Aquila: Adaptive Parallel Computation of Graph Connectivity Queries

Yuede Ji
Graph Computing Lab
George Washington University
yuedeji@gwu.edu

H. Howie Huang
Graph Computing Lab
George Washington University
howie@gwu.edu

ABSTRACT

Graph connectivity algorithms answer whether two nodes in a graph are connected under specific conditions, which are beneficial to a number of applications, such as pattern recognition and cybersecurity. Unfortunately, existing graph computing frameworks support only a small number of connectivity algorithms and achieve low computation parallelism. In this paper, we have designed an adaptive parallel computation framework, Aquila, that covers a wide range of different highly optimized graph connectivity algorithms. Given a graph, Aquila first transforms the query if it can be answered with partial computation. During the computation, Aquila is able to greatly reduce the workload by up to 98%. Furthermore, Aquila identifies the irregular tasks in the connectivity algorithms and applies different parallel strategies for different tasks. As a result, Aquila significantly outperforms existing systems such as Multistep, Galois, Ligra, GraphChi, X-Stream, DFS, and Boost, by average 13×, 53×, 264×, 364×, 1, 369×, 45×, and 255×, respectively.

KEYWORDS

graph; connectivity; parallel computation

ACM Reference Format:
Yuede Ji and H. Howie Huang . 2020. Aquila: Adaptive Parallel Computation of Graph Connectivity Queries. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3369583.3392690

1 INTRODUCTION

Graph is a natural representation for various types of data, such as social network [8, 26, 40], road map [9], and computer network [7]. To mine useful knowledge from the graphs, multiple graph computation systems have been proposed recently [4, 11, 19, 30, 32, 34, 39], which are able to efficiently compute some commonly used algorithms, including breadth-first search (BFS), PageRank, single source shortest path, and triangle counting. However, there are a group of important algorithms — the graph connectivity algorithms,

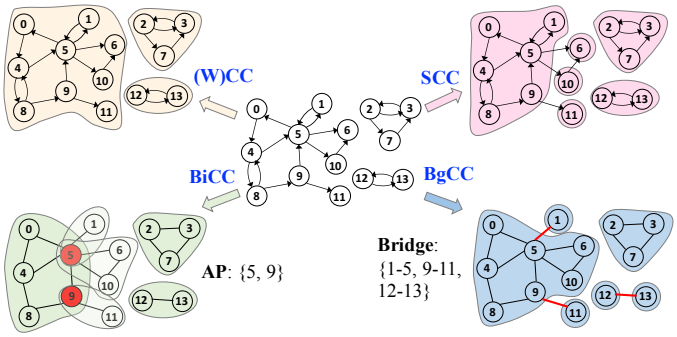


Figure 1: For the example graph, the WCC and SCC are shown on the top. Assuming the edges are undirected, the CCs are shown in the top left, BiCC and BgCC in the bottom. Each shaded area is an XCC.

which unfortunately are not well supported by existing graph systems. The graph connectivity algorithms answer whether two nodes in a graph are connected under some conditions [17]. Depending on the edge type, the commonly used connectivity algorithms can be classified into two categories:

For directed graphs, weakly connected component (**WCC**) and strongly connected component (**SCC**) are most popular. A WCC is the maximal subgraph where there is at least one path between any two nodes if each directed edge were considered as undirected. The example graph in Figure 1 has three WCCs as shown on the top left. On the other hand, an SCC is the maximal subgraph where any node has at least one directed path to all the others. The example graph has six SCCs as shown on the top right of Figure 1.

For undirected graphs, the algorithms such as connected component (**CC**), biconnected component (**BiCC**), and bridgeless connected component (**BgCC**) are commonly used. A CC is the maximal subgraph where any two nodes share at least one path. The CCs of the above example graph are same to WCC, without edge directions, as shown on the top left of Figure 1. A BiCC is the maximal subgraph without any articulation point (AP), i.e., cut vertex, whose removal will increase the number of connected components. Assuming the edges in the example graph are undirected, there are two APs, i.e., vertex 5 and 9, and six BiCCs, where AP vertex 5 appears in three different BiCCs. Moreover, a BgCC is the maximal subgraph without any bridge, i.e., cut edge, whose removal will also increase the number of connected components. There are three bridges and six BgCCs for the example graph shown on the bottom

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392690

right of Figure 1. In this paper, we use the term of **XCC** to represent the set of WCC, SCC, CC, BiCC, and BgCC.

1.1 Motivation

This work is motivated by three main observations. First, existing parallel graph computation frameworks only support a limited number of connectivity algorithms. While most of them, e.g., PowerGraph [19], GraphChi [32], GraphX [20], X-Stream [39], Ligra [42], PowerLyra [11], and Galois [37], can compute CC, only a few such as GraphChi and X-Stream are able to compute SCC. Unfortunately, none of these systems are able to compute more challenging algorithms such as BiCC or BgCC. There are also prior works targeting a specific problem, e.g., Hong's method [23] for SCC, iSpan [27] for SCC, and Multistep [45] for CC and SCC. As a result, many high-level applications continue to use inefficient implementations of connectivity algorithms. For example, a recent work [50] on betweenness centrality computation still uses the serial Tarjan's algorithm to compute the biconnected components.

Second, existing methods only provide the complete computation of the algorithm on the entire graph. For example, for a frequently asked query of "is this graph connected?", one can identify all the CCs and later verify whether there exist more than one CC. This method is used in current graph frameworks [32, 37]. In this work, we have observed that many connectivity queries can actually be answered with *partial computation*, which does not require computing the entire graph. As a result, we can transform such queries to simpler, faster computations. To answer the aforementioned query, one can simply compute one CC, to be even faster, the smallest CC. For the example graph in Figure 1, we only need to compute the CC with vertices 12 and 13, which would avoid the more expensive computation on the majority of the graph.

On the other hand, even for the cases requiring complete computation, existing methods are suboptimal. For example, the current method for BiCC computation takes every vertex as the root and runs breadth-first search (BFS) to find whether the root's parent is an articulation point (AP). Unfortunately, most (up to 99%) of these BFSes will not find any AP (discussed in Section 4). One can see that there are only two APs among all the 14 vertices for the example graph in Figure 1. In this work, we will identify and eliminate such inefficiency to dramatically increase the algorithm performance.

Third, existing systems fail to take advantage of the heterogeneous tasks, resulting in low computation parallelism. Naturally, the tasks of connectivity algorithms share the same irregular property when running on real-world graphs. That is, a few tasks compute the majority of vertices and edges in the graph, while the remaining tasks consist of a large number of small connected components. One can get a glimpse of such phenomenon from Figure 1. In each XCC case, there are always one big XCC with several small ones. Such power-law task distribution would require a good parallel strategy in order to increase the system utilization and performance.

1.2 Contribution

In this work, we have designed an adaptive parallel computation framework, Aquila, that covers a wide range of different highly optimized graph connectivity algorithms. As shown in Figure 2,

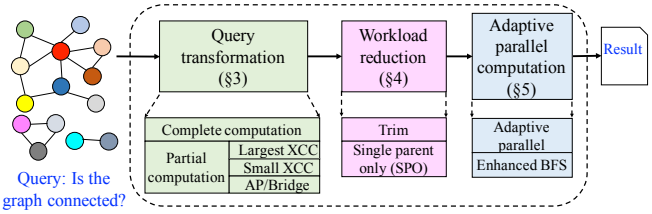


Figure 2: The framework of Aquila.

it takes a graph and the query as inputs, and applies three main techniques:

Query Transformation. Aquila transforms the computation queries if possible. In particular, we classify the queries into four categories, (1) complete computation in addition to three partial computation types, i.e., (2) computing the largest XCC, (3) computing small XCCs, and (4) computing AP and bridge. For each category, Aquila utilizes various strategies to speed up the computation. As a result, Aquila is able to significantly improve the performance by up to three orders of magnitude compared with the current strategies.

Workload Reduction. Aquila further reduces the workload with trim and single parent only (SPO) techniques. The trim technique is generic to all the connectivity algorithms by removing specific subgraph patterns. Trim is able to significantly reduce the workload by up to 21%. For the BiCC and BgCC queries that may need to run a large number (up to vertex count) of BFSes, Aquila applies the SPO technique to remove the unnecessary BFSes that would not lead to any BiCC or BgCC. As a result, SPO is able to further reduce the number of BFSes by up to 77%. Together, the two techniques are able to reduce up to 98% workloads.

Adaptive Parallel Computation. Aquila adaptively applies BFS for the few large tasks, and label propagation and concurrent BFS for the large number of small tasks. In contrast, current methods directly apply a single method (either BFS, depth-first search, or label propagation) to compute the connectivity algorithms [5, 23, 44]. Noticing that the few large tasks take the majority of the run time, we further enhance the parallel BFS with multi-pivot and relaxed synchronization techniques. In the end, such adaptive parallel strategy is able to achieve 6.7× speedup on average for different connectivity algorithms, including BiCC and BgCC which are not supported in many existing systems.

Evaluation. We have implemented Aquila[1] and compared with ten related systems, including four popular graph computation systems, i.e., Galois [37], X-Stream [39], GraphChi [32], and Ligra [42], four implementations for specific connectivity problems, i.e., Multistep for CC and SCC [45], iSpan for SCC [27], Slota for BiCC [44], and Hong's SCC method [23], and two serial implementations, i.e., DFS-based, and the boost graph library [43]. We have tested eleven different graphs, including nine real-world graphs and two synthetic graphs. Our evaluation shows that Aquila is able to significantly improve the performance on average by 1.1×, 3.7×, 13×, 21×, 53×, 264×, 364×, 1, 369×, 45×, and 255× compared to iSpan, Hong's SCC, Multistep, Slota's BiCC, Galois, Ligra, GraphChi, X-Stream, DFS, and Boost, respectively. Specifically, Aquila is able to outperform the state-of-the-art method for each XCC on average

[1]The source code of Aquila is available at https://github.com/iHeartGraph/Aquila

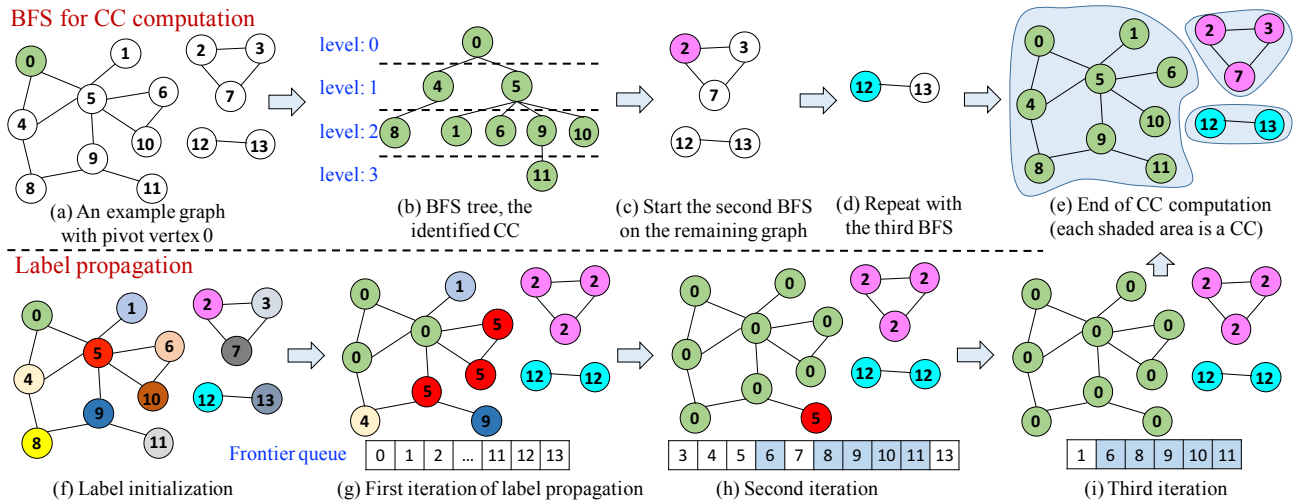


Figure 3: CC computation with BFS and label propagation. Given an example graph in (a), the BFS-based solution runs the first BFS from pivot vertex 0 in (b), the second BFS from pivot vertex 2 in (c), and the third one from pivot vertex 12 in (d). The final CC computation result is shown in (e). Label propagation method initializes all the vertices with their own labels in (f), and runs three iterations from (g) to (i), and gets the final CC result shown in (e).

by 5.9×, 1.1×, 20.7×, and 9.4×, for (W)CC, SCC, BiCC, and BgCC, respectively.

Comparison. Aquila is different from prior works in several aspects. First, unlike the traditional graph frameworks [32, 37, 39, 42] that aim to support different graph algorithms, Aquila is a unique framework that focuses on a variety of graph connectivity algorithms. Second, Aquila not only optimizes the complete computation for connectivity algorithms, but also identifies the frequently asked queries that can be quickly answered with partial computation. To the best of our knowledge, Aquila is the first work that provides such partial computation for connectivity queries. Third, although some techniques such as trim have been used in several works [23, 27, 45], they are limited to several connectivity problems, e.g., CC and SCC. In this paper, Aquila extends those techniques as well as designs new ones (e.g., single parent only in Section 4, multi-pivot sampling and concurrent BFS in Section 5) for additional connectivity algorithms, especially BiCC and BgCC.

Paper Organization. The rest of paper is organized as follows. Section 2 presents the background. Section 3 discusses the partial computation, and Section 4 describes workload reduction. Section 5 presents adaptive parallel computation. Section 6 evaluates Aquila. Section 7 discusses the related work, and Section 8 concludes.

2 BACKGROUND

We use $G(V, E)$ to denote a graph, where V denotes the set of vertices (nodes) and E is the set of edges.

2.1 Applications

The graph connectivity algorithms are widely used in many applications, where they not only serve as the fundamental steps for other graph algorithms, but also are key modules for many applications. Below, we will discuss five applications in three areas, namely, graph analytics, pattern recognition, and cybersecurity.

Graph Analytics. (1) Many graph algorithms, such as topological sort, and reachability query [12], require a directed acyclic graph (DAG). A regular directed graph is converted to DAG with the strongly connected component (SCC) algorithm by representing each SCC as a super node [52]. (2) Another frequently used graph algorithm, betweenness centrality (BC), measures the importance of each node by counting the number of the shortest paths. The state-of-the-art parallel solution divides the graph into multiple biconnected components (BiCCs) by identifying the articulation points, computes the BC for each BiCC, and merges the values [50]. This is motivated by the fact that a path crossing two BiCCs must pass the AP between them.

Pattern Recognition. (3) In computer vision, the connected component is used to label all the connected pixels as one object, which is known as the connected-component labeling [22].

Cybersecurity. (4) In spamming botnet detection, an effective method, BotGraph [53], constructs a user-to-user relationship graph to model the spamming attacks targeting the major web email providers. They find that the botnet controlled accounts usually form a large CC while the normal users form a number of small CCs. (5) The suspicious network activities can be mined from the DNS query graph [28], which is a directed graph. The SCC is used to identify the failed DNS graph, which is more likely to be malicious. Other security applications can be found in malware detection [25], malicious domain detection [51], and vulnerability detection [16].

2.2 Graph Connectivity Computation

There are two main parallel processing strategies for graph connectivity computation, as shown in Figure 3. First, **breadth-first search (BFS)**, due to its easy parallelism design, serves as the core technique for many existing graph connectivity computation methods [23, 44, 45]. BFS starts from a selected root vertex, a.k.a, *pivot vertex*, which is vertex 0 in Figure 3(a). Later, it constructs the BFS

tree where all the vertices belonging to the same CC are visited as shown in Figure 3(b). To find all the CCs, one needs to run BFS repeatedly till all the vertices are assigned to CCs as shown in Figure 3(c)(d)(e).

A recent work designs a direction-optimizing BFS, which switches between conventional top-down traversal and the newly designed bottom-up traversal [3]. The top-down traversal takes the visited vertices from the previous level as the frontier queue, inspects its neighbors, and marks the unvisited neighbor vertices as visited. Reversely, the bottom-up traversal takes the unvisited vertices as the frontier queue, inspects the neighbors, and terminates the inspection early if any neighbor is found to be visited from the previous level. The bottom-up method benefits from the early termination design, and shares different workload (frontier queue) with top-down. Thus, an efficient BFS design usually starts from top-down, switches to bottom-up, and switches back to top-down. The switch is dependent on the amount of the workload.

On the other hand, **label propagation** is another parallel algorithm for graph connectivity [44, 45]. Its frontier queue includes the vertices that are updated in the previous level. Initially, it assigns every vertex a label (vertex id) and all the vertices are in the frontier queue shown in Figure 3(f). Later, it scans the frontier queue and inspects the neighbors of each vertex. If the neighbor label is higher (or lower), it will propagate its label to the neighbor (or take the neighbor label to itself). The frontier queue is shrank to the newly updated vertices. Iteratively, the algorithm will stop if the frontier queue becomes empty. For the example, the first iteration is shown in Figure 3(g), and the frontier queue after the first iteration becomes {3–11, 13} as shown in Figure 3(h). After two more iterations, the propagation converges and the final result is shown in Figure 3(e).

3 QUERY TRANSFORMATION

Given a query, Aquila will analyze and classify it into a specific category, and make the appropriate computation strategy. Below, we will discuss the categories and computation strategies.

Complete computation is the conventional XCC computation strategy. The queries in this category include how many XCCs, what is the histogram, any query that does not fall into other categories. In this case, Aquila performs the complete computation with our workload reduction and adaptive parallel computation techniques.

Partial Computation of the Largest XCC. The largest XCC preserves the richest information, which makes it the most interesting XCC among all. For example, in the email user relationship graph, the security expert usually investigates the largest CC for potential botnet controlled malicious accounts, while the small CCs contain the normal users [53]. The queries targeting the largest XCC are classified into this category, including what is the largest CC, how big is the largest CC, and whether a specific vertex is in the largest XCC.

To identify the largest XCC, one needs to not only find it, but also make sure the others are smaller. Previous works turn to complete computation. Differently, Aquila firstly computes the largest XCC by heuristically selecting a vertex inside it based on vertex degree, which works for most real-world graphs [45]. To make sure it is the largest, we will compare its size with the remaining graph. If it

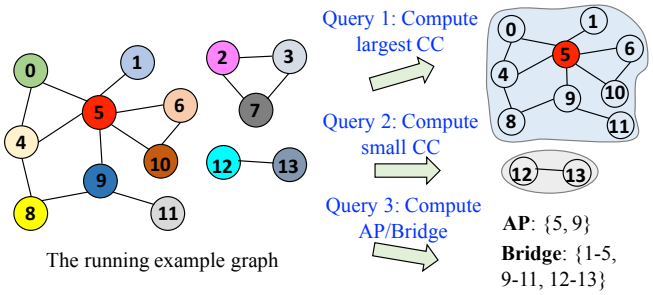


Figure 4: Examples of partial query transformation.

is smaller, Aquila will use the fast workload reduction techniques to identify the massive small XCCs. If the remaining graph is still greater, Aquila will do the complete computation. For the example graph in Figure 4, Aquila will find the largest CC by choosing vertex 5 as pivot and terminate the computation early as the found CC is greater than the remaining. In this example, Aquila saves the computation for vertices {2, 3, 7, 12, 13}.

Partial Computation of Small XCC. A typical query that can be answered with small XCC computation is whether a graph is connected. Such a query usually serves as the prerequisite checking to help make smart decisions for further computation. For example, the community detection algorithm only needs to work on a connected graph since the vertices in a community are densely connected internally than the rest of the graph [2]. Obviously, the vertices in different CCs should exist in different communities. One can use this query to decide whether they should further split the graph into CCs or directly compute the communities.

To answer such queries with existing graph systems, one needs to do a complete computation and later checks whether the number of XCCs equals to one [32, 37]. To simplify the computation, some methods would select an arbitrary pivot, find the XCC containing this pivot, and check whether the XCC size equals to the graph size [21]. Note that, the largest XCC contains the majority of the vertices, which makes such a strategy often compute the largest XCC. Differently, Aquila firstly checks whether any vertex can be trimmed and terminates when one is found. Otherwise, Aquila randomly selects a vertex and computes the XCC with our newly designed techniques. For the example graph in Figure 4, Aquila can quickly answer the query by trimming vertices {12, 13}.

Partial Computation of AP and Bridge. Besides BiCC and BgCC, the APs and bridges are of interests to many applications because they reveal the critical vertices and edges [50]. For the example graph in Figure 4, the APs are {5, 9}, and the bridges are {1-5, 9-11, 12-13}. They are "must" passing vertices and edges in that CC with any graph traversal methods, i.e., DFS, or BFS. In a computer network, the APs and bridges are usually critical as they can lead to the serious single point failures.

In the computation of BiCC, one needs to not only figure out what vertices are APs, but also recursively identify the BiCCs induced from the APs. The similar case happens for bridges. To answer such queries faster, we only compute the APs and bridges. Conventional methods are DFS-based, which is usually serial since DFS is hard to parallelize [38]. A recent work uses parallel BFS for AP,

while is limited to run too many BFSes. As DFS-based solution only needs to run DFS once with the workload equalling to the graph size, the parallel BFS solution may not be able to outperform it for all the graphs. In our test, the DFS solution is faster for 4 out of 11 tested graphs. Differently, we leverage our newly designed workload reduction techniques to remove a large amount of BFSes, up to 98%. For the remaining BFSes, we will leverage the adaptive computation strategy to fast compute them.

4 WORKLOAD REDUCTION

Our workload reduction strategy includes two techniques, single parent only (SPO) and trim.

Single parent only (SPO) is designed for BiCC and BgCC to significantly reduce the fairly large number of BFSes, which could be up to $|V|$. We will elaborate the BiCC computation process to explain why $|V|$ BFSes are needed.

The pseudocode of BiCC computation is shown in Algorithm 1. One builds a BFS tree firstly (line 1, 2). Later, for any vertex v in the reverse order on the BFS tree, one runs a constrained BFS by removing its parent vertex p (line 3–5). If it can not reach the same level of its parent vertex p, then p is an articulation point (AP) since at least the pivot vertex cannot reach v without p. If an AP is found, it will mark all the newly visited edges as one BiCC (line 6, 7).

Algorithm 1: bfsBiCC(G, *level, *parent)

```
1 pivot = selectPivot(G);
2 bfs(G, level, parent);
3 foreach v ∈ reverseBfsOrder(G) do
4     p = parent[v];
5     l = bfsConstrained(G, v, p, level);
6     if l < level[p] then
7         Mark the visited edges as one BiCC;
```

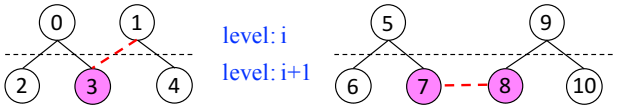
As one needs to run up to $|V|$ BFSes, we reduce the workload with the single parent only technique, which is based on the following two lemmas.

Lemma 1. *On the constructed BFS tree, for any non-root vertex p, after removing vertex p, if any of its children v cannot reach a vertex at the same level of p, then p is an articulation point. Similarly, after removing edge <p, v>, if v cannot reach a vertex at the level of p, then edge <p, v> is a bridge.*

Proof. Since a child vertex v cannot reach a vertex at the same level of p after removing p, that means, v cannot reach the root vertex. Thus, removing vertex p would at least disconnect the root vertex and v, which makes vertex p an articulation point. Similarly, we can prove the bridge. □

Lemma 2. *For any non-root vertex p, after removing vertex p, if a child v can reach a vertex at the same level of p, then p is not an articulation point from the view of v. Not checking vertex v will not affect the correctness. Similarly for the bridge.*

Proof. Let the reached vertex sharing the same level of p be q, since v is able to reach q, that means, v is able to reach the root vertex which is then able to reach all the other vertices except the children of p. Thus, from vertex v, one cannot tell whether p is an AP or not. Further, assume there is one child u that cannot be reached by v, then u cannot reach the root vertex, which means u cannot reach any vertex at the same level of p. According to Lemma 1, one can tell p is an AP from the view of u. Thus, not checking vertex v will not affect the correctness of finding all the APs. □



(a) Direct second parent (b) Sibling induced second parent

Figure 5: Two types of vertices that have second parent.

From Lemma 2, one can see that checking vertex like v would not find any AP. Motivated, our SPO technique is designed to quickly remove such vertices. For a vertex v, its second parent here represents a vertex at the same level of its parent. In the implementation, we save the space by simply recording whether a vertex has a single parent, instead of recording its second parent index. We identify the single parent vertices after the BFS tree construction instead of mixing them together, since doing so would cancel out the early termination benefits in the bottom-up traversal.

Currently, our SPO technique identifies two types of second parent, direct second parent and sibling induced second parent, as shown in Figure 5. A vertex has a direct second parent if one of its neighbors (not parent vertex) has the same level with its parent. An example is shown in Figure 5(a), the parent of vertex 3 is 0. Assume there is an edge between vertex 3 and 1 in the original graph, and vertex 1 shares the same level with 0, then vertex 3 actually has a second parent. We can prune vertex 3.

On the other hand, vertex has a sibling induced second parent if one of its neighbors has the same level but different parent. An example is shown in Figure 5(b). Assume there is an edge between 7 and 8, as they have different parents, this makes both vertices have second parent. One can find other types of vertices that have a second parent, for example, after finding a vertex with direct second parent as shown in Figure 5, if there is an edge between 2 and 3, then vertex 2 also has a second parent. However, we have tested these types and found that the cost of computing more complex types overweights the benefit.

Our SPO technique is able to reduce 74% and 77% workload on average for BiCC and BgCC, respectively, as shown in Figure 6. The graph benchmarks are presented in Section 6. Combining together with trim (discussed next), our workload reduction strategy is able to reduce 95% and 98% workload on average for BiCC and BgCC, respectively. We also estimate the reduction upper bound, which is defined as the number of BFSes that will not find an articulation point or bridge. One can see that, our workload reduction strategy gets close to the upper bound, which is 99.6% and 99.7% for BiCC and BgCC, respectively, as shown in Figure 6.

Trim is designed to quickly remove the trivial XCCs motivated by the fact that a large number of trivial XCCs exist in real-world

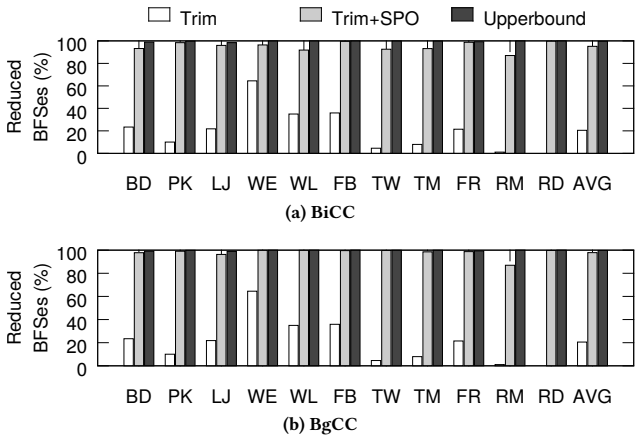


Figure 6: The percentage of reduced BFSes for (a) BiCC and (b) BgCC.

graphs. Trim is shown to be able to greatly reduce the workload for CC and SCC [23, 45]. In this work, we further design new trim techniques for BiCC and BgCC. Together, the trimmed subgraph patterns are shown in Figure 7.

The subgraph pattern shown in Figure 7(a) is an orphan vertex, who does not have any connections with other vertices. For CC and WCC, we further trim a size-2 pattern as shown in Figure 7(b), where two vertices are connected by one edge without connecting to any other vertices. For the directed WCC, the edge can be in either direction, or there can be two edges in each direction. For SCC, we trim size-1 pattern like vertex 3 shown in Figure 7(c). Such vertex has either zero indegree or zero outdegree, thus it can not involve in any other SCCs. Further, we trim size-2 SCC as shown by vertex 4 and 5. In such pattern, the two vertices mutually point to each other and their other edges are either all going out or coming in. In this way, one can make sure that they will not be involved in other SCCs. For BiCC and BgCC, we further trim the patterns shown in Figure 7(d). Vertex 7 only connects to vertex 6, which makes this edge a bridge because vertex 7 cannot connect to other vertices without this edge, and also makes vertex 7 a BgCC. If vertex 6 has other edges, it will become an articulation point and the subgraph involving vertex 6 and 7 is a BiCC because removing vertex 6 will disconnect vertex 7 and others. If vertex 6 only has one edge, it will not be an AP but the subgraph is still a BiCC.

On average, trim is able to reduce 21% workload for both BiCC and BgCC, as shown in Figure 6.

5 ADAPTIVE PARALLEL COMPUTATION

This section will discuss our adaptive parallel computation, including irregular tasks for connectivity, adaptive parallel strategy, and parallel traversal.

5.1 Irregular Tasks for Connectivity

We observe that different connected components share the same irregular task property for real-world graphs. That is, a few large XCCs take the majority of the graph whose size is close to the order of graph size, and the rest is a large number of trivial XCCs whose

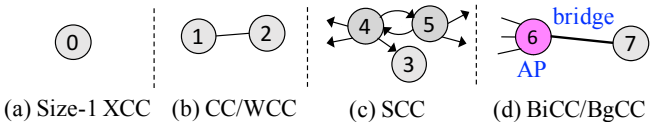


Figure 7: Trim patterns for connectivity algorithms, (a) is for all XCC, (b) is for (W)CC, (c) is for SCC, and (d) is for BiCC and BgCC.

count is close to the order of graph size as well. Figure 8 shows such irregular property for two representative graphs, a social network graph — Twitter graph (TwitterMpi) with 53 million vertices and 3.2 billion edges, and a web graph — Wikipedia graph (WikiLinkEn) with 11 million vertices and 517 million edges [31]. The graphs will also be used in Section 6. One can see that both graphs show the irregular task property for connectivity algorithms. Taking the BgCC as an example, one can see that, the Wikipedia graph has a single large BgCC with 3.6M vertices accounting for 32% of the graph size, and two BgCCs in one order of magnitude smaller, and the rest are two orders of magnitude smaller. Especially for the size-1 BgCCs, its count accounts for 93% of the total BgCCs.

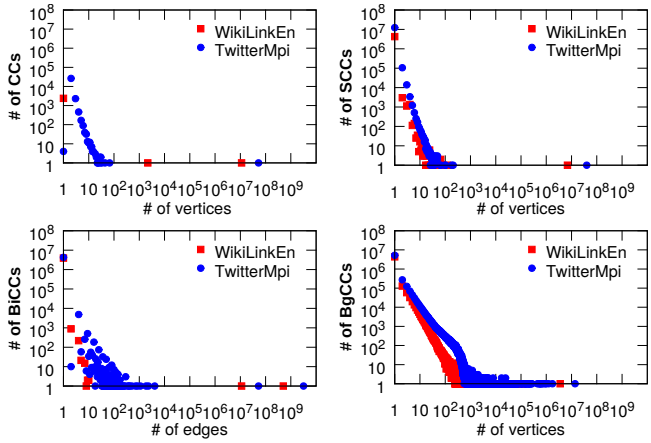


Figure 8: The number of XCCs against their sizes, in the order of (W)CC, SCC, BiCC, and BgCC, respectively. BiCC size is measured by edge count, and the others are measured by vertex count.

5.2 Adaptive Parallel Strategy

Motivated by the irregular task property, we design an adaptive parallel strategy. As shown in Figure 9, we firstly classify the task into large and small tasks. For the large task, we apply the data parallel strategy, that is, the enhanced parallel BFS. For the small tasks, we apply the task parallel strategy, that is, the label propagation technique for CC and SCC, and the concurrent BFS for BiCC and BgCC. In the following, we will discuss the details of this adaptive design.

Data vs. Task Parallel. The data parallel method utilizes all the computation resources to compute one task at a time, while the task parallel method computes a number of tasks concurrently.

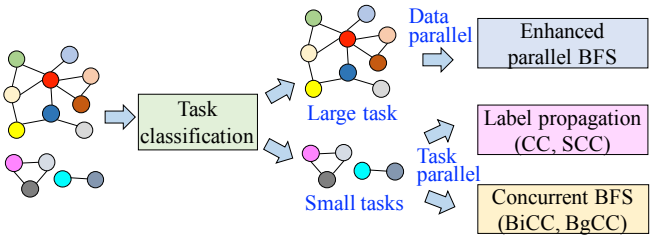


Figure 9: Overview of our adaptive parallel strategy.

Motivated by the irregular task property in XCCs, we apply the data parallel methods for the large tasks, and the task parallel methods for the large number of small tasks. Particularly, there is usually one large task for CC and SCC, while may be several for BiCC and BgCC. But still, the largest one in BiCC and BgCC is several orders of magnitude greater than others.

BFS vs. Label Propagation. Parallel BFS method shows the advantage of quickly finding one specific XCC. However, it is low efficient for finding many XCCs as it needs to run one BFS for each XCC and most XCCs are small which will result in the small frontier queue (often less than the number of available hardware threads) such that most hardware threads are wasted. Differently, label propagation has the advantage of fully utilizing all the hardware threads, and is able to identify all the XCCs by running only once. However, it faces the inflated workload challenge where a vertex may appear in multiple frontier queues. For the example graph in Figure 3(f), vertices {6, 8, 9, 10, 11} appear in all the three frontier queues. That means, one has to inspect their neighbors three times. Note that, the BFS (conventional top-down) inspects each vertex once. Such a limitation greatly lowers the performance for the large XCC. To this end, BFS works better for data parallel, and label propagation fits for task parallel.

Concurrent BFS can also be used for task parallel if one can assign a separate BFS to each thread. However, it faces the challenge of how to select a number of pivots from different tasks so that no duplicate works are performed. It is difficult for CC and SCC, while relatively easy for BiCC and BgCC since one needs to take almost every vertex as a pivot as shown in Algorithm 1. As BFS has less workload than label propagation, we apply the BFS-based task parallel method for BiCC and BgCC.

5.3 Enhancement to BFS

After applying the previous optimizations, the BFS-based data parallel method for the few large tasks take the majority of the runtime. We further improve the performance by optimizing the parallel BFS for the large tasks.

We observe that the graph connectivity computation only requires the connectivity information, where the correct BFS traversals are not required. Particularly, one does not have to run the BFS from a single pivot and synchronize after each iteration. Motivated, we design a fast parallel traversal method on top of the parallel BFS with two techniques, multi-pivot sampling and adaptive synchronization.

Table 1: Graph benchmarks (Dir, Und are short for directed and undirected).

Graph	Abbr.	# Nodes	# Dir Edges	# Und Edges	# CCs	Largest CC Percentage
Baidu	BD	2.1M	17.8M	34.0M	15,561	98.4%
Pokec	PK	1.6M	30.6M	44.6M	1	100%
Livejournal	LJ	4.8M	68.5M	85.7M	1,876	99.9%
WikiEn	WE	18.3M	172.2M	253.8M	1,366	99.9%
WikiLinkEn	WL	11.2M	340.3M	516.9M	3,061	99.9%
Facebook	FB	96.1M	679.7M	1.2B	5	99.9%
TwitterWww	TW	41.7M	1.5B	2.4B	1	100%
TwitterMpi	TM	52.6M	2.0B	3.2B	29,533	99.9%
Friendster	FR	68.3M	2.6B	3.6B	323,276	98.7%
RMAT	RM	4M	256M	506.2M	1.9M	52.1%
Random	RD	4M	256M	512.0M	1	100%

Multi-pivot sampling will sample multiple pivots so that our graph traversal will run from multiple roots instead of only one. Multi-pivot sampling faces two challenges, one is that all the pivots should belong to the same task, and the other is that this task should be the single large one. We solve them by firstly selecting a master pivot, which is the vertex with highest degree (vertex 5 in Figure 3(a)). Such a vertex is shown to be always in the single large task from our test and recent works for both real-world and synthetic graphs [27, 45]. Later, we sample a number of interesting pivots that are connected with the master pivot. In this way, we are able to sample multiple pivots which belong to the same large task. Currently, we sample the neighbors of the pivot vertex with the number equalling to the available hardware threads. Such sampling is an online process and counted as part of the enhanced BFS.

Adaptive Synchronization. Starting from the selected multi-pivots, Aquila traverses the graph by adaptively switching between the conventional synchronization (Sync), asynchronization (Async), and a recently designed relaxed synchronization (Rsync) from [27]. At the same time, it switches between the top-down and bottom-up traversal models. Such a design is proved to not only reduce the workload but also reduce the number of synchronizations. Specifically, Aquila starts from Sync top-down, switches to Rsync bottom-up, and switches back to Async top-down. For the connectivity computation where the correct BFS levels are not required, we apply such adaptive synchronization strategies.

6 EXPERIMENT

The experiments are performed on a server with two Intel Xeon Gold 6126 CPUs, each has 12 cores. The server runs CentOS 7.6 with hyper-threading enabled. All the results are reported with an average of ten runs.

6.1 Graph Benchmarks

We test Aquila on 11 graphs, including 9 real-world and 2 synthetic graphs shown in Table 1. The real-world graphs are collected from the KONECT [31] and SNAP project [33]. There are two popular graph types, social network and web graph. The synthetic graphs are generated from two popular graph generators, R-MAT [10]

Table 2: Runtime (ms) of Aquila and compared works. The hyphen denotes the test cannot complete. The best performance for each test is highlighted. The real-world graphs are ordered by edge number from left to right.

Alg	Method	BD	PK	LJ	WE	WL	FB	TW	TM	FR	RM	RD	Avg. speedup
CC	Boost	951	1,220	2,536	7,478	11,701	93,937	100,383	145,275	231,247	10,253	11,157	359.3
	DFS	232	265	666	2,054	3,016	11,236	17,992	23,873	44,611	1,717	2,492	67.1
	X-Stream	630	1,259	2,181	64,858	1,115,409	110,845	76,433	28,010	110,845	4,373	7,339	1,547.8
	Galois_Async	1,667	431	1,503	14,296	15,292	124,871	53,653	227,345	46,240	13,841	2,851	325.2
	Galois_LP	207	203	470	1,525	3,262	8,949	11,117	16,672	20,016	1,875	2,325	52.6
	GraphChi_LP	8,967	4,730	16,004	75,236	121,280	364,772	402,086	674,599	-	59,610	85,243	12,773.0
	GraphChi_UF	36	31	98	409	224	1,823	754	976	-	51	57	31.4
	Ligra_LP	420	619	1,006	5,035	11,670	34,690	66,560	79,700	139,600	12,550	7,764	264.0
	Ligra_SC	358	475	868	4,240	11,400	33,670	80,270	113,500	172,900	12,040	6,850	281.3
	Multistep	41	131	233	333	151	1,001	439	571	1,007	232	116	5.9
	Aquila	14	9	27	95	79	285	156	222	410	14	21	
SCC	Boost	617	1,635	4,121	13,395	14,694	26,562	112,058	124,197	240,553	8,972	17,639	182.7
	DFS	542	600	1,457	4,794	7,463	23,940	40,365	49,675	84,901	2,930	5,624	82.8
	X-Stream	818	2,215	4,447	118,303	1,138,397	28,900	179,068	189,875	164,021	8,822	14,011	1,191.0
	GraphChi	225	242	330	1,513	-	-	-	-	-	1,358	1,839	918.0
	Multistep	166	87	138	302	270	429	700	2,207	4,447	142	114	3.6
	Hong	49	46	121	228	352	1,171	1,670	1,618	-	137	112	3.7
	iSpan	22	28	54	120	113	89	603	572	1148	44	48	1.1
	Aquila	24	29	50	105	106	85	579	558	957	39	43	
BiCC	Boost	4,854	6,696	13,232	40,112	80,455	260,858	527,819	763,802	761,041	68,739	91,957	222.9
	DFS	631	719	1,751	5,753	8,714	28,121	45,049	54,464	91,495	4,376	7,453	22.4
	Slota_LP	555	190	681	7,060	9,136	68,323	44,576	114,529	65,480	4,801	-	22.7
	Slota_BFS	1,378	258	621	9,982	13,177	24,942	44,790	76,658	21,962	4,278	2,555	20.7
	Aquila	43	25	129	222	346	3,683	1,324	4,808	1,697	168	1,304	
BgCC	DFS	536	607	1,501	5,145	8,100	26,691	44,153	54,242	92,980	3,041	5,484	9.4
	Aquila	195	42	145	315	471	8,096	10,383	5,641	5,880	499	1,969	

and GTgraph [1]. The graphs are stored in the commonly used compressed sparse row (CSR) format [6]. It includes two arrays, one is the begin position array with the length of $|V| + 1$, the other is the adjacent list array with the length of $|E|$.

Initially, all the graphs are directed. As CC, BiCC, and BgCC work on the undirected graphs, we convert the directed graphs to undirected. Specifically, we generate the undirected graph by creating a reversely directed edge for any two vertices that share only one directed edge. After the conversion, the undirected graph shares the same vertex number with the directed one, but less than twice the edges. The details can be found in Table 1.

6.2 Implementation

We implement Aquila with over 5,000 lines of C++ code by supporting five most popular graph connectivity algorithms, WCC, SCC, CC, BiCC, and BgCC. Aquila is compiled by GCC (v4.8.5) with *O3* optimization level. We use OpenMP (v3.1) as the multi-threading library. As WCC and CC share the same computation process with the only difference in edge direction, we regard CC to represent both of them. For CC and SCC, Aquila starts with trim, computes the largest one with the optimized parallel BFS, and applies the label propagation for the remaining ones. For BiCC and BgCC, Aquila applies trim, builds a BFS tree for the largest CC, reduces workload with single parent only technique. Next, Aquila runs concurrent BFS for the vertices at levels greater than one, and optimized parallel BFS for the vertices at levels zero and one.

6.3 Compared Works

We compare Aquila with ten related works shown in Table 2. Among them, Multistep is the state-of-the-art method for CC [45], iSpan achieves the state-of-the-art performance for SCC [27], and Slota provides the BiCC implementations with either label propagation or BFS [44]. We also compare with four popular graph computation systems, Galois [37], X-Stream [39], GraphChi [32], and Ligra [42]. Further, we compare with one optimized SCC computation method, Hong [23]. In addition, we compare with two serial implementations, the classical DFS-based and the boost graph library [43]. The compared parallel works are running on our server with the same number of threads. i.e., 48, except for X-Stream with 32 threads because it requires the thread number to be the power of two. Note that, we only report the runtime of compared works and ours without any pre- or post-processing, i.e., graph loading, graph conversion, or writing to disk.

6.4 Performance of Complete Computation

In this section, we will compare the performance of Aquila with recent works in terms of complete computation. From Table 2, we can observe that Aquila is able to outperform all other graph processing frameworks. Of all the connectivity algorithms, Aquila is able to significantly improve the performance on average by 1.1×, 3.7×, 13×, 21×, 53×, 264×, 364×, 1, 369×, 45×, and 255× compared to iSpan, Hong's SCC, Multistep, Slota's BiCC, Galois, Ligra, GraphChi, X-Stream, DFS, and Boost, respectively. For each framework, the speedup number is calculated by averaging its method with the

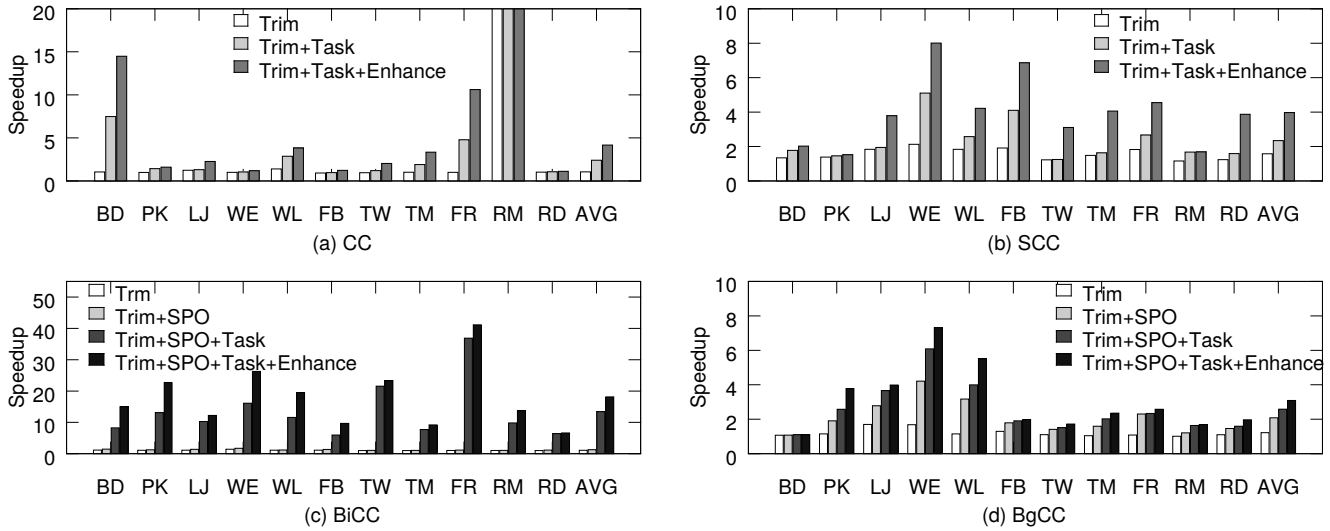


Figure 10: Benefits of the newly designed techniques for (a) CC, (b) SCC, (c) BiCC, and (d) BgCC.

best performance among all supported connectivity algorithms. For example, the speedup for GraphChi is calculated with its union find solution (GraphChi_UF) for CC and the method for SCC.

For CC computation, Aquila outperforms Multistep, Galois_LP, Galois_Async, GraphChi_UF, GraphChi_LP, Ligra_LP, Ligra_SC, X-Stream, DFS, and Boost by 5.9×, 53×, 325×, 31×, 12,773×, 264×, 281×, 1,548×, 67×, and 359× speedup, respectively. We compare with two fastest methods from Galois, the label propagation based Galois_LP, and asynchronous union find based Galois_Async. For GraphChi, we compare with its implementations using label propagation (GraphChi_LP) and union find (GraphChi_UF). For Ligra, we compare with its implementations using label propagation (Ligra_LP) and short-cut label propagation (Ligra_SC), which is an optimized label propagation method from [46].

The speedup comes from both our workload reduction and adaptive task computation strategies. Interestingly, for the graphs PK, TW, and RD, which only have one CC, the speedup shows the effectiveness of our enhanced parallel BFS for the large task. Also, the best union find-based solution, i.e., GraphChi_UF, performs better than the best label propagation solution, i.e., Multistep, for small graphs (BD, PK, LJ, RM, RD), while performs worse for large graphs (from WE to FR). The reason is that the atomic operations in union find solutions are costly as the number of conflicts increases when graph becomes larger.

For SCC, Aquila achieves 1.1×, 3.7×, 3.6×, 918×, 1,191×, 83×, and 183× speedup over iSpan, Hong, Multistep, GraphChi, X-Stream, DFS, and Boost, respectively. We observe that iSpan achieves better performance for graphs BD and PK. The graph BD has a largest SCC accounting for only 28% vertices and most of the rest are small SCCs (≤ 40). Although Aquila is able to outperform iSpan for the largest SCC computation, iSpan benefits from its optimized trim design for the small SCCs. Although the graph PK has 80% vertices in the largest SCC, the graph itself is small which offsets the benefits of our adaptive computation strategies.

The two parallel graph computation frameworks, X-Stream and GraphChi are not fast enough because they only apply the forward-backward algorithms without any other techniques, e.g., trim. Such an implementation is not efficient towards a graph with many SCCs, especially when there are a large number of trimmable SCCs.

For BiCC, Aquila achieves more than 20× speedup over Slota, DFS, and Boost. We compare with two implementations from Slota, label propagation-, and BFS-based. Although they share similar average performance, one can see that the label propagation method outperforms BFS for the small graph and BFS is better for large ones. We also observe that Slota's implementation gets similar performance with the classical DFS-based solution due to its heavy workload, up to $|V|$ BFSes. Aquila avoids such drawback by greatly reducing the workload with our newly designed workload reduction technique.

For BgCC, we only compare with DFS as we are not able to find any available parallel implementations. Aquila achieves 9.4× speedup over DFS. We notice that Aquila achieves better performance especially for the large graphs. For the largest graph, FR, which has 3.6 billion edges, Aquila can get all the BgCCs in 6 seconds, while the DFS solution needs 93 seconds.

6.5 Technique Benefits for Complete Computation

Figure 10 presents the speedup of using our newly designed techniques over the baseline method, which is parallel BFS-based for CC, SCC, BiCC, and BgCC. For the workload reduction techniques, trim brings 1.2× speedup, and single parent only (SPO) brings 1.4× speedup on average. Particularly, trim brings 1.1×, 1.6×, 1.1×, and 1.2× speedup for CC, SCC, BiCC, and BgCC, respectively. Interestingly, trim achieves up to 1,318× speedup for CC computation on RM graph. RM has up to 1.9 million trivial CCs which will greatly benefit from the trim technique. We omit this extreme speedup in the average speedup calculation. The SPO technique is able to bring another 1.2× and 1.9× speedup for BiCC and BgCC, respectively.

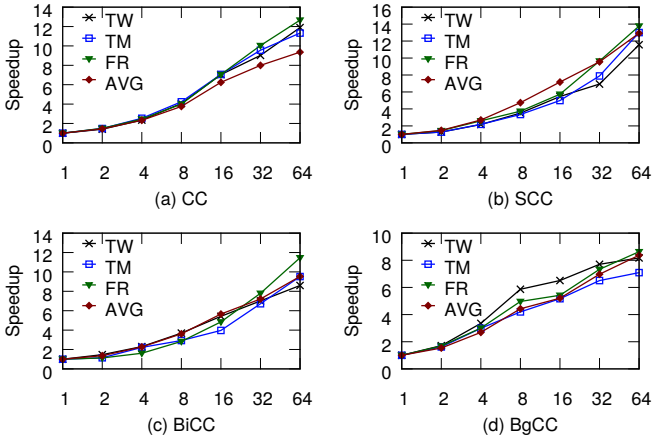


Figure 11: The scalability of Aquila against thread count (shown in x-axis) for the three largest graphs (TW, TM, FR) and the average of all the eleven graphs. (a) CC, (b) SCC, (c) BiCC, and (d) BgCC.

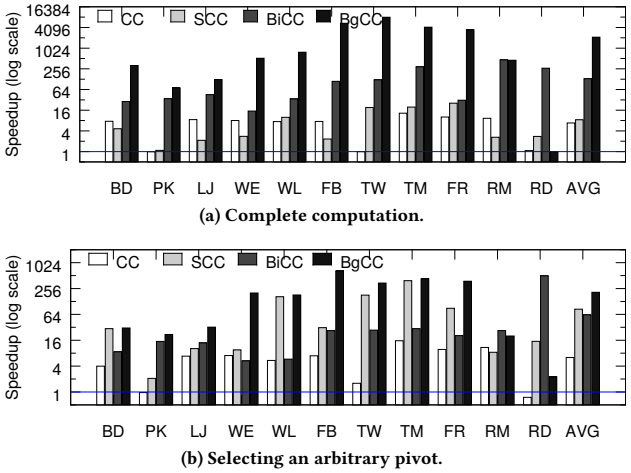


Figure 12: Speedup of small XCC computation over (a) complete computation, and (b) selecting an arbitrary pivot.

Workload reduction achieves the highest speedup, 4.2×, for BgCC computation on graph WE. For this graph, the two techniques are able to reduce the workload (number of BFSes) by 99.9%, which is the highest among all.

For the adaptive parallel computation techniques, the adaptive parallel strategy achieves 2.3×, 1.8×, 13.2×, and 1.5× speedup for CC, SCC, BiCC, and BgCC, respectively. The enhanced parallel BFS technique brings 2.8×, 2.6×, 5.7×, and 1.5× speedup for CC, SCC, BiCC, and BgCC, respectively. Together, the two adaptive parallel techniques bring 6.7× speedup on average. Particularly, they bring up to 25× speedup for BiCC computation on graph WE because it shows obvious irregular task distribution.

6.6 Scalability

This sections studies the scalability of Aquila against different number of running threads. We present the scalability results for the three largest graphs (TW, TM, FR) and the average of all the eleven graphs in Figure 11. One can see that, Aquila shows stable scalability when increasing the thread count for both large graphs and on average. Compared with the single thread implementation, Aquila with 64 threads is able to get 9.4×, 12.9×, 9.5×, and 8.4× speedup on average for CC, SCC, BiCC, and BgCC, respectively. For the largest graph FR, Aquila is able to achieve 12.7×, 13.8×, 11.5×, and 8.6× speedup for CC, SCC, BiCC, and BgCC, respectively.

6.7 Performance of Partial Computation Queries

This section presents the performance of computing the queries that can be answered with partial computation in terms of small XCC, largest XCC, AP, and Bridge.

Small XCC. For the queries in this category, we compare our strategy with two commonly used strategies in current works, i.e., complete computation and selecting an arbitrary pivot. We use Aquila as the complete computation baseline, which outperforms state-of-the-art works as shown previously. We implement the selecting an arbitrary pivot strategy with the optimized techniques from Aquila by randomly selecting a pivot, finding the XCC induced from this pivot, and terminating early when one is found.

The speedup of our strategy over complete computation is shown in Figure 12(a). On average, our strategy is able to get 6.8×, 8.4×, 132×, and 2, 137× speedup for CC, SCC, BiCC, and BgCC, respectively. Our strategy significantly improves the performance for most graphs, and gets similar performance with complete computation even for the worst case graphs. For CC, our performance is similar with complete computation on graphs PK, TW, and RD as they only have one CC. For BgCC, the synthetic graph RD has only one BgCC, which makes our performance similar with complete computation.

The speedup of our strategy over selecting an arbitrary pivot is shown in Figure 12(b). On average, ours is able to get 6.3×, 84.6×, 62.4×, and 209× speedup for CC, SCC, BiCC, and BgCC, respectively. We observe that most arbitrarily selected pivots end up with running in the largest XCC. Particularly, all the 110 pivots for CC fall in the largest CC. For BiCC and BgCC, all the pivots fall in the largest CC. Although they usually terminate early before computing the largest BiCC or BgCC, they still have to build the BFS tree for the largest CC which lowers the performance. For SCC, 65% pivots fall in the largest one. However, the complete computation is faster because trim helps to remove 55% vertices before running the two parallel BFSes. Interestingly, for the synthetic graph RD with only one CC, the arbitrarily selected pivots get better performance for CC benefited from the BFS traversals starting from different root vertices.

Largest XCC. The speedup of finding the largest XCC over Aquila's complete computation is shown in Figure 13. Our strategy achieves 1.2×, 1.1×, 1.03×, and 1.1× speedup over the baseline. For CC and SCC computation, we are able to find the largest XCC after trimming and computing the largest XCC. Thus, we benefit from eliminating the small XCC computation. Differently for BiCC and BgCC, our solution firstly computes the small XCCs due to the checking order is from the highest BFS level to the lowest where

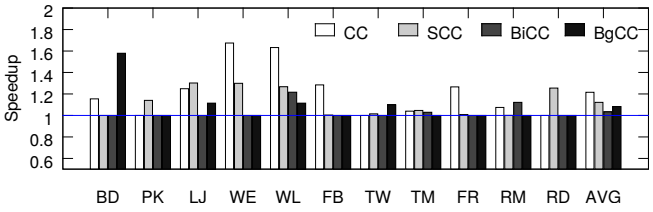


Figure 13: Speedup of largest XCC over complete computation of Aquila.

the largest XCC is usually found at the zero or first level. Simply reversing the checking order would result in wrong XCCs since the largest one will involve the small ones. However, we still achieve 1.03× and 1.1× speedup for BiCC and BgCC by eliminating some traversals at the zero and first level.

AP and Bridge. For AP only computation, besides comparing with Aquila's complete computation, we also compare with the AP only computation method from Slota, DFS, and Boost. As shown in Figure 14(a), on average, Aquila achieves 2.5×, 2.3×, 50×, 43×, and 423× speedup over Aquila's complete computation, Slota's BFS, Slota's label propagation, DFS, and Boost, respectively. The benefit is mainly from workload reduction because one does not need to check an already identified AP vertex which is different in BiCC computation. For bridge only computation, we are able to get 12× and 1.3× speedup over DFS and Aquila's BgCC computation as shown in Figure 14(b).

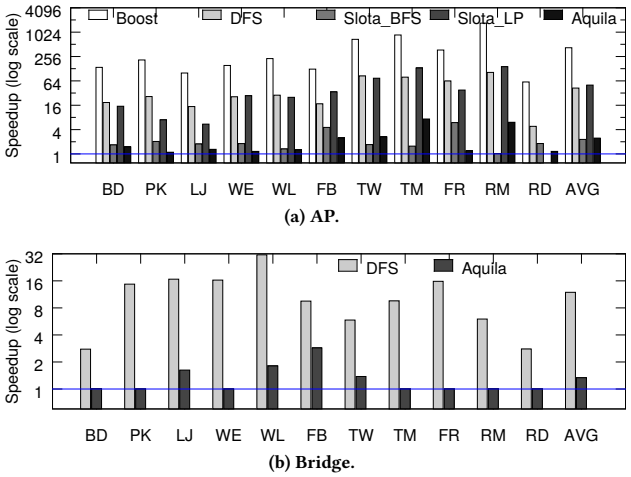


Figure 14: Speedup of AP (a) and bridge (b) computation.

7 RELATED WORK

In this section, we will discuss the works related to Aquila in terms of computing CC, SCC, BiCC, and BgCC.

The study of connected component (CC) starts from a serial DFS-based solution by Hopcroft and Tarjan [24]. It has linear time complexity and is used in several graph libraries as the serial solution, including Boost [43], and Galois [29]. Shiloach et al. proposed the first parallel CC algorithm [41]. It uses the union-find data structure with hooking and pointer jumping, where hooking is the union operation and the pointer jumping is a path compression technique to reduce the length of the path. The Galois graph library provides up to seven parallel CPU implementations, including both union-find based and label propagation based [29]. Ligra [42] has two implementations, label propagation and parallel BFS based. GraphChi [32] implements two parallel CC algorithm, label propagation and union find based. Later, Multistep fuses the parallel BFS, label propagation, and serial DFS-based Tarjan algorithm for shared-memory computation with a significant improvement [45].

Strongly connected component (SCC) is firstly studied by Tarjan with DFS-based solution [47]. Although a recent DFS-based work gets good performance for on-the-fly SCC detection [5], most parallel algorithms choose BFS for better parallelism [15]. Fleischer et al. first propose the parallel forward backward (FW-BW) algorithm [18]. Starting from a pivot vertex, FW-BW algorithms perform two BFSes, one with out edge and the other with in edge. The vertices covered in both BFSes belong to the SCC induced from the pivot vertex. Later, Mclendon et al. propose the trim technique motivated by the large number of size-1 SCCs in a graph [36]. Recently, Hong designs the trim-2 technique to fast trim size-2 SCCes and applies WCC-guided FW-BW algorithm for the small SCCs [23]. Slota et al. design the Multistep method, which applies trim, FW-BW algorithm, label propagation, and serial Tarjan's algorithm [45]. iSpan designs a fast spanning tree-based method with a relaxed synchronization technique for the FW-BW algorithm [27].

Initially, biconnected component (BiCC) and bridgeless connected component (BgCC) are also computed with DFS-based serial solution [24]. Later, Tarjan and Vishkin design a parallel solution by constructing an auxiliary graph [48]. Further, Cong and Bader add a preprocessing step to improve the performance by greatly reducing the size of the auxiliary graph [13]. Edwards and Vishkin extend such algorithm to explicit multi-threading many core platform [15]. The SCC-based solution is firstly introduced by Eckstein [14] and extended to parallel by Tsin [49]. Another direction of computing BiCC is open ear decomposition-based [35].

8 DISCUSSION AND CONCLUSION

For real-world graphs, the adaptive strategies, i.e., partial computation, workload reduction, and adaptive parallel computation, are shown to be effective. It is possible that some of the techniques may not bring obvious benefits under special cases. Particularly, on some graphs, the partial computation strategy for some specific connectivity algorithms may not bring obvious speedup, e.g., the largest Aquila in a large graph (FR). If a graph does not have trimmable patterns, the workload reduction strategy will not work. However, the real-world graphs usually have such patterns. Also, some trimmable patterns (the SPO patterns) usually exist as it is impossible to build a graph that every node is an articulation point.

Currently, Aquila implements five connectivity algorithms which are the most popular among all the connectivity algorithms. Not limited, one can extend our techniques to other connectivity algorithms, such as k-vertex connectivity and k-edge connectivity ($k > 2$). With the optimized connectivity computation, one can also improve the performance of related applications, such as betweenness centrality computation, and reachability query computation.

This work designs Aquila, an adaptive computation framework for answering queries on graph connectivity. Particularly, given a graph and the query, Aquila analyzes the query and determines a fast computation strategy. Aquila further simplifies the workload by reducing unnecessary tasks with our newly designed single parent only and trim techniques. Later, Aquila improves the performance with our adaptive parallel computation techniques. The evaluation on eleven graphs shows that Aquila significantly improves the performance on average by 13×, 53×, 264×, 364×, 1, 369×, 45×, and 255× compared to Multistep, Galois, Ligra, GraphChi, X-Stream, DFS, and Boost, respectively. Specifically, Aquila is able to outperform the state-of-the-art methods on average by 5.9×, 1.1×, 20.7×, and 9.4×, for (W)CC, SCC, BiCC, and BgCC, respectively.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous HPDC'20 reviewers for their valuable suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

REFERENCES

[1] David A Bader and Kamesh Madduri. 2006. Gtgraph: A synthetic graph generator suite. (2006).
[2] Seung-Hee Bae and Bill Howe. 2015. GossipMap: A distributed community detection algorithm for billion-edge directed graphs. *Proceedings of SC* (2015).
[3] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. *Proceedings of SC* (2012).
[4] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. *Proceedings of SIGMOD* (2019).
[5] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. 2016. Multi-core on-the-fly SCC decomposition. *Proceedings of PPoPP* (2016).
[6] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. *Proceedings of SPAA* (2009).
[7] Kenneth L Calvert, Matthew B Doar, and Ellen W Zegura. 1997. Modeling internet topology. *IEEE Communications magazine* (1997).
[8] Jian Cao, Qiang Li, Yuede Ji, Yukun He, and Dong Guo. 2016. Detection of forwarding-based malicious URLs in online social networks. *International Journal of Parallel Programming* (2016).
[9] Lili Cao and John Krumm. 2009. From GPS traces to a routable road map. *Proceedings of ACM SIGSPATIAL* (2009).
[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. *Proceedings of ICDM* (2004).
[11] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. *Proceedings of EuroSys* (2015).
[12] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. *Proceedings of SIGMOD* (2013).
[13] Guojing Cong and David A Bader. 2005. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). *Proceedings of IPDPS* (2005).
[14] DM Eckstein. 1979. BFS and biconnectivity. *Technical Report 79-11* (1979).
[15] James A Edwards and Uzi Vishkin. 2012. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. *Proceedings of International Workshop on PMAM* (2012).
[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. *Proceedings of NDSS* (2016).
[17] Shimon Even and R Endre Tarjan. 1975. Network flow and testing graph connectivity. *SIAM journal on computing* (1975).
[18] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On identifying strongly connected components in parallel. *Proceedings of IPDPS* (2000).
[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. *Proceedings of OSDI* (2012).
[20] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. *Proceedings of OSDI* (2014).
[21] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. Exploring network structure, dynamics, and function using NetworkX. (2008).
[22] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. 2009. Fast connected-component labeling. *Pattern recognition* (2009).
[23] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. *Proceedings of SC* (2013).
[24] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* (1973).
[25] Yuede Ji, Benjamin Bowman, and H Howie Huang. 2019. Securing malware cognitive systems against adversarial attacks. *Proceedings of ICCC* (2019).
[26] Yuede Ji, Yukun He, Xinyang Jiang, Jian Cao, and Qiang Li. 2016. Combating the evasion mechanisms of social bots. *Computers & Security* (2016).
[27] Yuede Ji, Hang Liu, and H Howie Huang. 2018. iSpan: Parallel Identification of Strongly Connected Components with Spanning Trees. *Proceedings of SC* (2018).
[28] Nan Jiang et al. 2010. Identifying suspicious activities through dns failure graph analysis. *Proceedings of ICNP* (2010).
[29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. 2007. Optimistic parallelism requires abstractions. *Proceedings of PPoPP* (2007).
[30] Pradeep Kumar and H Howie Huang. 2019. GraphOne: A data store for real-time analytics on evolving graphs. *Proceedings of FAST* (2019).
[31] Jérôme Kunegis. 2013. Konect: the koblenz network collection. *Proceedings of the International Conference on World Wide Web* (2013).
[32] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. *Proceedings of OSDI* (2012).
[33] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (2014).
[34] Hang Liu and H Howie Huang. 2015. Enterprise: breadth-first graph traversal on GPUs. *Proceedings of SC* (2015).
[35] Yael Maon, Baruch Schieber, and Uzi Vishkin. 1986. Parallel ear decomposition search (EDS) and st-numbering in graphs. *Aegean Workshop on Computing* (1986).
[36] William Mclendon Iii, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel and Distrib. Comput.* (2005).
[37] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. *Proceedings of SOSP* (2013).
[38] John H Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* (1985).
[39] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. *Proceedings of SOSP* (2013).
[40] John Scott. 1988. Social network analysis. *Sociology* (1988).
[41] Yossi Shiloach and Uzi Vishkin. 1980. An O (log n) parallel connectivity algorithm. (1980).
[42] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. *Proceedings of PPoPP* (2013).
[43] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. 2002. The boost graph library: user guide and reference manual. (2002).
[44] George M Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. *Proceedings of HiPC* (2014).
[45] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. *Proceedings of IPDPS* (2014).
[46] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. *International Conference on Web Search and Data Mining* (2018).
[47] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* (1972).
[48] Robert E Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.* (1985).
[49] Yung H Tsin and Francis Y Chin. 1984. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. Comput.* (1984).
[50] Lei Wang, Fan Yang, Liangji Zhuang, Huimin Cui, Fang Lv, and Xiaobing Feng. 2016. Articulation points guided redundancy elimination for betweenness centrality. *Proceedings of PPoPP* (2016).
[51] Sandeep Yadav, Ashwath Kumar Krishna Reddy, AL Narasimha Reddy, and Supranamaya Ranjan. 2010. Detecting algorithmically generated malicious domain names. *Proceedings of SIGCOMM* (2010).
[52] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, Lijun Chang, and Xuemin Lin. 2015. I/O efficient: computing sccs in massive graphs. *Proceedings of VLDB* (2015).
[53] Yao Zhao, Yinglian Xie, Fang Yu, Qifa Ke, Yuan Yu, Yan Chen, and Eliot Gillum. 2009. BotGraph: Large Scale Spamming Botnet Detection. *Proceedings of NSDI* (2009).