

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

Debarshi Dutta, Meher Chaitanya¹, Kishore Kothapalli

International Institute of Information Technology, Hyderabad, India 500 032

Email: {debarshi.dutta@research., meher.c@research., kkishore@}iiit.ac.in

Debajyoti Bera

Indraprastha Institute of Information Technology, New Delhi, India 110 020

Email: dbera@iiitd.ac.in

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.

I. 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], [26], [6], [32].

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 parallel algorithmics for efficient 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 is a renewed necessity to look for novel algorithmic enhancements

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

also. An approach to address this problem is to understand the structural properties of graphs and redesign algorithms that can better exploit such structural 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] that identifies the strongly connected components of a directed graph.

In this direction, in this paper we start with the problem of computing shortest paths between all pairs of nodes in a weighted graph, denoted APSP. In our algorithms presented in this paper, we utilize the ear decomposition of a graph to compute shortest paths. We show that using ear decomposition can help obtain significant improvements on APSP apart from making the algorithms scalable. Along the way, we introduce novel and non-trivial pre- and post-processing steps that are crucial in obtaining the algorithm design. 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 [30].

As an application of the APSP problem, we also study the problem of obtaining a minimum weight cycle basis of a graph, denoted MCB. The MCB problem on a weighted graph is to find a set of cycles of 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 such as three dimensional surface reconstruction from a point cloud [15] and as a preprocessing step in electric networks [11] and chemistry and biochemistry [14].

Our work in this paper can be summarized below. It is well known (cf. [31]) that a graph has an ear decomposition if and only if the graph is biconnected. The decomposition of such a graph into its ears allows us to systematically *remove* the nodes of degree two and focus on nodes of degree greater than two. 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. In a post-processing step, we mimic the computation corresponding to the removed nodes using the computation done at nodes with

degree greater than two. The exact pre- and post-processing steps and the computation depend on the problem at hand.

Our main technical contributions of this work are summarized below.

- We arrive at appropriate pre- and post-processing routines that help us solve graph problems such as APSP (Section II) and MCB (Section III) efficiently.
- 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 analyse 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.

A. 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.

a) *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 [29] 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. [6], Matsumoto et al. [26] 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 Parnetis [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. [25] 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.

b) *Minimum Cycle Basis*: 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. [24] have brought down the complexity to $O(m^2n + mn^2 \log n)$ by using

a fast matrix multiplication based approach. Mehlhorn et al. [27] 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 $O(m^w)$ Monte Carlo algorithm.

II. OUR APPROACH FOR APSP

We start with graphs that are biconnected (Section II-A) and extend our approach to graphs that are not biconnected in Section II-B.

A. 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}:  $G^r = \text{Reduce}(G)$ 
3: /* Phase II: Processing */
4: {cpu,gpu}:
5: for each  $s \in V(G^r)$  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  $s \in G \setminus G^r$  in parallel do
11:   UPDATE_DISTANCE( $s$ ).
12: end for

```

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, \dots , as follows (see also [31]).

- 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, \dots, P_{i-1} , and path P_i has no other nodes common with the nodes on the paths $\cup_{j=0}^{i-1} P_j$.

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

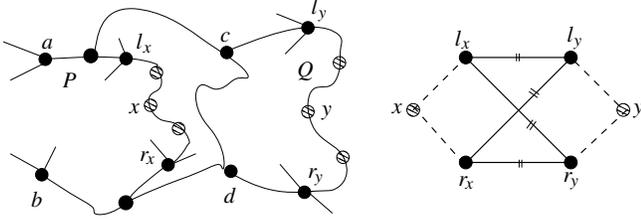


Fig. 1. The shortest path between nodes x and y has to use one of ℓ_x or r_x to leave the ear $P = (a \cdots \ell_x \cdots x \cdots x \cdots r_x \cdots b)$ and one of ℓ_y or r_y to enter the ear $Q = (c \cdots \ell_y \cdots y \cdots r_y \cdots d)$. If the four pairwise shortest paths between ℓ_x, r_x , and ℓ_y, r_y are given, marked with double lines in the figure on the right, the shortest path from x and y can be obtained as the shortest among the four possible paths. In the figure on the left, nodes completely filled appear in the reduced graph and shaded nodes are removed in Stage II of preprocessing.

ear P . The weight of an edge vw in G^r set as the sum of the weights of the edges $vx_1, x_1x_2, \dots, x_iw$ in G such that nodes x_1, x_2, \dots, 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 $\text{left}(x_i)$ and $\text{right}(x_i)$, 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 $v, x_1, x_1, x_2, \dots, x_i, w$ being on the same ear in that order with v and w having degree more than 2 and the x_i s having degree of 2, $\text{left}(x) = v$ and $\text{right}(x) = w$.

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) *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 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.

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 $\text{right}(x) = r_x$ (refer to Figure 1). 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 $x - \ell_x - r_x - y$. Paths from x that end at nodes y such that x and y have different left and/or right nodes, 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 . 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.

B. 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 an ear decomposition as being 2-edge-connected is a necessary (and sufficient) condition for having an ear decomposition [31].

To make use of Algorithm 1, in a preprocessing step we start by partitioning G into its biconnected components G_1, G_2, \dots each of which is 2-connected. We now obtain an ear decomposition of G_1, G_2, \dots , and obtain their respective reduced graphs G_1^r, G_2^r, \dots . Let A_1^r, A_2^r, \dots , denote the set of *articulation points (APs)* in G_1^r, G_2^r, \dots , respectively. We let $a = \left| \cup_i A_i^r \right|$. 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 $i = 1, 2, \dots$ 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 II-A. These results are also stored in tables A_i for $i = 1, 2, \dots$. 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 .

C. 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.

D. 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.

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.

TABLE I

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.

Graph	V	E	#BCCs	Largest BCC (%) (% E)	Nodes Removed (%) (% V)
Graphs taken from [10]					
nopoly	10K	30K	1	100	0.018
OPF 3754	15K	86K	1	100	1.98
ca-AstroPh	18K	198K	647	98.43	15.85
as-22july06	22K	48K	13	99.9	77.60
c-50	22K	90K	1	100	52.04
cond mat 2003	31K	120K	2157	80.52	26.88
delaunay n15	32K	98K	1	100	0
Rajat26	51K	247K	5053	95.17	32.92
Wordnet3	82K	132K	156	98.92	77.24
soc-sign-epinions	131K	841K	609	99.7	67.86
Graphs generated using the OGDF framework [8]					
Planar_1	19K	54K	46	99.55	12.42
Planar_2	25K	64K	164	93.65	5.63
Planar_3	30K	70K	298	96.53	19.72
Planar_4	36K	94K	175	98.37	18.56
Planar_5	41K	128K	223	95.63	16.34

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 I 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 I. 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%.

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.

a) *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 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].

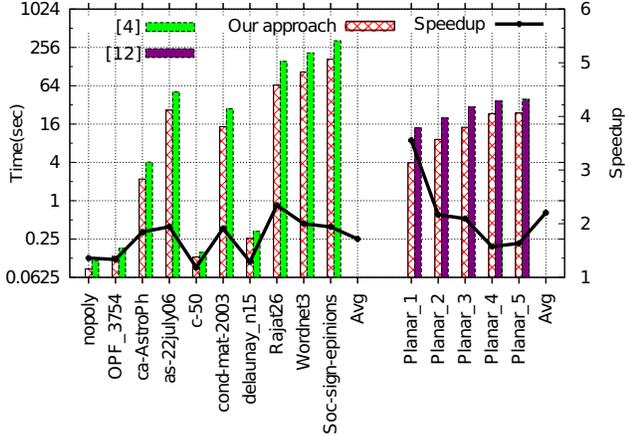


Fig. 2. Figure displays the absolute time taken by our approach, labeled "Our Approach" compared to [4] for general graphs and [12] for planar graphs.

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 the 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. An average speedup of 2.2x achieved is mentioned in the right most column of the Figure. 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].

b) 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 plot contains the overall timings for both the implementations on the Y1-axis on the left. The timings displayed on the Y1-axis are on a logarithmic scale. The Y2-axis

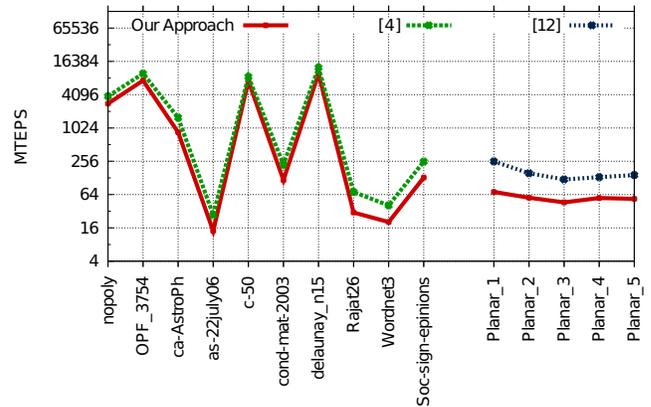


Fig. 3. MTEPS achieved by our algorithm, labeled "Our Approach" and that of [4] for general graphs and [12] for planar graphs.

denotes the speedup achieved by our algorithm with respect to that Banerjee et al. [4]. The average speedup achieved is 1.7x.

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 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].

III. MINIMUM WEIGHTED CYCLE BASIS (MCB)

A cycle in an undirected graph is a subgraph where every vertex has a degree of two. A cycle can be represented by its incidence vector on $E(G)$. These vectors are known to belong to a space of dimension $m - n + k$ if there are k connected components in the graph. A maximal set of linearly independent incidence vectors of cycles is known as a cycle basis of the graph. In a graph G with a weight function $W : E \rightarrow R^+$, the weight of a cycle 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 weighted graph.

A. Sequential Algorithms

We will now summarize the deterministic sequential algorithms from [1], [11], [18], [27] for obtaining an MCB. Horton [18] provided the first polynomial time algorithm for computing an MCB. Horton showed that the 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], [27] use the idea that the Horton cycles of G with respect to a feedback vertex set of $V(G)$ suffices. A feedback vertex set is a set of vertices that are present in every cycle [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 $E' = E \setminus T$ 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 $\{e_1, e_2, \dots, e_f\}$ and consider the vector representation of a cycle C incident on the restricted edge set consisting of edges in 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 $i = 1, 2, \dots, f$, with \vec{S}_i having 1 in the i th component and 0 in all other components. Each such vector \vec{S}_i is called as a *witness*. 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  $i = 1, \dots, f$  do
3:   find  $\vec{C}_i$  that satisfies  $\langle \vec{C}_i, \vec{S}_i \rangle = 1$ 
4:   for  $j = i + 1, \dots, f$  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 perform an *independence test* that updates the remaining witnesses to make them orthogonal to cycles $\vec{C}_1, \dots, \vec{C}_i$. This is done by taking the symmetric difference between \vec{S}_i and \vec{S}_j .

B. Our Approach for Parallel MCB

In this section we describe our algorithmic approach for obtaining an MCB in parallel. To the best of our knowledge, we provide the first parallel implementation for the MCB problem. 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.

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 $MCB(G^r)$ be a cycle basis of minimum weight on the graph G^r . Then,

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

Proof: For each cycle $C \in \text{Horton-Set}(G^r)$, there can be two cases, i.e. either C contains some $e_{P_1}, e_{P_2}, \dots, e_{P_t}$ 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 $i \in 1, \dots, t$. Hence, C is equivalent to a cycle C_k obtained as $C_k = C - \{e_{P_1}, e_{P_2}, \dots, e_{P_t}\} \cup (\cup_{i=1}^t P_i)$ 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.

The dimensions of the orthonormal basis vectors are based on 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 $\dim|MCB(G)| = \dim|MCB(G^r)|$. 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 $i \in 1, \dots, |\mathcal{P}|$, 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 $u_1, u_2, \dots, u_{k+1}, u_{k+2}$ where u_1 and u_{k+2} are non degree-two endpoints of P_i . Exactly one of the edges $e = u_j u_{j+1}$ for some $j \in 2, \dots, k+1$ is a non-tree edge. Nodes u_l for $l \in 2, \dots, k+1$ 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, 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 MCB depends upon the number of non-tree edges in G and G^r , this proves statement 3 of the lemma.

Since the weights of cycles in $\text{Horton-Set}(G)$ are same as those in $\text{Horton-Set}(G^r)$, hence $W|\text{MCB}(G)| = W|\text{MCB}(G^r)|$ and this proves statement 4 of the lemma.

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

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.

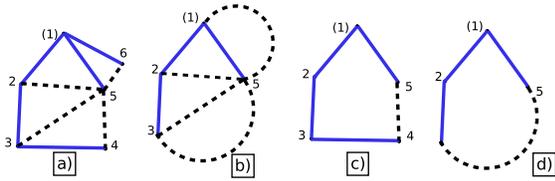


Fig. 4. In the above figure, (a) is the original graph G while (b) is the corresponding reduced graph G^r . The non-tree edges in graph G and G^r are displayed as dashed lines. Nodes 4, 6 are degree two vertices, that are pruned out using Ear-Decomposition. The degree-two chains $\{1,6,5\}$ and $\{3,4,5\}$ in G are replaced by respective edges $\{1,5\}$ and $\{3,5\}$ in G^r . (c) represent a subset of $\text{Horton-Set}(G)$ with node 1 as the root node. (d) represent the cycle in G^r corresponding to (c) with its degree two chain replaced by single edges. Note that, $W(c) = W(d)$.

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 the next subsection, we explain in detail about the witness update step (Independence Test).

a) *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} .

The algorithm needs to compute single source shortest path trees from each node to find cycles that can be added to an MCB. For each node present in the reduced graph, a shortest path tree is constructed with the node as root. 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 next describe Mehlhorn's approach in brief for computing such a cycle, and skip the proof of correctness.

- *A reduced collection*: Recall that the Horton cycles with respect to an FVS Z of G is a superset of the cycles in an MCB. Mehlhorn [27] shows that a further reduction in the number of Horton cycles is possible. For $z \in Z$, 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 cycle 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 [27].
- *Label Computation*: 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 $z \in Z$. The labels are computed as follows.

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 in the tree T_z . In the first pass for every edge, $e = uv \in T_z$ s.t. v is parent of u , we set $c_z(u) = 0$, if $e \notin E'$, 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 , $l_z(u) = l_z(v) \oplus c_z(u)$. The overall work done for all labels in all the $|\mathcal{Z}|$ trees is $O(n|\mathcal{Z}|)$. This step can be computed in parallel by allocating each thread to work on a single tree T_z . Level-order traversals can be carried out from every $z \in \mathcal{Z}$ in parallel. The computation is simple and only requires linear extra storage for every traversal.

- *Cycle Identification*: In this step, using the labels computed previously, we search for a minimum weighted cycle in \mathcal{A} which is non-orthogonal to \vec{S}_{curr} . The labels for every tree T_z are already computed. 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 a restricted edge set of E' as defined in the basic 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 $\langle \overrightarrow{path}_z(u), \vec{S}_{curr} \rangle$. 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 [27], 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

$\langle \vec{C}_i, \vec{S}_{curr} \rangle = 1$ and remove it from \mathcal{A} . There are $O(mn)$ such cycles, and hence it takes $O(mn)$ work in the worst case for every cycle. This task can be parallelized with an early-exit terminating condition.

We execute the cycle inspection step in parallel for an initial set of cycles and store the index. We do a reduction to find the minimum such index and if no such cycle has been found, we proceed to the next set of the cycles. We stop when an index is present in any iteration.

b) *Witness Update*: We update each witness \vec{S}_j , where $\langle \vec{S}_j, \vec{C}_i \rangle = 0$. Witnesses are updated to make them orthogonal to each \vec{C}_k , for $k \in 1, \dots, i$. 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.

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 .

C. Implementation Details

Our heterogeneous algorithm for MCB is divided into *tasks* as listed in Figure 5. Figure 5 (a) provides a timeline displaying every task for a single instance of a graph that reveals the inherent sequential dependencies among the tasks. Figure 5 (b) contains the description of each task. A brief pseudocode of our algorithm is listed in Algorithm 3. The label **{cpu,gpu}** in Algorithm 3 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). We next describe the implementation aspects with respect to each of the tasks in rest of the subsections.

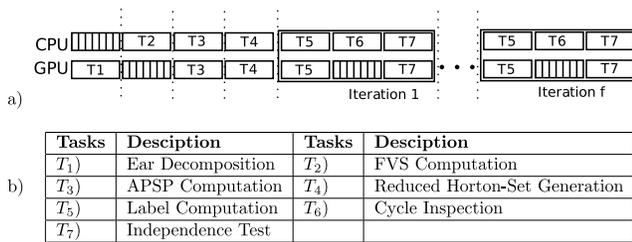


Fig. 5. a) lists the timing diagram for every task with respect to CPU and GPU based on the heterogeneous implementation. b) contains the Task IDs mapped to their description.

a) *Task T_1 : Ear-Decomposition*: In this task, we first obtain the ear decomposition of the input graph G and obtain the corresponding reduced graph G^r . We implement this algorithm on the GPU following the recent work from [30].

b) *Task T_2 : FVS Computation*: In this task, we obtain a 2-approximate FVS of the vertices in G^r . This is done by using the algorithm of Bafna et al. [3]. We use the sequential algorithm from [3].

Algorithm 3 Hybrid MCB algorithm(G)

```

1: {gpu}:  $G^r \leftarrow \text{ear\_decomposition}(G)$  {Task  $T_1$ }
2: {cpu}:  $\mathcal{Z} \leftarrow \text{FVS}(G^r)$  {Task  $T_2$ }
3: {cpu,gpu}:
4: for  $z = 1, \dots, |\mathcal{Z}|$  in parallel do
5:    $T_z \leftarrow \text{compute\_apsp}(G^r)$  {Task  $T_3$ }
6: end for
7: {cpu}:  $\mathcal{A} \leftarrow \text{construct\_cycles}(T_z, G^r), \forall z \in |\mathcal{Z}|$  {Task  $T_4$ }
8: for  $i \in 1, \dots, f$  do
9:   {cpu,gpu}:  $l_z(u) \leftarrow \text{label\_computation}(T_z), \forall z \in |\mathcal{Z}|,$ 
    $\forall u \in 1, \dots, n$  {Task  $T_5$ }
10:  {cpu}:  $\vec{C}_i \leftarrow \text{cycle\_inspection}(l_z(u), \mathcal{A}), \forall z \in |\mathcal{Z}|,$ 
    $\forall u \in 1, \dots, n$  {Task  $T_6$ }
11:  {cpu,gpu}:  $\text{independence.test}(\vec{C}_i)$ . {Task  $T_7$ }
12: end for

```

c) *Task T_3 : APSP Computation*: In this task, we need to compute shortest paths from each $z \in \mathcal{Z}$ to all vertices in G^r . For this, we use Algorithm 1 from Section II. The output of this task is a collection of shortest path trees, one for each $z \in \mathcal{Z}$.

d) *Task T_4 : Generating Horton Cycles \mathcal{A}* : In this task, we generate the reduced collection of Horton cycles \mathcal{A} in parallel in both CPU and the GPU. For this, note that the computation with respect to each shortest path T_z is independent. For each node u in T_z , we first calculate a value $V_z(u)$ that represents the first children of root in z to u path in T_z . Note that $V_z(z) = -1$. This can be done in $O(n)$ time per T_z . A cycle, C_{ze} induced by an edge $e = uv$ in T_z , would be added in \mathcal{A} only when LCA of u and v in T_z is z . This can be done in constant time per thread for every e by checking whether $V_z(u) \neq V_z(v)$ which indicates that z is the LCA of u, v .

e) *Task T_5 : Label Computation*: The computation in this task is done with respect to the shortest path tree for each $z \in \mathcal{Z}$. However, the computation of each tree can be done independently. Hence, we have the CPU and the GPU share the computation in this task.

Let T_z be the shortest path tree for some $z \in \mathcal{Z}$. We perform a BFS traversal on T_z and assign the labels as described in Section III-B.2.a. We note that on GPUs and CPUs there are existing implementations such as [28], [5] that are work-efficient for general graphs. Our approach instead requires a traversal on trees and can hence be simplified. To this end, we first describe our representation for trees followed by the procedure used to compute the labels.

- *Tree Representation*: We use two arrays similar to the compressed sparse row (CSR) representation. The leveloffset array is used to store the offset of each level indexed from 0 to the maximum number of levels in a given rooted tree. Note that $\text{leveloffset}[i + 1] - \text{leveloffset}[i]$ gives the number of nodes in the i^{th} level. The Index_Parent array contains the index of parent of the current node. The crucial property is that parents of all nodes in a level belong to the previous level. This helps maintain the coalescent property for accessing the parents

in the GPU and also making it cache-friendly for CPU. The above modified representation uses space similar to a CSR representation.

- *Prefix label computation in GF(2)*: There are two computations in this step. In the first computation, we update corresponding edges in every root to leaf path once. We maintain a linear storage L_z for every tree T_z . For every edge $e = uv$ with v as the parent of u in T_z , $l_z(u)$ is set to $\vec{S}_{curr}(e)$ if $e \in E'$, and $L_z(u)$ is set to 0 otherwise. For this step, in a GPU kernel, we assign each tree to a cuda *block*. Every thread in a block processes one edge of the tree. L_z contains the final label values for every $u \in T_z$ at the end of next phase. Threads on the CPU work on different T_z 's independently and compute the value of $L_z(u)$ in a sequential manner for every $e = uv$ with v as the parent of u in T_z .

In the second step, we calculate a label for each node in the tree using a BFS traversal as mentioned in Section III-B.2.a. In the GPU kernel each warp begins a BFS independently on a tree T_z using the data structure defined in Section III-C.0.e and proceeds one level at a time. Let lv denote the current level being processed by a warp. All the threads of a warp compute the labels of all the nodes in lv in batches. If there are less nodes than the batch size, then the threads remain idle. Each thread processes one node per batch. For each node u , $L_z(u)$ is obtained as $L_z(u) = L_z(u) \oplus L_z(\text{parent}(u))$ where $\text{parent}(u)$ indicates parent of u in T_z and lies in level $lv - 1$. Warps are executed in a lock-step basis and hence can be implicitly synchronized at every level.

We next describe the computation of final label values for the CPU. BFS traversal in CPU is done in a level-ordered manner using the above datastructure. We begin with vertices in level 1 and update the label values, $L_z(u)$ in a sequential manner for all the vertices corresponding to the current level before moving on to the next level.

f) *Task T_6 : Cycle Identification*: In this task we look for the minimum cycle amongst those in \mathcal{A} that satisfies the condition in Section III-B.2.a. We do this in CPU. Note that \mathcal{A} contain cycles in a sorted order. We arrange the cycles in \mathcal{A} into logical batches. We check for a cycle satisfy 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. We record the removal of the chosen cycle by setting a Boolean flag.

g) *Task T_7 : Independence Test*: This step implements the witness update step as specified in Section III-B.2.b. 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 dot 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

TABLE II
LIST OF TIMINGS (IN KS (KILO SECONDS)) OF FOUR DIFFERENT IMPLEMENTATIONS FOR OUR APPROACH. LABELS 'w' AND 'w/o' INDICATE WITH AND WITHOUT USING LEMMA 3.1.

Graphs	Sequential		Multi-Core		GPU		CPU + GPU	
	w	w/o	w	w/o	w	w/o	w	w/o
nopoly	7.83	7.83	2.34	2.35	0.602	0.604	0.624	0.624
OPF_3754	44.58	44.58	11.8	11.8	3.8	3.8	3.2	3.2
ca-Astro	246.3	271.3	75.06	81.5	38.04	40.15	27.6	27.6
as-22july06	0.57	7.4	0.17	1.8	0.134	1.29	0.09	0.94
c-50	17.05	28.07	6.17	9.8	2.90	4.278	2.02	3.03
cond_mat_2003	141.3	177.6	35.9	44.2	14.89	17.97	10.9	13.2
delanay_n15	272.5	272.5	59.5	59.5	18.37	18.37	15.8	15.8

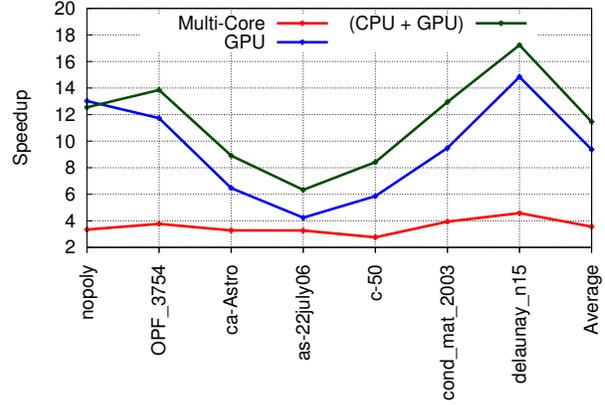


Fig. 6. Figure showing the speedup of the Multi-core, GPU and Heterogenous(CPU+GPU) versions of our algorithm with respect to a sequential implementation. All implementations use Lemma 3.1.

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 do a symmetric-difference of \vec{S}_{curr} with \vec{S}_j in parallel for the entire block.

We now comment on the aspects that affect all parts of our implementation. We can see that our tree representation induces negligible branch divergence and our memory accesses are mostly coalescent and cache-friendly. Tasks T_4 , T_5 and T_7 are done simultaneously on both the GPU and the CPU. 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.

D. Experimental Results

In this section, we describe the results of implementing our heterogeneous algorithm for obtaining an MCB. For our

experiments, we use the first seven graphs listed in Table I. 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 II lists the total time spent in each of four implementations. Figure 6 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 II 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 tasks T5, T6 and T7 have a major impact on the overall execution time. Task T5 precedes T6 and T6 precedes T7 which creates a sequential dependency. This has a limiting effect on the parallelism of the overall algorithm. We observe that task T5 takes on an average about 76% of the total time, task T6 about 14% and task T7 about 8%. Thus optimizing task T5 leads to a massive improvement on the overall execution time. Graphs such as ca-AstroPh display a lower speedup due to a large percentage of time spent in task T6 and T7 (41% and 31% respectively), whereas graphs like delaunay_n15 has roughly 96% of its time spent in task T5 thus leading to higher speedups.

IV. CONCLUSIONS

In this paper, we considered path based problems on graphs and proposed a two stage preprocessing based on graph decomposition as well as graph reduction. Our results on APSP and MCB indicate that our technique can be useful and practical. We believe that our technique can have independent interest and can be applied to other graph problems.

REFERENCES

- [1] AMALDI, E., IULIANO, C., JURKIEWICZ, T., MEHLHORN, K., AND RIZZI, R. Breaking the $o(m^2n)$ barrier for minimum cycle bases. In *European Symposium on Algorithms* (2009), Springer, pp. 301–312.
- [2] BADER, D. A., AND CONG, G. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Journal of Parallel and Distributed Computing* 65, 9 (2005), 994–1006.
- [3] BAFNA, V., BERMAN, P., AND FUJITO, T. A 2-approximation algorithm for the undirected feedback vertex set problem. *SIAM Journal on Discrete Mathematics* 12, 3 (1999), 289–297.
- [4] BANERJEE, D. S., KUMAR, A., CHAITANYA, M., SHARMA, S., AND KOTHAPALLI, K. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *Journal of Parallel and Distributed Computing* 76 (2015), 81–93.
- [5] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3-4 (2013), 137–148.
- [6] BULUC, A., GILBERT, J. R., AND BUDAK, C. Solving path problems on the gpu. *Parallel Computing* 36, 5 (2010), 241 – 253.
- [7] CHHUGANI, J., SATISH, N., KIM, C., SEWALL, J., AND DUBEY, P. Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In *IPDPS* (2012), pp. 378–389.
- [8] CHIMANI, M., GUTWENGER, C., JÜNGER, M., KLAU, G. W., KLEIN, K., AND MUTZEL, P. The open graph drawing framework (ogdf). *Handbook of Graph Drawing and Visualization* (2011), 543–569.
- [9] CORMEN, T., LEISERSON, C., RIVEST, R., AND STEIN, C. Introduction to algorithms, 2001.
- [10] DAVIS, T. A., AND HU, Y. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
- [11] DE PINA, J. C. *Applications of shortest path methods*. PhD thesis, University of Amsterdam, Netherlands, 1995.
- [12] DJIDJEV, H., CHAPUIS, G., ANDONOV, R., THULASIDASAN, S., AND LAVENIER, D. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing* 85 (2015), 91–103.
- [13] GHARAIBEH, A., COSTA, L. B., SANTOS-NETO, E., AND RIPEANU, M. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *Proc. of IEEE IPDPS* (2013).
- [14] GLEISS, P. M. *Short cycles: minimum cycle bases of graphs from chemistry and biochemistry*. na, 2001.
- [15] GOTSMAN, C., KALIGOSI, K., MEHLHORN, K., MICHAIL, D., AND PYRGA, E. Cycle bases of graphs and sampled manifolds. *Computer Aided Geometric Design* 24, 8-9 (2007), 464–480.
- [16] HARISH, P., AND NARAYANAN, P. J. Accelerating Large Graph Algorithms on the GPU using CUDA. In *Proc. of HiPC* (2007).
- [17] HONG, S., RODIA, N. C., AND OLUKOTUN, K. On fast parallel detection of strongly connected components (sc) in small-world graphs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for* (2013), IEEE, pp. 1–11.
- [18] HORTON, J. D. A polynomial-time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing* 16, 2 (1987), 358–366.
- [19] INDARAPU, S. B., MARAMREDDY, M. K., AND KOTHAPALLI, K. Architecture- and workload- aware heterogeneous algorithms for sparse matrix vector multiplication. In *ACM COMPUTE* (2014), pp. 3:1–3:9.
- [20] KARP, R. M. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 1972, pp. 85–103.
- [21] KARYPIS, G., AND KUMAR, V. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Intl. Conf. Par. Proc.* (1995), pp. 113–122.
- [22] KARYPIS, G., AND KUMAR, V. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proc. ACM SC* (1996).
- [23] KATZ, G. J., AND KIDER, JR, J. T. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM Symp. Grap. Hard.* (2008), pp. 47–55.
- [24] KAVITHA, T., MEHLHORN, K., MICHAIL, D., AND PALUCH, K. A faster algorithm for minimum cycle basis of graphs. In *International Colloquium on Automata, Languages, and Programming* (2004), Springer, pp. 846–857.
- [25] KAVVADIAS, D. J., PANTZIOUC, G. E., SPIRAKIS, P. G., AND ZAROLIAGIS, C. D. Hammock-on-ears decomposition: A technique for the efficient parallel solution of shortest paths and other problems. *Theoretical Computer Science* 168 (1996), 121154.
- [26] MATSUMOTO, K., NAKASATO, N., AND SEDUKHIN, S. Blocked All-Pairs Shortest Paths Algorithm for Hybrid CPU-GPU System. In *Proc. HPCC* (2011), pp. 145–152.
- [27] MEHLHORN, K., AND MICHAIL, D. Minimum cycle bases: Faster and simpler. *ACM Transactions on Algorithms (TALG)* 6, 1 (2009), 8.
- [28] MERRILL, D. G., AND GRIMSHAW, A. S. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques* (New York, NY, USA, 2010), PACT '10, ACM, pp. 545–546.
- [29] MICIKEVICIUS, P. General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem. In *PDPTA'04* (2004), pp. 1359–1365.
- [30] PACHORKAR, C., CHAITANYA, M., KOTHAPALLI, K., AND BERA, D. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In *High Performance Computing (HiPC)* (2016), IEEE, pp. 301–310.
- [31] RAMACHANDRAN, V. Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity, 1993.
- [32] SARYUCE, A. E., SAULE, E., KAYA, K., AND CATALYUREK, U. V. Betweenness centrality on gpus and heterogeneous architectures, 2013.