Shared Memory Parallelism Implementation of Triangle Counting

Dylan Xu, Farhan Baig, Forrest Gao

April 2025

Abstract

In this report, we provide a readily implementable shared-memory parallelism approach for the triangle counting problem. Triangle counting is an important problem in domains such as social networks, spam detection, database handling, biological networks, and recommendation systems. As the size of graphs/datasets increases, the need for fast triangle counting algorithms continues to rise, and as such, it serves as an interesting avenue of research for us.

Our approach was originally an attempt to implement a shared-memory version of the solution developed by Liu et al., which had used Massively Parallel Computation (MPC). However, as we adapted the algorithm, many of the details for the MPC system were either removed or modified in the shared-memory parallel solution, resulting in a novel triangle counting implementation for shared-memory parallelism that performs as well as *GBBS*. Our solution implements a heuristic for searching for shared vertices, and it takes advantage of a "look-ahead" strategy to avoid over-counting triangles.

This algorithm achieves asymptotic bounds of $O(m \cdot n)$ work and $O(n)$ span. Although the span is asymptotically high due to our node-dependent use of a two-pointer approach, we found that, in practice, it achieved better performance than another method that had $O(\log n)$ span.

Algorithmic Framework for Optimization

This section introduces the common framework that all of our optimizations build upon. While each optimization modifies different parts of the algorithm, the high-level structure remains consistent.

We iterate over all edges $\{u, v\}$ in parallel. For each edge, we compute the number of vertices shared between u and v. Each shared vertex corresponds to a triangle formed by u, v, and the shared neighbor. Since every triangle contains three edges, each triangle is counted three times in total–once from the perspective of each of its edges. Consequently, we divide the total number of shared vertices by three to obtain the final triangle count.

Across the following algorithms, we explore different strategies for computing the number of shared vertices between the two endpoints of each edge. To test runtimes, we ran each algorithm five times on the *email-Enron* dataset from Stanford SNAP's large network dataset collection. This dataset has 36,692 vertices, 183,831 edges, and 727,044 triangles. The runtimes are listed next to each algorithm's name.

Baseline Algorithm (159.553 ms)

First, we present a naive algorithm. For this algorithm and the first optimization, we use an edge list for our graph representation. We begin by constructing a set of edges from the edge list. Given an edge $\{u, v\}$, we identify the vertex of lower degree, which we will arbitrarily denote by u. We then iterate over the neighbors of u, and for each neighbor w, we check whether the edge $\{w, v\}$ exists in the set. If the query succeeds, we increment the total shared-vertices count by one. Because this count is shared between different processors, we use a fetch-and-add operation to ensure atomicity while incrementing.

Optimizations

1 Deferred Aggregation via Parallel Reduction (34.564 ms)

Although fetch-and-add operations are relatively efficient, we hypothesized that avoiding atomic operations on a shared variable would lead to faster performance. To achieve this, we introduced a sequence whose length is the number of edges. During the parallel iteration over the edges, each iteration maintains its own local counter in the corresponding entry of the sequence, incrementing it by one each time a shared vertex is found. This design preserves the overall logic of the baseline algorithm, while eliminating the need for synchronization. Localizing updates in this way is safe because each iteration is independent and writes exclusively to its own entry in the sequence. After all iterations are complete, we perform a parallel sum reduction over the sequence to compute the total number of shared vertices. This approach eliminates contention on a shared counter, deferring aggregation to an efficient parallel reduction step.

2 Binary Search Edge Querying (6.508 ms)

Initially, we believed that using an edge-set for our edge-queries would be an optimal strategy, as it allows for an *expected* constant-time query when checking for the existence of a particular edge. However, we explored an alternative approach based on sorted adjacency lists. In this graph representation, each vertex maintains a sorted list of its neighboring vertex IDs. Importantly, this more structured data format was also used by *GBBS*, meaning our performance comparison discussed later is still an equitable comparison. In the binary search strategy, given an edge $\{u, v\}$, we iterate over the neighbors of the lower-degree vertex, say u, and for each neighbor w, we perform a binary search for w within v's neighbor list.

Within this strategy, there exists an implementation detail that drastically changes practical runtime. Once again, assume that for an edge $\{u, v\}$, u is the lower-degree vertex and v is the higher-degree vertex. We need to check if edge $\{v, w\}$ exists $\forall w \in \mathit{neighbor}(u)$. In theory, we could either binary search for the existence of w within v's neighbor list as discussed above, OR we could binary search for the existence of v within w's neighbor list. On our initial attempt, we made this decision by binary searching the shorter adjacency list.

However, we quickly discovered that it is almost never optimal to binary search w's neighbor list for v, even if it is significantly shorter than v's adjacency list. This is due to the principle of cache locality: if we fix v and iterate over each $w \in \mathit{neighbor}(u)$, then the list of v's neighbors constantly remains in cache. In fact, when we tried the dynamic approach of switching out v's adjacency list and w's adjacency list per iteration depending on their lengths, we saw significant slowdowns due to cache thrashing. Therefore, our optimal solution uses the static approach of always binary searching v's adjacency list, as the cache locality results in practical gains.

3 Two Pointer (4.246 ms)

Our next optimization is based on a two-pointer approach. Since each vertex's neighbor list is stored in sorted order, we can efficiently compute the number of intersections by traversing the two lists simultaneously. Given an edge $\{u, v\}$, we initialize two pointers at the beginning of u's and v's neighbor lists, respectively. At each step, we compare the vertex IDs pointed to by the two pointers: if u's vertex ID is smaller, we advance u's pointer; if v's vertex ID is smaller, we advance v's pointer. If the two IDs are equal, we increment a local counter and advance both pointers.

The correctness of this approach follows from the sorted order of the neighbor lists. If the vertex ID at u's pointer is smaller, it cannot match any future vertex in v's list, so it must be advanced. Similarly, if the vertex ID at v's pointer is smaller, we must advance v's pointer. When the two IDs are equal, a shared neighbor has been found, so we can increment the counter accordingly, and advance both pointers. Since at least one pointer advances at each step, the algorithm always terminates after a finite number of steps.

4 Combining Binary Search and Two Pointer (2.996 ms)

Let $a = \min(deg(u), deg(v))$
Let $b = \max(deg(u), deg(v))$

Recall Optimization 2. We iterate over the neighbors of the lower-degree vertex and perform a binary search for each neighbor within the other vertex's neighbor list. This approach yields a runtime of $O(a \log b)$. On the other hand, Optimization 3 uses a two-pointer technique, which traverses both neighbor lists simultaneously in $O(a + b)$ time.

We observed that neither approach is uniformly better across all edges. If $b \approx a$, then $O(a \log b)$ would be slower than $O(a + b)$ due to the additional logarithmic factor. On the other hand, if $b \gg a$, then the dominating factor is b, and a can be approximated as a constant. So, to a first order correction, the binary search approach can be approximated as $O(\log b)$, while the two-pointer approach would still require $O(b)$ time. This motivated us to combine both strategies into a hybrid approach.

In the hybrid algorithm, we introduce a tunable parameter β, called the binary search factor. For each edge $\{u, v\}$, if the degree ratio $\frac{b}{a}$ exceeds β, we use binary search; otherwise, we apply the two-pointer method. After experimenting with different values of β using a bash script to automate testing, we found that setting $\beta \approx 100$ yielded the best performance empirically.

5 Other Optimizations (1.644 ms)

5.1 Manual Loop Control for Early Termination

In our two-pointer approach, the loop exits early if either pointer reaches the end of its respective neighbor lists. We further optimized this traversal by manually restructuring the loop to minimize condition checking overhead. Instead of checking for termination at the beginning of each iteration, we perform only the relevant boundary checks immediately after advancing the pointers, and exit early when necessary.

5.2 Compiler Flags

1. -O3: most aggressive GCC opimization level
2. -march=native: generates machine code optimized for the CPU that the program is compiled on

5.3 Forward Adjacency List

At this point, we were still counting the total number of shared vertices and dividing that value by 3, so we knew a direct counting method would significantly improve performance. To achieve this, we modified the graph representation to store only "forward" edges: for each vertex u, we retained only neighbors v such that $u < v$. In other words, each vertex's adjacency list contains only neighbors with greater vertex IDs.

This approach correctly avoids triple counting due to the properties of the forward adjacency list. Let (a, b, c) denote the three vertices of a triangle, where $a < b < c$. In the original adjacency list representation, c would be a shared neighbor between a and b, b would be a shared neighbor between a and c, and a would be a shared neighbor between b and c, leading to the need to divide the total count by three. However, with the forward adjacency list, only c remains a shared neighbor between a and b. The edges (a, c) and (b, c) would not find b and a as neighbors, respectively. Thus, each triangle is counted exactly once without requiring any post-processing to correct for overcounting.

6 Experiment: ParlayLib Merge (6.981 ms)

This section describes an algorithm we experimented with but ultimately did not include in our final implemenation due to its poor empirical performance.

We explored a merge-based strategy using ParlayLib *merge*. Given an edge $\{u, v\}$, we first merged u's sorted neighbor list and v's sorted neighbor list into a single sorted sequence. In the merged list, each pair of consecutive duplicate vertex IDs corresponds to a shared neighbor between u and v. To identify these, we looped through the merged sequence in parallel, marking indices whenever two consecutive entries matched. We then performed a parallel reduction over these marks to compute the total number of shared neighbors.

Because *merge* has a theoretical span of $O(\log n)$, we initially expected this approach to yield better performance. However, empirical results showed significantly worse runtimes compared to our optimal algorithm. We hypothesize that the limited number of available processors constrained the potential benefit of *merge*'s low span. In a setting with a very large number of processors, we anticipate that this algorithm could perform much closer to its theoretical potential.

Pseudocode

Algorithm 1 Parallel Triangle Counting Algorithm

```
Require: 1. A simple undirected graph $G$ in "forward" adjacency list format[1] 2. A list of the edges,
    specifically in the format $(a, b)$ where $a < b$. For $|G(V)| = n$, the indices of the vertices must be in
    $\{0, ..., n-1\}$.[2]
Ensure: Output: The number of triangle contained in the graph
 1: $\beta \leftarrow 100$
 2: Instantiate a list counts where index $i$ will accumulate the triangle count for the ith edge. $counts \leftarrow$ list
    of size $m$
 3: for parallel $e$ in edges do
 4:     Let $e = (u, v)$ where $u$ is smaller-degree vertex, $v$ is larger-degree vertex
 5:     if $deg(v)/deg(u) > \beta$ then
 6:         $\forall w \in adj(u)$, perform $binary\_search(w, adj(v))$
 7:         increment $counts[e]$ based on hits in the above step
 8:     else
 9:         $\forall w \in adj(u)$, perform $two\_pointer\_search(w, adj(u), adj(v))$
10:         increment $counts[e]$ based on hits in the above step
11:     end if
12: end for return $parallel\_reduce(counts)$
```

[1]Check section 5.3 for definition of forward adjacency list

[2]Note: in terms of implementation detail, we read in an edge list representing an undirected graph, where (a, b) and (b, a) are both included in the list. But, the sequence *edges* only contains each edge once.

Runtime Guarantees

In this section, we present a brief analysis of the asymptotic work and span of our algorithm. We focus only on worst-case bounds. Recall that in the hybrid approach, the two-pointer method is used in most cases. To simplify the analysis, we assume that whenever the binary search method is selected, its runtime is strictly better than that of the two-pointer method. Thus, we treat every iteration of the outer loop as using the two-pointer strategy. Let m represent the number of edges in our graph, and n represent the number of vertices.

Asymptotic Work Analysis

We begin by considering an arbitray edge $\{u, v\}$, which corresponds to a single iteration of the outer loop. Each iteration performs the two-pointer strategy, which requires traversal through the entire list of neighbors for at least one of the two vertices u and v. Since the maximum degree for a vertex is n, in the worst case, we have to loop through n neighbors. Hence, the worst-case work for a single iteration is $O(n)$. Since we conduct this procedure for each edge, we conclude that the work required to fill our *counts* sequence is $O(m \cdot n)$. Lastly, we consider the work of the parallel reduction over *counts*, which is simply $O(n)$. Therefore, the total work of our algorithm is $O(m \cdot n + n) = O(m \cdot n)$.

Asymptotic Span Analysis

Assuming access to an infinite number of processors, we can parallelize the outer loop and focus on the span of each individual iteration, which consists of a single execution of the two-pointer method. Since the two-pointer approach must be done sequentially, the worst-case span per iteration is $O(n)$ (see work analysis for explanation). Lastly, we consider the span of the parallel reduction over *counts*, which requires $O(\log n)$ span. Thus, the total span of our algorithm is $O(n + \log n) = O(n)$.

In theory, the span could be improved to $O(\log n)$ given infinite processors, as we experimented in Optimization 6. However, empirically, we found that Optimization 6 did not yield a speedup due to a limitation of processors, which prevented nested parallelism from being effective.

Results and Testing

In this section, we describe our results and findings from testing our algorithm on graphs from the Stanford SNAP database. We compiled a comprehensive test-suite of 49 graphs from the database spanning graphs of various density and size. Note that our runtimes do NOT include the input parsing time. Since our parsing time is higher than the whole *GBBS* runtime for larger graphs, we assumed that either *GBBS* did not include parsing time, or that it has fast enough of a parser to not constitute a large portion of the runtime, thus still allowing for meaningful comparison of the algorithmic results.

Before we examined different graph properties (number of nodes, number of edges, arboricity), we first performed a baseline check over the distribution of all the test graphs to view our speedup over *GBBS*. Note that Fig 1. shows that we had a median speedup over *GBBS* of 1.75 times, and we can also see that on some graphs, the speedup was much more substantial.

Next, we examined how our performance scaled as the number of edges increased. Fig 2. shows the log-scaled time in milliseconds versus the log-scaled edges across all of our test samples. Based on the line of best fit, it seems that our implementation is significantly faster than the *GBBS* implementation for graphs of up to 4 million edges. However, after this, the lines of best fit indicate that *GBBS* starts outperforming our implementation.

Visually, it would appear less obvious that *GBBS* is truly performing better than our implementation for the larger graphs (more edges). So, we decided to perform a couple of statistical tests to examine the results. Note that the underlying distribution of data (i.e. the pairwise differences) are likely not normally distributed, so we decided to utilize a Wilcoxon signed-rank test instead of a t-test.

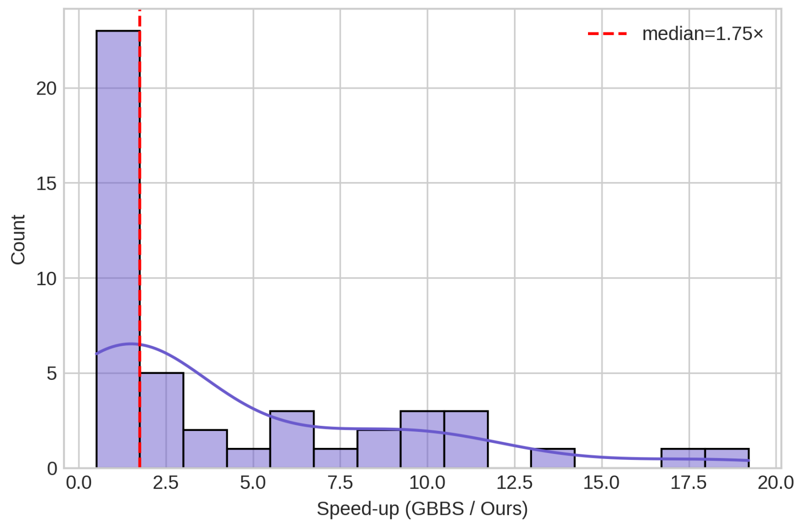

Figure 1: Distribution of speedup from GBBS implementation to our implementation over 49 test graphs.

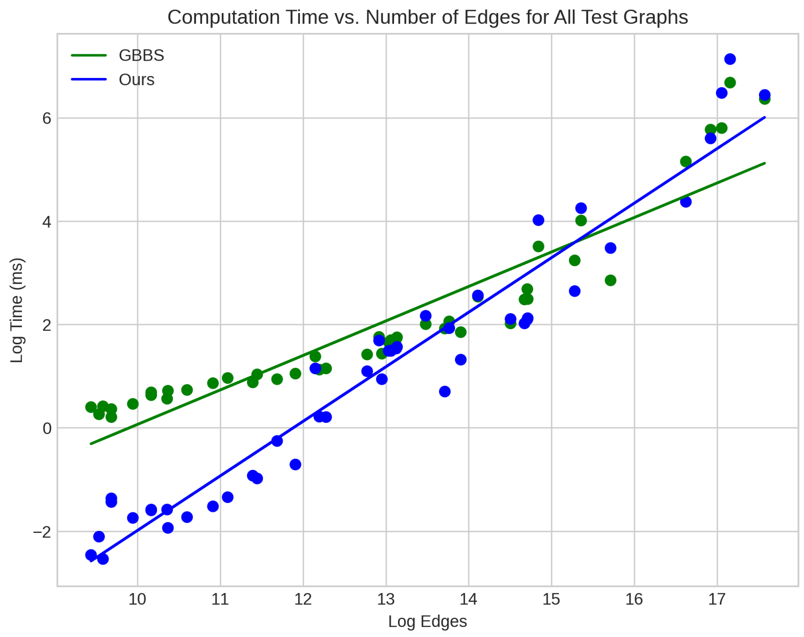

Figure 2: Log Time vs. Log Edges across 49 test graphs.

We began by segmenting the data into two subsets: *(i)* As the first test set, we chose the bottom 75 percent of graphs by number of edges (nominally the “small graphs”) where we seem to perform better than *GBBS* *(ii)* As the second test set, we chose the top 25 percent of graphs by edges (nominally the "large graphs") where $GBBS$ seems to perform better than us. We performed the Wilcoxon signed-rank test on both subsets of data to see if we have a significantly different performance compared to $GBBS$ for small graphs and also to see if we have a significantly different performance compared to $GBBS$ for large graphs.

Performing the Wilcoxon signed-rank test on the small graphs resulted in *p-value* ≈ 0. This suggests that we can strongly reject the null hypothesis that there is no difference between our performance and $GBBS$ performance. In other words, there is a statistically significant difference in performance between us and $GBBS$ for small graphs. On the other hand, the Wilcoxon signed-rank test for the large graphs yielded *p-value* ≈ 0.47. This large *p-value* indicates that we must fail to reject the null hypothesis; in this case, the null hypothesis is that our median performance and the median $GBBS$ performance have no difference. Thus, our current data has failed to provide compelling statistical evidence that $GBBS$ outperforms us for larger graphs. Admittedly, our sample size of large graphs was very limited. So, it would be an interesting exercise to run tests on even larger graphs to get a better sense of how we perform against $GBBS$ at a truly asymptotic level.

One final metric by which we measured performance was arboricity. The arboricity α of an undirected graph is the minimum number of forests into which its edges can be partitioned. In some sense, arboricity is a measure of density. We wished to see how our performance scaled with the arboricity. After removing some obvious outliers, Fig 3. displays the general trend of performance as the log of $(nodes * arboricity)$ increases We chose this term as we believe it best captures the effects of both the size of the graph and its density. As we can see, it seems that our performance tracks closely with the $GBBS$ performance across the test points. We decided to further analyze the performance as a function of arboricity.

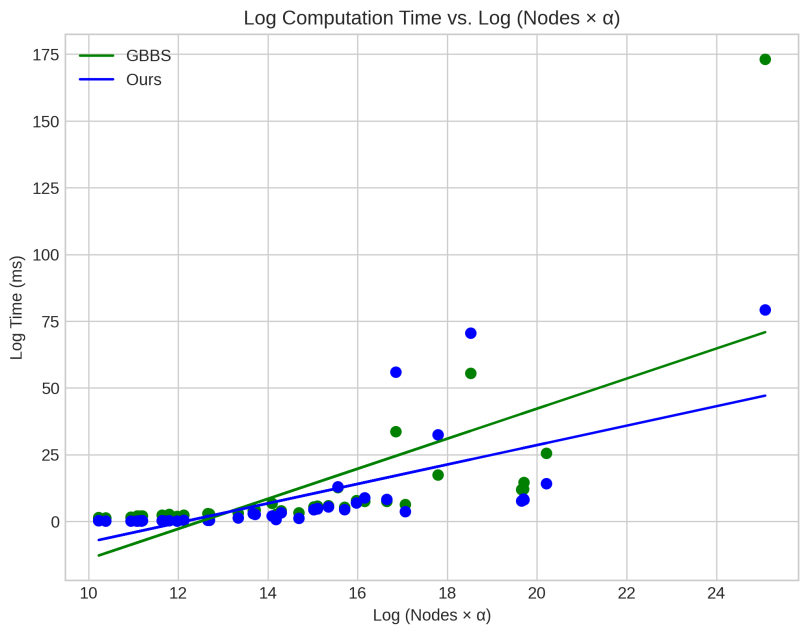

Figure 3: $log(runtime)$ versus $log(nodes * arboricity)$. Note that obvious visual outliers were removed

In particular, we were interested in analyzing the speedup compared to $GBBS$ across graphs of differing arboricity. We defined performance speedup as $GBBS$ time divided by our time. When we did this, we interestingly noted two different distributions. Upon examining the data points, we discovered that one cluster of points belonged to low-arboricity graphs ($\alpha \leq 20$), while another cluster of points belonged to high-arboricity graphs.

We split the data into these two clusters, and Fig 4. contains the resulting plots (the color bar displays the number of nodes per graph). As we can see, for the low-arboricity graphs, we start with a higher log fold change, but rapidly decline as the arboricity increases. However, for the higher arboricity graphs, although we have lower performance increases, and sometimes performed worse than *GBBS*, the trend is less obviously downwards. This supports the theory that our implementation is significantly faster at low-medium density graphs, but perhaps struggles at greater density.

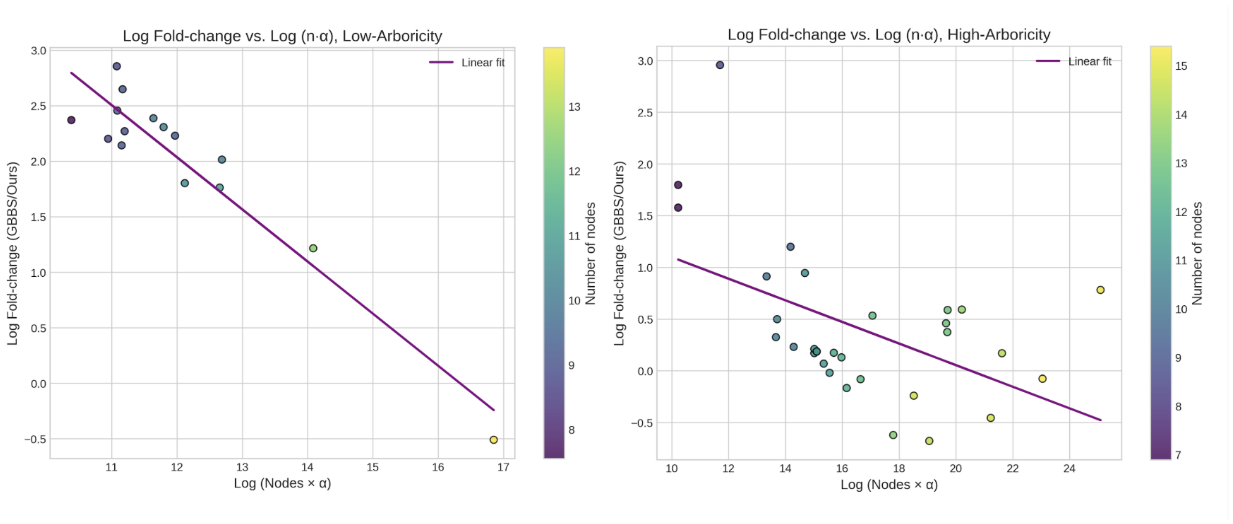

Figure 4: log fold change in performance compared to $log(nodes * arboricity)$. The points in the first plot correspond to graphs of low arboricity, whereas the points in the second plot correspond to graphs of high arboricity.

Conclusion

In conclusion, we have implemented a shared-memory parallel approach to counting triangles, which is a problem that has many important computational applications. We took advantage of graph properties, such as node-degree, and principles of cache locality to develop a dynamic approach for counting triangles.

We have seen that our implementation has a statistically significant performance increase compared to the *GBBS* implementation for graphs of small to medium size (up to roughly 4 million edges). Although we are less sure that our results scale asymptotically with *GBBS*, we were not able to find evidence among our test set that *GBBS* outperformed our algorithm for the large graphs that were included in our test sample.

Further, we found that for graphs of high arboricity, our fold increase in performance over *GBBS* decreases at a lower rate compared to the decrease we see among graphs of low arboricity. This may suggest that our program has superior performance in small-medium sized graphs, but on an asymptotic scale, *GBBS* may perform better than us by at least a constant factor. As a point of further exploration, it would be interesting to determine some applications that would require rapid calculation of triangles in medium-sized graphs, and it would also be interesting to optimize the $log(n)$ span solution that we experimented with in Optimization 6—doing so may result in a better asymptotic and practical performance compared to *GBBS* for truly large graphs, such as on the order of half a billion edges.