affect of non-determinism can be omitted.

Name	1-worker vs 16-worker AC3Trim	1-worker vs 16-worker AC4Trim	1-worker vs 16-worker AC6Trim	AC3Trim vs AC6Trim	AC4Trim vs AC6Trim
cambridge.6	13.04	15.97	11.74	**58.29**	2.08
bakery.6	12.90	15.61	12.58	25.44	2.22
leader-filters.7	13.22	15.15	13.23	4.71	1.75
dbpedia	14.13	5.94	10.26	42.43	6.69
baidu	13.49	13.07	11.19	5.79	9.35
livej	13.69	15.90	13.38	7.58	13.51
patent	14.63	8.09	15.07	1.04	3.72
wiki-talk-en	13.33	15.76	13.17	4.87	35.33
wikitalk	10.89	15.55	11.05	4.31	**36.51**
com-friendster	16.78	**2.00**	1.07	5.57	1.00
twitter	14.54	15.92	**14.95**	5.45	33.31
twitter-mpi	13.66	15.87	13.56	5.77	32.00
ER	**20.00**	**16.00**	12.47	1.87	6.24
BA	**5.90**	4.84	1.33	**0.43**	**0.55**
RMAT	10.23	2.07	**1.05**	10.39	1.02

Table 8: Compare the ratio for the maximum number of traversed edges per worker. The best and worst cases are in **bold** for each column.

In Table 8, columns 2 - 4 compare the ratio of the maximum number of traversed edges per worker between using a single worker and using 16 workers for *AC3Trim*, *AC4Trim* and *AC6Trim*, respectively. We can see that for *AC3Trim*, the ratio is at least 5.9 as *AC3Trim* is easy to be parallelized without using locks or atomic primitives. We also can see that for *AC3Trim* the ratio is larger than 16 in some graphs, e.g. *com-friendster* and *ER*. The reason is that parallel *AC3Trim* is non-deterministic; that is, different trimming orders lead to different numbers of traversed edges; if numerous vertices are early determined as `DEAD`, the time complexity is close to the best case. For *AC4Trim* and *AC6Trim*, the ratio is relatively low in some graphs with large α, e.g. *RMAT* and *com-friendster*, as large α always leads to high working depths.

In columns 5 and 6 of Table 8, we fix using 16 workers and compare the ratio of traversed edge numbers between *AC3Trim* and *AC6Trim* and between *AC4Trim* and *AC6Trim*. We can see that *AC6Trim* traverses much fewer edges

than *AC3Trim*, up to 58 times over the graph *cambridge.6*; *AC6Trim* traverses much fewer edges then *AC4Trim*, up to 36 times over the graph *wikitalk*. *AC3Trim* traverses the fewest edges in some graphs, e.g. *BA*, as the time complexity is close to the best case.

9.4 Evaluating the Real Running Time

In this experiment, we exponentially increase the number of workers from 1 to 32 and evaluate the real running time over graphs in Table 6. The plots in Figure 5 depict the performance of the three compared methods. The x-axis is the number of workers and the y-axis is the execution time (millisecond). The first look over all testing graphs reveals that the trends for the running time are much different from the trends for the number of traversed edges shown in Figure 4. This can be explained as follow:

- Although *AC6Trim* always traverses the fewest numbers of edges, *AC6Trim* is slower than *AC4Trim* in some graphs, e.g. *cambridge.6, livej, pokec* and *ER*, and even *AC3Trim* is the fastest in some graphs, e.g. *BA*. The main reason is that *AC6Trim* is cache-unfriendly while *AC3Trim* and *AC4Trim* are cache-friendly. That means *AC6Trim* cannot fully use caches to archive the best performance even if *AC6Trim* traverse the least number of edges. The other reason is that maintaining the supporting sets in *AC6Trim* costs much more computational time than maintaining the out-degree counters in *AC4Trim*; there is no auxiliary data structure in *AC3Trim* so that no computational time is spent on this part.

- In *AC4Trim*, the running times have a wide variation in certain graphs, e.g. *bakery.6*, *leader-filter.7*, *livej*. The reason is that *AC4Trim* is sometimes less cache-friendly. The unexpected missing cache leads to the performance decreased. However, *AC3Trim* is always cache-friendly and *AC6Trim* is always cache-unfriendly so that their performance is more stable than *AC3Trim*.

- In *AC6Trim*, the running times begin to increase when using more than 4 workers in certain graphs, e.g. *dbpedia* and *baidu*. The reason is that the supporting set shared by multi-worker is synchronized by busy waiting, which leads to contention. At the same time, *AC4Trim* still has not obvious speedup as there is less contention to use atomic primitive updating the out-degree counters. However, *AC3Trim* always has a speedup by multiple workers and even has the best performance with 16 workers in some graphs, e.g. *BA*, as *AC3Trim* has no shared data structures and thus no contention.

In columns 2 - 4 of Table 9 we compare the running time speedup between using one worker and 16 workers for *AC3Trim*, *AC4Trim* and *AC6Trim*, respectively. It is clear that *AC3Trim* achieves the best speedup and *AC4* achieves the worst speedup. This is because of the contention on shared data structures with multiple workers. In columns 5 and 6 of Table 9 we fix using 16 workers and compare the speedups for the running time between *AC6Trim* and *AC3Trim* and between *AC6Trim* and *AC4Trim*. We can see that our *AC6Trim* is up to 24 times faster than *AC3Trim* over *RMAT* and up to 7.8 times faster than *AC4Trim* over *leader-filters.7*. However, in some graphs, *AC4Trim* and *AC3Trim* have better performances than *AC6Trim*.

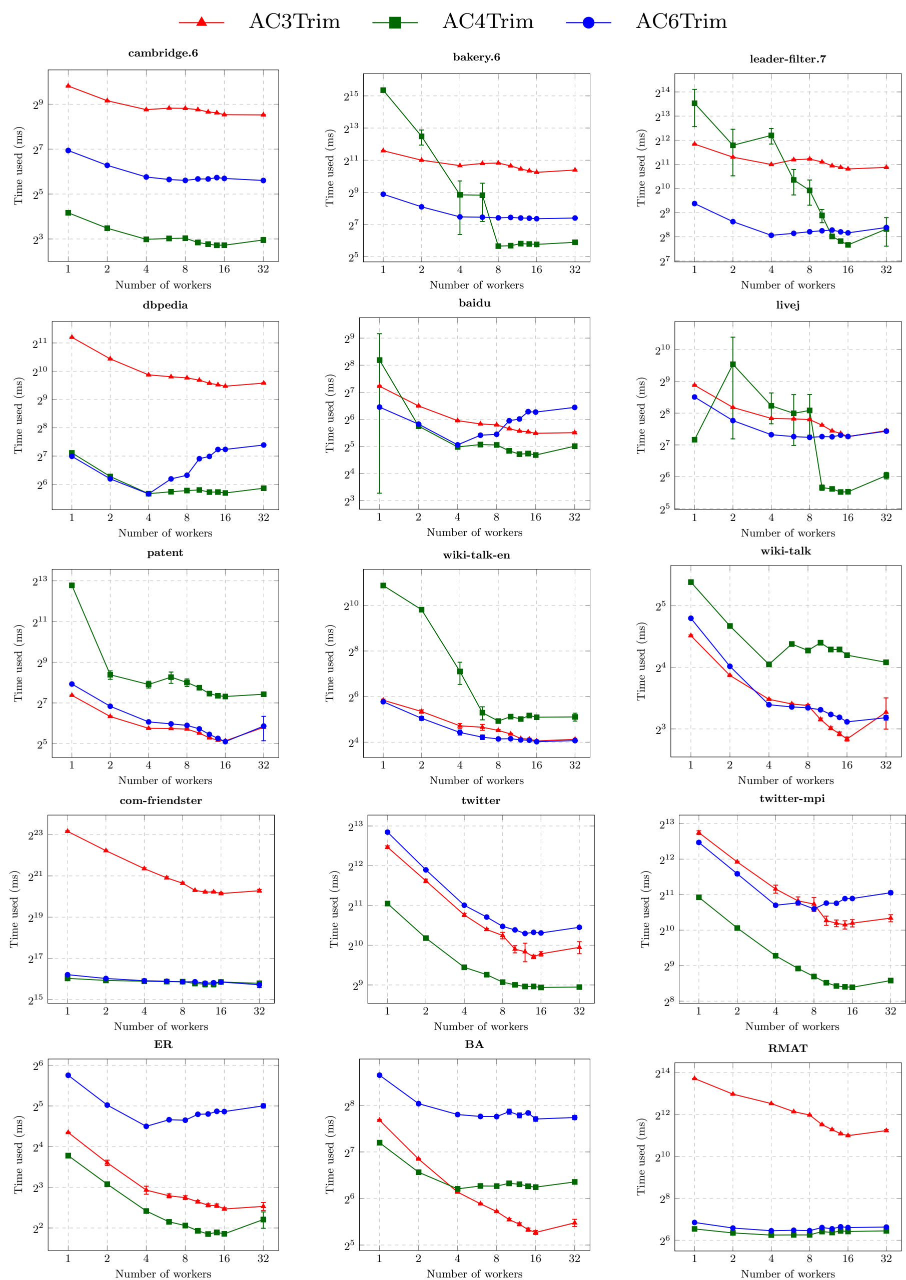

Fig. 5: The real running time for *AC3Trim*, *AC4Trim* and *AC6Trim* by varying the number workers.

Name	16-workers speedup vs 1-worker AC3Trim	AC4Trim	AC6Trim	AC6Trim speedup vs AC3Trim	AC4Trim
cambridge.6	2.42	2.72	2.37	7.15	0.13
bakery.6	2.54	**771.96**	2.87	7.39	0.33
leader-filters.7	**2.04**	58.42	2.32	6.27	0.71
dbpedia	3.31	2.67	**0.84**	4.71	0.34
baidu	3.34	11.38	1.14	0.58	0.33
livej	3.07	3.11	2.36	1.00	0.30
patent	4.71	44.02	**7.11**	1.03	**4.65**
wiki-talk-en	3.45	54.85	3.37	1.02	2.10
wikitalk	3.20	2.28	3.21	0.82	2.12
com-friendster	8.09	1.13	1.28	19.62	1.00
twitter	6.42	4.32	5.80	0.70	0.39
twitter-mpi	5.85	5.79	2.99	0.62	0.18
ER	3.68	3.79	1.86	0.19	**0.12**
BA	5.32	1.94	1.92	**0.18**	0.36
RMAT	**6.62**	**1.09**	1.18	**20.97**	0.88

Table 9: Compare the speedups for running times between using 1-worker and 16-worker for *AC3Trim*, *AC4Trim*, and *AC6Trim*, respectively; by fixing with 16 workers, compare the running time speedup between *AC6Trim* and *AC3Trim* and between *AC6Trim* and *AC4Trim*. The best and worst cases are in **bold** for each column.

9.5 Evaluating Stability

One issue is the stability of the trimming algorithms when executing the same algorithm multiple times. In this experiment, we compare 50 testing result over three chosen graphs, *leader-filters.7*,*livej*, and *wiki-talk-en*. In Figure 6, the x-axis of plots is the index of the repeating times. The upper three plots in Figure 6 depict the number of traversed edges for three trimming methods, in which the y-axis is the number of traversed edges. We observe that the number of traversed edges is well bounded for all three trimming methods. The lower three plots in Figure 6 depict the running time for three trimming methods, in which the y-axis is the running time. We observe that *AC4Trim* always has a wider variation than other methods. The reason is that parallel *AC4Trim* is non-deterministic, which means each time the order of removed vertices is different; *AC4Trim* is not always cache-friendly as vertices are not sequentially traversed; there is a high probability that the performance decreases due to the unexpected missing cache. Even *AC3Trim* and *AC6Trim* are also non-deterministic, *AC3Trim* is cache-friendly and *AC6Trim* is cache-unfriendly; cache-friendliness does not always lead to a wide performance variation.

9.6 Evaluating Scalability

An issue is the scalability of the trimming algorithms when the size of graphs is varied. In this experiment, we test the scalability of the trimming algorithms over the three largest graphs, i.e., *com-friendster, twitter*, and *twitter-mpi*. Using 16 workers, we vary the number of edges and vertices by randomly sampling at a ratio from 10% to 100%, respectively. By sampling the edges, we simply remove the

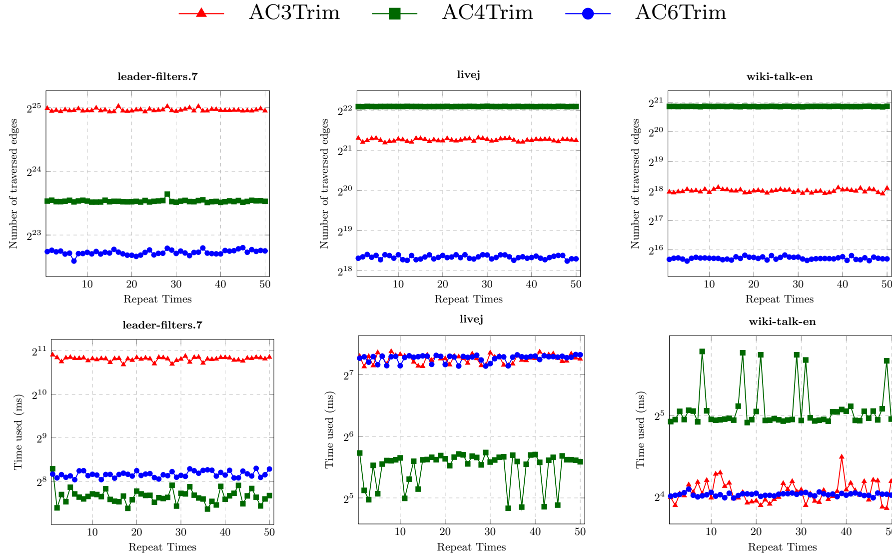

Fig. 6: The stability of the traversed edge number and the running time for *AC3Trim*, *AC4Trim* and *AC6Trim*.

unsampled edges. By sampling the vertices, we simply set the unsampled vertices to `DEAD`. As shown in Figure 7, we can see that the smaller ratio of sampling edges or vertices always leads to the higher ratio of trimable vertices, e.g. *twitter* and *twitter-mpi*; but for *com-friendster* all vertices are always trimmable with any ratio of sampling. Especially when sampling 10% edges or vertices, nearly 60% of vertices can be trimmed for *twitter* and *twitter-mpi*, and without sampling less than 20% of vertices can be trimmed. This is result is reasonable as more unsampled edges or vertices will lead to more vertices without out-going edges and thus can be trimmed.

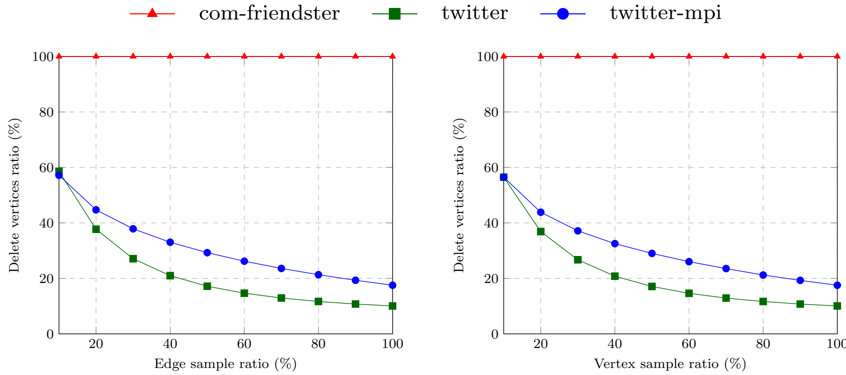

Fig. 7: The ratio of trimmable vertices.

We show the result of sampling edges in Figure 8, in which the x-axis of plots is the ratio of sampled edges. The upper three plots in Figure 8 depict the maximum number of traversed edges per worker. We observe that the number of traversed edges is generally increasing with the ratio of the sampled edges. Not surprisingly, *AC6Trim* traverses the least number of edges, and *AC3Trim* traverses the highest number of edges. But for *AC3Trim* the number of traversed edges fluctuates when increasing the sampling ratio of edges as the number of peeling steps α may fluctuate with a different sampling ratios of edges. The lower three plots in Figure 8 depict the real running time. We make three observations.

- Over *com-friendster*, *AC6Trim* has the best performance. The reason is that 100% of vertices can be trimmed so that *AC4Trim* accesses all vertices almost randomly. In this case, *AC4Trim* is likely too cache-unfriendly, and the cache can not provide an obvious speedup.

- Over *twitter* and *twitter-mpi*, *AC4Trim* has a wide variation and *AC4Trim* performs worse than *AC6Trim* in most of cases. The reason is that for *AC4Trim* more trimmable vertices lead to the cache being less effective, and sometimes the cache can provide a speedup but sometimes not; but *AC6Trim* is cache-unfriendly, and the cache cannot affect the running time.

- Over *twitter* and *twitter-mpi*, *AC3Trim* always has as good performance as *AC6Trim* even if *AC3Trim* traverse much more edges than *AC6Trim*. The reason is that *AC3Trim* is cache-friendly and achieve a high speedup with caching.

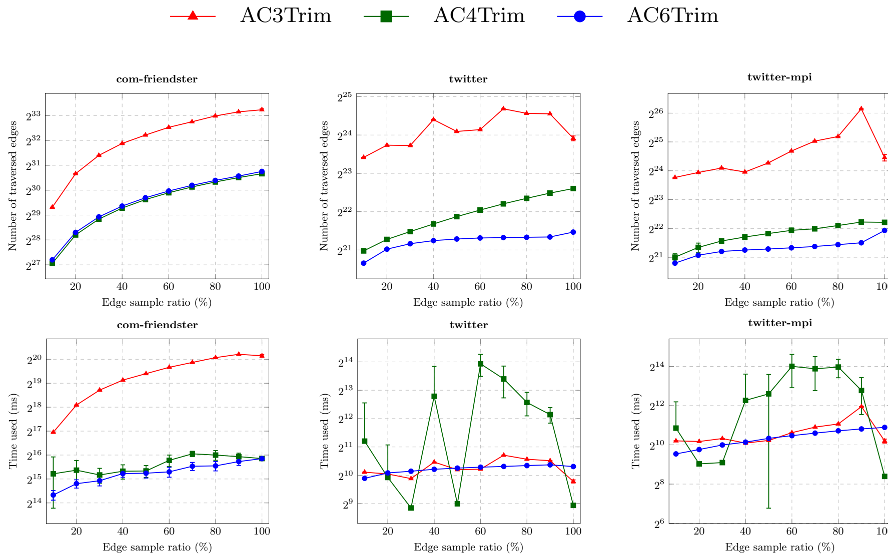

Fig. 8: The scalability of *AC3Trim, AC4Trim* and *AC6Trim* by using 16 workers. The number of edges is varied by randomly sampling from 10% to 100%

Analogously, we show the result of sampling vertices in Figure 9, in which the x-axis of plots is the ratio of sampling vertices. There are almost the same trends as shown in Figure 8. One difference is in upper three plots; that is, over

twitter and *twitter* we can see *AC4Trim* traverse more edges than *AC6Trim* as the the unsampled vertices are set to `DEAD` and their out-degree counters are still calculated. The other difference is in lower three plots; that is, over *twitter* and *twitter* we can see *AC6Trim* performs much better than *AC4Trim*, except when vertices are 100% sampled. In this experiment, for implicit graphs loaded into memory, we can see that *AC6Trim* is most scalable no matter how many vertices are trimmed and how many edges or vertices are sampled.

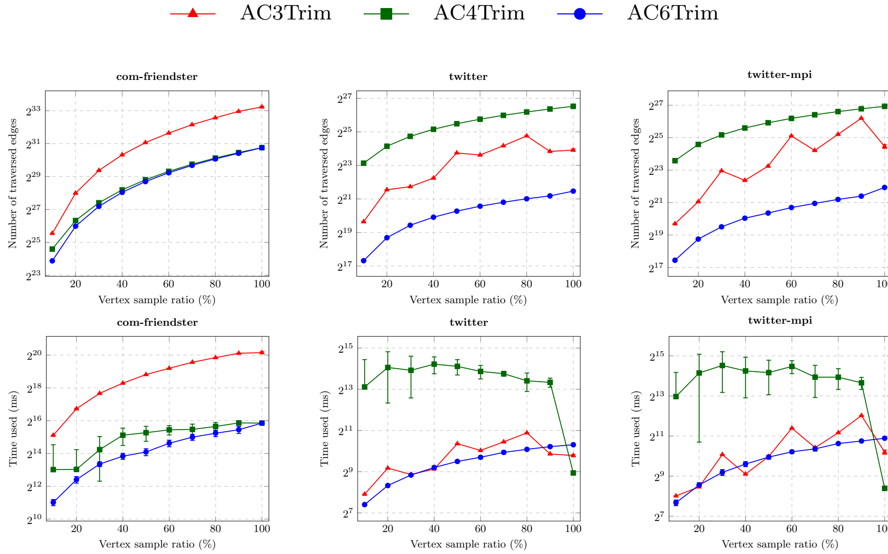

Fig. 9: The scalability of *AC3Trim, AC4Trim* and *AC6Trim* by using 16 workers. The vertices are varied by randomly sampling at radio from 10% to 100%

10 Conclusions

In this work, we study graph trimming algorithms for removing vertices without outgoing edges. The arc-consistency algorithms, in particular AC-3, AC-4, and AC-6, can be applied to graph trimming, leading to the so-called AC-3-based, AC-4-based, and AC-6-based trimming algorithms, respectively. Based on that, we propose parallel AC-4-based and AC-6-based trimming algorithms that have better worst-case time complexities than AC-3-based. For these three trimming algorithms, we summarize the trend and test results bellow:

- The common existing graph trimming method is actually parallel AC-3-based, which has worst-case time complexity. Although AC-4-based and AC-6-based algorithms have similar worst-case time complexities, the AC-6-based algorithm traverses fewer edges per worker and requires less memory usage than the AC-4-based one.

– For implicit graphs in which edges are generated on-the-fly, the AC6-based algorithm does not rely on the reversed graphs, unlike the AC4-based algorithm, and thus is more suitable for trimming implicit graphs.
– For explicit graphs in which all edges are linearly stored in memory, our AC-6-based algorithm does not always outperform the other methods, but always traverses the least number of edges and has the best stability and scalability.

In future work, we can apply graph trimming to Strong Connected Components (SCC) decomposition as a great percentage of size-1 SCCs can be trimmed in parallel. We also can apply graph trimming to cycle directions as the trimmable vertices cannot be in cycles and can be trimmed in parallel. Both applications depend on Depth First Search (DFS), which is hard to parallelize. However, our trimming techniques can efficiently trim graphs in parallel if there are a large portion of trimmable vertices in graphs. In particular, we can apply the AC-6-based algorithm to trim the model checking graphs in which edges are expensively calculated on-the-fly; fewer traversed edges will likely save the running time. In addition, we can device the graph trimming algorithms for distribute system to address the large scale parallelism.

References

1. Aggarwal, A., Anderson, R.J.: A random NC algorithm for depth first search. Combinatorica **8**(1), 1–12 (1988)
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBpedia: A nucleus for a web of open data. In: The Semantic Web, pp. 722–735. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-76298-0_52. URL https://doi.org/10.1007%2F978-3-540-76298-0_52
3. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge Discovery and Data Mining, pp. 44–54. Association for Computing Machinery (ACM) (2006). DOI 10.1145/1150402.1150412. URL https://doi.org/10.1145%2F1150402.1150412
4. Batagelj, V., Zaversnik, M.: An $O(m)$ algorithm for cores decomposition of networks. CoRR **cs.DS/0310049** (2003). URL http://arxiv.org/abs/cs/0310049
5. Bessière, C.: Arc-consistency and arc-consistency again. Artificial Intelligence **65**(1), 179–190 (1994). DOI 10.1016/0004-3702(94)90041-8. URL https://doi.org/10.1016%2F0004-3702%2894%2990041-8
6. Blelloch, G.E., Maggs, B.M.: Parallel algorithms. In: Algorithms and theory of computation handbook: special topics and techniques, pp. 25–25 (2010)
7. Bloemen, V.: On-the-fly parallel decomposition of strongly connected components. Master's thesis, University of Twente (2015)
8. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core on-the-fly SCC decomposition. ACM SIGPLAN Notices **51**(8), 1–12 (2016). DOI 10.1145/3016078.2851161. URL https://doi.org/10.1145%2F3016078.2851161
9. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, K.: Measuring user influence in twitter: The million follower fallacy. In: Proceedings of the International AAAI Conference on Web and Social Media, vol. 4 (2010)
10. Cha, M., Haddadi, H., Benevenuto, F., Gummadi, K.P.: Measuring user influence in twitter: The million follower fallacy. In: ICWSM. Washington DC, USA (2010)
11. Chen, X., Chen, C., Shen, J., Fang, J., Tang, T., Yang, C., Wang, Z.: Orchestrating parallel detection of strongly connected components on GPUs. Parallel Computing **78**, 101–114 (2018). DOI 10.1016/j.parco.2017.11.001. URL https://doi.org/10.1016%2Fj.parco.2017.11.001
12. Chen, Y., Guo, B., Huang, X.: δ-transitive closures and triangle consistency checking: a new way to evaluate graph pattern queries in large graph databases. The Journal of Supercomputing (2019). DOI 10.1007/s11227-019-02762-4. URL https://doi.org/10.1007%2Fs11227-019-02762-4

13. Cooper, P.R., Swain, M.J.: Arc consistency: parallelism and domain dependence. Artificial Intelligence **58**(1-3), 207–235 (1992). DOI 10.1016/0004-3702(92)90008-l. URL https://doi.org/10.1016%2F0004-3702%2892%2990008-l
14. Coppersmith, D., Fleischer, L., Hendrickson, B., Pinar, A.: A divide-and-conquer algorithm for identifying strongly connected components. Tech. rep., Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US) (2003). DOI 10.2172/889876. URL https://doi.org/10.2172%2F889876
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
16. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science and Engineering **5**(1), 46–55 (1998). DOI 10.1109/99.660313. URL https://doi.org/10.1109%2F99.660313
17. Defo, R.K., Wang, R., Manjunathaiah, M.: Parallel bfs implementing optimized decomposition of space and kmc simulations for diffusion of vacancies for quantum storage. Journal of Computational Science **36**, 101018 (2019)
18. Dhulipala, L., Blelloch, G., Shun, J.: Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 293–304 (2017)
19. Dib, M., Abdallah, R., Caminada, A.: Arc-consistency in constraint satisfaction problems: A survey. In: 2010 Second International Conference on Computational Intelligence, Modelling and Simulation. IEEE (2010). DOI 10.1109/cimsim.2010.18. URL https://doi.org/10.1109%2Fcimsim.2010.18
20. Erlebach, T., Hagerup, T., Jansen, K., Minzlaff, M., Wolff, A.: Trimming of graphs, with application to point labeling. Theory of Computing Systems **47**(3), 613–636 (2010)
21. Fleischer, L.K., Hendrickson, B., Pınar, A.: On identifying strongly connected components in parallel. In: International Parallel and Distributed Processing Symposium, pp. 505–511. Springer (2000). DOI 10.1007/3-540-45591-4_68. URL https://doi.org/10.1007%2F3-540-45591-4_68
22. Fleischer, L.K., Hendrickson, B., Pınar, A.: On identifying strongly connected components in parallel (November 2014), 505–511 (2007). DOI 10.1007/3-540-45591-4_68
23. Freuder, E., Régin, J.C.: Using constraint metaknowledge to reduce arc consistency computation. Artificial Intelligence **107**(1), 125–148 (1999). DOI 10.1016/s0004-3702(98)00105-2. URL https://doi.org/10.1016%2Fs0004-3702%2898%2900105-2
24. Gao, Y., Dong, W., Wu, W., Chen, C., Li, X.Y., Bu, J.: Scalpel: Scalable preferential link tomography based on graph trimming. IEEE/ACM Transactions on Networking **24**(3), 1392–1403 (2015)
25. Harabor, D., Grastien, A.: Online graph pruning for pathfinding on grid maps. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 25 (2011)
26. Heule, M.J.: Trimming graphs using clausal proof optimization. In: International Conference on Principles and Practice of Constraint Programming, pp. 251–267. Springer (2019)
27. Hojati, R., Brayton, R.K., Kurshan, R.P.: BDD-based debugging of designs using language containment and fair CTL. In: International Conference on Computer Aided Verification, pp. 41–58. Springer (1993). DOI 10.1007/3-540-56922-7_5. URL https://doi.org/10.1007%2F3-540-56922-7_5
28. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: a dsl for easy and efficient graph analysis. In: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, pp. 349–362 (2012). DOI 10.1145/2248487.2151013
29. Hong, S., Rodia, N.C., Olukotun, K.: On fast parallel detection of strongly connected components (SCC) in small-world graphs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '13. ACM Press (2013). URL https://doi.org/10.1145%2F2503210.2503246
30. III, W.M., Hendrickson, B., Plimpton, S.J., Rauchwerger, L.: Finding strongly connected components in distributed graphs. Journal of Parallel and Distributed Computing **65**(8), 901–910 (2005). DOI 10.1016/j.jpdc.2005.03.007. URL https://doi.org/10.1016%2Fj.jpdc.2005.03.007
31. JéJé, J.: An introduction to parallel algorithms. Reading, MA: Addison-Wesley (1992)
32. Ji, Y., Liu, H., Huang, H.H.: iSpan: Parallel identification of strongly connected components with spanning trees. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE (2018). DOI 10.1109/sc.2018.00061. URL https://doi.org/10.1109%2Fsc.2018.00061

33. Kirousis, L.M.: Fast parallel constraint satisfaction. Tech. Rep. 1 (1993). DOI 10.1016/0004-3702(93)90063-h. URL https://doi.org/10.1016%2F0004-3702%2893%2990063-h
34. Kumar, R., Novak, J., Tomkins, A.: Structure and evolution of online social networks. In: Link Mining: Models, Algorithms, and Applications, pp. 337–357. Springer New York (2010). DOI 10.1007/978-1-4419-6515-8_13. URL https://doi.org/10.1007%2F978-1-4419-6515-8_13
35. Kunegis, J.: KONECT. In: Proceedings of the 22nd International Conference on World Wide Web - WWW '13 Companion. ACM, ACM Press (2013). DOI 10.1145/2487788.2488173. URL https://doi.org/10.1145%2F2487788.2488173
36. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Signed networks in social media. In: Proceedings of the SIGCHI conference on human factors in computing systems, pp. 1361–1370 (2010)
37. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pp. 177–187 (2005)
38. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data (2014)
39. Lowe, G.: Concurrent depth-first search algorithms based on Tarjan's Algorithm. International Journal on Software Tools for Technology Transfer **18**(2), 129–147 (2016). DOI 10.1007/s10009-015-0382-1
40. Mackworth, A.K.: Consistency in networks of relations. Artificial Intelligence **8**(1), 99–118 (1977). DOI 10.1016/0004-3702(77)90007-8. URL https://doi.org/10.1016%2F0004-3702%2877%2990007-8
41. Mackworth, A.K., Freuder, E.C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artificial intelligence **25**(1), 65–74 (1985). DOI 10.1016/0004-3702(85)90035-9. URL https://doi.org/10.1016%2F0004-3702%2885%2990035-9
42. Merz, S.: Model checking: A tutorial overview. In: Modeling and Verification of Parallel Processes, pp. 3–38. Springer Berlin Heidelberg (2001). DOI 10.1007/3-540-45510-8_1. URL https://doi.org/10.1007%2F3-540-45510-8_1
43. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the fourteenth annual ACM symposium on Parallel Algorithms and Architectures - SPAA '02. ACM Press (2002). DOI 10.1145/564870.564881. URL https://doi.org/10.1145%2F564870.564881
44. Milman, G., Kogan, A., Lev, Y., Luchangco, V., Petrank, E.: Bq: A lock-free queue with batching. In: Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures - SPAA '18. ACM Press (2018). DOI 10.1145/3210377.3210388. URL https://doi.org/10.1145%2F3210377.3210388
45. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence **28**(2), 225–233 (1986). DOI 10.1016/0004-3702(86)90083-4. URL https://doi.org/10.1016%2F0004-3702%2886%2990083-4
46. social network, F.: Friendster: The online gaming social network. https://archive.org/details/friendster-dataset-201107
47. Niu, X., Sun, X., Wang, H., Rong, S., Qi, G., Yu, Y.: Zhishi.me - weaving chinese linking open data. In: The Semantic Web – ISWC 2011, pp. 205–220. Springer Berlin Heidelberg (2011). DOI 10.1007/978-3-642-25093-4_14. URL https://doi.org/10.1007%2F978-3-642-25093-4_14
48. Pelánek, R.: BEEM: Benchmarks for explicit model checkers. In: Model Checking Software, pp. 263–267. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-73370-6_17. URL https://doi.org/10.1007%2F978-3-540-73370-6_17
49. Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters **20**(5), 229–234 (1985). DOI 10.1016/0020-0190(85)90024-9. URL https://doi.org/10.1016%2F0020-0190%2885%2990024-9
50. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Parallel explicit model checking for generalized Büchi automata. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 9035, pp. 613–627. Springer Verlag (2015). DOI 10.1007/978-3-662-46681-0_56
51. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: AAAI (2015). URL http://networkrepository.com
52. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, USA (2009)

53. Shun, J.: Shared-memory parallelism can be simple, fast, and scalable. PUB7255 Association for Computing Machinery and Morgan & Claypool (2017)
54. Slota, G.M., Rajamanickam, S., Madduri, K.: BFS and coloring-based parallel algorithms for strongly connected components and related problems. In: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, pp. 550–559. IEEE Computer Society (2014). DOI 10.1109/IPDPS.2014.64
55. Sun, J., Kunegis, J., Staab, S.: Predicting user roles in social networks using transfer learning with feature transformation. In: 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), pp. 128–135. IEEE, IEEE (2016). DOI 10.1109/icdmw.2016.0026. URL https://doi.org/10.1109%2Ficdmw.2016.0026
56. Takac, L., Zabovsky, M.: Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations, vol. 1 (2012)
57. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972). DOI 10.1137/0201010. URL https://doi.org/10.1137%2F0201010
58. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing - PODC '95, pp. 214–222. ACM Press (1995). DOI 10.1145/224964.224988. URL https://doi.org/10.1145%2F224964.224988
59. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440 (1998). URL https://doi.org/10.1515/9781400841356.301
60. Xiaoping, G., Mengyu, R., Hong, Z., Ping, W., Ruijun, R., Feng, G.: Construction technology of knowledge graph and its application in power grid. In: E3S Web of Conferences, vol. 256, p. 01039. EDP Sciences (2021)