New Parallel Algorithms for Planarity Testing

by

Amelia Y. Hu

S.B., Computer Science and Engineering and Mathematics, Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Amelia Y. Hu Department of Electrical Engineering and Computer Science May 10, 2024
Certified by:	Julian Shun Associate Professor of EECS, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

New Parallel Algorithms for Planarity Testing

by

Amelia Y. Hu

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Planar graphs (defined as graphs in which no edges cross) have special properties and are often used in applications such as circuit design or transportation networks. While many linear work implementations of planarity testing algorithms exist, to our best knowledge, there is no practical implementation of a parallel planarity testing algorithm. In this thesis, we will describe and analyze two new parallel algorithms for planarity testing, both derived from the Boyer-Myrvold algorithm. First, we will present a divide-and-conquer approach, where the graph's edges are evenly distributed among worker threads. Each thread independently executes the sequential Boyer-Myrvold algorithm on its designated subgraph. Then, pairs of subgraphs are merged by embedding the edges between subgraphs with modified Boyer-Myrvold methods. The primary challenge of the divide-and-conquer approach is the merge step as determining the relative positions of subgraphs is a complicated and difficult process. Next, we describe the design and implementation of a new and simpler parallel algorithm. This algorithm modifies the Boyer-Myrvold algorithm by processing vertices in layers from the bottom-up (rather than sequentially by reverse DFI order). The computation in each layer is parallelized. On planar graphs, this algorithm achieves 2.4–2.7 times speedup over the sequential algorithm when run on 16 cores. On non-planar graphs, the performance gain is even more significant, with speedups ranging from 9 to 22 times on 16 cores.

Thesis supervisor: Julian Shun
Title: Associate Professor of EECS

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Julian Shun, for his invaluable guidance, unwavering support, and insightful feedback throughout these past few semesters. His mentorship has been truly instrumental in shaping my academic journey at MIT, and I appreciate his dedication and patience in advising me.

I am also immensely thankful to Junhong Lin for his exceptional mentorship, continuous encouragement, and hands-on guidance throughout our direct collaboration. Junhong's expertise, patience, and willingness to share his knowledge have significantly contributed to the success of this work. His guidance has been invaluable, and I am deeply grateful for the opportunity to work alongside him.

I would like to extend my sincere appreciation to Shangdi Yu and other members of the MIT ParAlg group for their support on this project.

I am grateful to all my colleagues, friends, and family members for their support and encouragement, and I would like to express my heartfelt thanks to all the professors, instructors, and staff members who have contributed to my academic and personal growth during my time at MIT.

Contents

Title page **1**

Abstract **3**

Acknowledgments **5**

List of Figures **9**

1 Introduction **11**
1.1 Motivation . . . 11
1.2 Terminology . . . 11
1.3 Related Work . . . 12
1.3.1 Previous Work in Planarity Testing . . . 13
1.3.2 Boyer-Myrvold Algorithm . . . 13
1.3.3 Parallelization Attempts . . . 16
1.4 Overview . . . 16

2 Divide-and-Conquer Algorithm **19**
2.1 Initialization . . . 19
2.2 Temporary Vertices . . . 20
2.3 Embedding Pending Edges . . . 22
2.4 Flippable Components . . . 23
2.5 Algorithm Outline . . . 24
2.6 Proof of Correctness . . . 25
2.7 Implementation . . . 26

3 Parallelization by Layer **27**
3.1 Algorithm Outline . . . 27
3.2 Proof of Correctness . . . 28
3.3 Theoretical Bounds . . . 29
3.4 Implementation . . . 30
3.5 Test Graphs . . . 31
3.6 Results . . . 32
3.6.1 Planar Graphs . . . 32
3.6.2 Non-Planar Graphs . . . 34
3.7 Performance Bottlenecks . . . 36

3.7.1 Initialization . . . 36
3.7.2 Theoretical Bounds . . . 37
3.8 Tuning . . . 37
3.8.1 Granularity . . . 37
3.8.2 Stopping Criteria . . . 38
3.8.3 Branching Criteria . . . 39

4 Conclusion **43**
4.1 Summary . . . 43
4.2 Future Work . . . 44

References **47**

List of Figures

1.1 Left: The graph is not biconnected, because removing vertex b would separate the graph into two components. Right: The graph is biconnected, because, for every pair of vertices, there exists at least two distinct paths between them. Figure from [7]. Below each graph is its corresponding planar embedding. . . 12

1.2 In this example, vertices with a superscript represent virtual vertices. For example, 1^2 is a virtual vertex of node 1 associated with child node 2. Left: the graph is separated into three biconnected components: 0^1 and 1; 1^2, 2, and 3; 3^4, 4, 5, and 6. Center: Since 3 is still detached from its virtual vertex 3^4, the component can flip freely. Right: When the edge from 1^2 to 4 is embedded, the biconnected components rooted at 1^2 and 3^4 merge and vertex 3 merges with its virtual vertex. Figure from [7]. 14

1.3 In this example, the square nodes are considered externally active during the processing of vertex v. z is marked externally active because it is incident to the forward edge originating from u (ancestor of v). Consequently, d and x are considered externally active as well because they have a descendant biconnected component that contains an externally active vertex. Figure from [5]. 14

1.4 An outline of the Boyer-Myrvold algorithm as described in [5]. 15

2.1 Left: In this example, the component containing the light blue vertices is embedded within the face of the red subgraph. Right: We replace the component with the temporary vertex (in dark blue) and attach all pending edges to temporary vertex instead. Figure from [7]. 20

2.2 Left: the red and blue vertices each below to a different subgraph. (This is possible if the components of the blue subgraph represent the two descendant subtrees of the vertices in the red subgraph.) Right: we construct two temporary vertices (in dark blue): one for the component on the outside and one for the component on the inside. 21

2.3 The red and blue vertices represent two different subgraphs to be merged, and the green and yellow vertices are part of the red subgraph. Left: In the current orientation, the graph appears to be non-planar. Right: If we flip the green and yellow vertices, we derive a planar graph. 24

3.1 The embedding time of small planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm. . . . 32
3.2 The embedding time of medium planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm. . . . 33
3.3 The speedup relative to sequential is calculated by taking the runtime of the sequential algorithm and dividing it by the runtime of the parallel algorithm. The values plotted represent the mean taken over all the planar graphs in the small graphs dataset. . . . 33
3.4 The embedding time of large planar graphs. The blue curve represents the embedding time from the original sequential algorithm. . . . 34
3.5 The embedding time of small non-planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm. . . . 35
3.6 The speedup is calculated by taking the runtime of the sequential algorithm and dividing it by the runtime of the parallel algorithm. The values plotted represent the mean taken over all the non-planar graphs in the small graphs dataset. . . . 35
3.7 The breakdown of runtime into initialization, embedding and post-processing steps on two sample graphs. Left: Data from a small graph with 105,000 vertices and about 315,000 edges. Right: Data from a medium graph with 1.3 million vertices and about 3.9 million edges. Note that, in both cases, the parallel algorithm primarily improves the embedding time and incurs some overhead in the initialization step. . . . 36
3.8 The embedding time of planar graphs with granularity set to default (GD), 1 (G1) and 2 (G2). . . . 38
3.9 The embedding time of small planar graphs where the threshold for the base case was 10 (S10), 50 (S50), 100 (S100), 500 (S500), and 1000 (S1000) vertices. 39
3.10 The embedding time of medium planar graphs where the threshold for the base case was 10 (S10), 50 (S50), 100 (S100), 500 (S500), 1000 (S1000), 5000 (S5000), and 10000 (S10000) vertices. . . . 39
3.11 The embedding time of small planar graphs where the threshold for using a parallel for loop is set to 2 (B2), 4 (B4), 8 (B8), and 16 (B16) iterations. . . . 40
3.12 The embedding time of medium planar graphs where the threshold for using a parallel for loop is set to 2 (B2), 4 (B4), 8 (B8), and 16 (B16) iterations. . . . 41

Chapter 1

Introduction

1.1 Motivation

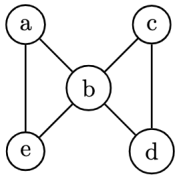
A *planar graph* is a graph that can be embedded in the plane without any edges crossing [1]. Planar graphs have special properties and are often used in applications such as circuit design or transportation networks. Planarity testing algorithms can take in an input graph as a set of vertices and edges, and output whether or not the graph is planar and, in some cases, generate a planar embedding (a representation of the graph in the plane). Most existing planarity testing algorithms are sequential in nature and operate relatively well on small to moderate size graphs [2]–[5]. However, given larger, more complex graphs, it may be beneficial to find a planarity testing algorithm that can take advantage of modern parallel computing architectures to speedup the analysis of large-scale graphs and efficiently detect whether a given input is planar. Previous works have discussed the possibility of a parallel planarity testing algorithm [6]–[8], but, as far as we know, there have not been practical implementations of a parallel planarity testing algorithm. This thesis will discuss two different parallel planarity testing algorithms.

In the following sections, this thesis will first discuss the existing sequential planarity testing algorithms and the progress towards a parallel planarity testing algorithm. We will then outline two new parallel algorithms and discuss performance, complications, etc.

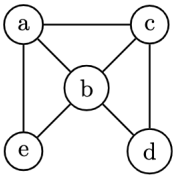
1.2 Terminology

First, let us define a few key terms that will be continually used throughout the rest of the proposal. A graph G can be represented by vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, where $n = |\mathcal{V}|, m = |\mathcal{E}|$. A graph is connected if for every pair of vertices, (v_1, v_2), there exists a path from v_1 to v_2 via edges in $\mathcal{E}$. A graph is *biconnected* if, for the removal of any one vertex and its incident edges, the graph remains connected. That is, for every pair of vertices, (v_1, v_2), there exists at least two distinct paths between v_1 and v_2. A *connected component* is a subset of the graph that is connected and a *biconnected component* is a subset of the graph that is biconnected. A graph is planar if it can be drawn in the plane with no edges crossing. A *planar embedding* is a representation of a planar graph. In particular, given a planar graph, we can generate

Figure 1.1: Left: The graph is not biconnected, because removing vertex b would separate the graph into two components. Right: The graph is biconnected, because, for every pair of vertices, there exists at least two distinct paths between them. Figure from [7]. Below each graph is its corresponding planar embedding.



a	b,e
b	c,d,e,a
c	d,b
d	b,c
e	b,a



a	c,b,e
b	c,d,e,a
c	d,b,a
d	b,c
e	b,a

a unique embedding by listing the edges incident to each vertex in clockwise order (see the example shown in Figure 1.1). A *complete graph* is a graph in which there exists an edge from every vertex to every other vertex in the graph and a complete graph with n vertices can be represented by the notation K_n. A *bipartite graph* is a type of graph whose vertices can be divided into two disjoint sets such that no two vertices within the same set are adjacent. A bipartite graph with disjoint sets of size n_1 and n_2 can be represented with the notation K_{n_1,n_2}.

A *depth-first traversal* (also known as depth-first search or DFS) of a graph starts at an initial node and explores each of its neighbors recursively (traveling as far along each branch as possible) before returning to the initial node. A node's depth-first index (DFI) is the label corresponding to the order in which the node is first visited in the DFS. A *breadth-first traversal* starts at an initial node and explores each of its neighbors first before moving to the next level of neighbors.

In this thesis, we use a parallel computing model that centers around multicore shared-memory architectures. In this setup, multiple processors collaborate on executing tasks concurrently while accessing the same global memory pool. Throughout the discussion, the terms "thread", "worker" and "processor" are used interchangeably to denote the hardware cores on a machine. To analyze the complexity of our algorithms, we will use the work-span model [9]. *Work* refers to the total number of operations required to complete a task, encompassing the work performed by all threads. *Span* is the critical path length or the longest chain of dependencies among operations.

1.3 Related Work

To provide some background on planarity testing, we will first go over some of the previous works on planarity testing algorithms.

1.3.1 Previous Work in Planarity Testing

One of the foundational results in planarity testing is Kuratowski's theorem [1] which states that a finite graph is planar if and only if it does not contain a subgraph that is a subdivision of the complete graph K_5 (the complete graph on five vertices) or the complete bipartite graph $K_{3,3}$ (a bipartite graph with three vertices in each partition). In addition, as a corollary of Euler's formula, we have that $m \leq 3n - 6$. That is, $m = O(n)$.

One of the earliest works on planarity testing was an algorithm developed by Hopcroft and Tarjan which used the idea of path addition [2]. The algorithm performs‘ a depth-first traversal of the graph. After the traversal, the algorithm starts with a cycle in the graph (found during the traversal) and adds new paths to the graph one at a time. While the Hopcroft and Tarjan algorithm has been surpassed by newer, more performant planarity testing algorithms, the theme of using a depth-first traversal appears in many subsequent works.

Later on, Booth and Lueker approached planarity testing by introducing the PQ-tree data structure and Shih and Hsu introduced the PC-tree data structure (a generalization of PQ-trees) [3], [4]. In a PQ-tree, P vertices are defined as vertices where the children can be permuted in any order and Q vertices as vertices where the children must stay in the order in which they are listed or the reverse order. PC-trees explore a similar concept but contain C vertices instead of Q vertices which allow for a circular ordering of children rather than a linear one. The idea behind these tree data structures is that in the planarity testing process, the children of some vertices in the tree must remain in a specific order to prevent any edge crossings in the embedding.

1.3.2 Boyer-Myrvold Algorithm

The standard planarity testing for present-day graph libraries and the algorithm upon which we aim to build our work is the *Boyer-Myrvold algorithm* (summarized in Figure 1.4). The Boyer-Myrvold algorithm seeks to create a planar embedding by adding edges into the graph one by one [5]. Like Hopcroft and Tarjan's algorithm, the Boyer-Myrvold algorithm begins by constructing a depth-first-search tree (line 1 of Figure 1.4). For each internal node in the tree, we create a corresponding virtual vertex and we detach the tree into biconnected components (see Figure 1.2 for an example). Eventually, the virtual vertices will be merged with the original vertices, but the tree is initially separated into biconnected components in order for sections of the graph to be able to flip or change their orientation independently before additional edges are embedded to keep it in place. Any edges that are not already present in the DFS tree must be a forward edge from an ancestor to one of its descendants. The general goal of the algorithm is to embed each of the forward edges in a particular order and to merge the detached biconnected components as the forward edges are embedded.

A child w of vertex v is considered *externally active* if there is a path from w to a DFS ancestor of v consisting of zero or more descendant DFS tree edges plus one unembedded forward edge (refer to Figure 1.3). If w is externally active, it will be involved in a future

Figure 1.2: In this example, vertices with a superscript represent virtual vertices. For example, 1^2 is a virtual vertex of node 1 associated with child node 2. Left: the graph is separated into three biconnected components: 0^1 and 1; 1^2, 2, and 3; 3^4, 4, 5, and 6. Center: Since 3 is still detached from its virtual vertex 3^4, the component can flip freely. Right: When the edge from 1^2 to 4 is embedded, the biconnected components rooted at 1^2 and 3^4 merge and vertex 3 merges with its virtual vertex. Figure from [7].

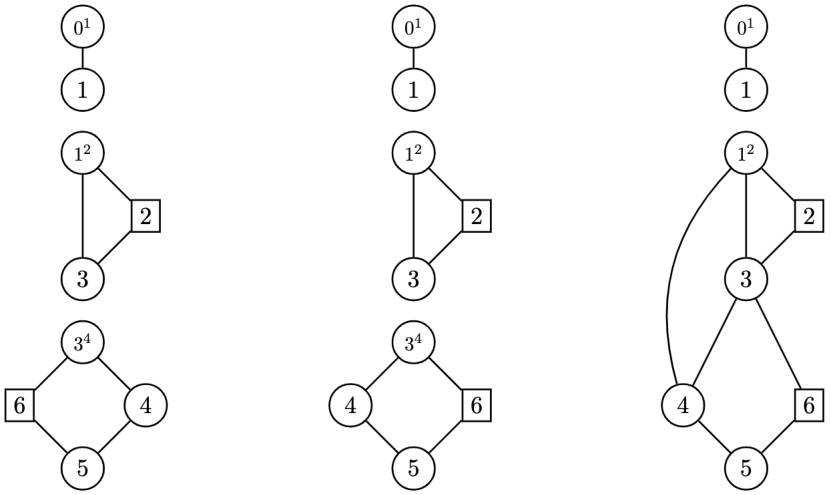
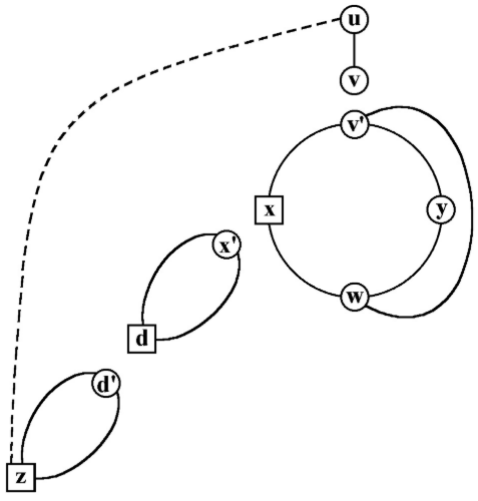


Figure 1.3: In this example, the square nodes are considered externally active during the processing of vertex v. z is marked externally active because it is incident to the forward edge originating from u (ancestor of v). Consequently, d and x are considered externally active as well because they have a descendant biconnected component that contains an externally active vertex. Figure from [5].



embedding after vertex v has processed. To ensure that future embeddings can be successful, the Boyer-Myrvold algorithm maintains the invariant that all externally active vertices re-

main on the external face at all times. That is, the internal faces of the graph already have a complete embedding and any unembedded forward edges will eventually be embedded on the external face of the existing graph. We keep all externally active vertices on the external face so that any future forward edges can reach the vertex without crossing other edges. Each time the Boyer-Myrvold algorithm embeds a single forward edge, it forms a new internal face consisting of the recently embedded forward edge and the existing edges on the external face (as shown in Figure 1.2). The new forward edge becomes a part of the new external face of the component. The *lowpoint* of a vertex is a piece of metadata to help determine the order of embedding and whether a vertex is externally active. The lowpoint is defined as the DFS ancestor of least DFI that can be reached by a path of zero or more descendant DFS tree edges plus one back edge and it can be calculated in a quick traversal during the preprocessing step.

Figure 1.4: An outline of the Boyer-Myrvold algorithm as described in [5].

```
(1) Perform depth first search and lowpoint calculations for G.
(2) Create a DFS child list for each vertex, sorted by child lowpoint.
(3) For each vertex v from DFI = n - 1 to 0
(4)     for each DFS child c of v in G
(5)         Embed tree edge (v^c, c) as a biconnected component in G
(6)     for each back edge of G incident to v and a descendant w
(7)         Walkup(G, v, w)
(8)     for each DFS child c of v in G
(9)         Walkdown(G, v^c)
(10)        if Walkdown fails
(11)            return NONPLANAR
(12) return PLANAR
```

To generate an embedding, the Boyer-Myrvold operates in reverse order, starting with the vertices of highest DFS index, which are typically the leaves or vertices in the lowest part of the tree (see line 3 of Figure 1.4). For each vertex v, the algorithm embeds all forward edges from v. For each descendant w where (v, w) is a forward edge, the algorithm finds a path from w to v by "walking up" the external face of the biconnected components (refer to lines 6-7 of Figure 1.4). During this walk-up, it will mark the roots of biconnected components that it visits as *pertinent*. Then, after a walk-up from all descendants w where (v, w) is a forward edge, the algorithm will perform a walk-down starting at v by visiting all pertinent components (see lines 8-11 of Figure 1.4). During this walk-down, the traversal is not permitted to bypass any externally active vertices. Because each component is biconnected, there should always be two possible paths from the root. If there exists an externally active vertex on one side, the traversal will simply choose the other side to reach its target vertex. And if both sides are blocked, then the graph is not planar. Once the walk-down from v reaches a target vertex w, it can embed the edge (v, w) and merge all biconnected components on the path from v to w, forming a internal face of the graph. During the traversal, the algorithm keeps track of which side of each biconnected component is traversed and in cases where the traversal changes from one side to another, the biconnected component needs to be flipped.

After embedding all pending edges, the algorithm should terminate and return a planar embedding. In an optimal implementation (using techniques such as lazy orientation of flippable components), the Boyer-Myrvold algorithm runs in linear time. During the walk-up and walk-down methods, traversal costs are directly proportional to the size of the faces formed by embedding the forward edges. Specifically, any edges traversed during these processes become embedded as part of new internal faces within the graph. Consequently, the traversal cost during walk-up and walk-down is directly proportional to the total number of edges in the graph. Similarly, the loops invoked in lines 4-11 of Figure 1.4 are associated with forward edges and tree edges and therefore incur a constant cost per embedded edge [5].

1.3.3 Parallelization Attempts

There have been few attempts to parallelize existing planarity testing algorithms.

Parallel open ear decomposition is a technique discussed in a paper by Ramachandran that may have applications to planarity testing [8]. However, the algorithm suggested in the paper is too complicated for most readers to understand and as far as we know, there has been no practical implementation. Later, Bader and Sreshta present a parallel algorithm based off of the PQ-tree data structure, but do not provide an implementation either [6].

Another work that explores the parallelization of planarity testing algorithms is Ingram's thesis, which seeks to parallelize the Boyer-Myrvold algorithm [7]. The general idea is a divide-and-conquer algorithm which sorts the edges and divides them evenly among worker threads. Each worker thread runs a sequential Boyer-Myrvold algorithm on the resulting subgraph. The challenging part of the algorithm lies within the merge step when subgraphs are combined. Since the edges are divided among worker threads, there are bound to be edges that connect vertices that are present in different workers. Ingram's thesis provides a high-level summary of how to embed these pending edges between subgraphs and even discusses the challenges of submerging (where one subgraph is completely embedded within another). However, the thesis omits a number of details (especially when it comes to the submerging process) and do not provide a working implementation of the algorithm.

1.4 Overview

In the following chapters, we will outline two new parallel algorithms for planarity testing, both derived from the Boyer-Myrvold algorithm. In Chapter 2, we present a divide-and-conquer approach, where the graph's edges are evenly distributed among worker threads. Each thread independently executes the sequential Boyer-Myrvold algorithm on its designated subgraph. Then, pairs of subgraphs are merged by embedding the edges between subgraphs with modified walk-up and walk-down methods. The primary challenge of the divide-and-conquer approach is the merge step as determining the relative positions of subgraphs is a complicated and difficult process. In Chapter 3, we describe the design and implementation of a new and simpler parallel algorithm. This algorithm modifies the Boyer-

Myrvold algorithm by processing vertices in layers from the bottom-up (rather than sequentially by reverse DFI order). The computation in each layer is parallelized. On planar graphs, this algorithm achieves 2.4–2.7 times speedup over the sequential algorithm when run on 16 cores. On non-planar graphs, the performance gain is even more significant, with speedups ranging from 9 to 22 times on 16 cores. We conduct an analysis of the algorithm's components, identify primary performance bottlenecks, and explore various parameters and their tuning for optimization. Finally, Chapter 4 provides a synthesis of our thesis work and outlines potential future research directions along with associated challenges.

Chapter 2

Divide-and-Conquer Algorithm

Our initial approach to parallelization is to build off of Ingram's divide-and-conquer algorithm [7]. The basic steps of the algorithm are to: 1. divide the edges among worker threads; 2. run a sequential Boyer-Myrvold algorithm on the subgraph assigned to each worker in parallel; 3. merge the embedded subgraphs. The correctness of the sequential Boyer-Myrvold algorithm depends on embedding the edges in a specific order and when transitioning to a parallel algorithm, we need to make adjustments to account for the dependencies across subgraphs. Section 2.1 details how the edges are split among subgraphs. Section 2.2 discusses temporary vertices which help define how subgraphs are positioned relative to one another. Section 2.3 explains how we modify the walk up and walk down procedures to embed edges that span two different subgraphs.

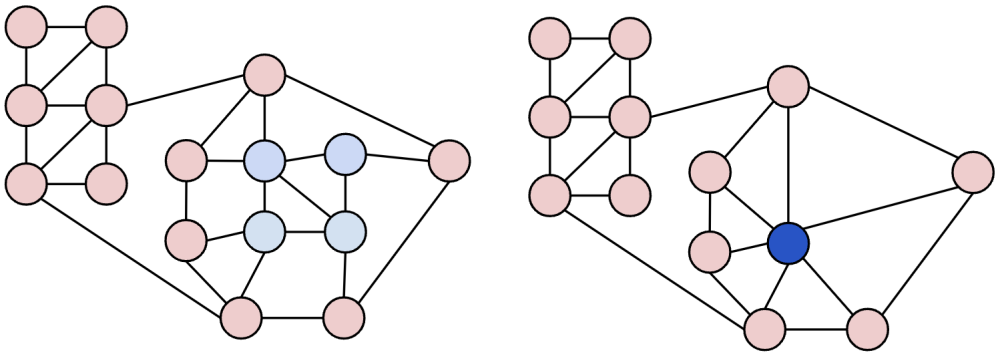
2.1 Initialization

Similar to Ingram's divide-and-conquer algorithm, we begin by labelling vertices with their DFI and splitting the edges evenly among the t worker threads. (From here on out, the notation vertex v will denote the vertex with DFI of v.) For a given thread i, G_i is the subgraph associated with the thread. To mimic the order of embedding in the Boyer-Myrvold algorithm, for edges (v, w) where $v < w$, the edges are first sorted by descending order of v and ascending order of w before they are divided among the workers [7]. G_i begins as a graph with no vertices, but as edges are allocated to thread i, vertices associated with tree edges are gradually added to G_i, resulting in a set of vertices that are present in G_i. An edge (v, w), allocated to thread i, is marked as a pending edge "if w is not a child of v in the DFS tree AND v or w is not already present in G_i" [7]. Pending edges are not embedded during the Boyer-Myrvold execution on the subgraph but rather during the merge step. Each thread i runs the Boyer-Myrvold algorithm on the subgraph G_i in parallel, and during the merge step, pairs of subgraphs are combined by embedding the pending edges from one subgraph to another.

2.2 Temporary Vertices

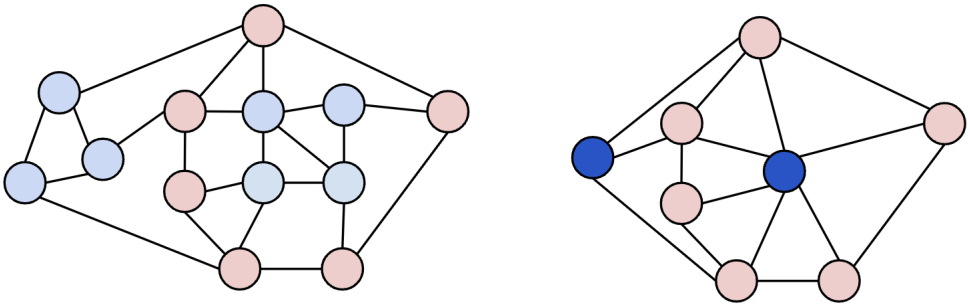
The primary challenge in this divide-and-conquer algorithm is figuring out how subgraphs should be positioned and connected relative to each other. In particular, it is possible for subgraphs to be submerged within one another and we cannot simply connect them by embedding edges on the external face. In the algorithm, we must quickly identify which internal face a particular subgraph will be submerged in. A key component to facilitate the process of positioning subgraphs relative to each other is the construction of temporary vertices. Given two subgraphs G_i and G_{i+1} that will be merged eventually, Ingram's algorithm constructs **one** temporary vertex to represent subgraph G_{i+1} [7]. As shown in Figure 2.1, the temporary vertex will be created at the initialization step in subgraph G_i to represent subgraph G_{i+1} and any pending edges between subgraph G_i and G_{i+1} will be embedded to the temporary vertex during the initial Boyer-Myrvold embedding on subgraph G_i. By incorporating the temporary vertex as part of the embedding, the connections between subgraph G_i and subgraph G_{i+1} will be prelimarily embedded. During the merge step, the attachment edges to the temporary vertex will be resolved one by one as they are replaced by the embedding of the underlying pending edge.

Figure 2.1: Left: In this example, the component containing the light blue vertices is embedded within the face of the red subgraph. Right: We replace the component with the temporary vertex (in dark blue) and attach all pending edges to temporary vertex instead. Figure from [7].



While Ingram's idea of temporary vertices provides a baseline idea for our divide-and-conquer approach, **we have found that the claim that there should only be one temporary vertex for each subgraph is incorrect**. As we can see in Figure 2.2, there are cases where not all of the components of subgraph G_{i+1} are embedded in the same face of subgraph G_i. This case would require the construction of multiple temporary vertices. We define a missing child of graph G_i to be a child c of node v such that v is in G_i but c is not in G_i. We can show that the number of temporary vertices in G_i is equal to the number of missing children in G_i via a series of proofs.

Figure 2.2: Left: the red and blue vertices each below to a different subgraph. (This is possible if the components of the blue subgraph represent the two descendant subtrees of the vertices in the red subgraph.) Right: we construct two temporary vertices (in dark blue): one for the component on the outside and one for the component on the inside.



The forward edge property is the principle that in a DFS tree, the only edges that exist are tree edges or edges between an ancestor and descendant and there will never be edges between two separate child components of a single node. This is an important property that has implications not only in this divide-and-conquer algorithm, but in the parallelization by layer approach as well.

Lemma 1 (The Forward Edge Property)**.** *If there is an edge (v,w) where $v < w$, v must be an ancestor of w.*

Proof. Let us consider the edge (v,w) where $v < w$. Since $v < w$, we know that in the DFS tree, v was added first. Based on the DFS algorithm, at vertex v, we loop through its adjacency list. For each edge (v,c), if vertex c hasn't been added to the tree yet, we add tree edge (v,c) and recurse on vertex c. In this case, there are two possibilities when we reach vertex w in the adjacency list:

1. It hasn't been added to the tree yet. In this case, we add (v,w) as a tree edge and now w is a direct child of v.

2. It has already been added to the DFS tree. In this case, it must have been added during some other iteration in the adjacency list of v. Thus, it must be the descendant of some other child of v, making it a descendant of v as well.

In both cases, v is an ancestor of w. □

For the following lemmas, let us consider a given vertex v in subgraph G. Consider children c_i of v where $v \in G$ and $c_i \notin G$.

Lemma 2. *Let us consider children c_i and c_j of v as described above. There cannot exist edges between the components rooted at c_i and c_j.*

Proof. Without loss of generality, assume $c_i < c_j$. Assume for the sake of contradiction that there exists an edge $(d_i,\ d_j)$ where d_i is a descendant of c_i and d_j is a descendant of c_j.

Based on the DFS algorithm, we know that all descendants of c_i are numbered before c_j, so we have that $d_i < d_j$. From Lemma 1, d_i must be an ancestor of d_j. If d_i is an ancestor of d_j and a descendant of c_i, then c_i must be an ancestor of d_j. By definition, c_j is also an ancestor of d_j. The only way both c_i and c_j can be ancestors of d_j is if c_i is an ancestor of c_j. However, c_i and c_j are both direct children of v so c_i cannot be an ancestor of c_j and this is a contradiction. Thus, the edge (d_i, d_j) cannot exist and there cannot be edges between the components rooted at c_i and c_j. □

Lemma 3. *The component rooted at a child vertex c must be embedded in a single face in G to maintain planarity.*

Proof. Assume for the sake of contradiction that the connected component rooted at c is embedded across multiple faces, including F_0 and F_1 of G. That is, there must exist descendants d_i and d_j of c such that d_i is in face F_0 and d_j is in face F_1 of G. Given that both d_i and d_j are both descendants of c, there must exist a path from d_i to d_j through c or a descendant of c. By definition of a face, there must be one or more edges between faces F_0 and F_1. For a path to get from d_i to d_j, it must cross one of these edges between the two faces. Unless one of the vertices along the path is a vertex in both faces, this would violate planarity. However, c is not in G and none of c's descendants are in G (due to the division of edges in Step 2 of the algorithm). Therefore, none of the vertices along the path from d_i to d_j can be part of F_0 or F_1. This means that planarity *must* be violated if d_i and d_j were in separate faces of G. Thus, all the descendants of child c must be embedded within a single face of G to maintain planarity. □

Theorem 4. *Each component connected to a child c of v (as described above) can be represented as a single temporary vertex.*

Proof. From Lemma 2, we know that there do not exist any edges between the components c_i and c_j rooted at distinct children c_i and c_j of v. From Lemma 3, we know that the entire component c_i must be embedded in a single face in G and same with the component rooted at c_j. Thus, for any given child c_i of v, we can represent its component with a temporary vertex t_i and embed it in any face adjacent to v. □

Therefore, as an initialization step to creating each subgraph, we must construct a temporary vertex for each missing child of the subgraph.

2.3 Embedding Pending Edges

During the merge step, we must embed all pending edges from one subgraph to another. To embed pending edges to temporary vertices, we iterate through the pending edges in the initialized order (descending v, ascending w) and embed them one by one using a modified version of the walk-up and walk-down methods as in the original Boyer-Myrvold algorithm.

Let us consider the merge of subgraphs G_i and G_{i+1}. In the original Boyer-Myrvold algorithm, the walk-up and walk-down traversals were performed by examining the external face data structure for each vertex to reach each subsequent vertex on the external face. In

the new algorithm, there is no longer the guarantee that the pending edge to be embedded will be on the external face. Instead, we will sometimes traverse an internal face of the subgraph G_i if subgraph G_{i+1} is submerged in subgraph G_i. To find the next node in the traversal of the internal face of subgraph G_i, we can simply examine the planar embedding of subgraph G_i and find the vertices adjacent to the current one in the embedding order.

Consider a pending edge (v, w) from subgraph G_i to G_{i+1} where w is part of a connected component rooted at child vertex c that was represented by temporary vertex T. We can adjust the walk-up procedure by walking up from w with the following four steps.

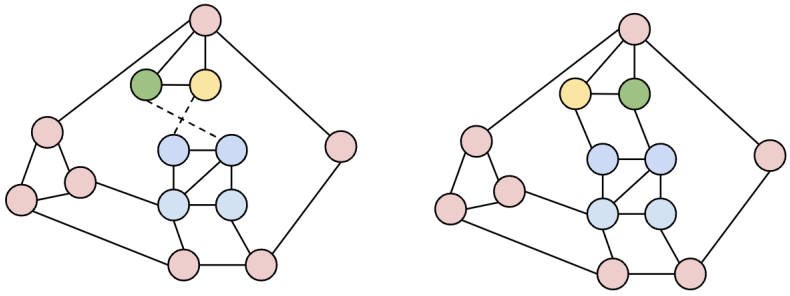
1. Perform a typical walk-up from node w in subgraph G_{i+1} until you reach the missing child c that T represents.
2. Let p be the parent of missing child of c.
3. Use the planar embedding of subgraph G_i to find the vertices adjacent to the tree edge (p, c) on the internal face of subgraph G_i.
4. Continue to use the planar embedding to traverse the internal face of subgraph G_i until you reach v.

We can modify the walk-down procedure in a similar manner. Beyond the method of finding the next node in the traversal, all other aspects of the walk-up and walk-down procedure are the same as in the original Boyer-Myrvold algorithm.

2.4 Flippable Components

Another consideration that was overlooked in previous works is the issue of flippable components. In the original Boyer-Myrvold algorithm, each DFS child was detached from its parent, creating a series of flippable, biconnected components, but as edges were embedded, the flippable components would merge into an internal face of a larger biconnected component and remain fixed for the rest of the algorithm. In the new algorithm, when submerging one subgraph within another (say subgraph $G_i + 1$ in subgraph G_i), there may be flippable components on the internal face of subgraph G_i that need to remain flippable as the pending edges from subgraph G_i to $i+1$ are embedded. However, during the initial run of the Boyer-Myrvold algorithm on subgraph G_i, these flippable components are merged into an internal face of i as the edges to the temporary vertices are attached and therefore are no longer free to flip. Then, we have a scenario like that of Figure 2.3 where resolving the pending edges requires flipping a component on an internal face of i. In order to resolve this issue, the current algorithm will traverse the internal face once per temporary vertex to detach all flippable components. However, this approach may introduce unnecessary computation and, for future work, we will find a more efficient resolution to the issue.

Figure 2.3: The red and blue vertices represent two different subgraphs to be merged, and the green and yellow vertices are part of the red subgraph. Left: In the current orientation, the graph appears to be non-planar. Right: If we flip the green and yellow vertices, we derive a planar graph.



2.5 Algorithm Outline

The algorithm will take as input a set of vertices and edges and output whether the graph is PLANAR or NONPLANAR. The divide-and-conquer algorithm is summarized below.

1. Perform DFS to number all vertices, perform lowpoint calculations and generate the DFS tree.

2. Divide edges as evenly as possible among threads in sorted order. Note that we assign all edges (v, w_i) originated from the same v into the same thread to simplify the merging process slightly.

3. Label tree edges and pending edges and sort the non-pending edges into their respective subgraphs.

   (a) For each child c_i of vertex v where $v \in G$ and $c_i \notin G$ (known as a missing child), create a temporary vertex t_i in G.

   (b) For each pending edge (y, z) where z is in the missing component rooted at c_i, create an attachment edge (y, t_i) and add the attachment edge to the list of edges in the subgraph.

4. In parallel, run the Boyer-Myrvold algorithm on the subgraph in each thread (including temporary vertices). If the algorithm returns NONPLANAR for any of the subgraphs, return NONPLANAR immediately.

5. Merge threads pairwise, by replacing each temporary vertex with its component as follows.

   (a) The temporary vertex should already be connected to the host subgraph by a tree edge since the temporary vertex corresponds to a DFS child of a vertex in the subgraph.

(b) Detach all flippable components from the boundary cycle surrounding the temporary vertex.

(c) Embed all pending edges from the host subgraph to the component one by one in sorted order, using modified walk-up and walk-down methods (discussed in the sections below) and remove corresponding attachment edges to the temporary vertex.

(d) If at any point, the algorithm is unable to embed a pending edge (this may occur if the walk-down method is blocked by externally active vertices), return NONPLANAR.

(e) Remove the temporary vertex from the subgraph if there is only a tree edge connection to the temporary vertex and no more attachment edges.

6. Repeat this process until all subgraphs have been merged. If all edges are embedded successfully, return PLANAR.

2.6 Proof of Correctness

Since there exists proof that the sequential Boyer-Myrvold algorithm is correct, to show that the divide-and-conquer approach is correct, we simply need to prove that the merge process will embed the pending edges correctly.
There are two aspects to this proof: 1. we need to show that any externally active vertices will remain on the external face even after the merge process and 2. we need to show that the walk-down procedure prevents the embedding of a pending edge if it would cross with any other edges.

Theorem 5. *If a vertex is externally active, it will remain on the external face after the merge process.*

Proof. Let us consider an externally active vertex in the inner component. If there exists such a vertex, then the temporary vertex representing the missing component would be marked externally active during Step 4 and the temporary vertex would be embedded on the external face of G. As the temporary vertex gets replaced by the inner component, any externally active vertices in the inner component remain on the external face of the inner component as dictated by the walk down procedure in Section 2.3. Since the external face of the temporary vertex is the external face of G, the external face of the missing component is also the external face of G. Thus, any externally active vertices being embedded will remain on the external face. □

Theorem 6. *A pending edge will only be embedded if it doesn't cross with any other existing edges.*

Proof. Let us consider a pending edge (v, w) where v is a vertex in subgraph G_i and w is a vertex in subgraph $i+1$. Assume that the missing component associated with w was rooted at missing child c and represented by temporary vertex T during the embedding in Step 4.

Assume that T was embedded in an internal face of subgraph G_i (the proof should be valid regardless of whether T is on the external face or internal face of subgraph G_i).

According to the walk-up and walk-down algorithm, a pending edge (v, w) will be embedded if and only if we can traverse from v to w without being blocked by externally active vertices on both sides. The pending edge can only be blocked if there exist two or more externally active vertices on the boundary cycle. A vertex is only externally active if it is incident to a pending edge that has not yet been embedded.

Let us consider any two externally active vertices x and y that could potentially block the traversal from v to w. For x and y to be externally active, the following properties must hold.

1. x and y must both be incident to a pending edge to T. According to the Forward Edge Property, this means they must be ancestors of c.

2. The DFI of x and y must be lower than that of v, since x and y are still active and pending edges are embedded in sorted order. Assume without loss of generality that x has a lower DFI than y.

Since (v, w) is a forward edge, there must exist a path of descendant tree edges from v to c. Since x and y are both ancestors of the same missing child, there must exist a path of descendant tree edges from x to y.
There are two possible cases for the placement of x and y, relative to v and c.

1. If the path from v to x to c is not equal to the path from v to y to c, then the pending edge would have crossed with an existing edge and the algorithm will fail to embed the (v, w) edge and return NONPLANAR.

2. If x and y are on the same side of v on the boundary cycle, then (v, w) can be embedded safely in the algorithm and the pending edge (v, w) doesn't present any immediate conflicts.

□

2.7 Implementation

The implementation of this divide-and-conquer algorithm will be left as future work. Several potential challenges and implementation details must be considered for the divide-and-conquer approach. For example, we must determine how to efficiently detach flippable components before embedding pending edges and we must handle situations where a temporary vertex is involved in multiple merge steps. These challenges necessitate careful design and optimization to ensure the algorithm's effectiveness and scalability. In the next chapter, we will propose a simpler parallel algorithm that we implement.

Chapter 3

Parallelization by Layer

3.1 Algorithm Outline

We propose a new and simple parallel algorithm whose correctness relies on the invariant that, in a planar graph, there are only forward edges in the DFS tree and there are never edges between two child subtrees of a given vertex. This principle gives us the opportunity to create a parallel algorithm where, given a vertex, the planar embeddings for each child can be generated independently and in parallel. The algorithm works as follows.

1. Perform the same initialization steps as in the Boyer-Myrvold algorithm (generate a DFS tree, perform lowpoint calculations, etc.). From this step forward, the notation vertex v will denote the vertex with DFI of v.

2. Divide the DFS tree into layers from the bottom up as follows.

   (a) During the lowpoint calculation, we traverse the vertices in reverse DFI order. We will set the layer number of each vertex during this traversal. We use $l(v)$ to denote the layer number of vertex v.

   (b) We only add a vertex into a layer if it either meets the base case condition or if it is the root of a tree and has no parent. Currently, the base case condition returns true for a vertex v if the subtree rooted at v has size greater than $STOPSIZE$, where $STOPSIZE$ is a tuned parameter.

   (c) For vertices v that are not added into a layer, we set its layer value equal to -1 ($l(v) = -1$).

   (d) To determine the layer number of a particular vertex, we take the maximum layer value among all of its children and add one. That is, $l(p) = \max_{c \in C} l(c) + 1$ where C is the set of p's children.

   (e) After all layer numbers have been assigned, sort the vertices by layer number.

3. Iterate through the layers from the bottom to the top and parallelize the computations at each layer.

(a) In parallel, for each vertex v in the bottom-most layer ($l(v) = 0$), we run the sequential Boyer-Myrvold algorithm on the subtree rooted at v. If the sequential algorithm returns NONPLANAR for any of the subtrees, return NONPLANAR immediately.

(b) For each subsequent layer i (starting at layer 1), we do the following in parallel: for each vertex v, we embed all forward edges from v using the original walk-up and walk-down methods from the Boyer-Myrvold algorithm. If the algorithm is unable to embed any of the forward edges (this may occur if the walk-down method is blocked by externally active vertices), return NONPLANAR.

4. If all edges are embedded successfully, return PLANAR.

3.2 Proof of Correctness

The primary difference between the parallel algorithm and the sequential Boyer-Myrvold algorithm is that the parallel algorithm will process vertices by layer rather than in reverse DFI order. The sequential Boyer-Myrvold algorithm processes vertices in reverse DFI order "so that every unprocessed vertex has a path of unprocessed DFS ancestors leading to the last vertex to be processed, the DFS tree root" [5]. In particular, we care about the ordering of vertices when there are edge dependencies. As proven in Lemma 2, there cannot exist edges between two child subtrees c_i and c_j of a single vertex v. Thus, it would not matter if c_i was processed before c_j or if c_j was processed before c_i. However, c_i and c_j must both be processed before v as there might be unembedded forward edges between v and the components rooted at c_i and c_j. To show that the parallel algorithm is correct, we will prove that the order in which vertices is processed mirrors that of the sequential Boyer-Myrvold algorithm.

Lemma 7. *For every edge (v, w) in G, where $v < w$ (recall that, in our notation, vertex v denotes the vertex with DFI of v), if v is assigned a non-negative layer number, v will be assigned a layer number that is strictly greater than w's layer number. That is, $l(v) > l(w)$.*

Proof. According to Lemma 1 (the forward edge property), v must be an ancestor of w. By this definition, either v must be a parent of w or v must be an ancestor of a parent of w. Step 2 of the algorithm guarantees that every parent vertex will have a layer number that is strictly greater than every single one of its children. If v is the parent of w, v will have a layer number strictly greater than w. If v is an ancestor of w's parent, then there must exist one or more vertices p_i such that p_1 is w's parent, p_2 is p_1's parent, etc., and v is p_k's parent. Because parent vertices are guaranteed to have a layer number strictly higher than its children, we have that $l(w) < l(p_1) < l(p_2) < \cdots < l(p_k) < l(v)$. Thus, $l(w) < l(v)$. □

Lemma 8. *For every edge (v, w) in G, where $v < w$, if v is not assigned to a layer ($l(v) = -1$), then w is not assigned to a layer ($l(w) = -1$).*

Proof. Step 2 implies that a vertex will *not* be assigned to a layer if it fails to meets the base case condition *and* if it has a parent. If v is not assigned to a layer, that means the subtree rooted at v has size less than or equal to $STOPSIZE$. According to Lemma 1, v

is an ancestor of w. Thus, w must have a parent and the subtree rooted at w must have size less than the subtree rooted at v. Thus, the subtree rooted at w would still be smaller than $STOPSIZE$ and w satisfies both the criteria for unassigned vertices. This means $l(w) = -1$. □

Theorem 9. *For a given edge (v, w), where $v < w$, the parallel algorithm will process vertices v and w in the same order as they would be processed in the sequential Boyer-Myrvold algorithm.*

Proof. The sequential Boyer-Myrvold processes vertices in reverse DFI order. This means that w would be processed before v. In the parallel algorithm, there are two possibilities: either v is assigned a non-negative layer number or it is assigned layer -1. If v is assigned a non-negative layer number, by Lemma 3, $l(v) > l(w)$. In Step 3 of the algorithm, layers are processed in order from 0 on up. Then, in this scenario, w would be processed before v. If v is assigned layer number -1, it will be embedded during the sequential Boyer-Myrvold algorithm. By Lemma 7, w will be assigned layer number -1 as well. Then, since v is an ancestor of w, v and w are part of the same subtree and will both be processed during the sequential Boyer-Myrvold algorithm on that subtree in Step 3. Thus, in both cases, w will be processed before v, mirroring the order in the sequential Boyer-Myrvold algorithm. □

3.3 Theoretical Bounds

The new parallel algorithm has an $O(n)$ work bound overall. We can derive this bound by analyzing the work at each step of the algorithm.

1. In Step 1, the DFS traversal runs in $O(n)$.

2. In Step 2, the computation first calculates lowpoint values and sets layer numbers. The lowpoint calculation requires $O(n)$ work (as discussed in [5]) and since setting layer number is a constant operation during the lowpoint calculation, this step takes $O(n)$ overall. To initialize the layers, the vertices are sorted by layer number via a parallel integer sort which is $O(n)$ work.

3. In Step 3, the work required is equivalent to that of the sequential Boyer-Myrvold algorithm. Each individual vertex is processed in the same manner as in the original sequential algorithm. The amount of work during the walk-up and walk-down methods is proportional to the size of the faces formed by the newly embedded edges. Thus, the embedding process incurs a cost that is linear in the number of edges in the graph.

While the algorithm demonstrates parallelism in practice, the theoretical bound of the span is still $O(n)$. Processing the vertices one layer at a time creates a chain of dependencies whose length is roughly equal to the depth of the DFS tree. Quantifying the span of each layer is challenging due to the variable amount of work required to process a single vertex. Recall that, in the Boyer-Myrvold algorithm, the amount of work to perform walk-up and walk-down methods for a single vertex is proportional to the size of the faces formed by the forward edges emanating from that vertex. If, for example, a vertex v had forward edges to

every descendant w on the external face of the partially embedded graph, it would result in a notably high span.

Regardless, the algorithm is work-efficient and yields substantial speedups in practice as detailed in Section 3.6.

3.4 Implementation

The algorithm was implemented in C++17 and the development and testing were primarily conducted on an Ubuntu virtual machine, equipped with a Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz processor, with 24 cores and 96 GB of RAM. Later on, we transitioned to a more powerful testing environment to evaluate the algorithm's performance under significantly higher computational loads. In particular, we performed trial runs on large graphs with tens of millions of vertices. These additional trials required greater computational resources than our initial setup could provide and were run on a 4-socket machine (2.1 GHz Intel Xeon Platinum 8176 CPUs) with 1.5 TB of DRAM and a total of 112 cores and 2-way hyper-threading. The g++ compiler (version 11.4.0) was used to compile the source code.

For parallelization, we chose to use the ParlayLib library, a C++ library for shared-memory parallel computations [10]. ParlayLib contains a number of data types suited for parallel computations as well as a suite of parallel routines and algorithms. Using ParlayLib's work-stealing scheduler outperforms other libraries such as OpenMP in terms of code execution speed. In addition, setting up ParlayLib is remarkably straightforward, making it a more convenient option compared to other libraries.

The primary source of parallelism in the algorithm is that the vertices in each layer can be processed in parallel. We used a parallel for loop to parallelize this computation. In the initialization step, we set up the layers by using the integer sort function to sort vertices by layer number and then the filter function to calculate offset indices for where each layer ends.

In the process of setting up layers, we discovered that there are two different ways to generate layers of the tree and the two methods have drastically different performance results. At first, we attempted to generate layers from top down — the first layer would be the root of the tree, the second layer contained the children of the root, the third layer was composed of the children of the second layer, etc. When we tested the performance of the implementation with top down layers, the parallel algorithm did not produce any speedups - in fact, it even ran slower than the sequential algorithm sometimes. On the second attempt, we switched to bottom up layer generation (as described in Section 3.1 above) - the first layer would start as close to the leaves as possible, the second layer consisted of the parents of the first layer, the third layer contained the parents of the second layer, etc. The bottom up layering performed much better and it achieved speedup over the sequential algorithm. In a balanced tree, these two layering methods would produce the exact same result, but, usually in practice, the DFS tree in the algorithm will not be balanced. The top down method produces a large number of layers with just a few vertices in each while the bottom

up method produces a few layers with a large number of vertices and then many layers with only one vertex. Parallel for loops incur quite a bit of overhead during runtime and the larger the parallel for loop is, the more the overhead gets amortized out and the more benefits there are to parallelizing the computation. The parallel for loops generated by the top down approach are not large enough to offset the overhead, but the parallel for loops generated by the bottom up approach are.

Similarly, for smaller graphs, the overhead of configuring parallel computation outweighs the marginal performance gain achieved through parallelization. In these base cases, it is more efficient to execute the sequential algorithm rather than its parallel counterpart. Notably, in Step 2, we employ a filtering mechanism to identify vertices with subtree sizes surpassing a predefined threshold ($STOPSIZE$). For the first vertices that exceed this threshold (layer 0), we will execute the sequential Boyer-Myrvold algorithm on their respective subtrees (see Step 3 of the algorithm). If a vertex does not meet the threshold, it will be processed as part of a sequential execution. In Section 3.8, we discuss the tuning of this threshold parameter. Another implementation detail to account for the overhead of parallel computation involves restricting the use of parallel for loops to cases where the number of iterations surpasses a specific threshold. If the number of iterations does not meet the threshold, we execute a simple sequential for loop instead to reduce the overhead. We will conduct an analysis and evaluation of this parameter in Section 3.8 as well.

3.5 Test Graphs

To generate test graphs, we used the Python NetworkX library. For non-planar graphs, we generated random graphs with n vertices and $2n$ edges where n ranged from 1000 to 100000. Generating planar graphs was a little more tricky since most random graphs are non-planar and NetworkX doesn't have a function to generate planar graphs. We used Delaunay triangulation to generate planar graphs that each had n vertices and roughly $3n$ edges. Since the planarity testing algorithm will run to completion on planar graphs (it may return early on non-planar graphs), we did most of the performance testing on planar graphs and generated planar graphs with order of magnitude ranging from 1000 vertices to 10 million vertices.

In the following sections, we will continue to reference a few datasets:

1. Small graphs. These graphs have size ranging from 10,000 vertices to 105,000 vertices. By default, we conduct testing on the planar graphs in this dataset. However, there will be an analysis on the non-planar graphs in this dataset as well.

2. Medium graphs. These graphs have size ranging from 500k vertices to 1.3 million vertices. All graphs in this dataset are planar.

3. Large graphs. These graphs have size ranging from 10 million to 90 million vertices. All graphs in this dataset are planar.

3.6 Results

To test the performance of the new algorithm, we ran trials on a tuned implementation of the algorithm with 1, 2, 4, 8, and 16 threads. (Parameter tuning will be discussed in more detail in Section 3.8.) We ran 4 trials of each experiment, discarded the first trial to account for cold start times and took the median embedding time of the remaining 3 trials. The overall performance of the parallel algorithm surpassed that of the sequential counterpart, demonstrating substantial speedups across different graph sizes and types. Specifically, on small graphs, the embedding portion of the parallel algorithm achieved a speedup of 1.4 to 1.8 times when executed on 16 cores, and, on medium-sized graphs, the speedup ranged from 2.4 to 2.7 times. On small non-planar graphs, the parallel algorithm exhibited an even more remarkable performance, with a 9 to 22 times speedup on 16 cores.

3.6.1 Planar Graphs

Figures 3.1 and 3.2 show the embedding time of the parallel algorithm compared to the sequential algorithm on small and medium planar graphs. We can see in Figure 3.7 that the parallel algorithm achieves significant speedup over the sequential algorithm, but the increase in speedup plateaus at around 8 threads. In Section 3.7, we will discuss bottlenecks that prevent the algorithm from scaling indefinitely with the number of processors. Notice in Figure 3.1 how the "Parallel 1" curve is higher than the "Sequential" curve — this indicates that our parallel algorithm incurs some overhead over the sequential algorithm even in the embedding step.

Figure 3.1: The embedding time of small planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm.

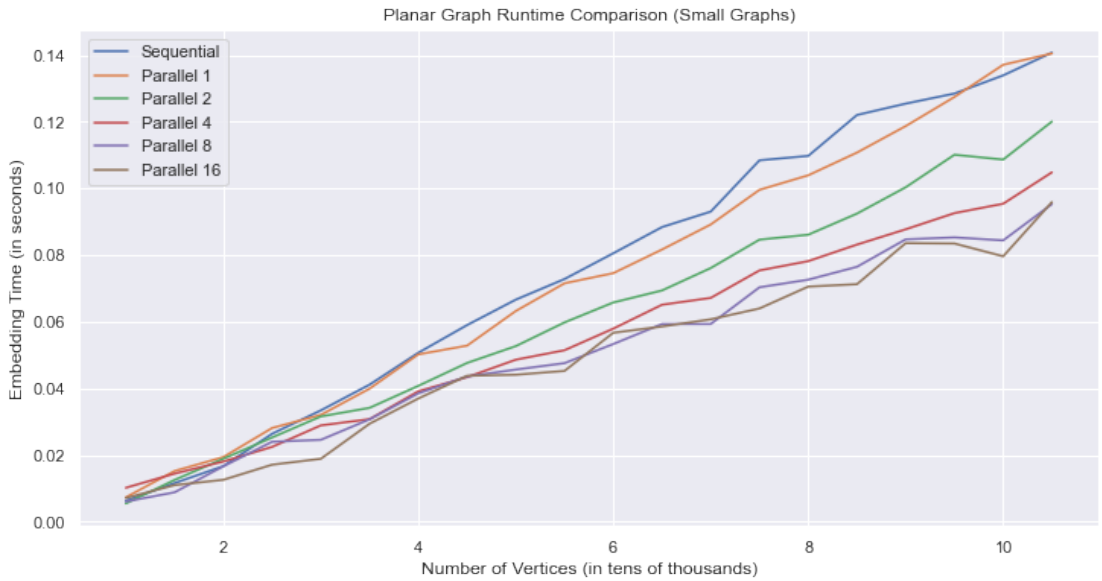


Figure 3.2: The embedding time of medium planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm.

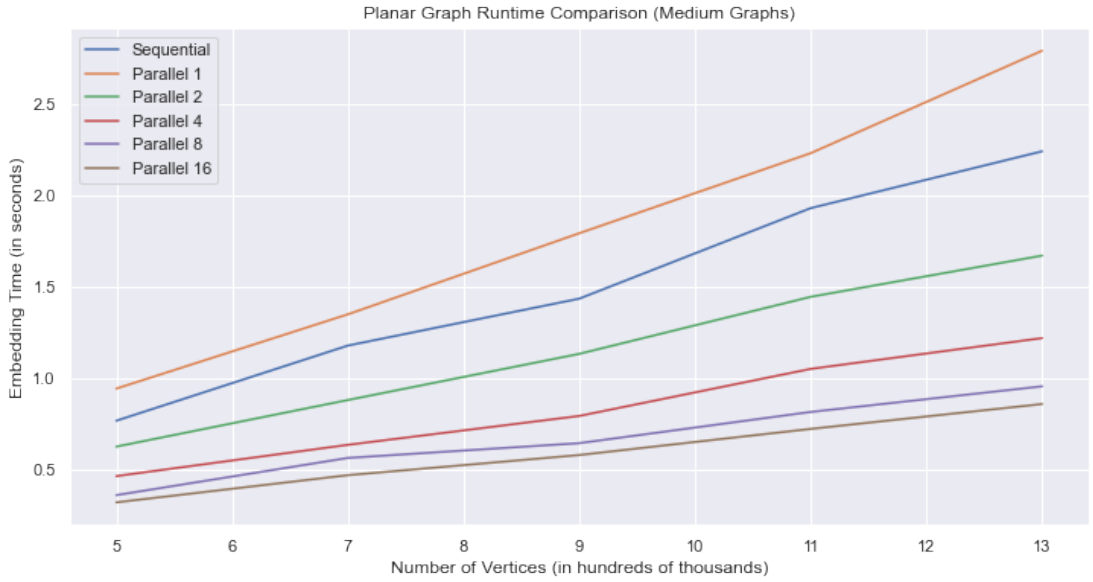
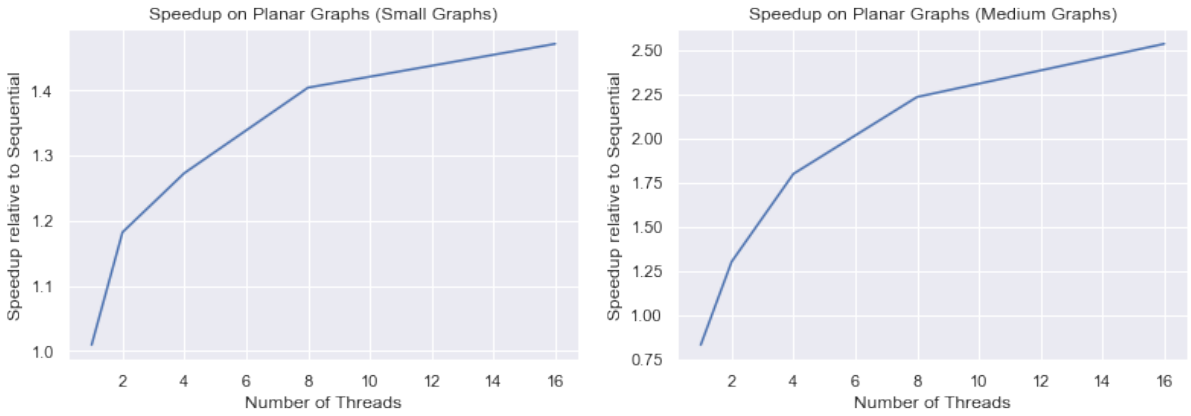
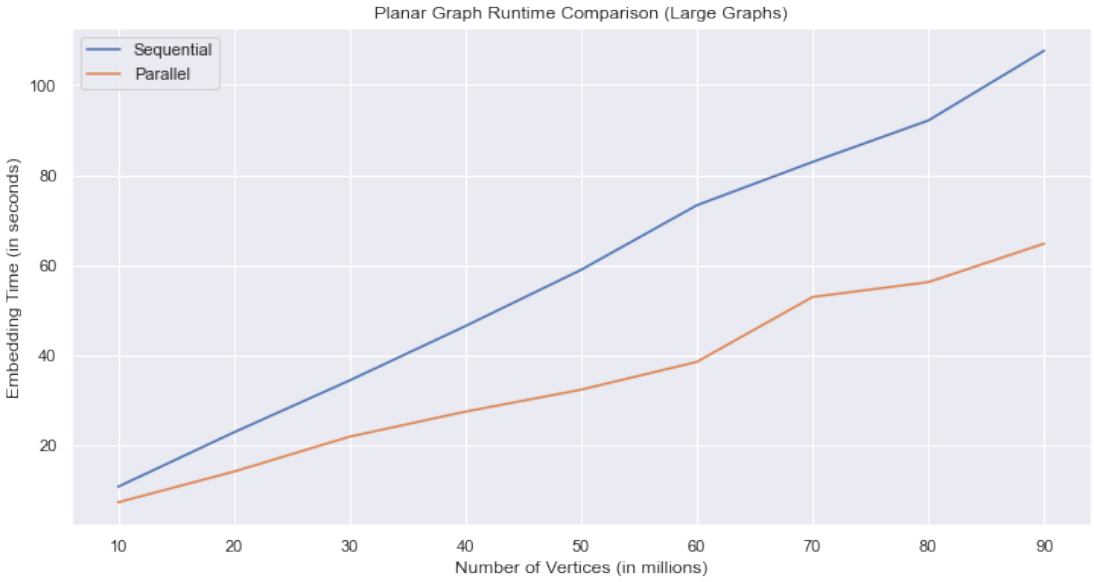


Figure 3.3: The speedup relative to sequential is calculated by taking the runtime of the sequential algorithm and dividing it by the runtime of the parallel algorithm. The values plotted represent the mean taken over all the planar graphs in the small graphs dataset.



In addition, we tested our algorithm on large graphs with tens of millions of vertices. Many practical applications involve processing massive graphs and it was important that our algorithm could perform effectively on large-scale datasets and scale appropriately to meet computational demands. Figure 3.4 shows the performance of the parallel algorithm relative to the sequential on our sample of large graphs. The results demonstrate a significant speedup, with the parallel algorithm outperforming the sequential algorithm by a substantial margin. As we can see in Figure 3.4, with larger graphs, even a 1.5 times speedup could cut computation time down by tens of seconds.

Figure 3.4: The embedding time of large planar graphs. The blue curve represents the embedding time from the original sequential algorithm.



3.6.2 Non-Planar Graphs

In Figure 3.5 and 3.6, we present results on non-planar graphs. Notice that the parallel algorithm achieves significant speedup over the original Boyer-Myrvold algorithm even with only one core. In both algorithms, when the algorithm encounters a vertex whose forward edge it is not able to embed, it will return immediately. Thus, the speed of detecting non-planar graphs is largely determined by the number of vertices processed before the unembeddable forward edge. In the original sequential Boyer-Myrvold algorithm, the vertices are embedded in reverse DFI order. If the faulty vertex has DFI i, then the number of vertices processed before i would be $n - i$. In the parallel algorithm, we change the order in which vertices are processed by processing vertices in layers by height from the bottom of the tree. This means the number of vertices processed before a given vertex is more dependent on the height of the vertex from the bottom rather than the vertex's DFI. In the random non-planar graphs generated, we have found that in most cases, the faulty vertex is relatively close to the bottom of the tree (for example, in a sample non-planar graph of 105,000 vertices and 210,000 edges, the faulty vertex was in layer 123 from the bottom when the entire tree has depth of around 63k). In cases like these, the original Boyer-Myrvold algorithm would process all vertices below the faulty vertex and all subtrees to the right of the faulty vertex. On the other hand, the new algorithm described above will process all vertices that are in the layer equal to or below the faulty vertex that have a lower DFI. Since the number of processed vertices is largely constrained by height, the parallel algorithm processes much fewer vertices than the sequential one. For example, the previously mentioned non-planar graph of 105,000 vertices and 210,000 edges, the sequential algorithm would process 28k vertices while the parallel algorithm would only process 4500 vertices. Thus, even with one core, the new algorithm significantly improves the runtime of planarity testing on non-planar graphs due to its more efficient ordering.

With multiple cores, we see even more improvement as the bottom-most layers of the tree contain the most opportunity for parallelization and the parallel for loops of those layers are the largest out of the entire tree.

Figure 3.5: The embedding time of small non-planar graphs, run on 1, 2, 4, 8, and 16 threads. The blue curve represents the embedding time from the original sequential algorithm.

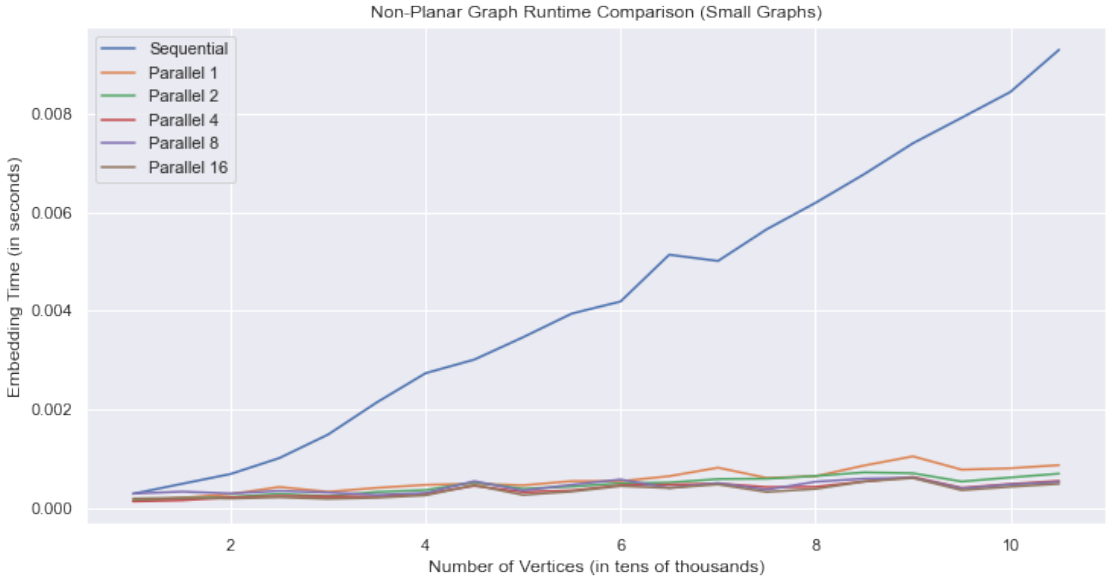
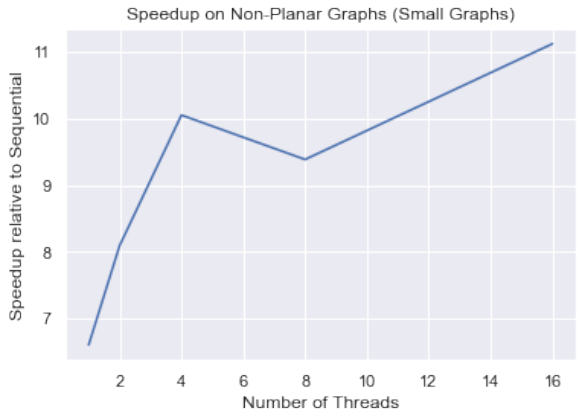


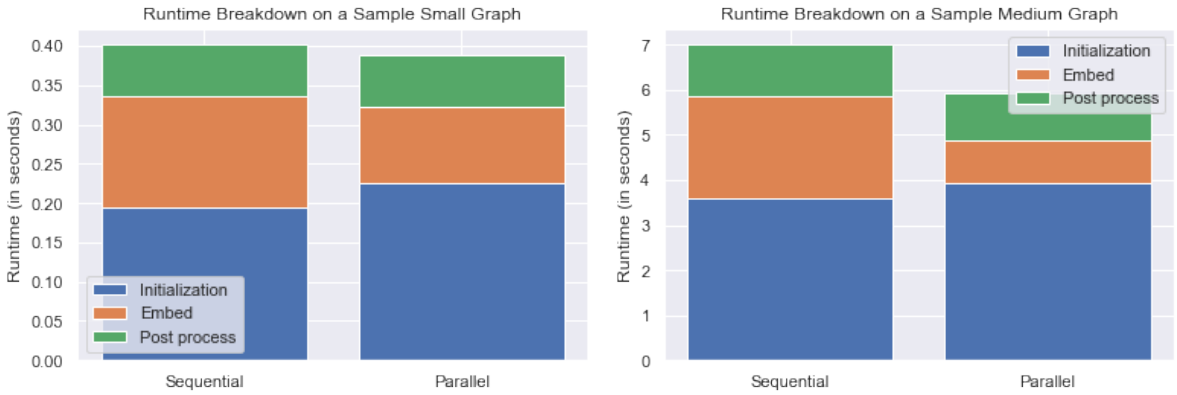
Figure 3.6: The speedup is calculated by taking the runtime of the sequential algorithm and dividing it by the runtime of the parallel algorithm. The values plotted represent the mean taken over all the non-planar graphs in the small graphs dataset.



3.7 Performance Bottlenecks

Even though the parallel algorithm exhibits speedup over the sequential Boyer-Myrvold algorithm, there are some significant bottlenecks that prevent the algorithm from improving speedup further. In Figure 3.7, we breakdown the runtime into initialization, embedding, and post-process components. As we can see, while the embedding process performs much better in the parallel setting, the initialization step incurs some overhead and the performance of the parallel algorithm seems to be limited by the initialization step.

Figure 3.7: The breakdown of runtime into initialization, embedding and post-processing steps on two sample graphs. Left: Data from a small graph with 105,000 vertices and about 315,000 edges. Right: Data from a medium graph with 1.3 million vertices and about 3.9 million edges. Note that, in both cases, the parallel algorithm primarily improves the embedding time and incurs some overhead in the initialization step.



3.7.1 Initialization

Even though the embedding process is now parallelized, the initialization step still consists of a DFS traversal and an additional sequential for loop to set lowpoint values. Thus, the overall span of the algorithm is still $O(n)$ and the total runtime of the algorithm is constrained by the initialization step.

The parallel algorithm incurs some overhead in the initialization step (as shown Figure 3.7) as it must initialize and set layers (this step was not present in the sequential algorithm). To set the layers, we create an array of vertices sorted by layer number and an array of offset values that denote where each layer starts and ends. The primary operations involved in this step are running an integer sort to sort the vertices by their layer number and running a filter to derive the offset array. We experimented with another initialization option that involved adding vertices directly to an array of vectors, where the index represented the layer number and each value in the array was a vector of all the vertices in the layer. However, the original method provided better performance overall. In the current setup, the initialization step of the parallel algorithm runs about 10–20% slower than that of the sequential (see Figure 3.7).

3.7.2 Theoretical Bounds

The other primary constraint on the performance of the parallel algorithm is the theoretical bounds. The span of the embedding portion of the algorithm proportional to factors such as the number of layers and the density of forward edges from a given vertex. For a sample planar graph with n vertices, the number of layers is roughly 1/4 to 1/5 times n. In the bottom-most layers, each layer contains a large number of vertices and we can parallelize the work with multiple processors. However, the number of vertices in each layer is monotonically decreasing. In the small graphs dataset, only about 2% of the layers execute in parallel and the parallel layers compose about 8% of the embedding time when executed on 16 cores. (Recall that the branching criteria for the small graphs dataset is tuned to only use parallel for loops on loops with 8 or more iterations.) In the medium graphs dataset, about 19% of the layers execute in parallel and the parallel layers compose about 40% of the embedding time when executed on 16 cores. Even though most of the work is done in the parallel layers (the parallel layers contain of roughly 80% of the vertices), the sequential layers take up the majority of the embedding time. This indicates that the sequential portions of the embedding process serve as a bottleneck on the algorithm.

In an attempt to reduce this bottleneck, we experimented with different methods of generating DFS trees. DFS trees with lower depth would result in less layers and reduce the span of the embedding portion of the algorithm. The two techniques we tried were:

1. Generating a DFS tree by picking the root to be the vertex with the highest degree. We theorized that picking a vertex with a high degree would give us a higher chance of having more subtree children in the DFS tree.

2. Generating 5–10 random DFS trees and picking the tree of the lowest depth.

Neither one of these techniques worked. Picking the highest degree vertex as the root did not affect the depth of the DFS and taking the minimum depth was costly and produced very little improvement in the result.

3.8 Tuning

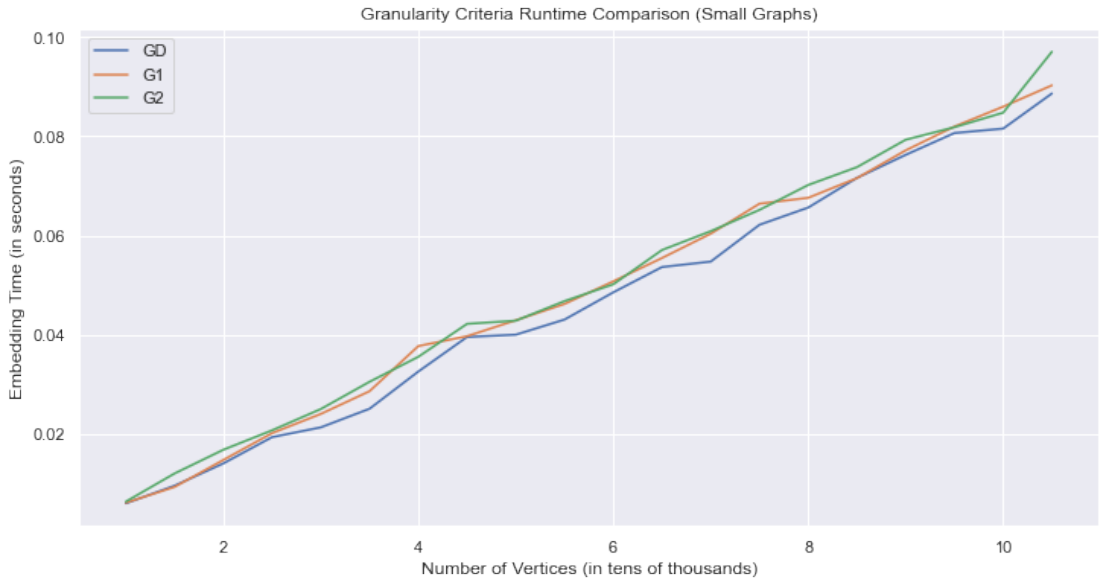
We did all parameter tuning on $n = 4$ threads. To generate the data below, we ran 4 trials of each experiment, discarded the first trial to account for cold startup times and took the median embedding time of the remaining 3 trials.

3.8.1 Granularity

The ParlayLib parallel for loop allows the user to designate a granularity level. According to the documentation [10], "the granularity of a parallel-for loop is a hint to the scheduler that the given number of iterations of the loop should be ran sequentially ... when left to its default value (zero), Parlay's scheduler will estimate a good granularity for the loop itself. The scheduler-estimated value is usually quite good and close to optimal, so it is rarely necessary to manually tune the granularity" [insert citation]. To be sure that we were using

an optimal granularity level, we tested the performance on the default granularity setting as well as granularity value equal to 1 and 2. We only chose to test the granularity value on small values, because many of the parallel for loops in our algorithm have a small number of iterations but we would like to make sure that we are parallelizing those iterations. However, it seems from the Figure 3.8 that the default granularity value is consistently achieves the least embedding time among the 3 options.

Figure 3.8: The embedding time of planar graphs with granularity set to default (GD), 1 (G1) and 2 (G2).



3.8.2 Stopping Criteria

Another parameter to tune for is the stopping criteria: at what point should we stop the parallelizing computation and run the sequential Boyer-Myrvold algorithm on the remaining subtrees? In the parallel algorithm, vertices are only added to a layer if they pass the base case condition or if they have no parents. Currently, the base case condition involves comparing the subtree size at the vertex to a fixed value and if the subtree size is larger than the fixed value, then we will add the vertex to a layer. Otherwise, one of its ancestors will be added to a layer and the vertex will be processed as part of the base case in a sequential Boyer-Myrvold algorithm run.

In Figures 3.9 and 3.10, it does not seem like stopping criteria makes a significant difference to embedding time. But, from here on out, we use a base case size of 500 vertices for small graphs dataset and use a base case of 50 vertices for the medium graphs dataset.

Figure 3.9: The embedding time of small planar graphs where the threshold for the base case was 10 (S10), 50 (S50), 100 (S100), 500 (S500), and 1000 (S1000) vertices.

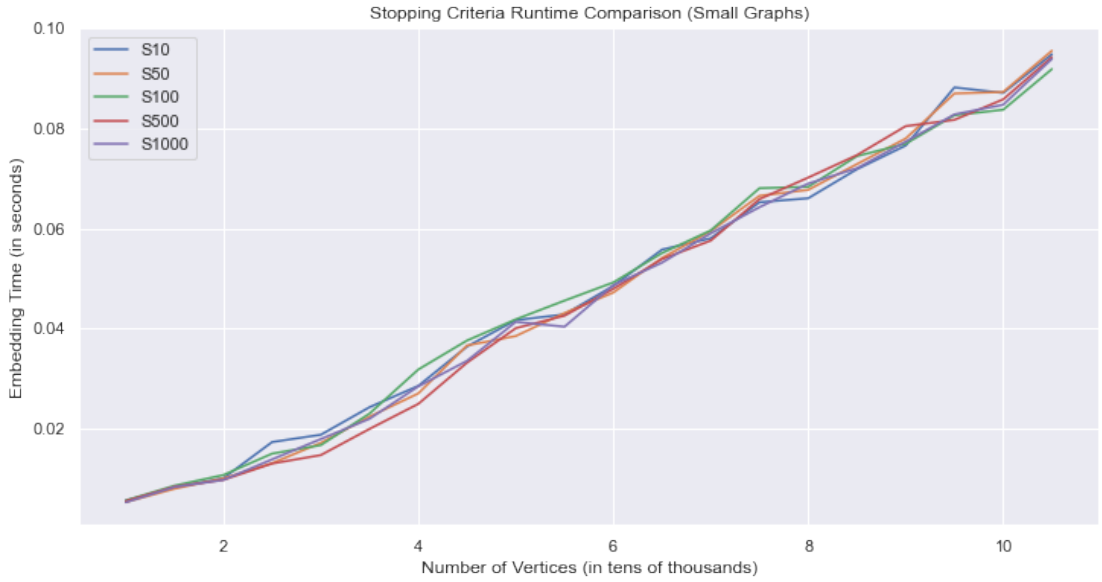
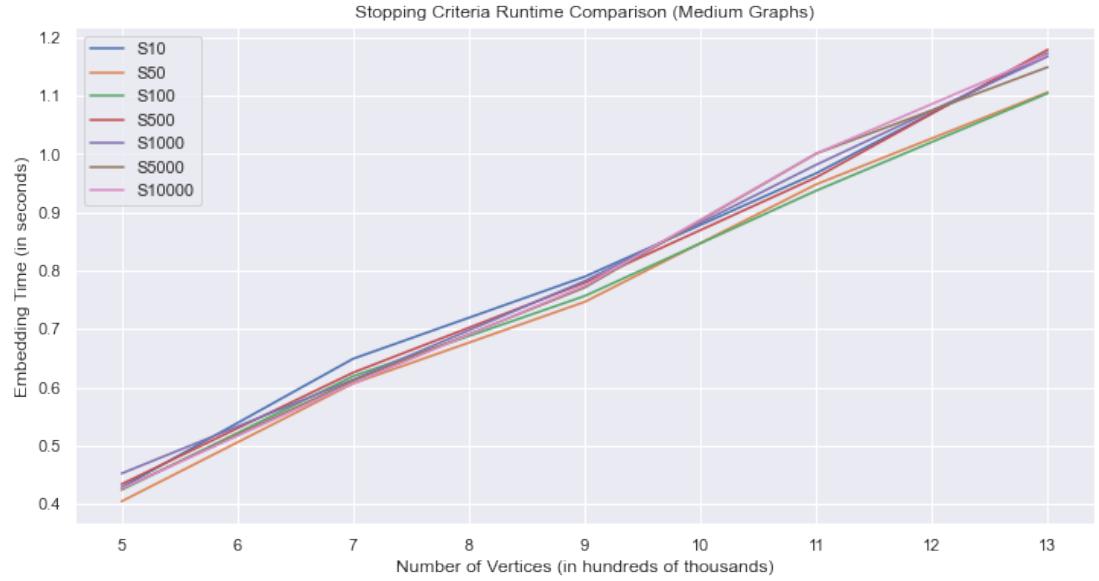


Figure 3.10: The embedding time of medium planar graphs where the threshold for the base case was 10 (S10), 50 (S50), 100 (S100), 500 (S500), 1000 (S1000), 5000 (S5000), and 10000 (S10000) vertices.



3.8.3 Branching Criteria

Since setting up a parallel for loop incurs some overhead, we added a condition for switching from a parallel for loop to a sequential for loop. We call this condition the branching criteria. If the number of iterations in the for loop is equal to or above some fixed threshold, we use a parallel for loop; otherwise, we use a sequential for loop.

For small graphs, we chose to set the branch criteria to 8 or more iterations (see Figure 3.11). For medium graphs, we chose to set the branch criteria to at least two iterations (see Figure 3.12). Because the medium graphs are larger, each iteration requires more work. Even when the loop only has two iterations, the benefits of parallelizing the work overcomes the overhead expenses of setting up the parallel for. The small graphs are much smaller and the benefits of the parallel for loop are only present for larger numbers of iterations.

Figure 3.11: The embedding time of small planar graphs where the threshold for using a parallel for loop is set to 2 (B2), 4 (B4), 8 (B8), and 16 (B16) iterations.

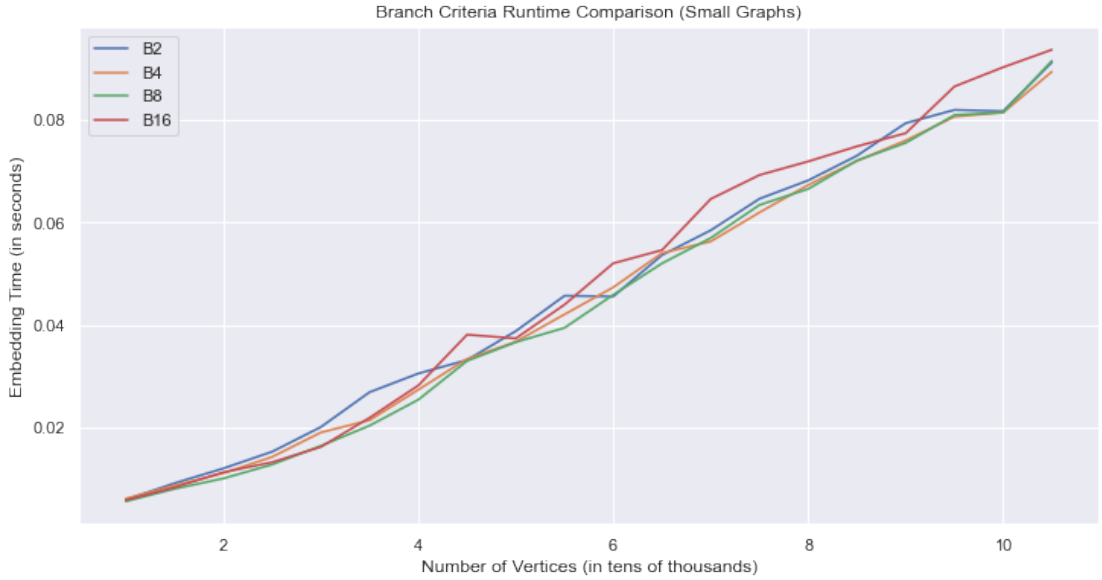
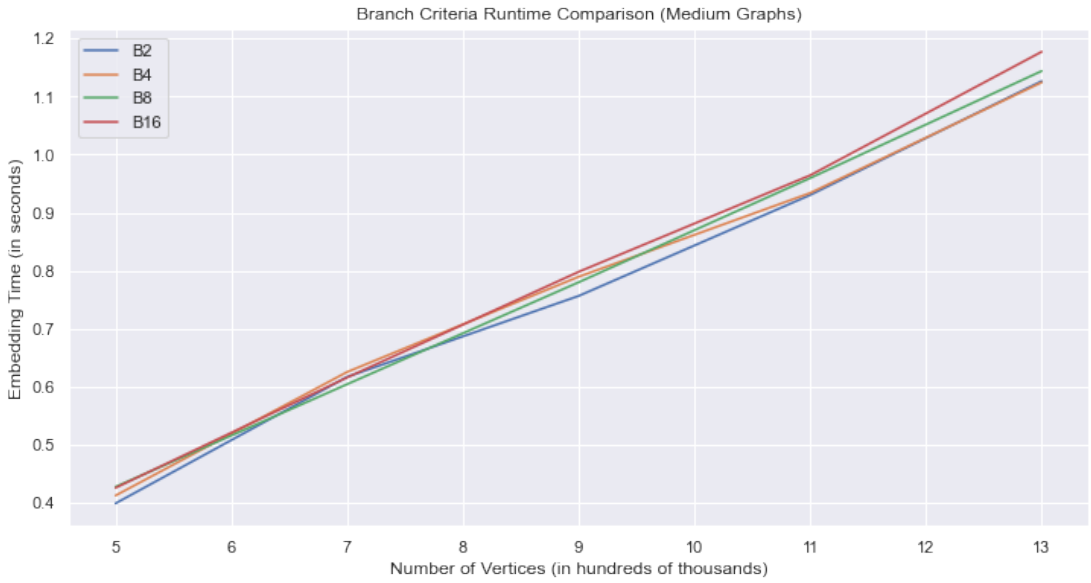


Figure 3.12: The embedding time of medium planar graphs where the threshold for using a parallel for loop is set to 2 (B2), 4 (B4), 8 (B8), and 16 (B16) iterations.



Chapter 4

Conclusion

4.1 Summary

In this thesis, we discussed two new parallel algorithms for planarity testing.

The first algorithm uses a divide-and-conquer method (based on [7]) to split the work among threads. Each thread receives a set of edges, constructs its own subgraph, and runs the sequential Boyer-Myrvold algorithm on its subgraph. We build off of [7]'s idea of using temporary vertices to represent subgraphs and prove that we must construct one temporary vertex per missing child in the subgraph to fully represent where subgraphs will be embedded relative to each other. The most challenging part of the algorithm is the merge step in which pending edges between subgraphs are resolved. We modify the walk-up and walk-down method to allow for a traversal across an internal face and prove that pending edges are correctly embedded by the algorithm. The implementation of this algorithm will be left as future work.

The second algorithm discussed avoids the complex dependencies in the divide-and-conquer algorithm by parallelizing computation over independent parts of the graph. With the Forward Edge Property 1, we know that, for a given node, each of its child subtrees can be embedded independently. The algorithm separates vertices into layers where all the vertices in a given layer can be embedded in parallel. We discovered that layers must be created from the bottom up rather than top down in order to minimize overhead of generating parallel for loops. We tuned for parameters such as base case size and criteria for switching from parallel for loop to sequential. We ran trials across datasets with graphs of size 10000 vertices and up to 10 million vertices. On planar graphs with hundreds of thousands of vertices, the parallel algorithm, when run on 16 cores, exhibited a 2.4–2.7 times speedup over the sequential algorithm. On non-planar graphs of tens of thousands of vertices, the parallel algorithm exhibited a 9–22 times speedup on 16 cores. Because our algorithm processes vertices from bottom up (the sequential algorithm processes vertices in reverse DFI order), it returns much more quickly than the sequential algorithm. Unfortunately, the algorithm incurs a little bit of overhead in the initialization step and because the span is bounded by the number of layers, the speedup scaling plateaus after about 8 cores. But, as far as we

know, this is the first implementation of a practical parallel algorithm for planarity testing.

4.2 Future Work

In the future, it would be interesting to implement the divide-and-conquer algorithm and compare the performance of the algorithm to the parallelization by layers algorithm. Some implementation details we need to consider are how to handle the transition between merge steps (in the case of more than two workers) and how to detach flippable components in a more efficient manner.

Another possibility that could be interesting to explore is a combination of the two algorithms. Currently, the parallelization by layers algorithm is bottlenecked by the sequential layers. If we were able to keep parallelizing by layers in the first half and introduce a method to parallelize the sequential layers safely in the second half, we would create a parallel algorithm that likely exceeds the performance of either one of the individual algorithms.

In both cases, the initialization step serves as a theoretical bottleneck on the entire algorithm as it requires an $O(n)$ DFS traversal. A future avenue to explore could be either replacing the DFS traversal with a parallel version of DFS or trying a DFS traversal with multiple starting vertices and adapting the algorithm to run on the resulting subtrees. Another potential approach is to attempt to parallelize a planarity testing algorithm that isn't based on DFS.

In addition, we could try to adapt our current algorithms to come up with an approximation algorithm that yields high performance and works correctly with high probability. Finally, it would be interesting to see if the techniques we developed for parallel planarity testing generalize to any other parallel graph algorithms.

Glossary

attachment edge An attachment edge is an edge from the boundary cycle to the temporary vertex. 24, 25

boundary cycle The boundary cycle is the internal or external face containing the vertices in the subgraph that the temporary vertex is attached to. 25

flippable component We define a flippable component C on the boundary cycle as follows. If we were to traverse the boundary cycle, the vertices corresponding to that component would be listed as "a, X, a" where X is a series of vertices on the outside of C. 23, 25, 44

forward edge A forward edge is an edge (v, w) where v is an ancestor of w but v is not w's direct parent. 13

missing component The missing component is the subtree represented by the temporary vertex. 24, 25

missing child Given vertices p, v, and c where p is the DFS parent of v and v is the DFS parent of c. Vertex c is designated as a missing child of v if the edge (v, c) is assigned to a different subgraph than edge (p, v). 20, 24, 43

pending edge A pending edge is an edge (v, w) that has not been embedded because vertices v and w are part of two distinct subgraphs. 19, 43

temporary vertex A temporary vertex is a constructed vertex used to represent the subtree rooted at a missing child when running Boyer's algorithm on each subgraph. Any and all edges to the subtree associated with the missing child are attached to the temporary vertex until the merging step of the algorithm. 20, 24

References

[1] "Planar graphs," in *Graph Theory*. London: Springer London, 2008, pp. 243–285, ISBN: 978-1-84628-970-5. DOI: 10.1007/978-1-84628-970-5_10. URL: https://doi.org/10.1007/978-1-84628-970-5_10.

[2] J. Hopcroft and R. Tarjan, "Efficient planarity testing," Cornell University, Tech. Rep. TR 73-165, Apr. 1973.

[3] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms," *Journal of Computer and System Sciences*, vol. 13, pp. 335–379, 1976.

[4] W.-K. Shih and W.-L. Hsu, "A new planarity test," *Theoretical Computer Science*, vol. 223, pp. 171–191, 1999.

[5] J. M. Boyer and W. J. Myrvold, "On the cutting edge: Simplified $O(n)$ planarity by edge addition," *Journal of Graph Algorithms and Applications*, vol. 8, no. 3, pp. 241–273, 2004.

[6] D. Bader and S. Sreshta, "A new parallel algorithm for planarity testing," May 2004.

[7] R. Ingram, "A new parallel algorithm for planarity testing," Bachelor's Thesis, Trinity University, 2009. URL: https://digitalcommons.trinity.edu/compsci_honors/24.

[8] V. Ramachandran and J. H. Reif, "Planarity testing in parallel," *J. Comput. Syst. Sci.*, vol. 49, pp. 517–561, 1994. URL: https://api.semanticscholar.org/CorpusID:15797132.

[9] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, Mar. 1996. DOI: 10.1145/227234.227246.

[10] G. Blelloch, D. Anderson, and L. Dhulipala, *Parlaylib: A toolkit for programming parallel algorithms on shared-memory multicore machines*, 2020. URL: https://github.com/cmuparlay/ParlayLib.