International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 8, Number 1, pages 73–92, January 2018

Applications of Ear Decomposition to Efficient Heterogeneous Algorithms for Shortest Path/Cycle Problems

Debarshi Dutta
Center for Security, Theory, and Algorithmic Research,
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad 500 032, India.

Meher Chaitanya[1]
Center for Security, Theory, and Algorithmic Research,
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad 500 032, India.

Kishore Kothapalli
Center for Security, Theory, and Algorithmic Research,
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad 500 032, India.

Debajyoti Bera
Indraprastha Institute of Information Technology, New Delhi
New Delhi, India 110 020.

Received: July 25, 2017
Revised: November 8, 2017
Accepted: December 1, 2017
Communicated by Susumu Matsumae

Abstract

Graph algorithms play an important role in several fields of sciences and engineering. Prominent among them are the All-Pairs-Shortest-Paths (APSP) and related problems. Indeed there are several efficient implementations for such problems on a variety of modern multi- and many-core architectures.

It can be noticed that for several graph problems, parallelism offers only a limited success as current parallel architectures have severe short-comings when deployed for most graph algorithms. At the same time, some of these graphs exhibit clear structural properties due to their sparsity. This calls for particular solution strategies aimed at scalable processing of large, sparse graphs on modern parallel architectures.

In this paper, we study the applicability of an ear decomposition of graphs to problems such as all-pairs-shortest-paths and minimum cost cycle basis. Through experimentation, we show that the resulting solutions are scalable in terms of both memory usage and also their speedup over best known current implementations. We believe that our techniques have the potential to be relevant for designing scalable solutions for other computations on large sparse graphs.

Keywords: shortest paths, ear decomposition, minimum cycle basis, heterogeneous algorithms

[1]Part of the work was done while the author was at International Institute of Information Technology, Hyderabad

1 Introduction

Graphs are of fundamental importance to several disciplines in sciences and engineering with applications to biological and social phenomenon. As graphs corresponding to real-world and practical applications have a massive size, parallel processing is often necessary. It is therefore natural that a lot of current research is directed towards efficient algorithmics on a variety of modern and emerging multi- and many-core architectures [4, 28, 5, 34].

On the other hand, it is observed by several authors that the characteristics of most modern architectures are not well-suited for efficient execution of graph algorithms. The highly irregular nature of memory accesses of graph algorithms induces a heavy burden on the I/O system of modern architectures. Recent work on efficient parallel algorithmics for graph problems is aimed at addressing this issue via novel data structures and memory layout optimizations [7, 13].

As the sizes of the graphs of interest are large, there has been a renewed interest in novel algorithmic enhancements for known graph problems. An approach to address this problem is to understand the structural properties of graphs and redesign algorithms that can better exploit such properties for gains in efficiency. Examples of this approach can be seen in works of Cong and Bader [2] for identifying the biconnected components of a graph, Banerjee et al. [4] for graph algorithms such as BFS, connected components, and APSP, and Hong et al. [17] for identifying the strongly connected components of a directed graph.

This paper presents some novel insights along the same direction on two well studied graph theoretic problems. We first consider the problem of computing shortest paths between all pairs of nodes in a weighted graph which we denote by `APSP`. As an application of the `APSP` problem, our second problem (denoted by `MCB`) is to obtain a minimum weight cycle basis of a weighted graph — essentially, to find a set of basis cycles with the least total weight such that every other cycle can be represented as a linear combination of the basis cycles. The `MCB` problem has applications to problems in biochemistry [14], three-dimensional surface reconstruction from a point cloud [15] and electric networks [11].

The main tool behind the algorithms presented in this paper is an ear decomposition of a graph. We observed that the degree-two nodes can be considered differently from the higher degree nodes for problems that are based on paths in graphs. Consider, e.g., a sequence of *connected degree-two* vertices $u_1 - u_2 - u_3 - \ldots u_k$ that is present in some graph. Then, for any s and t chosen not among them, any path between s and t either passes through *all or none* of the u_is. We also observed that several real-life instances contain a significant number of degree-two nodes (see Table 1). Ear decomposition is a technique to identify the structural composition of a graph in terms of sequences of degree-two nodes (*aka.* ears). We show how ear decomposition can lead to significant improvements on `APSP` (thereby also affecting `MCB`) apart from making the algorithms scalable. Along the way, we introduce novel and non-trivial pre-processing and post-processing steps that are crucial for obtaining an efficient algorithm for each of the problems. In a recent work, we have used the ear decomposition of a graph to obtain efficient parallel algorithms for computing the betweenness-centrality values at each node of a graph [32].

Our algorithms for `APSP` and `MCB` have a common blueprint. It is well known that a graph has an ear decomposition if and only if the graph is biconnected [33]. The decomposition of such a graph into its ears allows us to systematically *remove* the nodes of degree two in the preprocessing stage and focus on a smaller network built only on the higher degree nodes that we call as a *reduced graph*. Such a decomposition is helpful in the context of parallel graph algorithms to increase the available parallelism in the computation and decrease the work required. It should be noted that ear-decomposition is a linear-time process and moreover, has an efficient parallel implementation as well [33]. In the next stage we run a parallel adaptation of one of the existing algorithms for `APSP` and `MCB`, with appropriate modifications, on the reduced graph. We observed a drastic reduction in the running time due to the small size of this graph compared to the original one. In the post-processing stage, we extrapolate the results obtained on the reduced graph to the original graph which is again a linear-time process.

The main technical contributions of this work are summarized below.

- We design appropriate pre-processing and post-processing routines that leverage the ear-

decomposition of a graph to efficiently solve the `APSP` graph problem (Section 2) and the `MCB` problem (Section 3).

- We implement our algorithms on a heterogeneous platform consisting of an Intel i5 E2650 multicore CPU and an NVidia Tesla K40c GPU. For our implementation to be efficient, we introduce dynamic work balancing techniques via a work queue.

- We analyze the benefits of our approach by conducting a wide variety of experiments on real-world graphs of size ranging from 10 K nodes to 130 K nodes. Our approach results in an 1.7x improvement on average over the corresponding best known implementation of `APSP` on real-world graphs. When compared to a technique without ear-decomposition, our ear-decomposition based algorithm for `MCB` results in about 3x improvement.

1.1 Related Work

Graph algorithms on a variety of emerging architectures have been studied in several recent works. We focus only on those that are directly relevant to our paper.

APSP Parallel implementations for the APSP problem have been a topic of immense research interest over the decades on a variety of architectures. One of the earliest works on parallel shortest path problem was proposed by Micikevicius et al. in [30] that was subsequently improved by Harish and Narayanan [16]. The algorithm of Floyd and Warshall (cf. [9]) has been the choice of several parallel implementation as the algorithm allows one to study cache blocking techniques. Examples of this approach can be seen in Buluc et al. [5], Matsumoto et al. [28] and Katz et al. [23]. The above works report results on a variety of CPU and GPU architectures.

Recent GPU algorithms for the `APSP` problem are reported in Banerjee et al. [4] and Djidjev et al. [12]. Djidjev et al. [12] use graph decomposition via Parmetis [22], compute shortest paths within the partitions and extend the same to paths across partitions. They work mostly with planar graphs to ensure a good partition. Banerjee et al. [4] use a decomposition based on biconnected components to compute shortest paths in a large sparse graph.

A decomposition technique called the hammocks-on-ears decomposition has been proposed by Kavvadias et al. [27] along with a PRAM algorithm for obtaining such a decomposition. The hammocks they propose have stronger properties than an ear decomposition. But, the parallel computation of such a decomposition and the post-processing can turn out to be more time consuming in practice.

Minimum Cycle Basis Minimum Cycle Basis problem has been well studied in the sequential domain. There are several known algorithms for computing the MCB in an weighted undirected graph. The first polynomial algorithm was suggested by Horton [18]. Horton [18] computes an MCB in time $O(m^3n)$. De Pina [11] gave an $O(m^3 + mn^2)$ approach by using a different method. Recent contributions by Telikepalli et al. [25] have brought down the complexity to $O(m^2n + mn^2 \log n)$ by using a fast matrix multiplication based approach. Mehlhorn et al. [29] further describes a $O(m^2n/\log n + n^2m)$ algorithm for undirected weighted graphs and also provided for a simpler way to obtain the shortest cycle in each phase. Amaldi et al. [1] characterizes the Horton cycles to obtain a restricted set of cycles known as the isometric cycles and provides an improved $O(m^\omega)$ Monte Carlo algorithm where ω is the exponent of the fast matrix multiplication.

2 Our Approach for APSP

We start with graphs that are biconnected (Section 2.1) and extend our approach to graphs that are not biconnected in Section 2.2.

2.1 APSP for Biconnected Graphs

We now present our algorithm for APSP on biconnected graphs using the technique of ear decomposition. Algorithm 1 describes a brief pseudocode of our three phase algorithm followed by the details of each phase. The label {**cpu,gpu**} in Algorithm 1 is used to indicate that the corresponding task is computed in a heterogeneous manner on both the GPU and CPU. The labels {**cpu**} (*resp.* {**gpu**}) are used to indicate tasks that are executed solely on the CPU (GPU).

Algorithm 1 APSP(G)

```
1: /* Phase I: Preprocessing */
2: {gpu}: G^r = Reduce(G)
3: /* Phase II: Processing */
4: {cpu,gpu}:
5: for each s ∈ V(G^r) do
6:    DIJKSTRA(G^r, s) /* Find shortest paths from s */
7: end for
8: /* Phase III: Post-processing */
9: {cpu,gpu}:
10: for each s ∈ G \ G^r in parallel do
11:    UPDATE_DISTANCE(s).
12: end for
```

2.1.1 Preprocessing

Let G be a sparse and biconnected graph. It is known that a biconnected graph possesses an ear decomposition. An ear decomposition of a graph $G = (V, E)$ is a partitioning of the edges of G into simple paths (ears) $P_0, P_1, \cdots$, as follows (see also [33]).

- P_0 is an edge uv,
- $P_0 \cup P_1$ is a simple cycle, and
- The end points of path P_i, for $i \geq 2$, are on the paths $P_0, P_1, \cdots, P_{i-1}$, and path P_i has no other nodes common with the nodes on the paths $P_0, P_1, \cdots, P_{i-1}$.

In such a decomposition, nodes of degree two, except possibly those on ear P_0, appear on exactly one ear. We show that such nodes of degree two can be removed from G. We call the resulting graph of G as the reduced graph G^r. One can formalize the notion of the reduced graph $G^r = (V^r, E^r, W^r)$ as follows. The nodes of G^r are the nodes of G that have a degree at least three. Two nodes v and w in G^r are neighbors if and only if v and w belong to a common ear P of G and have no nodes of degree three or more in between them on the ear P. The weight of an edge vw in G^r set as the sum of the weights of the edges $vx_1, x_1x_2, \cdots, x_iw$ in G such that nodes $x_1, x_2, \cdots, x_i$ are consecutive vertices on P with degree two in G and are in between v and w on the ear P in G. For a node x_i, $i > 1$, of degree two on ear $P = (x_1x_2 \cdots x_k)$ in G, we define functions *left* and *right* of x_i in G^r, denoted left(x_i) and right(x_i), as the nodes of degree at least three on P that are closest to x_i towards x_1 and x_k respectively. For instance, in the above example with $v, x_1, x_1, x_2, \cdots, x_i, w$ being on the same ear in that order with v and w having degree more than 2 and the x_is having degree of 2, left(x) = v and right(x) = w.

Notice that during the construction of the reduced graph, there could be multiple edges between nodes in the reduced graph. In this case, since we are interested in shortest paths, we retain the edge with the shortest weight and discard the remaining edges.

2.1.2 Phase II: Processing

In this phase, we find the shortest paths between all pairs of nodes in the reduced graph G^r. From each node v in G^r, we essentially run the algorithm of Dijkstra [9] that finds the shortest paths from

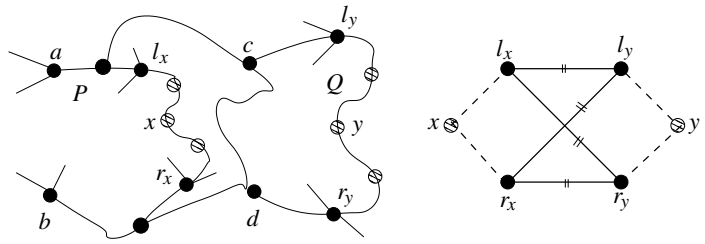

Figure 1: In the figure on the left, nodes completely filled appear in the reduced graph and shaded nodes are removed in Stage II of preprocessing.

v to all other nodes t in G^r. In short, we obtain all the shortest path values $S^r[s,t] \mid \forall\{s,t\} \in G^r$. We use the GPU implementation of Dijkstra's algorithm due to Harish et al. [16]. On CPU, we run multiple instances of Dijkstra's algorithm from different vertices of G^r. Each instance of Dijkstra's algorithm is run on an individual thread. The algorithm of Dijkstra is preferred over other shortest path algorithms for reasons including the ability to run each instance of Dijkstra's algorithm independently by a thread and the work involved in Dijkstra's algorithm depends linearly on the number of edges in the graph.

2.1.3 Post-processing

In this phase, we use the shortest paths in G^r to compute the shortest paths across all pairs of nodes in G. Consider the shortest paths originating from a node x in $G \setminus G^r$ with $\text{left}(x) = \ell_x$ and $\text{right}(x) = r_x$, $\text{left}(y) = \ell_y$ and $\text{right}(y) = r_y$ as shown in Figure 1. The shortest path between such nodes x and y has to use one of ℓ_x or r_x to leave the ear $P = (a \cdots \ell_x \cdots x \cdots r_x \cdots b)$ and one of ℓ_y or r_y to enter the ear $Q = (c \cdots \ell_y \cdots y \cdots r_y \cdots d)$. If the four pairwise shortest paths between ℓ_x, r_x, and ℓ_y, r_y are given, marked with double lines in the right figure of Figure 1, the shortest path from x and y can be obtained as the shortest among the four possible paths.

For paths from x that end at nodes y with $\text{left}(x) = \text{left}(y)$ and $\text{right}(x) = \text{right}(y)$, the shortest xy path is either the unique xy-path along P that does not use ℓ_x and r_x, or the path $x - \ell_x - r_x - y$ that crosses P via ℓ_x, reenters P via r_x and then reaches y. Paths from x that end at nodes y such that x and y have different left and/or rightnodes, have to necessarily go via ℓ_x or r_x.

Let $S[s,t]$ store the weight of the shortest path between s and t in G and $wt(a,b)$ represents the weight of the edge ab. Clearly, for all $u, v \in V^r$, $S[u,v] = S^r[u,v]$ since the reduced graph preserves shortest-path distances between vertices of degree at least 3. To compute shortest path $S[x,v]$ between any $v \in V^r$ and any $x \in V \setminus V^r$, consider the ear P on which x lies and let $\text{left}(x) = \ell_x$ and $\text{right}(x) = r_x$ (v may coincide with ℓ_x or r_x). We can compute

$$S[x,v] = \min\left\{S^r[\ell_x, v] + wt(x,\ell_x), S^r[r_x, v] + wt(x, r_x)\right\}$$

.

Now we consider the most general case of computing $S[x,y]$ for nodes $x, y \in V \setminus V^r$. For this case, let ℓ_x, r_x and ℓ_y, r_y be the left and right nodes of x and y respectively (ℓ_x may coincide with ℓ_y or r_y, and similar reasoning applies to r_x). Using the same idea as above, we can compute

$$S[x,y] = \min\left\{\begin{array}{l} wt(x,\ell_x) + S^r[\ell_x,\ell_y] + wt(\ell_y, y), \\ wt(x,\ell_x) + S^r[\ell_x, r_y] + wt(r_y, y), \\ wt(x, r_x) + S^r[r_x,\ell_y] + wt(\ell_y, y), \\ wt(x, r_x) + S^r[r_x, r_y] + wt(r_y, y). \end{array}\right\}$$

The call to UPDATE_DISTANCE(s) in this phase essentially computes $S[s,t]$ for all $t \in G$ by using the appropriate formula described above.

2.2 Extension to General Graphs

As we are interested in large sparse graphs, it is very likely that our graphs are not 2-connected or even 2-edge-connected. Quite contrary, large sparse graphs arising out of real-world phenomena tend to have several 2-connected components of varying sizes. In such a scenario, such graphs do not have a ear decomposition as being 2-edge-connected is a necessary (and sufficient) condition for having an ear decomposition [33].

To make use of Algorithm 1, in a preprocessing step we start by partitioning G into its biconnected components $G_1, G_2, \cdots$ each of which is 2-connected. We now obtain an ear decomposition of $G_1, G_2, \cdots$, and obtain their respective reduced graphs $G_1^r, G_2^r, \cdots$,. Let $A_1^r, A_2^r, \cdots$, denote the set of *articulation points (APs)* in $G_1^r, G_2^r, \cdots$, respectively. We let $a = |\cup_i A_i^r|$. The quantity a denotes the number of articulation points in G.

In the processing step, we now find the shortest paths between pairs of nodes in each G_i^r individually, and in parallel. We store the computed results in a table A_i that stores the shortest distance between pairs of nodes in G_i.

Our post-processing is now spread across two stages. In Stage 1, for each $i = 1, 2, \cdots$ we extend the shortest paths between pairs of nodes in the ear graph G_i^r to shortest path between pairs of nodes in G_i. This is done as described in Section 2.1. These results are also stored in tables A_i for $i = 1, 2, \cdots$. To compute shortest paths across pairs of nodes in different biconnected components we proceed as follows.

In Stage 2 of post-processing, we use the notion of the block-cut tree of a graph as described in [4]. The block-cut tree B of a graph G has nodes corresponding to the biconnected components of G. An *edge* exists between two nodes v and w in B if the corresponding biconnected components in G share an articulation point. We use the block-cut tree to find the shortest distance from each articulation point to every other articulation point in G. These results are stored in a table A of size $a \times a$. We use A to compute distance between nodes of different biconnected components, G_i and G_j.

For nodes $n_1 \in G_i$ and $n_2 \in G_j \mid i \neq j$, $d(n_1, n_2) = \min(d(n_1, a_1) + d(a_1, a_2) + d(a_2, n_2))$ where a_1 and a_2 are the AP's corresponding to G_i and G_j which are on the path from G_i and G_j.

2.3 Implementation Details

In our heterogeneous implementation of the processing and the post-processing step, we notice that work balancing is needed between the CPU and the GPU. Since a static approach for work balancing can fall short of the desired work balance, we use our custom work queue (from [19]). The workunits correspond to the processing (*resp.* post-processing) with respect to each biconnected component of the graph. For reasons of efficiency, the work units are sorted according to the size of the biconnected component and arranged in sorted order so that the GPU starts accessing the bigger workunits. If the graph is already biconnected and we are using Algorithm 1, then the workunits can correspond to the processing required with respect to a vertex. As is done in [19], the CPU and the GPU access workunits from the queue from either end points, and also in proportion to the number of threads supported on the CPU and the GPU.

Since the matrix A is needed by both the CPU and the GPU in the post-processing step, the matrix A is kept in the memory of both the CPU and the GPU. This forces us to limit our experiments to fit the available space on the GPU. One advantage of our method is that the space used to store all the shortest path values is in $O(a^2 + \sum_i n_i^2)$ where n_i refers to the number of nodes in G_i. In most sparse graphs, the above quantity is usually much smaller than $O(n^2)$ that is required to store the shortest path values. See also Table 1 for evidence for this phenomenon.

2.4 Results and Analysis

In this section, we show experimental results of our algorithm and also compare the results with related approaches. We start by describing our experimental platform and the datasets used.

2.4.1 Our Experimental Platform

Our experiments are conducted on a multicore CPU and an NVidia GPU. We use the Intel E5-2650 CPU for our experiments on a multicore CPU. The E5-2650 is a dual processor with each processor having 10 cores. With hyper-threading each core can support two logical threads. The cores operate at a frequency of 2.3 GHz that can be boosted up to 3 GHz using the turbo boost technology. The E5-2650 has 128 GB RAM and a memory bandwidth of 68 GB/s. In addition, the memory hierarchy includes a 64 KB L1 cache per core, a 256 KB L2 cache per core, and a shared 25 MB L3 cache.

The NVidia Tesla K40c GPU houses 2880 cores over 15 SMs, with each core clocked at 745 MHz, providing a peak double precision floating point performance of 1.43 TFLOPS and single precision floating point performance of 4.29 TFLOPS. The K40c GPU has an on board GDDR5 RAM of 12 GB that is served by a 288 GB/sec channel. Each SM also has a 64 KB configurable cache to exploit data locality.

CPU: OpenMP - Application Program Interface `OpenMP` (Open Multi-processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++, Fortran, on most platforms, processor architectures and operating systems. The API is simple and flexible enough for developing parallel applications for platforms ranging from standard desktop computer to the supercomputer.

Using `OpenMP`, the section of code that is meant to run in parallel is marked with a *preprocessor directive*, the master thread forks a specified number of slave threads and the system divides the task among them. The threads run concurrently, with the runtime environment (defined using *environment variables*) allocating threads to different processor. After the execution of parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

We refer the reader to `OpenMP` tutorial [6], for detailed information about *compiler directives, library routines, and environment variables.*

GPU: CUDA - Application Program Interface `CUDA` (Compute Unified Device Architecture) is an API created by `NVIDIA`. It allows programmers to use `CUDA`-enabled GPU for general purpose programming - an approach known as GPGPU. The `CUDA` platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computing elements. The `CUDA` platform is designed to work with programming languages such as C, C++ and Fortran.

We refer the reader to `CUDA` programming guide [31], for detailed information about programming model, interface, runtime, guidelines and extensions.

Table 1: List of sparse graphs that we use in our experiments. In the column labeled "Largest BCC (%)", we show the number of edges in the largest BCC of the graph as a percentage of the number of edges in the graph. The column labeled "Nodes Removed (%)" shows the percentage of the nodes removed by our algorithm during the preprocessing step. The column "Our's memory" lists the total memory used by our algorithm as compared to the memory required to store the table of shortest distance across all pairs of vertices. The latter value is shown in the column labelled "Max Memory". Note that storage requirements are shown rounded to the nearest Megabyte (MB).

Graph	**|V**	**|E|**	**#BCCs**	**Largest BCC (%) (% $\|E\|$)**	**Nodes Removed (% $\|V\|$)**	**Our's Memory** (MB)	**Max Memory** (MB)
Graphs taken from [10]							
nopoly	10K	30K	1	100	0.018	443	443
OPF 3754	15K	86K	1	100	1.98	873	909
ca-AstroPh	18K	198K	647	98.43	15.85	970	1344
as-22july06	22K	48K	13	99.9	77.60	851	2012
c-50	22K	90K	1	100	52.04	651	1914
cond mat 2003	31K	120K	2157	80.52	26.88	1826	3705
delaunay n15	32K	98K	1	100	0	4096	4096
Rajat26	51K	247K	5053	95.17	32.92	7176	9934
Wordnet3	82K	132K	156	98.92	77.24	4663	26071
soc-signs--epinions	131K	841K	609	99.7	67.86	12932	66294
Graphs generated using the OGDF framework [8]							
Planar_1	19K	54K	46	99.55	12.42	1278	1296
Planar_2	25K	64K	164	93.65	5.63	1627	1881
Planar_3	30K	70K	298	96.53	19.72	2068	2275
Planar_4	36K	94K	175	98.37	18.56	3890	4074
Planar_5	41K	128K	223	95.63	16.34	4350	4942

2.4.2 Datasets

We experiment on two datasets: general graphs and planar graphs. General graphs for our experiment are taken from the dataset of sparse graphs from the University of Florida Sparse Matrix Collection [10]. These graph come from domains such as geometric, social networks, collaboration, and peer-to-peer networks. The planar graphs shown in Table 1 were generated using the OGDF framework [8] using methods that generate connected graphs. Some of the characteristics of the graphs considered are listed in Table 1. It can be observed that our dataset has a good diversity. The size of the graphs ranges from 10 K to 130 K, and the number of nodes of degree two range between 0% to 60%. Further, the size of the largest BCC as a percentage of edges also varies between 80% to 98%.

2.4.3 Results

We now compare the results of our algorithm labeled as "Our Approach" with two related approaches: the approach of Djidjev et al. [12] that works for planar graphs, and the approach of Banerjee et al. [4] that works for general graphs. We start by briefly describing these approaches.

Comparison with Djidjev et al. [12] for planar graphs The algorithm of Djidjev et al. [12] works as follows. As part of their approach, Djidjev et al. [12] starts by partitioning the input graph into k parts using the METIS decomposition [21]. The partitioning is used to define a boundary graph that contains nodes of the input graph that are the end points of edges that go across partitions. Once the shortest paths in each partition are obtained, the boundary graph is augmented with edges uv such that u and v are in the same partition and the weight of the edge uv

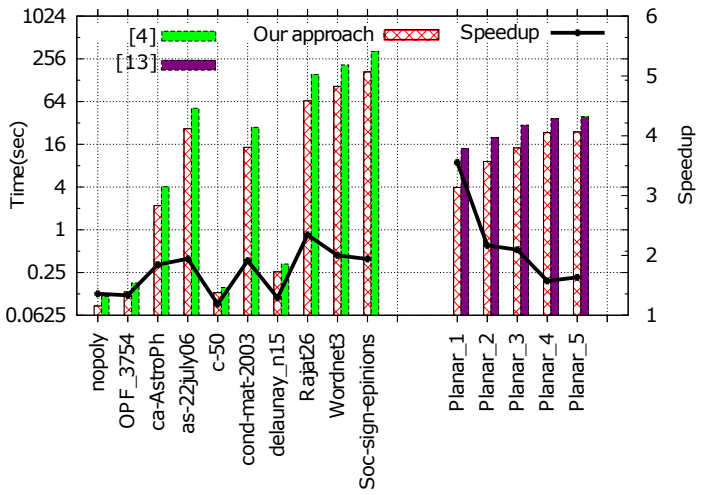

Figure 2: Figure displays the absolute time taken by our approach, labeled "Our Approach" compared to [4] for general graphs and [12] for planar graphs.

is set to be the shortest distance between u and v as computed in the previous step. The shortest paths in the boundary graph are computed in a recursive fashion followed by the shortest paths in each partition. For further details we reader can refer to [12].

It is worthwhile to note that while the algorithm presented by Djidjev et al. [12] works for any general graph, the approach is efficient for particular classes of graphs, including planar graphs with the property that the number of vertices in the boundary graph is guaranteed to be small. For this reason, their experimental results are shown only for planar graphs.

The speedup achieved by our implementation on planar graphs compared to Djidjev et al. [12] approach is shown in Figure 2. It contains the overall timings for our implementation along with Djidjev's for planar graphs on the Y1-axis on the right. The Y2-axis denotes the speedup achieved by our algorithm. The timings displayed on the Y1-axis are on a logarithmic scale. An average speedup of 2.2x achieved is mentioned in the right most column of Figure 2. As most planar graphs contain a good percentage of degree-2 vertices we conclude that our approach for real world planar graphs is more beneficial compared to [12].

Comparison with Banerjee et al. [4] The algorithm provided by Banerjee et al. [4] works by decomposing the graph as follows. Given an input graph G, it constructs a block-cut tree for G. It then computes the shortest paths within each biconnected component and later extends the computation of shortest paths across the blocks. The algorithm also optimizes the run time by removing the iterative pendants vertices. That is, it initially removes vertices of degree-1 from the graph. It then checks if the degree of any vertices adjacent to the vertices removed in the first iteration, degenerates to 1. This method, though reduces the computation time compared to other existing algorithms for real world sparse graphs, it does not effectively benefit from the degree-2 vertices present in the graph. Also this model requires more storage compared to our approach. For further details interested reader can refer to [4]. To illustrate our algorithm's computational efficiency we compare our results with Banerjee et al. [4] for general graphs.

Figure 2 shows the relative improvement of our approach compared to Banerjee et al. [4] implementation for general graphs. The Y2-axis denotes the speedup achieved by our algorithm with respect to that Banerjee et al. [4]. The average speedup achieved is 1.7x. Moreover, since we notice a small speedup even in cases where no nodes were removed, the speedup achieved is due to both implementation and using an ear decomposition.

Another way to study the scalability of parallel graph algorithms is to use the metric MTEPS standing for Million Traversed Edges Per Second. This metric is computed as the ratio of the product of the number of edges and number of vertices over the time taken in seconds. A higher MTEPS

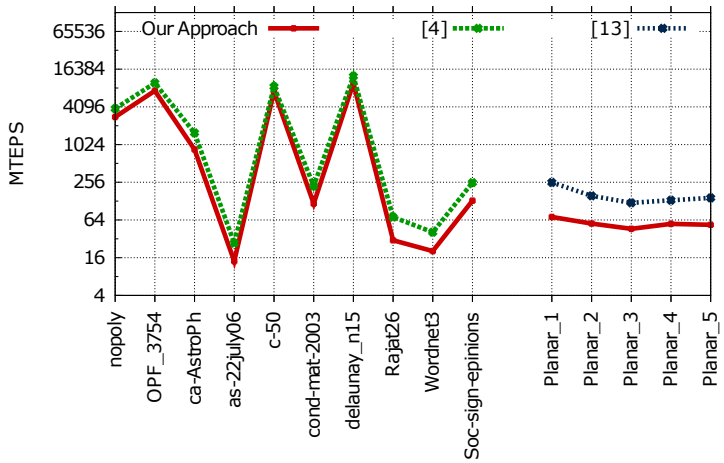

Figure 3: MTEPS achieved by our algorithm, labeled "Our Approach" and that of [4] for general graphs and [12] for planar graphs.

indicates a more scalable algorithm. Figure 3 provides the MTEPS achieved by the approaches of Djidjev et al. [12] and Banerjee et al. [4] on planar and non-planar graphs respectively in comparison to our approach. We finally note that we are limited in this comparison by the space available on the GPU although our approach needs lesser space compared to that of both [12, 4].

3 Minimum Weighted Cycle Basis (MCB)

3.1 Preliminaries

A cycle C, in an undirected graph is a subgraph where every vertex has a degree of two. A cycle can be represented by an incidence vector $\vec{C}$ on E. The vector space of the cycles are generated in the field $GF(2)$. If there are k connected components in the graph, then the cardinality of a cycle space of the graph is equal to $m - n + k$. A maximal set of linearly independent cycles form the cycle basis of the graph. In a graph G with a weight function $W : E \rightarrow R^+$, the weight of a cycle, denoted by $W(C)$ is the sum of weights of edges present in the cycle. The weight of a cycle basis is the sum of weights of all the cycles in the basis. We consider the problem of finding a cycle basis of minimum total weight, denoted MCB, in a graph.

3.2 Sequential Algorithms

We will now summarize the deterministic sequential algorithms from [1, 11, 18, 29] for obtaining an MCB. Horton [18] provided the first polynomial time algorithm for computing an MCB. Horton showed that an MCB can be extracted from a set of restricted cycles containing the fundamental cycles with respect to the shortest path trees from each node as the source. Such cycles are known as Horton cycles and the set of the Horton cycles is denoted as Horton-Set(G). Notice that there are $n \cdot (m - n + 1)$ cycles in Horton-Set(G).

Many recent works [1, 29] use the idea that the Horton cycles of G with respect to a feedback vertex set of $V(G)$ suffices. A Feedback Vertex Set (FVS) is a set of vertices such that every cycle in the graph contains some vertex in the set [20]. Since obtaining a smallest FVS is shown to be NP-Complete [20], one often uses a 2-approximate FVS that is easy to obtain (cf. [3]).

To describe the algorithms for an MCB, we need the following notation. For two vectors $\vec{x_1}$ and $\vec{x_2}$ in $GF(2)$, denote their inner product as $\langle \vec{x_1}, \vec{x_2} \rangle$. If $\langle \vec{x_1}, \vec{x_2} \rangle = 0$, then $\vec{x_1}$ and $\vec{x_2}$ are said to be orthogonal to each other. Let T be any spanning tree in the underlying unweighted graph $G(V, E)$. We denote the set $E' = E \backslash T$ to be the set of non-tree edges. Let the cardinality of this set be $f = |E'|$. Order the edges of E' in an arbitrary order $\{e_1, e_2, \cdots, e_f\}$. Fundamental cycle bases

consist of cycles induced by the non-tree edges corresponding to a spanning tree T. There is a cycle for each non-tree edge consisting of the non-tree edge plus the tree path connecting its endpoints. We consider the vector representation of a cycle C incident on the restricted edge set consisting of edges in E' and not E since it is sufficient to consider E'. Since each cycle can be represented uniquely in this manner, we can also treat each cycle as an incidence vector on E'. Each such vector is seen to lie in $\{0,1\}^f$.

For these vectors, one can also associate the standard orthonormal basis $\vec{S_i}$ for $i = 1, 2, \cdots, f$, with $\vec{S_i}$ having 1 in the ith component and 0 in all other components. Each such vector $\vec{S_i}$ is called as a *witness*. The product $\langle \vec{C}, \vec{S} \rangle = 0$ indicates that $\vec{S}$ is orthogonal to $\vec{C}$. Since we are in the field GF(2), observe that $\langle \vec{C}, \vec{S} \rangle = 1$ if and only if C contains an odd number of edges in $\vec{S}$. A cycle C_e formed by edges of T and e has intersection of exactly one edge with S_i. So there is always at least one cycle satisfying $\langle \vec{C}, \vec{S} \rangle = 1$.

De Pina [11] shows that given the witness vectors $\vec{S_i}$ from $\{0,1\}^f$, let $\vec{C_i}$ be the vector corresponding to the shortest cycle C_i in the graph G such that $\langle \vec{C_i}, \vec{S_i} \rangle = 1$. Then, $\sum_{i=1}^{f} W(C_i) \leq MCB(G)$. The algorithm from De Pina [11] therefore computes the minimum weighted cycles to form a basis.

Algorithm 2 Obtaining Minimum Cycle Basis

1: *Initialize* $\forall i, \vec{S_i}(e_i) = 1$ and $\forall j \ (j \neq i) \ S_i(e_j) = 0$
2: **for** $i = 1, \cdots, f$ **do**
3: find $\vec{C_i}$ that satisfies $\langle \vec{C_i}, \vec{S_i} \rangle = 1$
4: **for** $j = i + 1, \cdots, f$ **do**
5: **if** $\langle \vec{C_i}, \vec{S_j} \rangle = 1$ /*$\vec{S_j}$ is not orthogonal to $\vec{C_i}$*/ **then**
6: $\vec{S_j} = \vec{S_j} \oplus \vec{S_i}$ /* Make $\vec{S_j}$ orthogonal to $\vec{S_i}$*/
7: **end if**
8: **end for**
9: **end for**

Algorithm 2 presents the generic algorithm. It has f iterations (Step 2). Each iteration has two sections. In the first section (Step 3), we search for a $\vec{C_i}$ non-orthogonal to $\vec{S_i}$. This retrieves one minimum cycle of the basis. The next section corresponding to Steps 4-6 performs an *independence test* that updates the remaining witnesses to make them orthogonal to cycles $\vec{C_1}, \cdots, \vec{C_i}$. This is done by taking the symmetric difference between $\vec{S_i}$ and $\vec{S_j}$.

3.2.1 Searching for the cycle C_i

We now give the intuition behind obtaining the required minimum cycle at step three of the Algorithm 2. Given graph $G(V, E)$, we construct an auxiliary graph with the following rules.

1. For every node $x \in G$, construct two nodes $x+$ and $x-$.
2. Given the initial spanning tree T of the underlying unweighted graph of G and the set E', consider an edge $e = (u, v) \in G$
   (a) construct the edges $(u+, v+)$ and $(u-, v-)$ if $e \in T$ (connect similar signed nodes), else
   (b) construct the edges $(u+, v-)$ and $(u-, v+)$ if $e \in E'$ (connect opposite signed nodes).

For a given node $x \in G$, we obtain a single source shortest path(SSSP) in the auxiliary graph from $x+$. Every shortest path from $x+$ to $x-$ induces a minimum weighted cycle in G that contains x [24]. Also, it is proved in [26] that every such minimum weighted cycles have odd number of non-tree edges(i.e. edges $\in E'$). This in turn implies that such a cycle need to satisfy the condition $\langle \vec{C_i}, \vec{S_i} \rangle = 1$ in GF(2).

3.2.2 Computing the witnesses $\vec{S}_i$

We now consider the problem of computing $\vec{S}_i$ for $i \in 1, \cdots, f$. $\vec{S}_i$ should be orthogonal to cycles $C_1, \cdots, C_{i-1}$. We now show that we take the initial witnesses $\vec{S}_i, \cdots, \vec{S}_f$ and at every step i, we use $\vec{S}_i$ to compute $\vec{C}_i$ and update $\vec{S}_{i+1}, \cdots, \vec{S}_f$ to make them orthogonal to the cycles $\vec{C}_1, \cdots, \vec{C}_i$. At the beginning of phase i, we have $\vec{S}_i, \vec{S}_{i+1}, \cdots, \vec{S}_f$ which is a basis of the space C orthogonal to the space C spanned by $\vec{C}_1, \cdots, \vec{C}_{i-1}$. After ith step we end up having $\vec{S}'_{i+1}, \cdots, \vec{S}'_f$ and show that $\vec{S}'_{i+1}, \cdots, \vec{S}'_f$ are orthogonal to $\vec{C}_1, \cdots, \vec{C}_i$. Since each $\vec{S}'_j$, $i+1 \leq j \leq f$ is a linear combination of $\vec{S}_j$ and $\vec{S}_i$, it follows that $\vec{S}'_j$ is orthogonal to $\vec{C}_1, \cdots, \vec{C}_{i-1}$. If an $\vec{S}_j$ is already orthogonal to $\vec{C}_i$, then we leave it as it is, i.e., $\vec{S}'_j = \vec{S}_j$. Otherwise, $\langle \vec{C}_i, \vec{S}_i \rangle = 1$, and we update $\vec{S}_j$ as $\vec{S}'_j = \vec{S}_j \oplus \vec{S}_i$. Since both $\langle \vec{C}_i, \vec{S}_j \rangle$ and $\langle \vec{C}_i, \vec{S}_i \rangle$ are equal to 1, it follows that each $\vec{S}'_j$ is now orthogonal to $\vec{C}_i$ also. Hence, $\vec{S}'_{i+1}, \cdots, \vec{S}'_f$ belong to the subspace orthogonal to $\vec{C}_1, \cdots, \vec{C}_i$.

3.3 Our Approach for Parallel MCB

In this section we describe our algorithmic approach for obtaining an MCB in parallel. We use the generic algorithm (Algorithm 2) and describe each step in detail. Our algorithmic approach has three phases: pre-processing, processing, and post-processing.

3.3.1 Pre-Processing

We process each biconnected component separately as there can be no cycles in an MCB that span two different biconnected components. We begin with an ear decomposition on the graph G to obtain a reduced graph G^r. The following lemma shows that the `MCB` of G^r can be used to obtain an `MCB` of G.

Lemma 3.1 *Let $G^r(V^r, E^r)$ be the reduced graph obtained by an ear-decomposition on $G(V, E)$. Let $\mathcal{P}$ denote all maximal degree two chains with the endpoints as non-degree two nodes in G. Let e_P be the edge in G^r that replaces the path $P \in \mathcal{P}$ with $W(e_P) = W(P)$. Let $\mathrm{MCB}(G^r)$ be a cycle basis of minimum weight on the graph G^r. Then,*

1. *for every cycle $C \in$ Horton-Set(G^r) that contains edges $e_{P_1}, e_{P_2}, \cdots, e_{P_t}$ from $\mathcal{P}$, with $t \geq 1$, there exists a cycle $C' \in$ Horton-Set(G) such that C' contains all edges in $C - \{e_{P_1}, e_{P_2}, \cdots, e_{P_t}\} \cup (\cup_{i=1}^{t} P_i)$.*
2. *for every cycle $C \in$ Horton-Set(G^r) that does not contain any edge e_P for some $P \in \mathcal{P}$, $C \in$ Horton-Set(G),*
3. *$dim(MCB(G)) = dim(MCB(G^r))$, and*
4. *$W(MCB(G)) = W(MCB(G^r))$.*

Proof. We will start by proving statements 1 and 2. For each cycle $C \in$ Horton-Set(G^r), there can be two cases, i.e. either C contains some $e_{P_1}, e_{P_2}, \cdots, e_{P_t}$ from $\mathcal{P}$, with $t \geq 1$ or none at all. Considering the first case, note that $W(e_{P_i}) = W(P_i)$, for every $i \in 1, \cdots, t$. Hence, C is equivalent to a cycle C_k obtained as $C_k = C - \{e_{P_1}, e_{P_2}, \cdots, e_{P_t}\} \cup (\cup_{i=1}^{t} P_i)$ and C_k can substitute C in Horton-Set(G). C_k is guaranteed to contain every P_i in its entirety with P_i being a degree-two chain. For the other case, when a cycle C does not contain any e_P, C is also present in Horton-Set(G) since all edges of C are present in G as well. This proves statements 1 and 2 of the lemma.

For statement 3, the dimensions of the orthonormal basis vectors are equal to the number of non-tree edges with respect to any spanning tree. Let T be a spanning tree in G. We show that there exists a spanning tree T^r in G^r such that G and G^r have the same number of non-tree edges. This implies that $dim(\mathrm{MCB}(G)) = dim(\mathrm{MCB}(G^r))$. To construct T^r, we mainly focus on the chains in $\mathcal{P}$. Note that being chains of degree two nodes, each P contains at least 2 edges. For each chain $P \in \mathcal{P}$, there are two cases.

1. All edges in P are in T.

2. One edge in P is a non-tree edge and the remaining are tree edges with respect to T.

We will now study the effect of replacing the chains in $\mathcal{P}$ with corresponding e_P in G^r. For any chain having k degree-two nodes, the effect of substitution is to reduce $k+1$ edges and add 1 edge with a net effect of reduction of k edges. We will now substitute each P_i, for $i \in 1, \cdots, |\mathcal{P}|$, in succession one at a time. Let us assume the initial graph $G_0 = G, E_0 = E, T_0 = T$. For each iteration of i we get a graph G_i by replacing P_i with its corresponding e_{P_i}. Note that $G^r = G_K, E^r = E_K$ and T_K obtained at the end equal T^r. For every iteration i , P_i might belong to case 1 or case 2. In case 1, the graph G_i is obtained by substituting P_i. This removes from G_{i-1} all degree-two nodes present in chains which existed entirely in T_{i-1} and replacing them with a single edge, essentially compressing T_{i-1} while keeping it connected. Thus, $|E_i| = |E_{i-1}| - (|V_{i-1}| - |V_i|)$ as a total of $|V_{i-1}| - |V_i|$ nodes are removed from G_{i-1}. The number of tree-edges in G_i is $|V_i| - 1$ and non-tree edges in $G_i = |E_i| - |V_i| + 1 = |E_{i-1}| - |V_{i-1}| + 1$. This proves that chains in case 1 do not affect the number of non-tree edges with respect to T_{i-1} going from G_{i-1} to G_i.

For a P_i in case 2, lets consider the nodes in P_i as $u_1, u_2, \cdots, u_{k+1}, u_{k+2}$ where u_1 and u_{k+2} are non degree-two endpoints of P_i. Exactly one of the edge e $= u_j u_{j+1}$ for some $j \in 2, \cdots, k+1$ is a non-tree edge. Nodes u_l for $l \in 2, \cdots, k+1$ are removed from G_{i-1}. Thus, e_{P_i} connects u_1, u_{k+2} as a non-tree edge in G_i. This retains the number of non-tree edges with respect to T_{i-1} from G_{i-1} to G_i.

Thus, the overall number of non-tree edges in G_k with respect to the T_k is equivalent to number of non-tree edges in G_0 with respect to T_0 and hence, number of non-tree edges in G^r is equivalent to those in G. Since, the dimension of the MCB depends on the number of non-tree edges in G and G^r, this proves statement 3 of the lemma. Since the weights of cycles in Horton-Set(G) are same as those in Horton-Set(G^r), hence $W(MCB(G)) = W(MCB(G^r))$ and this proves statement 4 of the lemma.

This ensures that cycles in $MCB(G)$, extracted from Horton-Set(G) are equivalent to the cycles in $MCB(G^r)$ extracted from the Horton-Set(G^r). ■

Note that the graph G^r may contain multiple edges and self-loops. For the purpose of obtaining an `MCB`, we imagine that multiple edges and self-loops appear as nontree edges of any spanning tree of G^r. An example of the lemma and its proof is shown in Figure 4.

3.3.2 Processing

We divide this section in two parts, the first part describes in detail the algorithm for computation of the least weighted cycle satisfying the condition in Step 3 of Algorithm 2. In next subsection, we explain in detail about the witness update step (Independence Test).

Searching for the least weighted cycle: Step 3 of Algorithm 2 is the most time consuming step. Let $\vec{S}_{curr} \in \{0,1\}^f$ denote $\vec{S}_i$ for the i^{th} phase. A cycle represented by $\vec{C}_i$ needs to be obtained such that $\vec{C}_i$ is non-orthogonal to $\vec{S}_{curr}$.

We start by computing the single source shortest path trees from each node in the reduced graph. Horton cycles can then be obtained by inspecting the non-tree edges present in all such shortest path trees and stored in a list sorted by the weight of each cycle. We then make use of the approach of Mehlhorn and Michail [29] for computing such a cycle. This approach has the advantage that the potential set of cycles that need to be considered are smaller in number than the Horton cycles.

Recall that the Horton cycles with respect to an FVS $\mathcal{Z}$ of G is a superset of the cycles in an MCB. Mehlhorn and Michail [29] show that a further reduction in the number of Horton cycles is possible. For $z \in \mathcal{Z}$, let T_z denote the shortest path tree rooted at z and let $e = uv$ be a nontree edge with respect to T_z such that z is the least common ancestor of u and v in T_z. Consider the cycles C_{ze} of weight $W(C_{ze}) = d_z(u) + W(uv) + d_z(v)$ where $d_z(u)$ denotes the distance between z and u in T_z. The collection of such cycles, denoted $\mathcal{A}$, is shown to be a superset of the cycles in an MCB of G [29].

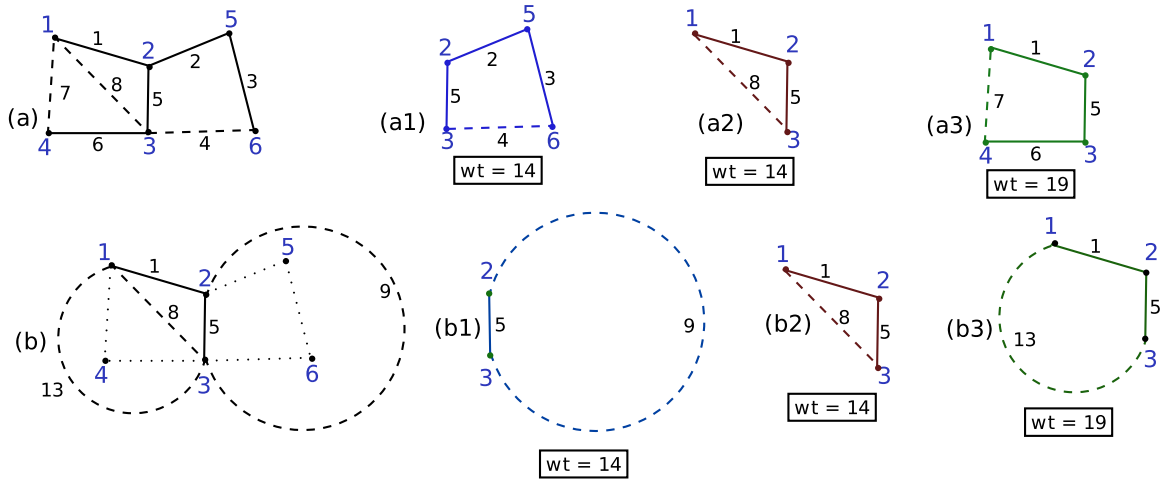

Figure 4: In the above figure, (a) is the original graph G while (b) is the corresponding reduced graph G^r. The non-tree edges in graph G and G^r are displayed as dashed lines. Nodes $4, 5, 6$ are degree two vertices, that are pruned out using Ear-Decomposition. The degree-two chains {1,4,3} and {2,5,6,3} in G are replaced by respective edges {1,3} and {2,3} in G^r. (a1,a2,a3) represent subsets of Horton-Set(G) with node 1 as the root node. (b1,b2,b3) represent the cycles in G^r corresponding to (a1,a2,a3) with their degree two chain replaced by single edges. Note that, $W(a1) = W(b1), W(a2) = W(b2), W(a3) = W(b3)$

Since the cycles in $\mathcal{A}$ always pass through the root of some shortest path tree T_z for z in an FVS of G, Mehlhorn and Michail [29] use a tree traversals from the root to the leaves to identify the required cycle. We briefly explain this procedure and its parallelization in the following. The algorithm from [29] is a two part algorithm. The first part computes partial labels and the second step uses these partially computed labels to determine orthogonality of a cycle with the current witness $\vec{S}_{curr}$.

One of the tasks in Step 3 of Algorithm 2 is to check if a cycle is non-orthogonal to vector $\vec{S}_{curr}$. To do this check in constant time per cycle, we associate labels to each node u in the shortest path tree T_z rooted at z, for every $z \in \mathcal{Z}$. We consider a path from root z to a node u in T_z and form a vector representation of this path, $\overrightarrow{path}_z(u)$ in the space induced by the restricted edge set of E' as defined in the algorithm. Edges in the above path that belong to E' are the components of $\overrightarrow{path}_z(u)$. For all node $u \in T_z$, the label $l_z(u)$ represents $\langle \overrightarrow{path}_z(u), \vec{S}_{curr} \rangle$. These labels are computed as below.

Given a tree T_z and a witness $\vec{S}_{curr}$, the tree is traversed from root to leaves. For every node $u \in T_z$, a label $l_z(u)$ is computed with respect to $\vec{S}_{curr}$. We maintain an additional variable $c_z(u)$ for each $u \in T_z$. We make two passes on the tree T_z. In the first pass for every edge, $e = uv \in T_z$ and $e \notin E'$ such that v is parent of u, we set $c_z(u) = 0$, otherwise we set $c_z(u) = \vec{S}_{curr}(e)$. Note that $c_z(z) = 0$ and $l_z(z) = 0$ for T_z. In the second pass, the traversal updates for every edge e, $l_z(u) = l_z(v) \oplus c_z(u)$. A pseudocode of the algorithm is shown in Algorithm 3.

The overall work done for all labels in all the $|\mathcal{Z}|$ trees is $O(n|\mathcal{Z}|)$. Since the computation with respect to each tree T_z is independent, this step can be performed in parallel by allocating each thread to work on a single tree T_z. In our heterogeneous algorithm, we also divide the computation across the CPU and the GPU.

In this step, using the labels computed previously, we search for the minimum weighted cycle in $\mathcal{A}$ which is non-orthogonal to $\vec{S}_{curr}$. The labels for every tree T_z are already computed. For every cycle, C_{ze} in $\mathcal{A}$, we can inspect in constant time, whether the cycle vector $\vec{C}_{ze}$ is non-orthogonal to $\vec{S}_{curr}$. Following [29], this can be done as follows. For $e = (u, v) \in C_{ze}$, $\langle \vec{C}_{ze}, \vec{S}_{curr} \rangle = (l_z(u) \oplus l_z(v) \oplus 0)$, if $e \notin E'$ or $(l_z(u) \oplus l_z(v) \oplus \vec{S}_{curr}(e))$, if $e \in E'$, we stop after obtaining the first such cycle, C_{ze} that satisfies $\langle \vec{C}_{ze}, \vec{S}_{curr} \rangle = 1$ and remove it from $\mathcal{A}$. There are $O(mn)$ such cycles, and hence it

Algorithm 3 Algorithm for computing labelled trees (from [29])

```
1: Initialize {∀z ∈ 𝒵, ∀u ∈ n}, c_z(u) = 0 , l_z(z) = 0
2: for each z ∈ 𝒵 do
3:    //Traverse T_z in level-order from root to leaves
4:    for each e = uv, e ∈ T_z do
5:       if  e ∈ E′  then
6:          c_z(u) = S⃗_curr(e)
7:       end if
8:    end for
9:    for each e = uv, e ∈ T_z do
10:      l_z(u) = l_z(v) ⊕ c_z(u)
11:   end for
12: end for
```

takes $O(mn)$ work in the worst case for every cycle. This task can be parallelized with an early-exit terminating condition.

Note that $\mathcal{A}$ contain cycles in a sorted order in increasing order of their weights. We arrange the cycles in $\mathcal{A}$ into logical batches. We check for a cycle satisfying the non-orthogonality condition in Step 3 of Algorithm 2 in each batch in parallel. If no cycle is found in batch B_1, then we move to check in batch B_2. We repeat this check until we find the required cycle. While the removal of the lists can be achieved by setting a boolean flag, the algorithm has to still travel through all the nodes in the worst case, whereas use of linked-lists are found to be lacking in efficiency due to higher penalty in access times. We use a hybrid of linked-list as well as linear arrays to store the cycles. Each linked-list node consists of a constant sized array as its base element and has a single next pointer. We first check within each position of the linked-list node and if not found skip to the next node. We mark the removal of elements by setting off the MSB and reorder the cycles within nodes when half of those in a node are removed.

Independence Test (Witness Update) This computation corresponds to steps 5 and 6 of Algorithm 2. We update each witness $\vec{S}_j$, where $\langle \vec{S}_j, \vec{C}_i \rangle = 1$. Witnesses are updated to make them orthogonal to each $\vec{C}_k$, for $k \in 1, \cdots, i$. This is done by $\vec{S}_i \oplus \vec{S}_j$ (Step 6 of Algorithm 2). We update witnesses in parallel. For each witness $\vec{S}_{i+1}$ through $\vec{S}_f$, each thread can carry out the steps 5 and 6 of Algorithm 2 independently.

We perform this in both CPU as well as GPU. For the CPU, we dedicate each thread to a witness $\vec{S}_j$. Each thread checks in a sequential manner whether $\vec{S}_j$ is non-orthogonal to $\vec{S}_{curr}$ and depending on the result, updates $\vec{S}_j$. We have observed in our experiments that allocating every thread to a single witness and calculating the inner product is more cache-efficient than allocating threads to all the elements of a single witness and then reducing them.

For the GPU, each block processes one single witness, We first do a per block pairwise-component product for the witness $\vec{S}_j$ with $\vec{C}_{curr}$. We then use a parallel reduce on the block to obtain the *Xor* of the entire product. If the resultant value of the reduction is 1, we compute a symmetric-difference of $\vec{S}_{curr}$ with $\vec{S}_j$ in parallel for the entire block.

3.3.3 Post-Processing

We refer to the lemma given in Section 3.1 to note that $MCB(G)$ is equivalent to $MCB(G^r)$. We maintain an additional identifier for each e_P in G^r corresponding to a $P \in \mathcal{P}$. The actual cycle with respect to $MCB(G^r)$ can be obtained per query basis from a cycle in $MCB(G^r)$ just by substituting every e_P present in the cycle with its corresponding P.

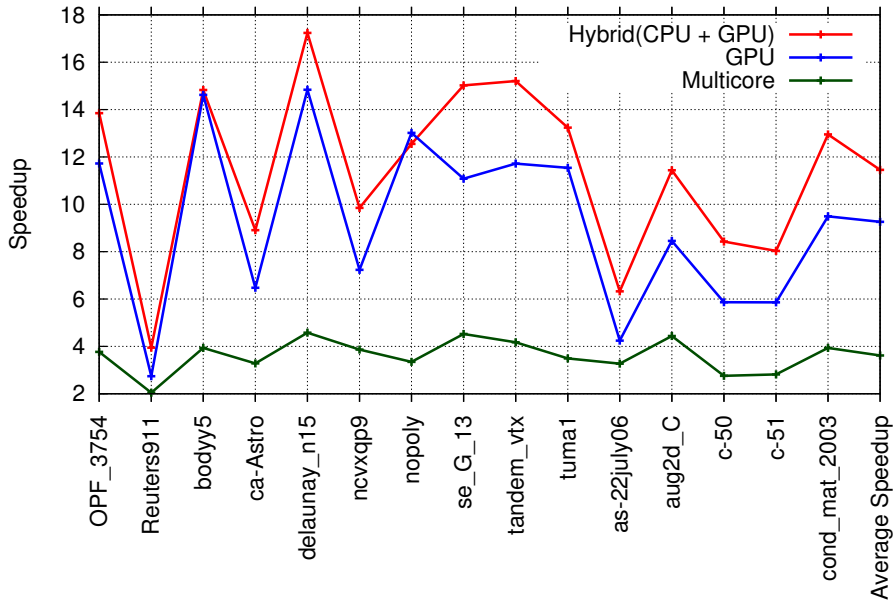

Figure 5: Displays the relative speedup of the Multi-core, GPU and Heterogeneous(CPU+GPU) w.r.t Sequential Approach

3.4 Implementation Details

We now comment on the aspects that affect all parts of our implementation. Our shortest path trees are constructed in a manner that lead to an increased efficiency of the GPU warp based SIMT parallelism while also maintaining the caching effectiveness for a CPU.

Many steps of our algorithm such as computing shortest path trees, identifying the least weighted cycle, and witness update are executed simultaneously on both the CPU and the GPU. Such a situation requires one to aim at an execution where the work is split among the CPU and the GPU in the right proportion. Since arriving at this proportion analytically is not easy, we use a dynamic mechanism based on the work queue framework [19]. Each task that uses the workqueue is organized into multiple independent workunits that can be executed either on the CPU or the GPU. These workunits are then kept in a double ended queue with the CPU and the GPU accessing the queue from either ends. The workunits are removed by the CPU and the GPU from the queue in batches whose size depends on the nature of the task. The computation finishes when the queue becomes empty.

3.5 Experimental Results

All our experiments are conducted using the computing platform described in Section 2.4.1. For our experiments, we use the first seven graphs listed in Table 1. The existing space limitations on the system renders it impossible to run our algorithm on larger graphs. Since there is no known parallel implementation available to the best of our knowledge, we limit ourselves to study the speedup of our heterogeneous implementation over a multi-threaded CPU, GPU, and sequential implementation. Table 2 lists the total time spent in each of four implementations. Figure 5 lists the speedup achieved by the above three implementations with respect to the sequential algorithm. We observe an average speedup of 3x, 9x and 11x respectively.

The speedup can be attributed to using the ear decomposition that results in reducing the number of nodes and the corresponding shortest path trees that have to be processed. In particular, if the number of degree two nodes removed is n_2, we now construct only $n - n_2$ shortest path trees. This leads to an overall reduction of $f \cdot n_2 \cdot (n - n_2)$ work with respect to the entire algorithm. Table 2

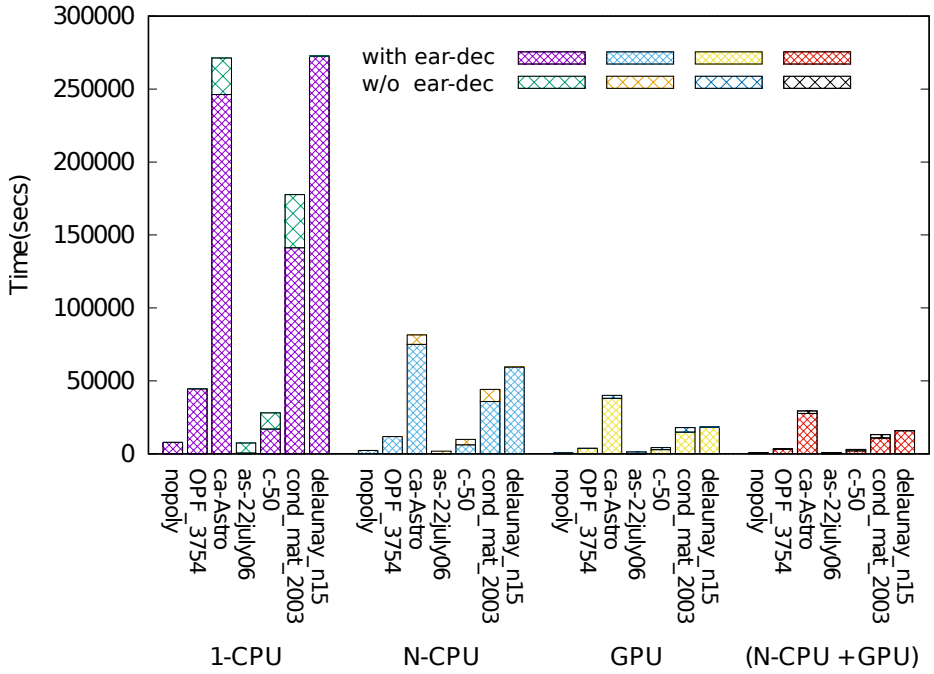

Figure 6: Displays the absolute speeds of the Sequential, Multi-core, GPU and Heterogeneous(CPU+GPU) approaches

Table 2: List of timings(in seconds) of four different implementations for our approach. Labels 'w' and 'w/o' indicate with and without ear-decomposition respectively.

	Sequential		**Multi-Core**		**GPU**		**CPU + GPU**	
Graphs	**w**	**w/o**	**w**	**w/o**	**w**	**w/o**	**w**	**w/o**
nopoly	7.83K	7.83K	2.34K	2.35K	.602K	.604K	.624K	.624K
OPF_3754	44.58K	44.58K	11.8K	11.8K	3.8K	3.8K	3.2K	3.2K
ca-Astro	246.3K	271.3K	75.06K	81.5K	38.04K	40.15K	27.6K	27.6K
as-22july06	.57K	7.4K	.17K	1.8K	0.134K	1.29K	0.09K	0.94K
c-50	17.05K	28.07K	6.17K	9.8K	2.90K	4.278K	2.02K	3.03K
cond_mat_2003	141.3K	177.6K	35.9K	44.2K	14.89K	17.97K	10.9K	13.2K
delaunay_n15	272.5K	272.5K	59.5K	59.5K	18.37K	18.37K	15.8K	15.8K

and Figure 6 also lists the impact of ear-decomposition on all the four implementations.

The average speedup due to Ear-Decomposition on each of the four implementations as specified in the table are 3.1x, 2.7x, 2.5x, 2.7x respectively. The speedup is proportional to the number of degree-two nodes e.g. as-22july06 has an average of 10x speedup across all the implementations.

In our experiments, we have observed that steps label computation, identifying the minimum weight cycle, and independence test have a major impact on the overall execution time at 76%, 14%, and 8% of the overall execution time respectively. Since independence test is dependent on label computation and identifying the minimum weight cycle, there is also a limit on the parallelism available in the overall algorithm.

4 Conclusions

In this paper, we considered two important path-based graph problems, namely "all-pairs shortest paths" and "minimum cycle basis computation" and proposed efficient parallel algorithms based on graph decomposition and reduction. We discussed several heuristics based on ear-decomposition of

biconnected graphs, both in the pre-processing and the post-processing stages, that considerably reduces the time taken compared to the state-of-the-art algorithms. We believe that similar techniques can be employed to obtain significant speedup for other graph problems too, especially the ones based on paths of a graph.

Acknowledgments

The authors would like to thank the anonymous reviewers for giving us detailed comments that helped improve the readability of the paper.

References

[1] Edoardo Amaldi, Claudio Iuliano, Tomasz Jurkiewicz, Kurt Mehlhorn, and Romeo Rizzi. Breaking the $o(m^2n)$ barrier for minimum cycle bases. In *European Symposium on Algorithms*, pages 301–312. Springer, 2009.

[2] David A Bader and Guojing Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing*, 65(9):994–1006, 2005.

[3] Vineet Bafna, Piotr Berman, and Toshihiro Fujito. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics*, 12(3):289–297, 1999.

[4] Dip Sankar Banerjee, Ashutosh Kumar, Meher Chaitanya, Shashank Sharma, and Kishore Kothapalli. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *Journal of Parallel and Distributed Computing*, 76:81–93, 2015.

[5] A. Buluc, J. R. Gilbert, and C. Budak. Solving path problems on the gpu. *Parallel Computing*, 36(5):241 – 253, 2010.

[6] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[7] J. Chhugani, N. Satish, Changkyu Kim, J. Sewall, and P. Dubey. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *IPDPS*, pages 378–389, 2012.

[8] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization*, pages 543–569, 2011.

[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to algorithms, 2001.

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[11] José Coelho de Pina. *Applications of shortest path methods.* PhD thesis, University of Amsterdam, Netherlands, 1995.

[12] Hristo Djidjev, Guillaume Chapuis, Rumen Andonov, Sunil Thulasidasan, and Dominique Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 85:91–103, 2015.

[13] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *in Proc. of IEEE IPDPS*, 2013.

[14] Petra Manuela Gleiss. *Short cycles: minimum cycle bases of graphs from chemistry and biochemistry*. na, 2001.

[15] Craig Gotsman, Kanela Kaligosi, Kurt Mehlhorn, Dimitrios Michail, and Evangelia Pyrga. Cycle bases of graphs and sampled manifolds. *Computer Aided Geometric Design*, 24(8-9):464–480, 2007.

[16] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC*, 2007.

[17] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE, 2013.

[18] Joseph Douglas Horton. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[19] SivaRamaKrishna Bharadwaj Indarapu, Manoj Kumar Maramreddy, and Kishore Kothapalli. Architecture- and workload- aware heterogeneous algorithms for sparse matrix vector multiplication. In *ACM COMPUTE*, pages 3:1–3:9, 2014.

[20] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Intl. Conf. Par. Proc.*, pages 113–122, 1995.

[22] George Karypis and Vipin Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proc. ACM SC*, 1996.

[23] Gary J. Katz and Joseph T. Kider, Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM Symp. Grap. Hard.*, pages 47–55, 2008.

[24] Telikepalli Kavitha and Kurt Mehlhorn. A polynomial time algorithm for minimum cycle basis in directed graphs. In *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*, pages 654–665, 2005.

[25] Telikepalli Kavitha, Kurt Mehlhorn, Dimitrios Michail, and Katarzyna Paluch. A faster algorithm for minimum cycle basis of graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 846–857. Springer, 2004.

[26] Telikepalli Kavitha, Kurt Mehlhorn, Dimitrios Michail, and Katarzyna E Paluch. An $O(m^2n)$ algorithm for minimum cycle basis of graphs.

[27] D. J. Kavvadias, G. E. Pantziouc, P. G. Spirakis, and C. D. Zaroliagis. Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems. *Theoretical Computer Science*, 168:121154, 1996.

[28] K. Matsumoto, N. Nakasato, and S.G. Sedukhin. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. In *Proc. HPCC*, pages 145–152, 2011.

[29] Kurt Mehlhorn and Dimitrios Michail. Minimum cycle bases: Faster and simpler. *ACM Transactions on Algorithms (TALG)*, 6(1):8, 2009.

[30] Paulius Micikevicius. General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem. In *PDPTA'04*, pages 1359–1365, 2004.

[31] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[32] Charudatt Pachorkar, Meher Chaitanya, Kishore Kothapalli, and Debajyoti Bera. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In *High Performance Computing (HiPC)*, pages 301–310. IEEE, 2016.

[33] V. Ramachandran. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity, 1993.

[34] A. Erdem. Saryuce, E .Saule, K. Kaya, and U. V. Catalyurek. Betweenness centrality on gpus and heterogeneous architectures, 2013.