2017 IEEE International Parallel and Distributed Processing Symposium Workshops

A Study of Graph Decomposition Algorithms for Parallel Symmetry Breaking

Sayyad Nayyaroddeen, Mahak Gambhir, and Kishore Kothapalli
International Institute of Information Technology, Hyderabad
Gachibowli, Hyderabad, India 500 032.
Email: {sayyad.n,mahak.gambhir}@research.iiit.ac.in, kkishore@iiit.ac.in

***Abstract*—Parallel algorithms on large graphs play a prominent role in various problems from several domains of sciences and engineering. Of these, symmetry breaking problems such as matchings and colorings are fundamental owing to a large number of applications. Algorithms for symmetry breaking problems are studied in several parallel/distributed settings over the decades.**

However, as the size of graphs corresponding to real-world phenomenon increase, it is imperative to use not only just parallelism but also algorithmic enhancements based on the structure of real-world graphs. A popular approach in this context is to use a graph decomposition to break the problem into multiple subproblems the solutions of which can be used to solve the original problem. A big question in this direction that is left unanswered by current research is to study which decomposition is appropriate for a given computation and a target architecture.

In this context, we address the above question with respect to three problems: Maximal Matching (MM), Vertex Coloring (COLOR), and Maximal Independent Set (MIS). For these three problems, we study three different decomposition techniques including one based on bridges, one based on a random partitioning, and one based on vertices of degree k for a given k. We show that existing algorithms for the above computations on a multi-core CPU and a GPU can be significantly improved by making use of an appropriate decomposition of the input graph. Our study indicates that the exact decomposition to use depends more on the problem and is largely independent of target architecture between the CPU and the GPU.

I. Introduction

Graphs have applications to a variety of problems in science and engineering. This, coupled with the emergence of multi- and many-core accelerators over the last decade, has led to a renewed interest in parallel algorithms for graphs. Several graph problems including shortest paths, traversals, spanning tree construction, now are known to have very efficient implementations on parallel architectures such as multi-core CPUs, GPUs, and heterogeneous collections, see e.g., [2], [11], [13], [25]. Library frameworks [30] that offer a collection of parallel implementations for graph primitives are also proposed [30].

In recent years, one is witnessing a trend where parallel algorithms and their implementations are designed with special focus on real-world graphs. A popular approach in this direction is to decompose the graph into subgraphs, solve the original problem on the subgraphs, and then assemble the solutions on the subgraphs to arrive at a solution on the original graph. Decompositions such as those based on Metis and ParMetis, based on biconnected components, based on edge partitioning, and random decompositions are currently used by various authors for problems such as shortest paths, and betweenness centrality [2], [11], [25], [27].

However, one important question that is left unanswered by most current works using the decomposition model is to understand which decomposition is most suitable for a given graph problem and a given parallel architecture. In this paper, we address the above question while focusing on symmetry breaking problems. Symmetry breaking problems include problems such as a Maximal Matching (MM), Coloring (COLOR), and Maximal Independent Set (MIS). These problems have applications to settings such as scheduling and efficient sparse matrix computations [29] and are being studied actively in the parallel computing community [1], [6], [10].

In our study, we consider four light-weight decomposition techniques: based on bridges (BRIDGE), based on random partitioning (RANDOM), and based on vertices of degree k for a given k (DEGk), Metis/ParMetis based (METIS). These decompositions induce 2-edge connected components, random induced subgraphs, induced subgraphs of high and low degree vertices, and induced subgraphs with fewer cross edges, respectively.

Our study in this paper has two lessons. Firstly, we show that graph decomposition can lead to significant parallel performance gains for MM, COLOR, and MIS. Secondly, our study indicates that within these problems, the best partitioning mechanism depends on the problem and is relatively agnostic to the architecture between the CPU and the GPU. For instance, for the coloring problem on multicore CPUs, we show that the DEGk partitioning with $k = 2$ is most suitable as the DEGk partitioning induces subgraphs whose independent coloring is also a valid coloring (See Section IV).

A summary of our results is contained in Table I where we show the best decomposition strategy and the corresponding speed up obtained for problems MM, COLOR, and MIS. It can be noticed from Table I that the improvement obtained by the decomposition based algorithm is significant except for the case of the COLOR problem on GPUs.

A. Related Work

One of the prominent works that suggests the use of graph decomposition in solving symmetry breaking problems is that of Hoschbaum [16]. Hoschbaum [16] uses a decomposition based on the biconnected components of the graph to arrive at solutions for problems including vertex cover, matching,

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.120

TABLE I: A Summary of our Results.

	CPU		GPU	
Problem	**Decomposition**	**Speedup**	**Decomposition**	**Speedup**
MM	RAND	3.5X	RAND	2.53X
COLOR	DEGk	1.27X	RAND	1X
MIS	DEGk	3.39X	DEGk	2.16X

coloring and maximum flow problems. In addition to studying the techniques proposed in [16], we focus on the applicability of other decomposition strategies and also study parallel algorithms.

Addressing the concerns of practicality, in recent years, symmetry breaking on modern parallel architectures is being keenly studied. Prominent works in this direction for MM include the algorithm and its implementation by Auer and Bisseling [1], Blelloch [6], Birn [15]. For COLOR, the vertex–based and the edge–based algorithms of [10] respectively improve on the early work of [7], [12]. For MIS, the latest reported results are from Blelloch [6]. More details of these and related works are provided in Sections III–V.

B. Organization of the Paper

The rest of the paper organized as follows. In Section II we present the decomposition techniques. In Sections III–V we discuss the application of the decomposition algorithms to MM, COLOR, abd MIS respectively. The paper ends with concluding remarks in Section VI.

II. Decomposition Algorithms

In this section, we briefly describe the decomposition techniques used in the subsequent sections. We also outline parallel algorithms that can arrive at the required decomposition.

A. **BRIDGE**: *A Bridge Based Decomposition Algorithm*

Algorithm 1 Bridge Decomposition Algorithm

1: Dcmp_Bridge(Graph G)
2: $T \leftarrow BFS(G)$ /*STEP 1*/
3: **for** all $e = (w, v) \in G \setminus T$ **do**
4: Mark the tree edges in the process of computing the least common ancestor of w and v
5: /* STEP 2 */
6: **for** $e \in E(T)$ **do**
7: If e is not marked then B $\leftarrow$ B$\cup$ $\{e\}$
8: **return** B and $\{G_1, G_2, ...\}$ as the connected components of G–B

Algorithm 1 (see also [8]) describes an approach to decompose the graph into 2-edge-connected components. In Algorithm 1, there are two major steps to decompose a graph. In Step 1, a BFS tree is found and in the next step the least common ancestor is found for each of non–tree edges. To compute the BFS tree we take an arbitrary vertex r as root and perform a parallel BFS. We fix vertex r as the root of the tree. The output of Step 1 is a parent array, $P(v)$ and a level array, $L(v)$. For the root r, we set $P(r) = -1$, and $L(r) = 0$. We use a basic property of the graph to find bridges. Any bridge in a graph can never be a part of a cycle. In Step 2, for each non–tree edge $e = xy$, we find the least common ancestor (LCA) of x and y by walking up the tree towards the root from x and y in parallel. Each edge encountered in this walk is marked. Edges of T that are not marked at the end of the for-loop in Step 2 are the bridges of G. Given the bridges of G, the graph G can be decomposed by removing these bridges. This results in a collection of 2-edge connected subgraphs $G_1, G_2, \cdots$, of G. An example for the BRIDGE decomposition is given in Figure 1 (b).

B. **RAND**: *A Randomized Decomposition Algorithm*

Algorithm 2 Randomized Decomposition Algorithm

1: Dcmp_Rand(Graph G, size)
2: **for** all $v \in V$ in parallel **do**
3: Choose an integer $i \in \{1, 2, \cdots, k\}$ uniformly at random
4: Assign v to V_i
5: **return** $G[V_i]$ for $1 \leq i \leq k$ and G_{k+1} as the edge induced subgraph of $\{uv | u \in V_i$ and $v \in V_j, i \neq j\}$.

We study a randomized decomposition of a graph G as follows. For a given $k \geq 2$, the vertex set is partitioned into $V_1, V_2, \cdots, V_k$ subsets. For each i, the set to which v_i belongs to is chosen uniformly at random. The graph G can be decomposed as the induced subgraphs $G_j = G[V_j]$ for $1 \leq j \leq k$. We also let G_{k+1} be the subgraph that has edges uv of G such that u and v do not belong to the same vertex partition. We also refer to the edges in G_{k+1} as *cross edges*. The graphs G_j, for $1 \leq j \leq k+1$, may be disconnected in nature. We expect that applications that use the RAND decomposition will work with graphs that are not necessarily connected. The interest in this decomposition technique is due to its simplicity. It may be noted that a similar decomposition is used in designing algorithms in the k-machine model [20]. A pseudocode for **RAND** decomposition is given in Algorithm 2. An example for the RAND decomposition is given in Figure 1 (c).

C. **DEGk**: *A Degree Based Decomposition Algorithm*

Algorithm 3 Degree-k Decomposition Algorithm

1: Dcmp_Degreek(Graph G, k)
2: $V_L \leftarrow \emptyset$
3: $V_H \leftarrow \emptyset$
4: **for** all $v \in V$ in parallel **do**
5: **if** degree(v) $\leq$ k **then**
6: $V_L \leftarrow V_L \cup \{v\}$
7: **else**
8: $V_H \leftarrow V_H \cup \{v\}$
9: **return** $G_L \leftarrow G[V_L]$; $G_H \leftarrow G[V_H]$; and $G_C \leftarrow G \setminus \{G_H \cup G_L\}$

In the Degree-k based decomposition, the graph is decomposed into three subgraphs G_H, G_L and G_C. The subgraph

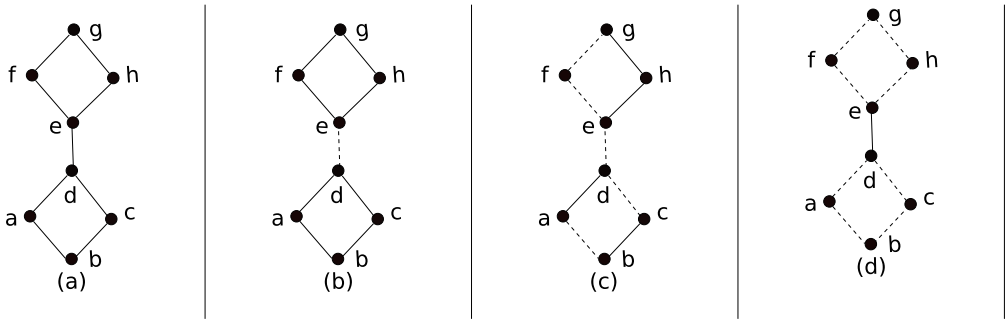


Fig. 1: (a) input graph G. (b) BRIDGE decomposition of G. (c) RAND decomposition of G with group size 2. Vertices $\{b, c, e, h, g\}$ are in group 1 and $\{a, d, f\}$ are in group 2. (d) DEG2 decomposition of G.

TABLE II: LIST OF GRAPHS USED IN OUR EXPERIMENTS

Graph Class	Graph Name	V	E	% DEG2	%BRIDGES	Avg Degree
Numerical simulations	c-73	169,422	1,109,852	48.7	14.9	6.6
	lp1	534,388	1,109,032	93.8	92.7	2.1
Collaboration	Cit-Patents	3,774,768	33,045,146	28.06	4.1	8.8
	coAuthorsCiteseer	227,320	1,628,268	28.97	3.7	7.2
Road	germany-osm	11,548,845	24,738,362	82.27	19.9	2.1
Clustering	road-central	14,081,816	33,866,826	50.91	25	2.4
Synthetic	kron-g500-logn20	1,048,576	89,238,804	42.1	0.3	85.1
	kron-g500-logn21	2,097,152	182,081,864	44.59	0.3	86.8
Random geometric	rgg-n-2-23-s0	8,388,608	127,002,794	0	0	15.1
	rgg-n-2-24-s0	16,777,216	265,114,402	0	0	15.8
Web	web-Google	916,428	10,296,998	30.67	4	11.2
	webbase-1M	1,000,005	4,216,602	87.35	38.3	4.2

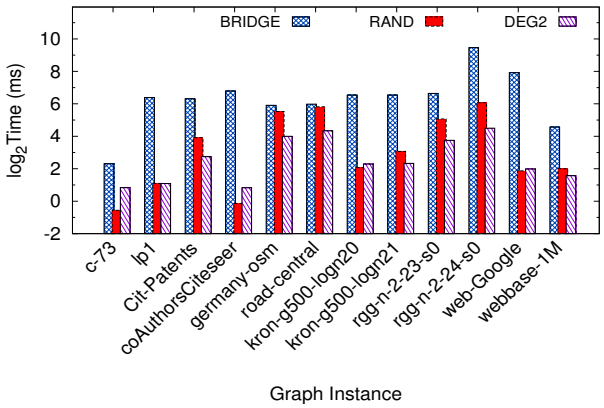


Fig. 2: Figure shows the time required for each decomposition technique for the graphs listed in Table II.

G_H is the subgraph induced by vertices degree more than k. The subgraph G_L induced by vertices of degree at most k. G_C includes the cross edges between G_H and G_L. Notice that these resulted subgraphs need not necessarily be connected. We expect that applications that use the DEGk decomposition will work with graphs that are not necessarily connected. This decomposition is seen to be helpful in cases where a parallel traversal of the graph, such as BFS, can be slow due to the large diameter of the underlying graph. A pseudocode for **DEGk** decomposition is given in Algorithm 3. An example for the DEGk decomposition is given in Figure 1 (d).

D. Decomposition Time

We briefly study the time taken by the decomposition algorithms. We start by describing the dataset and the experimental platform, also used in the rest of the paper.

1) Dataset: We experiment on different classes of undirected graphs with varying number of vertices and edges. The graphs are obtained from the University of Florida graph dataset [9]. The graphs are listed in Table II. For graphs that are not connected, we add additional edges to make the graph connected. Directed edges are converted to undirected edges and self-loops in the graphs are ignored.

2) 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 hyperthreading 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.

3) Results: Figure 2 shows the timings for each of the decomposition techniques on the E5-2650 CPU running 80 threads. Among all the decomposition techniques, DEG2 decomposition takes the least time since it involves a simple computation. RAND decomposition is the second fastest of the four techniques. In BRIDGE decomposition, the BFS

computation can become a bottleneck on graphs with a large diameter in particular. The timings given in Figure 2 are for only decomposing each graph to 10 sub-graphs when using the RAND decomposition algorithm. As the number of sub-graphs increases time also increases. A similar trend is observed also on GPUs.

Remark 1: PMETIS [19] is another popular graph decomposition technique that is used in several parallel graph algorithms. However, for the applications we consider in this paper such as MM, COLOR, and MIS, the current best practical implementations [6], [10] in most cases finish faster than the time it takes to decompose the graph using PMETIS. For this reason, we exclude PMETIS from our study.

III. Application to Maximal Matching

Given a graph $G = (V, E)$ on a set of vertices V and edges E, a matching M is set of edges such that each vertex v has at most one edge in M that is incident on v. The number of edges $|M|$ in matching M is called the cardinality of the matching. A matching M is called maximal if no proper superset of M also is a matching. Matching has applications to scientific computing such as in the solution of sparse matrix-vector multiplication [29] and in multi-level graph algorithms for partitioning [15]. We start by reviewing existing approaches to maximal matching in Section III-A. In section III-B, we show our algorithms and follow it with experimental results in Section III-C.

A. Existing Algorithms

One can summarize most existing algorithms [1], [5], [6], [17] on mutlicore CPUs and GPUs for obtaining a maximal matching as follows. Vertices that are yet unmatched find a potential mate and propose to match with the chosen mate. A subset of these proposals are found viable. This creates a set of matched vertices and a set of vertices are left unmatched. The vertices that are matched will also imply that some edges are forbidden to be used for extending the matching. After removing these forbidden edges, the algorithm will work on the unmatched vertices and the remaining edges.

a) Multicore CPU Algorithms: Blelloch et al. [6] propose a randomized greedy parallel algorithm for maximal matching. This algorithm, denoted Algorithm GM in the rest of the discussion, assigns random priorities for the edges of G. For each vertex, its adjacent edges are sorted by priority. The priorities on the edges can be used to induce a Directed Acyclic Graph (DAG) with edges that have no incoming neighboring edges with higher priority called as roots of the DAG. A matching can be found by removing the roots from the DAG. The process is repeated until there are no more roots to remove in the DAG. For more details, we refer the reader to [6].

b) GPU Algorithms: On GPUs, Fagginger and Bisseling [1] proposed a randomized matching algorithm. This has since been improved by Birn et al. [5] via a procedure that relies on local maxima of the edge weights. The algorithm from [5] starts with an empty matching M. Every vertex tries in parallel to find the adjacent heaviest edge. If a vertex v finds an edge vw as local maximum and the other end point w also finds that wv as the local maximum, then the edge vw is added to the matching M. The adjacent edges of vw, and the vertices v, w are removed from the graph G. This process repeats until there are are no more edges left to be matched. In the rest of the paper we refer to this algorithm as Algorithm LMAX.

B. Maximal Matching Algorithms Based on Graph Decomposition

For all the decomposition based maximal matching algorithms on multicore CPUs, we use Algorithm GM as a subroutine and also as a baseline in our experimental results. For our algorithms on GPUs, we use Algorithm LMAX as a subroutine and also as a baseline in our experimental results. In our pseudocodes we use MM acronym for corresponding algorithms.

1) Algorithm MM-Bridge: In Algorithm MM-Bridge, we apply the BRIDGE decomposition to the given input graph G. As described in Section II, BRIDGE decomposition reduces the graph G into a collection of two-edge-connected components $G_1, G_2, \cdots$, and a set of bridges $B := \{e_1, e_2, \cdots\}$. We call the end points of the edges in B as *bridge vertices*. We now process $G_1, G_2, \cdots$ in parallel to find maximal matchings $M_1, M_2, \cdots$ respectively using a suitable maximal matching algorithm. Based on the matching $M_c := \cup_i M_i$, we collect the unmatched bridge vertices, say V', in G_b. Next we augment the matching M by finding, in parallel, a maximal matching M_b in the subgraph of G induced by V'. The matching $M := M_c \cup M_b$ is a maximal matching of G. A pseudocode for Algorithm MM-Bridge is given in Algorithm 4.

2) Algorithm MM-Rand: In Algorithm MM-Rand, we apply the RAND decomposition given in Section II to the input graph G prior to computing a maximal matching in G. The RAND decomposition produces induced subgraphs $G_i = G[V_i]$ of the k partitions of V and an edge induced subgraph G_{k+1} consisting of cross edges. The nature of the subgraphs produced by the RAND decomposition depends upon the number of partitions we make. If the number of partitions increases, the sparsity of the decomposed graph also increases. So, we use the partition size *k* close to the average degree of the graph and obtain the decomposition. We compute a maximal matching M_j in the resulting induced subgraphs $G_j = G[V_j]$ for $1 \leq j \leq k$. Let $M_{IS} := \cup_j M_j$ be a matching in G. We use M_{IS} to remove vertices that are matched. We then process the remaining subgraph of G_{k+1} and obtain a maximal matching M_{k+1}. The matching $M := M_{IS} \cup M_{k+1}$ is a maximal matching in G. A pseudocode for Algorithm MM-Rand is given in Algorithm 5.

3) Algorithm MM-Degk: Algorithm MM-DEGk applies the DEGk decomposition given in Section II to the input graph G before computing a maximal matching of G. We use $k = 2$ in the DEGk decomposition. The DEGk decomposition produces three subgraphs G_H,G_L and G_C, that are induced by vertices of degree more than k, at most k, and the cross edges respectively. We first find a matching M_H in the graph G_H. We can then use M_H to remove vertices in $G_L \cup G_C$ that are matched. In the remaining $G_L \cup G_C$, we then find a matching

Algorithm 4 MM-Bridge

1: G_c, G_b = Dcmp_Bridge(G)
2: $M_c \leftarrow \emptyset$
3: $M_b \leftarrow \emptyset$
4: $M_c \leftarrow$ MM(G_c)
5: $V^{'} \leftarrow$ unmatched vertices in G_b using M_c
6: $M_b \leftarrow MM(G[V'])$
7: **return** $M_b \cup M_c$

Algorithm 5 MM-Rand

1: $\{G_i : 1 \leq i \leq (k+1)\}$ = Dcmp_Rand(G)
2: $G_{IS} = G_i : 1 \leq i \leq k$
3: $M_{IS} \leftarrow \emptyset$
4: $M_{k+1} \leftarrow \emptyset$
5: $M_{IS} \leftarrow$ MM(G_{IS})
6: $V^{'} \leftarrow$ unmatched vertices in G_{k+1} using M_{IS}
7: $M_{k+1} \leftarrow$ MM($G_{k+1}[V']$)
8: **return** $M_{IS} \cup M_{k+1}$

Algorithm 6 MM-Degk

1: G_H, G_L, G_C = Dcmp_Degreek(G, k)
2: $M \leftarrow \emptyset$
3: $M_H \leftarrow \emptyset$
4: $G_{LC} \leftarrow G_L \cup G_C$
5: $M_H \leftarrow$ MM(G_H)
6: $V^{'} \leftarrow$ unmatched vertices in G_{LC} using M_H
7: $M_{LC} \leftarrow$ MM($G_{LC}[V']$)
8: **return** $M_H \cup M_{LC}$

M_{LC}. Now, $M := M_H \cup M_{LC}$ is a maximal matching in G. A pseudocode for Algorithm MM-Degk is given in Algorithm 6.

C. Experimental Results

In this section we study the performance of Algorithms MM-Bridge, MM-Rand, and MM-Degk for maximal matching compared to Algorithms GM and LMAX on the graphs listed in Table II. In our implementation of Algorithm GM, we use the vertex numbers to help in the selection of potential mates. In particular, for every vertex its neighbor with lowest id is the potential mate. In the worst case, if every vertex proposes to its potential mate it can result in a long chain of proposals. In the end, out of each such long chain of proposals, only one edge would get matched. Other vertices in the chain of proposals fail to do so. We call this behavior as *vain tendency*.

a) Experiments on Multicore CPUs: Figure 3 (a) shows the absolute timings of Algorithm GM and the decomposition based algorithms on the multicore CPU.

It can be seen from Figure 3(a) that decomposition based algorithms MM-Bridge and MM-Degk do not perform better than Algorithm GM. This is due to the fact that the number of edges removed in BRIDGE and DEG2 decomposition are small enough that the algorithms still cannot get out of the *vain tendency*. On the other hand, Algorithm MM-Rand tends to perform better than also Algorithm GM. For the RAND decomposition, we use 10 partitions in Algorithm MM-Rand. Since the graph obtained by the RAND decomposition would be sparser than the input graph G, Algorithm MM-Rand is often seen to avoid the *vain tendency* property.

For the graph instance `rgg-n-2-24-s0` it is observed that with ten partitions, RAND decomposition creates induced subgraphs which together contain 10% of the edges of G. Algorithm MM-Rand, applied to the induced subgraphs is seen to match about 70% of vertices in the induced subgraphs within 17 iterations and the remaining matches are found in another 400 iterations approximately. Algorithm GM requires on the order of 14,000 iterations to find a maximal matching on this instance. This huge difference in the behavior of the algorithm on random subgraphs of the entire graph results in a huge speedup on this instance. A similar behavior is observed on the instance `rg-n-2-23-so`. On the other hand, Algorithm MM-Rand tends to perform poorly on instances `kron-g500-logn20` and `kron-g500-logn21` due to the large average degree of these graphs. The average degree of these graphs is close to 85 and the partition size we used fails to create a sparse enough graph. However, if we increase the size of the partitions to 100 for these instances then a speedup is observed. The average speedup obtained by Algorithm MM-Rand is 3.5 [1].

b) Experiments on GPUs: Figure 3 (b) shows the absolute timings of Algorithm LMAX and the decomposition based algorithms on the GPU. As Algorithms GM and LMAX follow a similar model in finding potential mates and matches, we notice a similar trend in the performance of Algorithms MM-Rand on the CPU and the GPU. Just as on the CPUs, for graphs `rgg-n-2-24-s0` and `rgg-n-2-23-s0`, the high speedup acheived is on account of Algorithm MM-Rand being able to match a large portion of vertices in the induced subgraphs. On the other hand, on instance `lp1`, Algorithm LMAX matches the 90% of vertices in first iteration whereas Algorithm MM-Rand matches only 25% of vertices. Thus, there is a slowdown on the instance `lp1`. The average speedup obtained by the Algorithm MM-Rand on GPUs is 2.53[1].

D. Discussion

The above suggest that on both multicore CPUs and GPUs, using a random partitioning is a good strategy to obtain a maximal matching in a sparse graph. On both the CPU and the GPU, we did experiment varying the number of partitions when using the RAND decomposition. It is observed that as the number of partitions increases there is slowdown in the performance of Algorithm MM-Rand. As the number of partitions increases the induced subgraphs tend to be sparser resulting in less number of matches within the subgraphs. We used 10 partitions on the CPU and 4 on the GPU.

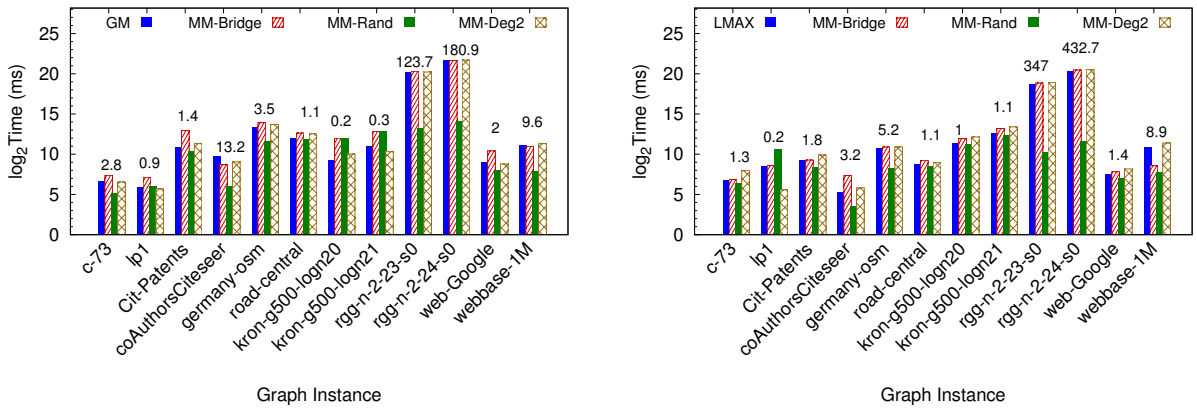
IV. Application to Coloring

A vertex coloring of a graph is an assignment of colors to vertices such that no two adjacent vertices are assigned the same color. The twin objectives of the graph coloring is to minimize both the number of colors used and the time required to obtain the coloring. Vertex coloring has applications to a variety of problems including task scheduling (see also [10], [26]).

A. Existing Algorithms

One of the most prominent algorithms for obtaining an $O(\Delta)$ vertex coloring in the parallel/distributed setting is

[1]The average speedup is calculated by excluding the instances `rgg-n-2-23-s0` and `rgg-n- 2-24-s0`.



(a) Runtime on the CPU

(b) Runtime on the GPU

Fig. 3: Figure (a) shows the absolute timings of GM and decomposition based algorithms for maximal matching for the graphs listed in Table II with 80 threads on the CPU. Figure (b) shows the absolute timings of algorithms LMAX and decomposition based algorithms for maximal matching for the graphs listed in Table II on the GPU. The speedup obtained by Algorithm MM-Rand is shown on the top of the bars with respect to Algorithms GM and LMAX on the CPU and the GPU respectively.

the algorithm of Luby [23]. The algorithm of Luby uses randomization and can be made to run as a PRAM algorithm that requires $O(\log n)$ time and $O(m+n)$ work. Since then there has been a lot of work in the distributed setting and also in the parallel setting. Important works include the algorithm of Barenboim and Elkin [3] for bounded arboricity graphs and the algorithm of Barenboim et al. for general graphs [4].

With respect to the practical setting, Jones and Plassman [18] proposed an algorithm based on independent sets. Hasenplaugh et al. [14] studied the algorithm of Jones and Plassman [18] with vertex ordering such as largest degree first and smallest degree first. These algorithms are proposed on multicore CPUs. Gebremedhin and Manne [12] and Catalyurek et al. [7] study a greedy/speculative optimistic coloring algorithm where each thread tries to find the coloring of a vertex independently. Vertices with a color conflict are uncolored and the process repeats until all vertices are colored. Rokos et al. [26] proposed a variant of speculative coloring algorithm from [7] on multicore and manycore architecture that uses a per-vertex FORBIDDEN array that stores colors forbidden for each vertex. The size of the FORBIDDEN array is set at the maximum degree of the graph. In a recent work, Deveci et al. [10] made an improvement over the speculative algorithm from [7] by using a fixed size FORBIDDEN array. Deveci et al. [10] proposed two variants: a vertex–based algorithm and an edge–based algorithm for manycore architectures.

The vertex–based algorithm from [10], denoted Algorithm VB, uses a fixed size FORBIDDEN array to assign the colors to vertices. Every vertex searches for valid color with in a fixed range which is the size of the FORBIDDEN array in parallel. If a valid color is not found then OFFSET variable is used to increase the color range. Once all vertices are colored then vertices with conflicting colors are detected. These vertices try to obtain a valid color in the next round. This process repeats until all vertices obtain a valid color. The edge–based algorithm from [10] is designed to suit SIMD (Single Instruction and Multiple data) architectures. The edge–based algorithm proceeds by giving the smallest available color to every vertex. Color conflicts are detected by checking whether the two end points of any edge have the same color or not. If they have the same color then the color of the endpoint with the lowest id is reset. This process repeats till all vertices get a valid coloring. Instead of a FORBIDDEN array, a 32 bit integer is used to represent the availability of the colors. In rest of the paper we refer to this algorithm as Algorithm EB.

B. Coloring Algorithms based on Graph Decomposition

From our experimentation, we noticed that Algorithm VB, originally proposed for manycore settings [10], is also suitable for multicore CPUs and outperforms the algorithm from [26]. Hence, on multicore CPUs, we use Algorithm VB as a subroutine to find the coloring and also as a baseline for our experiments. On GPUs, we use Algorithm EB as a subroutine to find the coloring and also as a baseline for our experiments.

1) Algorithm COLOR-Bridge: On the input graph G, we first use the BRIDGE decomposition as described in Section II. Let the 2-edge-connected components of G be $G_1, G_2, \cdots$ and let B be the set of bridges. Let $G_c := \cup_i G_i$. We can notice that the components G_c can be colored with the same set of colors and independently in parallel. Once a coloring C_c of G_c is obtained, the validity of C_c is tested with respect to G and vertices in a color conflict are identified. These vertices are recolored along with the graph consisting of the edges in B. The coloring thus obtained is a valid coloring of G. A pseudocode for COLOR-Bridge is given in Algorithm 7.

2) Algorithm COLOR-Rand: We start by using the RAND decomposition to decompose the input graph G. Recall that RAND decomposition creates a partitioning of vertices into $V_1, V_2, \cdots, V_k$ and a set of cross edges G_{k+1}. We color induced subgraphs $G_i = G[V_i]$ for $1 \leq j \leq k$ in parallel with

Algorithm 7 COLOR-Bridge

1: G_c, G_b = Dcmp_Bridge(G)
2: $C_c(v) \leftarrow 0$ for all v $\in V$
3: $C(v) \leftarrow 0$ for all v $\in V$
4: $C_c \leftarrow$ COLOR(G_c)
5: $C \leftarrow C_c$
6: $V^{'} \leftarrow$ conflicted vertices $G_c \cup G_b$ using C
7: Color $V^{'}$ in $G_c \cup G_b$ and update the color of V' in C
8: **return** C

Algorithm 8 COLOR-Rand

1: $\{G_i : 1 \leq i \leq (k+1)\}$ =Dcmp_Rand(G)
2: $G_{IS} = G_i : 1 \leq i \leq k$
3: $C(v) \leftarrow 0$ for all v $\in V$
4: $C_{IS}(v) \leftarrow 0$ for all v $\in V$
5: $C_{IS} \leftarrow$ COLOR(G_{IS})
6: $C \leftarrow C_{IS}$
7: $V^{'} \leftarrow$ conflicted vertices of G using C
8: Color $V^{'}$ in $G_{IS} \cup G_{k+1}$
9: upate the color of V' to C
10: **return** C

Algorithm 9 COLOR-Degk

1: G_H, G_L, G_C = Dcmp_Degreek(G, k)
2: $C(v) \leftarrow 0$ for all v $\in V$
3: $C_H(v) \leftarrow 0$ for all v $\in V$
4: $C_H \leftarrow$ COLOR(G_H)
5: $C \leftarrow C_H$
6: Color G_L in $G_L \cup G_C$ using $k+1$ sized FORBIDDEN array and update the color of vertices in G_L to C
7: **return** C

an identical palette. Let C_{IS} denote the coloring obtained for the induced subgraphs. We check the validity of C_{IS} with respect to G and vertices whose colors conflict are collected. In a latter step we recolor the vertices in a color conflict with respect to the coloring C_{IS}, along with the graph G_{k+1}. The coloring thus obtained is now a valid coloring of G. A pseudocode for COLOR-Rand is given in Algorithm 8.

3) Algorithm COLOR-Degk: In Algorithm Color-Degk, We decompose the input graph G using the DEGk decomposition where we use $k = 2$. Recall that Degk decomposition gives three subgraphs G_H,G_L,G_C. We first obtain a coloring of G_H and denote the coloring as C_H. As only one end point of the edges in G_C is colored according to the coloring C_H, no vertices are in color conflict with respect to the coloring C_H. When $k = 2$, vertices in G_L have degree at most 2. We use an additional $k+1$ colors $\max(C_H)+1$ to $\max(C_H)+k+1$ to obtain the coloring of G_L. We initialize the color of of vertices in G_L with $\max(C_H)+1$. We use a $k+1$ sized FORBIDDEN array to check the availability of colors. Vertices that have a color conflict use the FORBIDDEN array to obtain a new color that is potentially valid. This process repeats in iteration until all vertices in G_L are validly colored. When $k = 2$, Algorithm COLOR-Degk uses a FORBIDDEN array of size 3 for coloring G_L. Using a small sized FORBIDDEN array improves the performance of Algorithm COLOR-Degk. It is easy to see that the coloring obtained $C_H \cup C_{LC}$ is the coloring of G. A pseudocode for COLOR-Degk is given in Algorithm 9.

C. Experimental Results

We study the performance of the decomposition based algorithms Color-Bridge, Color-Rand, and Color-Degk on graphs listed in Table II.

a) Experiments on Multicore CPUs: In our implementation of Algorithm VB, we keep the size of the FORBIDDEN array to be the average degree of the graph being colored. Figure 4(a) shows the absolute timings of Algorithm VB and the decomposition based algorithms on the graphs listed in Table II.

For Algorithm COLOR-Rand, it can be noticed that for every cross edge uv, with $u \in G_i$ and $v \in G_j$ for distinct i and j, the colors of u and v obtained when coloring subgraphs G_i and G_j may conflict. This results in a necessary recoloring of such end points resulting in additional time taken to finish the coloring. As the number of partitions increase, the number of cross edges increase in expectation, and hence the number of end points whose colors conflict also increases. In our experiment with two partitions, we found that about 45% vertices enter into a color conflict after the coloring of induced subgraph. The recoloring of these vertices increases the overall time resulting in a decline in the speedup. Algorithm COLOR-Bridge also faces a similar problem.

On the other hand, in Algorithm COLOR-Degk once G_H is colored, no such recoloring is required. Further, the vertices in G_L can be colored using a smaller palette of $k+1$ colors. As real-world graphs tend to have a good number of vertices of degree at most 2, Algorithm COLOR-Deg2 can exploit this property and obtain a speedup over Algorithm VB. The average speedup obtained by Algorithm COLOR-Deg2 is 1.27X.

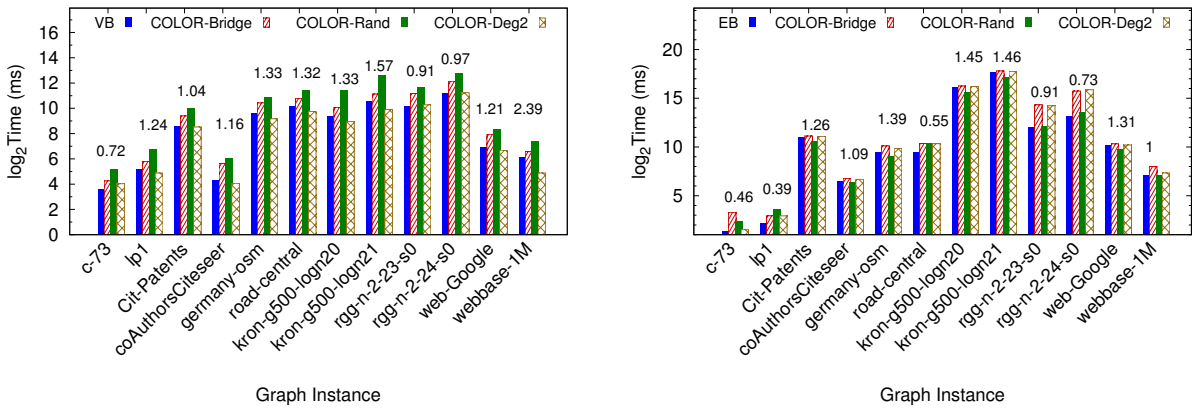
For the above reasons, among the decomposition based algorithms, Algorithm COLOR-Deg2 performs better than Algorithm VB on the CPU. On the other hand, there is a decline in the performance of Algorithm COLOR-Degk for instances `c-73`, `rg-n-2-22-so` and `rg-n-2-23-so`. Decomposing graph `c-73` using RAND decomposition takes time that is relatively large and hence no speedup is observed. Graphs `rg-n-2-22-so` and `rg-n-2-23-so` have very few vertices of degree at most 2 resulting in no improvement for Algorithm Color-Deg2.

b) Experiments on GPUs: Figure 4 (b) shows the absolute timings of Algorithm EB and the decomposition based algorithms. It can be seen that the decomposition based algorithms fail to achieve noticeable speedup in most instances. Moreover, for instances `c-73` and `lp1`, Algorithm EB finishes faster than the time taken for the decomposition of the graphs. An additional reason for the lack of speedup is the impact of cross edges in coloring.

D. Discussion

The above suggest that on multicore CPUs using the DEGk partitioning is a good strategy to obtain a coloring in a sparse graph. With respect to Algorithm Color-Rand, we did experiment varying the number of partitions when using the RAND decomposition. It is observed that as the number of partitions increases there is slowdown in the performance of Algorithm Color-Rand. As the number of partitions increases the increase in the number of cross edges results in more color conflicts.

The number of colors used by the decomposition based algorithms did not differ significantly compared to Algorithms



(a) Runtime on the CPU

(b) Runtime on the GPU

Fig. 4: Figure (a) shows the absolute timings of the VB and decomposition algorithms on Y-axis for the graphs listed in Table II with 80 threads configuration and numbers on the top of the bars shows the speedup obtained by the COLOR-Degk with respect to Algorithm VB. Figure (b) Shows the absolute timings of EB and decomposition algorithms on Y-axis for the graphs listed in Table II on GPU and numbers on the top of the bars shows the speedup obtained by the COLOR-Rand with respect to Algorithm EB.

Algorithm 10 MIS-Bridge

1: $\{G_i : i \geq 1\}$ =Dcmp_Bridge(G)
2: $G_B := G[B]$
3: $H_i \leftarrow G_i \setminus G_b$, for all $i \geq 1$
4: **if** $\Delta(\cup H_i) < \Delta(G_B)$ **then**
5: $I_A = LubyMIS(\cup H_i)$
6: **else**
7: $I_A = LubyMIS(G_B)$
8: Remove vertices which have neighbours in I_A from the remaining graph to get R
9: $I_B = LubyMIS(R)$
10: **return** $I_A \cup I_B$

Algorithm 11 MIS-Rand

1: $\{G_i : 1 \leq i \leq (k+1)\}$ = Dcmp_Rand(G)
2: $H_i \leftarrow G_i \setminus G_{k+1}$, for $1 \leq i \leq k$
3: **if** $\Delta(\cup H_i) < \Delta(G_{k+1})$ **then**
4: $I_A = LubyMIS(\cup H_i)$
5: **else**
6: $I_A = LubyMIS(G_{k+1})$
7: Remove vertices which have neighbours in I_A from the remaining graph to get R
8: $I_B = LubyMIS(R)$
9: **return** $I_A \cup I_B$

Algorithm 12 MIS-Degk

1: G_H, G_L, G_C = Dcmp_Degreek(G, k)
2: $I_C = LexMIS(G_C)$
3: Remove vertices from G_L and G_H based on I_C to get reduced graphs G'_L and G'_H
4: $G_R = G'_L \cup G'_H$
5: $I_R = LubyMIS(G_R)$
6: **return** $I_C \cup I_R$

VB and EB. Algorithm COLOR-Rand increases the average number of colors by 3.9%, 3.4% on CPU, GPU respectively. Algorithm COLOR-Degk uses 3% more colors on CPU and 4.6% on GPU. Algorithm COLOR-Bridge does not use any additional colors on CPU, where as on GPU it uses 4.5% more colors.

V. Application to MIS

The Maximal Independent Set (MIS) problem is one of the most fundamental problems in graph theory, especially in the parallel/distributed setting. Given a graph $G(V, E)$, an independent set of G is defined as a subset I of the vertex set V such that vertices in I are mutual non−neighbors in G. The set I qualifies as a maximal independent set (MIS) if every vertex of the graph either belongs to the set I, or has at least one neighbour in it. Computing an MIS is an important step in solving numerous graph problems in fields as varied as routing and networking, topological control, scheduling, and work distribution and optimization. See for instance [24].

A. Existing Algorithms

The first (poly)logarithmic round algorithm to find the MIS in a parallel/distributed setting, referred to as LubyMIS, is given by Luby [22]. This algorithm uses randomization to break symmetry. It guarantees that in each iteration of the algorithm, at least half the vertices are eliminated, leading to completion in an expected $O(\log n)$ parallel time/distributed rounds. Since the algorithm of Luby, there has been significant work on the problem, including work focusing on special classes of graphs such as graphs with bounded growth [28] and graphs with bounded arboricity [4]. Blelloch et al. [6] proposed a parallelization of the popular greedy algorithm to obtain the MIS of the graph. This algorithm has an expected run-time of $O(\log^2 n)$ in the PRAM sense.

B. Algorithms Based on Decomposition

In this section, we describe algorithms for obtaining an MIS using the decomposition techniques from Section II. In our algorithms, we use Algorithm LubyMIS as a subroutine. We also use Algorithm LubyMIS as a baseline for our experiments.

1) Algorithm MIS-Bridge: To obtain an MIS of a graph G, Algorithm MIS-Bridge uses the BRIDGE decomposition which breaks G into a set of 2-edge connected components $G_1, G_2, \cdots$ and a set B of bridges. Let $H_i := G_i \setminus B$, for $i \geq 1$, be the graph obtained by removing end points of edges in B

from each G_i. Let I_A be an MIS in $\cup_i H_i$. Notice that I_A is an independent set of the graph G but is not necessarily maximal. To extend I_A as an MIS, we eliminate from G vertices that are in I_A or have a neighbor in I_A to get a reduced graph R. Let I_B be an MIS of R. Then $I = I_A \cup I_B$ is an MIS of the graph G.

It is to be noted that computing I_A followed by obtaining the graph R, followed by computing I_B can be done alternatively by first computing I_B as an MIS of G_B, and using I_B to obtain the graph R followed by computing I_A. We observed that computing an MIS on the sparser of the graphs $\cup_i G_i \setminus B$, or G_B is beneficial in practice. Hence, we base the choice on the average degree of the graphs in question. The pseudocode for this algorithm is given in Algorithm 10.

2) Algorithm MIS-Rand: In Algorithm MIS-Rand, we use the RAND decomposition approach prior to computing an MIS of the input graph. The RAND decomposition produces k induced subgraphs $G_i = G[V_i]$ and the graph G_{k+1} consisting of cross edges according to the partitioning of V. Let $H_i := G_i \setminus G_{k+1}$ and $H := \cup H_i$. We start by computing an MIS I_A in H using Algorithm LubyMIS. Using I_A, we can then remove from G vertices that are either in I_A or have a neighbor in I_A to get a reduced graph R. We then compute an MIS I_B of the remaining graph using again the Algorithm LubyMIS. It can be seen that $I := I_A \cup I_B$ is an MIS of G. As in the case of Algorithm MIS-Bridge, I_A and I_B can be computed in any order. We base our decision on which of the two graphs $\cup_i G_i$ or G_{k+1} is sparser. The pseudo-code for this method is given in Algorithm 11.

3) Algorithm MIS-Deg2: In Algorithm MIS-Deg2, we use the DEGk decomposition approach with $k = 2$ prior to computing an MIS of the input graph. The output of the decomposition is seen as consisting of graphs G_L, G_H, and G_C, which are respectively the graphs induced by vertices of degree at most 2, vertices of degree greater than 2, and the cross edges. With its degree bounded by two, the graph G_C necessarily consists only of a set of paths that do not overlap with each other. We start by computing an MIS I_C in G_C. Instead of Algorithm *LubyMIS*, we use the algorithm from [21, Section III] that is designed for bounded degree graphs. The algorithm of [21] requires orientation on the edges of the graph. We use the vertex numbers to induce the required orientation. Once an MIS I_C in G_C is obtained, we use I_C to remove from G vertices that are either in I_C or have a neighbor in I_C. An MIS I_R is computed in the graph that remains using Algorithm *LubyMIS*. It can be seen that $I_C \cup I_R$ is an MIS of G. Algorithm 12 gives a high level description of the procedure.

C. Experimental Results

We study the performance of the decomposition based algorithms MIS-Bridge, MIS-Rand, and MIS-Degk on the graphs listed in Table II.

1) Experiments on the CPU: From Figure 5, it can be seen that Algorithm MIS-Bridge is the slowest in almost all cases. This can be attributed to the fact that the BRIDGE decomposition algorithm takes time comparable to that of finding the MIS using Luby's algorithm [22]. Although Algorithm MIS-Rand does away with the need to spend time finding bridges, it really is not the best way to partition a graph as the MIS computation has little to gain from it. Algorithm MIS-Deg2, on the other hand, achieves a considerable speedup compared to that of Algorithm LubyMIS. This can be attributed to the fact that real world graphs tend to have a large number of vertices of degree two or smaller. The Deg2 decomposition can be seen to remove them quickly. Some of the speedup is also due to using the algorithm of Kothapalli and Pindiproli [21] for computing an MIS of the graph induced by vertices of degree at most two.

On instance `lp1`, we see a speedup as high as 10.5x as the instance has more than 90% of vertices with degree at most 2. On instances such as `rgg-n-2-24`, Algorithm MIS-Deg2 performs poorer than Algorithm LubyMIS. This could be attributed to the fact that random graphs lack the properties required for Algorithm MIS-Deg2 to do better. Algorithm MIS-Degk achieves an average speedup of $3.3x$ compared to Algorithm LubyMIS.

2) Experiments on the GPU: From Figure 5(b), we observe a similar trend with respect to GPUs. The time consumed by the BRIDGE decomposition algorithm renders this decomposition strategy noncompetitive. No significant gains are observed using Algorithm MIS-Rand also. Algorithm MIS-Deg2, on the other hand, shows remarkable improvements for reasons similar to the performance gains on a multicore CPU. We see graphs like `lp1` and `c-73` giving speedups of the order of 50-150 times, while others gain a modest 3-4 times. On an average, we get a speedup of 2.16 times. [2]

3) Discussion: We observe that the Degk decomposition is highly suited for obtaining an MIS, owing to the little time spent in decomposition and also the strong structural property of isolating an induced sub-graph of degree at most 2. This also indicates that the decomposition strategy employed has a big influence on the performance obtained. A good decomposition allows the use of simple, special purpose algorithms on the sub-graph(s) which can easily outperform algorithms for general graphs.

VI. Conclusions

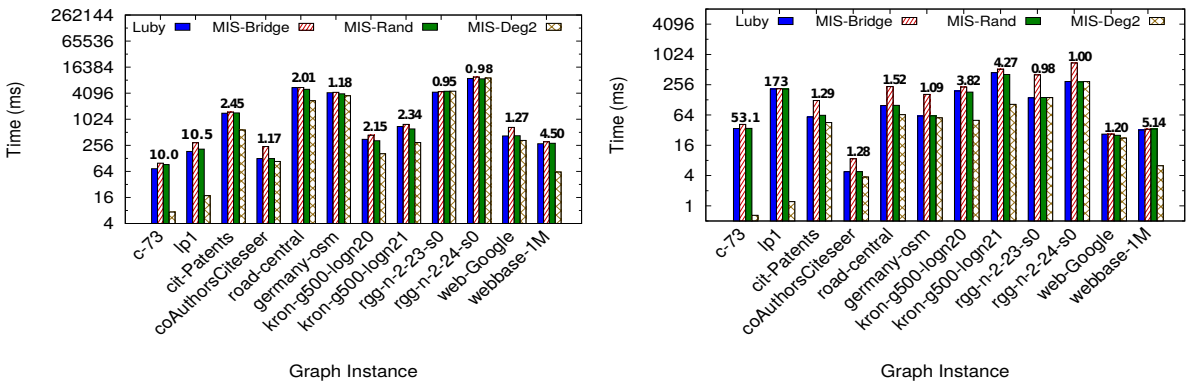
In this paper, we studied three graph symmetry breaking problems and four graph decomposition algorithms on two different parallel architectures. Our study shows that mainly the computation at hand, and, partially, the target architecture can significantly influence the decomposition algorithm to use. In future, we plan to study other parallel graph computations for suitability with respect to various decomposition algorithms.

References

[1] B. O. F. Auer and R. H. Bisseling. A gpu algorithm for greedy graph matching. In *Facing the Multicore-Challenge II*, pages 108–119. Springer, 2012.

[2] D. S. Banerjee, A. Kumar, M. Chaitanya, S. Sharma, and K. K. Work efficient parallel algorithms for large graph exploration on emerging heterogeneous architectures. *JPDC*, 76:81–93, 2015.

[2]The average speedup is calculated by excluding the instances `c-73` and `lp1`.



(a) Runtime on the CPU

(b) Runtime on the GPU

Fig. 5: Figure(a) shows the run times of the decomposition based algorithms for MIS on the E6-2650 CPU running 80 threads. Figure (b) shows the corresponding run times on the K40c GPU. The numbers atop the bars are speedups values for the Degk Decomposition with respect to Algorithm LubyMIS.

[3] L. Barenboim and M. Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. In *Proc. ACM PODC*, pages 25–34, 2008.

[4] L. Barenboim, M. Elkin, S. Pettie, and J. Schneider. The locality of distributed symmetry breaking. *J. ACM*, 63(3):20:1–20:45, June 2016.

[5] M. Birn, V. Osipov, P. Sanders, C. Schulz, and N. Sitchinava. Efficient parallel and external matching. In *Europar*, pages 659–670, 2013.

[6] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proc ACM SPAA*, pages 308–317, 2012.

[7] Ü. V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*, 38(10):576–594, 2012.

[8] M. Chaitanya and K. Kothapalli. Efficient multicore algorithms for identifying biconnected components. *IJNC*, 6:1:87–106, 2016.

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

[10] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam. Parallel graph coloring for manycore architectures. 2016.

[11] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier. Efficient multi-GPU computation of all-pairs shortest paths. 2014.

[12] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency - Prac. and Exp.*, 12(12):1131–1146, 2000.

[13] A. Gharaibeh, L. B. Costa, E. Santos-Neto, and M. Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *in Proc. of IEEE IPDPS*, 2013.

[14] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of ACM SPAA*, pages 166–177. ACM, 2014.

[15] J. Her and F. Pellegrini. Efficient and scalable parallel graph partitioning. *Parallel Computing*, 2010.

[16] D. S. Hochbaum and D. B. Shmoys. A unified approach to approximation algorithms for bottleneck problems. *Journal of the ACM (JACM)*, 33(3):533–550, 1986.

[17] A. Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.

[18] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993.

[19] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. In *In Intl. Par. Proc. Symp.*, pages 314–319, 1996.

[20] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson. Distributed computation of large-scale graph problems. In *ACM SODA*, pages 391–410, 2015.

[21] K. Kothapalli and S. K. Pindiproli. The power of orientation in symmetry-breaking. In *IEEE International Conference on Advanced Information Networking and Applications*, pages 369–376, 2010.

[22] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[23] M. Luby. Removing randomness in parallel computation without a processor penalty. *J. Comp. and Sys. Sci.*, 47(2):250–286, 1993.

[24] M. Naumov, P. Castonguay, and J. Cohen. Parallel graph coloring with applications to the incomplete-lu factorization on the gpu. Technical report, Nvidia Technical Report NVR-2015-001, 2015.

[25] C. Pachorkar, M. Chaitanya, K. Kothapalli, and D. Bera. Efficient parallel ear decomposition of graphs with application to betweenness-centrality. In *Proc. of HiPC*, 2016.

[26] G. Rokos, G. Gorman, and P. H. Kelly. A fast and scalable graph coloring algorithm for multi-core and many-core architectures. In *Proc. EuroPar*, pages 414–425. Springer, 2015.

[27] A. E. Sariyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *Proc.* SIAM Data Mining Conference (SDM). SIAM, 2013.

[28] J. Schneider and R. Wattenhofer. An optimal maximal independent set algorithm for bounded-independence graphs. *Distributed Computing*, 22(5-6):349–361, 2010.

[29] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.

[30] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the GPU. *In* ACM SIGPLAN Not., 50:265–266, 2015.