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,

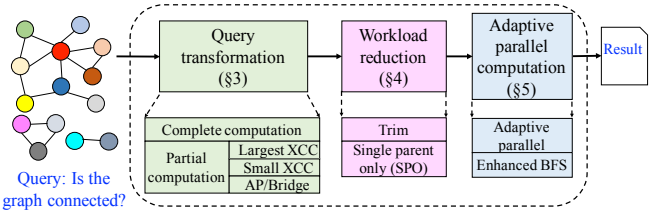


Figure 2: The framework of Aquila.

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

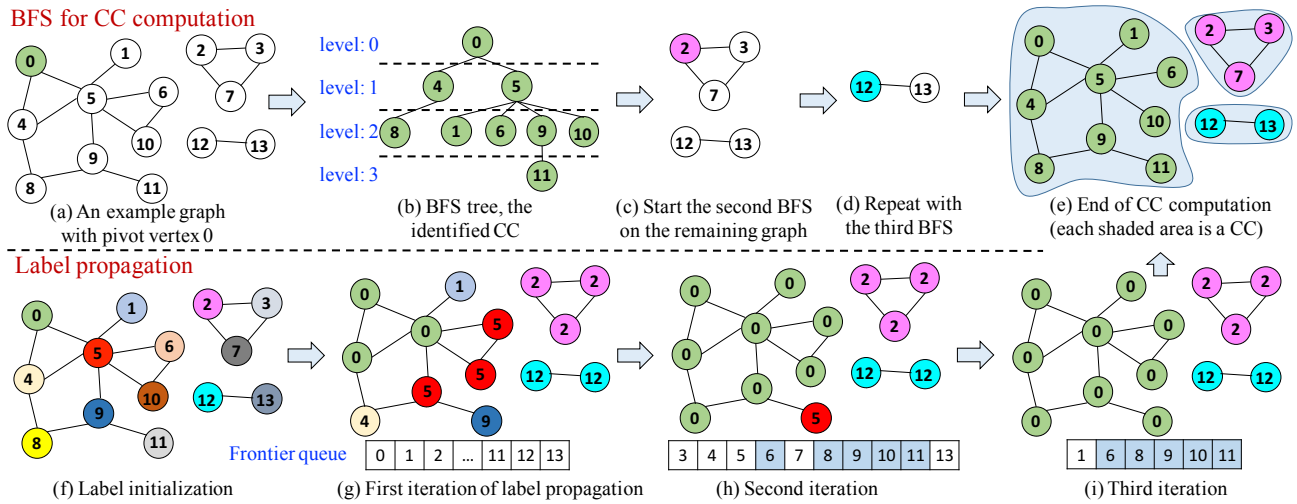


Figure 3: CC computation with BFS and label propagation. Given an example graph in (a), the BFS-based solution runs the first BFS from pivot vertex 0 in (b), the second BFS from pivot vertex 2 in (c), and the third one from pivot vertex 12 in (d). The final CC computation result is shown in (e). Label 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

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;
```

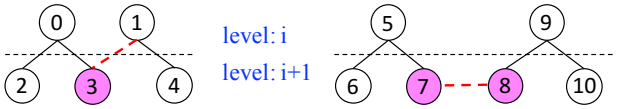
As one needs to run up to $|V|$ BFSes, we reduce the workload with the single parent only technique, which is based on the following two lemmas.

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

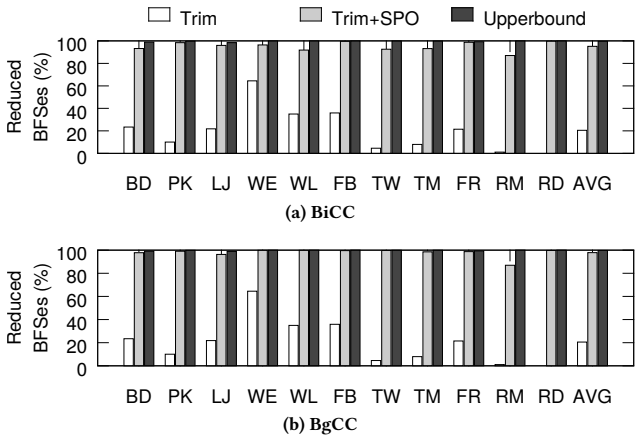


Figure 6: The percentage of reduced BFSes for (a) BiCC and (b) BgCC.

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

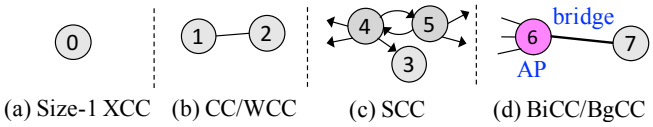


Figure 7: Trim patterns for connectivity algorithms, (a) is for all XCC, (b) is for (W)CC, (c) is for SCC, and (d) is for BiCC and BgCC.

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.

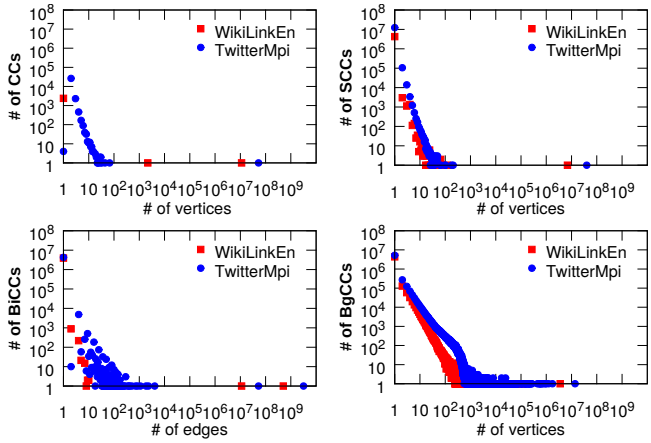


Figure 8: The number of XCCs against their sizes, in the order of (W)CC, SCC, BiCC, and BgCC, respectively. BiCC size is measured by edge count, and the others are measured by vertex count.

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.