ACYCLIC PARTITIONING OF LARGE DIRECTED ACYCLIC GRAPHS*

JULIEN HERRMANN†, M. YUSUF ÖZKAYA†, BORA UÇAR‡, KAMER KAYA§, AND ÜMIT V. ÇATALYÜREK†

Abstract. We investigate the problem of partitioning the vertices of a directed acyclic graph into a given number of parts. The objective function is to minimize the number or the total weight of the edges having end points in different parts, which is also known as edge cut. The standard load balancing constraint of having an equitable partition of the vertices among the parts should be met. Furthermore, the partition is required to be *acyclic*, i.e., the inter-part edges between the vertices from different parts should preserve an acyclic dependency structure among the parts. In this work, we adopt the multilevel approach with coarsening, initial partitioning, and refinement phases for acyclic partitioning of directed acyclic graphs. We focus on two-way partitioning (sometimes called bisection), as this scheme can be used in a recursive way for multi-way partitioning. To ensure the acyclicity of the partition at all times, we propose novel and efficient coarsening and refinement heuristics. The quality of the computed acyclic partitions is assessed by computing the edge cut. We also propose effective ways to use the standard undirected graph partitioning methods in our multilevel scheme. We perform a large set of experiments on a dataset consisting of (i) graphs coming from an application and (ii) some others corresponding to matrices from a public collection. We report improvements, on average, around 59% compared to the current state of the art.

Key words. directed graph, acyclic partitioning, multilevel partitioning

AMS subject classifications. 05C70, 05C85, 68R10, 68W05

1. Introduction.

The standard graph partitioning (GP) problem asks for a partition of the vertices of an undirected graph into a number of parts. The objective and the constraint of this well-known problem are to minimize the number of edges having vertices in two different parts and to equitably partition the vertices among the parts. The GP problem is NP-complete [11, ND14]. We investigate a variant of this problem, called *acyclic partitioning*, for directed acyclic graphs. In this variant, we have one more constraint: the partition should be acyclic. In other words, for a suitable numbering of the parts, all edges should be directed from a vertex in a part p to another vertex in a part q where $p \leq q$.

The directed acyclic graph partitioning (DAGP) problem arises in many applications. The stated variant of the DAGP problem arises in exposing parallelism in automatic differentiation [5, Ch.9], and particularly in the computation of the Newton step for solving nonlinear systems [3, 4]. The DAGP problem with some additional constraints is used to reason about the parallel data movement complexity and to dynamically analyze the data locality potential [8, 9]. Other important applications of the DAGP problem include (i) fusing loops for improving temporal locality, and enabling streaming and array contractions in runtime systems [15], such as Bohrium [16]; (ii) analysis of cache efficient execution of streaming applications on uniprocessors [1]; (iii) a number of circuit design applications in which the signal directions impose acyclic partitioning requirement [6, 23].

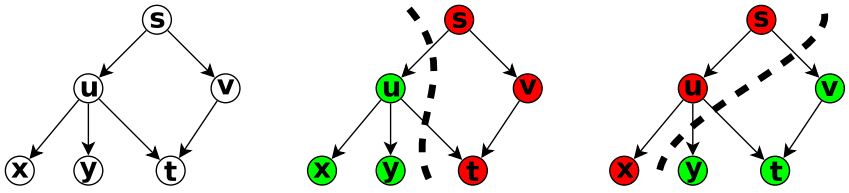
Let us consider a toy example shown in Figure 1.1(a). A partition of the vertices

*A preliminary version appeared in CCGRID'17 [12].

†School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332–0250, julien.herrmann@cc.gatech.edu, myozka@gatech.edu, umit@gatech.edu.

‡CNRS and LIP (UMR5668 CNRS-ENS Lyon-INRIA-UCBL), 46, allée d'Italie, ENS Lyon, 69364, France, bora.ucar@ens-lyon.fr.

§Sabancı University, Istanbul, Turkey, kaya@sabanciuniv.edu.



(a) A toy graph (b) A partition ignoring the directions; it is cyclic. (c) An acyclic partitioning.

Fig. 1.1: **a)** A toy example with six tasks and six dependencies, **b)** a non-acyclic partitioning when edges are oriented, **c)** an acyclic partitioning of the same directed graph.

of this graph is shown in Figure 1.1(b) with a dashed curve. Since there is a cut edge from s to u and another from u to t, the partition is cyclic, and is not acceptable. An acyclic partition is shown in Figure 1.1(c), where all the cut edges are from one part to the other.

We adopt the multilevel partitioning approach [**?**, **?**] with the coarsening, initial partitioning, and refinement phases for acyclic partitioning of DAGs. We propose heuristics for these three phases (Subsections 4.1, 4.2 and 4.3, respectively) which guarantee acyclicity of the partitions at all phases and maintains a DAG at every level. We strived to have fast heuristics at the core. With these characterizations, the coarsening phase requires new algorithmic/theoretical reasoning, while the initial partitioning and refinement heuristics are direct adaptations of the standard methods used in undirected graph partitioning, with some differences worth mentioning. We discuss only the bisection case, as we were able to improve the direct k-way algorithms we proposed before [12] by using the bisection heuristics recursively—we give a brief comparison in Subsection 5.4.

The acyclicity constraint on the partitions precludes the use of the state of the art undirected graph partitioning tools. This has been recognized before, and those tools were put aside [12, 18]. While this is sensible, one can still try to make use of the existing undirected graph partitioning tools [**?**, 13, 20, 22], as they have been very well engineered. Let us assume that we have partitioned a DAG with an undirected graph partitioning tool into two parts by ignoring the directions. It is easy to detect if the partition is cyclic since all the edges need to go from part one to part two. Furthermore, we can easily fix it as follows. Let v be a vertex in the second part; we can move all u vertices for which there is path from v to u into the second part. This procedure breaks any cycle containing v and hence, the partition becomes acyclic. However, the edge cut may increase, and the partitions can be unbalanced. To solve the balance problem and reduce the cut, we can apply a restricted version of the move-based refinement algorithms in the literature. After this step, this final partition meets the acyclicity and balance conditions. Depending on the structure of the input graph, it could also be a good initial partition for reducing the edge cut. Indeed, one of our most effective schemes uses an undirected graph partitioning algorithm to create a (potentially cyclic) partition, fixes the cycles in the partition, and refines the resulting acyclic partition with a novel heuristic to obtain an initial partition. We then integrate this partition within the proposed coarsening approaches to refine it at different granularities. We elaborate on this scheme in Subsection 4.4.

The rest of the paper is organized as follows: Section 2 introduces the notation

and background on directed acyclic graph partitioning and Section 3 briefly surveys the existing literature. We propose multilevel partitioning heuristics for acyclic partitioning of directed acyclic graphs in Section 4. Section 5 presents the experimental results, and Section 6 concludes the paper.

2. Preliminaries and notation. A *directed graph* $G = (V, E)$ contains a set of vertices V and a set of directed edges E of the form $e = (u, v)$, where e is directed from u to v. A *path* is a sequence of edges $(u_1, v_1) \cdot (u_2, v_2), \ldots$ with $v_i = u_{i+1}$. A path $((u_1, v_1) \cdot (u_2, v_2) \cdot (u_3, v_3) \cdots (u_\ell, v_\ell))$ is of length ℓ, where it connects a sequence of $\ell + 1$ vertices $(u_1, v_1 = u_2, \ldots, v_{\ell-1} = u_\ell, v_\ell)$. A path is called *simple* if the connected vertices are distinct. Let $u \rightsquigarrow v$ denote a simple path that starts from u and ends at v. Among all the $u \rightsquigarrow v$ paths, one with the smallest length ℓ is called a *shortest* path. A path $((u_1, v_1) \cdot (u_2, v_2) \cdots (u_\ell, v_\ell))$ forms a (simple) *cycle* if all v_i for $1 \leq i \leq \ell$ are distinct and $u_1 = v_\ell$. A *directed acyclic graph*, DAG in short, is a directed graph with no cycles.

The path $u \rightsquigarrow v$ represents a dependency of v to u. We say that the edge (u, v) is *redundant* if there exists another $u \rightsquigarrow v$ path in the graph. That is when we remove a redundant (u, v) edge, u remains to be connected to v, and hence, the dependency information is preserved. We use $\texttt{Pred}[v] = \{u \mid (u, v) \in E\}$ to represent the (immediate) predecessors of a vertex v, and $\texttt{Succ}[v] = \{u \mid (v, u) \in E\}$ to represent the (immediate) successors of v. We call the neighbors of a vertex v, its immediate predecessors and immediate successors: $\texttt{Neigh}[u] = \texttt{Pred}[v] \cup \texttt{Succ}[v]$. For a vertex u, the set of vertices v such that $u \rightsquigarrow v$ are called the *descendants* of u. Similarly, the set of vertices v such that $v \rightsquigarrow u$ are called the *ancestors* of the vertex u. Every vertex u has a weight denoted by w_u and every edge $(u, v) \in E$ has a cost denoted by $c_{u,v}$.

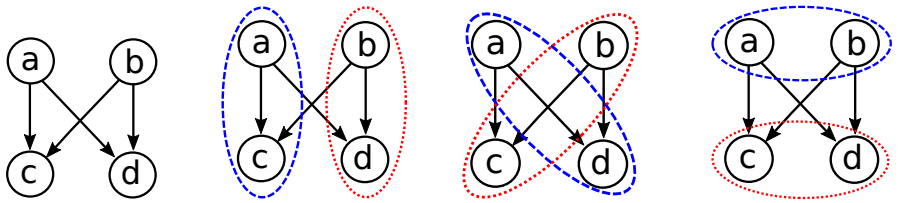
A k-way partitioning of a graph $G = (V, E)$ divides V into k disjoint subsets $\{V_1, \ldots, V_k\}$. The weight of a part V_i for $1 \leq i \leq k$ is equal to $\sum_{u \in V_i} w_u$, denoted as $w(V_i)$, which is the total vertex weight in V_i. Given a partition, an edge is called a *cut edge* if its endpoints are in different parts. The *edge cut* of a partition is defined as the sum of the costs of the cut edges. Usually, a constraint on the part weights accompanies the problem. We are interested in acyclic partitions, which are defined below.

DEFINITION 2.1 (**Acyclic k-way partition**). A partition $\{V_1, \ldots, V_k\}$ of $G = (V, E)$ is called an acyclic k-way partition if two paths $u \rightsquigarrow v$ and $v' \rightsquigarrow u'$ do not co-exist for $u, u' \in V_i$, $v, v' \in V_j$, and $1 \leq i \neq j \leq k$.

There is a related definition in the literature [9], which is called convex partition. A partition is convex if for any pair of vertices u and v in the same part, all vertices in any path from $u \rightsquigarrow v$ are also in the same part. Hence, if a partition is acyclic it is also convex. On the other hand, convexity does not imply acyclicity. Figure 2.1 shows that the definitions of an acyclic partition and a convex partition are not equivalent. For the toy graph in Figure 2.1(a), there are three possible balanced partitions shown in Figure 2.1(b), Figure 2.1(c), and Figure 2.1(d). They are all convex, but only that in Figure 2.1(d) is acyclic.

Deciding on the existence of a k-way acyclic partition respecting an upper bound on part weights and another upper bound on the cost of cut edges is NP-complete [11]. The formal problem treated in this paper is defined as follows.

DEFINITION 2.2 (DAG partitioning problem). *Given a DAG $G = (V, E)$ an imbalance parameter ε, find an acyclic k-way partition $P = \{V_1, \ldots, V_k\}$ of V such that*



(a) A toy graph (b) Cyclic and convex (c) Cyclic and convex (d) Acyclic and convex

Fig. 2.1: A toy graph (left), two cyclic and convex partitions (middle two), and an acyclic and convex partition (right).

the balance constraints

$$w(V_i) \leq (1+\varepsilon)\frac{\sum_{v \in V} w_v}{k} \tag{2.1}$$

are satisfied and the edge cut is minimized.

In the related partitioning tools, a common value of ε is 0.03.

3. Related work. Fauzia et al. [9] propose a heuristic for the acyclic partitioning problem to optimize data locality when analyzing DAGs. To create partitions, the heuristic categorizes a vertex as ready to be assigned to a partition when all of the vertices it depends on have already been assigned. Vertices are assigned to the current partition set until the maximum number of vertices that would be active during the computation of the partition set reaches a specified cache size. This implies that partition sizes can be larger than the size of the cache. This differs from our problem as we limit the size of each partition to the cache size.

Kernighan [14] proposes an algorithm to find a minimum edge-cut partition of the vertices of a graph into subsets of size greater than a lower bound and inferior to an upper bound. The partition needs to use a fixed vertex sequence that cannot be changed. Indeed, Kernighan's algorithm takes a topological order of the vertices of the graph as an input and partitions the vertices such that every vertex in a subset are adjacent in the given topological order. This procedure is optimal for a given, fixed topological order and has a run time proportional to the number of edges in the graph, if the part weights are taken as constant. We used a modified version of this algorithm as a heuristic in the earlier version of our work [12].

Cong et al. [6] describe two approaches for obtaining acyclic partitions of directed Boolean networks, modeling circuits. The first one is a single-level Fiduccia-Mattheyses (FM)-based approach. In this approach, Cong et al. generate an initial acyclic partition by splitting the list of the vertices (in a topological order) from left to right into k parts such that the weight of each part does not violate the bound. The quality of the results is then improved with a k-way variant of the FM heuristic [10] taking the acyclicity constraint into account. Our previous work [12] employs a similar refinement heuristic. The second approach of Cong et al. is a two-level heuristic; the initial graph is first clustered with a special decomposition, and then it is partitioned using the first heuristic.

In a recent paper [18], Moreira et al. focus on an imaging and computer vision application on embedded systems and discuss acyclic partitioning heuristics. They

propose a single level approach in which an initial partitioning is obtained using a topological order and then refined using four local search heuristics while respecting the balance constraint and maintaining the acyclicity of the partition. Three heuristics pick a vertex and move it to an eligible part when the move respects the constraints and improves the cut. They differ in the set of eligible parts for each vertex (from a very restrictive to a more general one allowing arbitrary target parts so long as acyclicity is maintained). The fourth one tentatively realizes the moves that hurt the cut in order to escape from the local minima. This fourth one delivers better results than the others. In a follow-up paper, Moreira et al. [17] discuss a multilevel graph partitioner and an evolutionary algorithm based on this multilevel scheme. Their multilevel scheme starts with a given acyclic partition. Then, the coarsening phase contracts edges that are in the same part until there is no edge to contract. Here matching-based heuristics from undirected graph partitioning tools are used without taking the directions of the edges into account. Therefore, the coarsening phase can create cycles in the graph; however the induced partitions are never cyclic. Then, an initial partition is obtained, which is refined during the uncoarsening phase with moved-based heuristics. In order to guarantee acyclic partitions, the vertices that lie in cycles are not moved. In a systematic evaluation of the proposed methods, Moreira et al. note that there are many local minima and suggest using relaxed constraints in the multilevel setting. The proposed methods have high run time, as the evolutionary method of Moreira et al. is not concerned with this issue. Improvements with respect to the earlier work [18] are reported.

Previously, we had developed a multilevel partitioner [12]. In this paper, we propose methods to use an undirected graph partitioner to guide the multilevel partitioner. We focus on partitioning the graph in two parts since we can handle the general case with a recursive bisection scheme. We also propose new coarsening, initial partitioning, and refinement methods specifically designed for the 2-partitioning problem. Our multilevel scheme maintains acyclic partitions and graphs through all the levels.

Other related work on acyclic partitioning of directed graphs include an exact, branch-and-bound algorithm by Nossack and Pesch [19] which works on the integer programming formulation of the acyclic partitioning problem. This solution is, of course, too costly to be used in practice. Wong et al. [23] present a modification of the decomposition of Cong et al. [6] for clustering, and use this in a two-level scheme.

4. Directed multilevel graph partitioning.

We propose a new multilevel tool for obtaining acyclic partitions of directed acyclic graphs. Multilevel schemes [**?**, **?**] form the de-facto standard for solving graph and hypergraph partitioning problems efficiently, and used by almost all current state-of-the-art partitioning tools [2, **?**, 13, 20, 22]. Similar to other multilevel schemes, our tool has three phases; the coarsening phase, which reduces the number of vertices by clustering them; the initial partitioning phase, which finds a partition of the coarsened graph; and the uncoarsening phase, in which the initial partition is projected to the finer graphs and refined along the way, until a solution for the original graph is obtained.

4.1. Coarsening.

In this phase, we obtain smaller DAGs by coalescing the vertices, level by level. This phase continues until the number of vertices becomes smaller than a specified bound or the reduction on the number of vertices from one level to the next one is lower than a threshold. At each level ℓ, we start with a finer acyclic graph G_ℓ, compute a valid clustering $\mathcal{C}_\ell$ ensuring the acyclicity, and obtain a coarser acyclic graph $G_{\ell+1}$. While our previous work [12] discussed matching based algorithms for

coarsening, we present agglomerative clustering based variants here. The new variants supersede the matching based ones. Unlike the standard undirected graph case, in DAG partitioning, not all vertices can be safely combined. Consider a DAG with three vertices a, b, c and three edges (a, b), (b, c), (a, c). Here, the vertices a and c cannot be combined, since that would create a cycle. We say that a set of vertices is contractible (all its vertices are matchable), if unifying them does not create a cycle. We now present a general theory about finding clusters without forming cycles, after giving some definitions.

DEFINITION 4.1 (**Clustering**). *A clustering of a DAG is a set of (disjoint) subsets of vertices without common vertices.*

DEFINITION 4.2 (**Coarse graph**). *Given a DAG G and a clustering C of G, we let $G_{|C}$ denote the coarse graph created by contracting all sets of vertices of C.*

The coarse graph is a quotient graph of G if the clustering C is extended to a partition with singleton vertex sets.

DEFINITION 4.3 (**Feasible clustering**). *A feasible clustering C of a DAG G is a clustering such that $G_{|C}$ is acyclic.*

THEOREM 4.1. *Let $G = (V, E)$ be a DAG. For $u, v \in V$ and $(u, v) \in E$, the coarse graph $G_{|\{(u,v)\}}$ is acyclic if and only if there is no path from u to v in G avoiding the edge (u, v).*

Proof. Let $G' = (V', E') = G_{|\{(u,v)\}}$ be the coarse graph, and w be the merged, coarser vertex of G' corresponding to $\{u, v\}$.

If there is a path from u to v in G avoiding the edge (u, v), then all the edges of this path are also in G', and the corresponding path in G' goes from w to w, creating a cycle.

Assume that there is a cycle in the coarse graph G'. This cycle has to pass through w; otherwise, it must be in G which is impossible by the definition of G. Thus, there is a cycle from w to w in the coarse graph G'. Let $a \in V'$ be the first vertex visited by this cycle after w and $b \in V'$ be the last one, just before completing the cycle. Let $\mathbf{p}$ be an $a \rightsquigarrow b$ path in G' such that $(w, a) \cdot \mathbf{p} \cdot (b, w)$ is the said $w \rightsquigarrow w$ cycle in G'. Note that a can be equal to b and in this case $\mathbf{p} = \emptyset$. By the definition of the coarse graph G', $a, b \in V$ and all edges in the path $\mathbf{p}$ are in $E \backslash \{(u, v)\}$. Since we have a cycle in G', the following two items must hold: (i) either $(u, a) \in E$ or $(v, a) \in E$, or both; and (ii) either $(b, u) \in E$ or $(b, v) \in E$, or both. We now investigate these nine cases. Here we investigate only four of them, as the "both" cases will be implied by the others.

- $(u, a) \in E$ and $(b, u) \in E$ is impossible because otherwise, $(u, a) \cdot \mathbf{p} \cdot (b, u)$ would be a $u \rightsquigarrow u$ cycle in the original graph G.
- $(v, a) \in E$ and $(b, v) \in E$ is impossible because otherwise, $(v, a) \cdot \mathbf{p} \cdot (b, v)$ would be a $v \rightsquigarrow v$ cycle in the original graph G.
- $(v, a) \in E$ and $(b, u) \in E$ is impossible because otherwise, $(u, v) \cdot (v, a) \cdot \mathbf{p} \cdot (b, u)$ would be a $u \rightsquigarrow u$ cycle in the original graph G.

Thus $(u, a) \in E$ and $(b, v) \in E$. Therefore, $(u, a) \cdot \mathbf{p} \cdot (b, v)$ is a $u \rightsquigarrow v$ path in G avoiding the edge (u, v), which concludes the proof. ∎

Theorem 4.1 can be extended to a set of vertices by noting that this time all paths connecting two vertices of the set should contain only the vertices of the set. The theorem (nor its extension) does not imply an efficient algorithm, as it requires at least one transitive reduction. Furthermore, it does not describe a condition about

two clusters forming a cycle, even if both are individually contractible. In order to address both of these issues, we put a constraint on the vertices that can be in a cluster, based on the following definition.

DEFINITION 4.4 (**Top level value**). *For a DAG $G = (V, E)$, the top level value of a vertex $u \in V$ is the length of the shortest path from a source of G to that vertex. The top level values of all vertices can be computed in a single traversal of the graph with a complexity $O(|V| + |E|)$. We use $\texttt{top}[u]$ to denote the top level of the vertex u.*

The top level value of a vertex is independent of the topological order used for computation. By restricting the set of edges considered in the clustering to the edges $(u, v) \in E$ such that $\texttt{top}[u] + 1 = \texttt{top}[v]$, we ensure that no cycles are formed by contracting a unique cluster (the condition identified in Theorem 4.1 is satisfied). Let C be a clustering of the vertices. Every edge in a cluster of C being contractible is a necessary condition for C to be feasible, but not a sufficient one. More restrictions on the edges of vertices inside the clusters should be found to ensure that C is feasible. We propose three coarsening heuristics based on clustering sets of more than two vertices, whose pair-wise top level differences are always zero or one.

4.1.1. Acyclic clustering with forbidden edges. To have an efficient heuristic, we rely only on static information computable in linear time while searching for a feasible clustering. As stated in the introduction of this section, we rely on the top level difference of one (or less) for all vertices in the same cluster, and an additional condition to ensure that there will be no cycles when a number of clusters are contracted simultaneously. In Theorem 4.2, we give two sufficient conditions for a clustering to be feasible (that is, the graphs at all levels are DAGs) and prove their correctness.

THEOREM 4.2 (Correctness of the proposed clustering). *Let $G = (V, E)$ be a DAG and $C = \{C_1, \ldots, C_k\}$ be a clustering. If C is such that:*

- *for any cluster C_i, for all $u, v \in C_i$, $|\texttt{top}[u] - \texttt{top}[v]| \leq 1$,*
- *for two different clusters C_i and C_j and for all $u \in C_i$ and $v \in C_j$ either $(u, v) \notin E$, or $\texttt{top}[u] \neq \texttt{top}[v] - 1$,*

then, the coarse graph $G_{|C}$ is acyclic.

Proof. Let us assume (for the sake of contradiction) that there is a clustering with the same properties above, but the coarsened graph has a cycle. We pick one such clustering $C = \{C_1, \ldots, C_k\}$ with the minimum number of clusters. Let $t_i = \min\{\texttt{top}[u], u \in C_i\}$ be the smallest top level value of a vertex of C_i. According to the properties of C, for every vertex $u \in C_i$, either $\texttt{top}[u] = t_i$, or $\texttt{top}[u] = t_i + 1$. Let w_i be the coarse vertex in $G_{|C}$ obtained by contracting all vertices in C_i, for $i = 1, \ldots, k$. By the assumption, there is a cycle in $G_{|C}$, and let $\mathbf{c}$ be one with the minimum length. This cycle passes through all the w_i vertices. Otherwise, there would be a smaller cardinality clustering with the properties above and creating a cycle in the coarsened graph, contradicting the minimal cardinality of C. Let us renumber the w_i vertices such that $\mathbf{c}$ is a $w_1 \rightsquigarrow w_1$ cycle which passes through all the w_i vertices in the non-decreasing order of the indices.

After the reordering, for every $i \in \{1, \ldots, k\}$, there is a path in $G_{|C}$ from w_i to w_{i+1} (for the rest of the proof, let $w_0 = w_k$ and $w_{k+1} = w_1$ to simplify the notation). Given the definition of the coarsened graph, for every $i \in \{1, \ldots, k\}$ there exists a vertex $u_i \in C_i$, and a vertex $u_{i+1} \in C_{i+1}$ such that there exists a path $u_i \rightsquigarrow u_{i+1}$ in G. Thus, $\texttt{top}[u_i] + 1 \leq \texttt{top}[u_{i+1}]$. According to the second property, either there is at least one intermediate vertex between u_i and u_{i+1} and then $\texttt{top}[u_i] + 1 < \texttt{top}[u_{i+1}]$;

or $\texttt{top}[u_i] + 1 \neq \texttt{top}[u_{i+1}]$ and then $\texttt{top}[u_i] + 1 < \texttt{top}[u_{i+1}]$. Thus, in any case, $\texttt{top}[u_i] + 1 < \texttt{top}[u_{i+1}]$.

By definition, we know that $t_i \leq \texttt{top}[u_i] + 1$ and $\texttt{top}[u_{i+1}] \leq t_{i+1}$. Thus for every $i \in \{1, \ldots, k\}$, we have $t_i < t_{i+1}$, which leads to the self-contradicting statement $t_1 < t_{k+1} = t_1$ and concludes the proof. ∎

The main heuristic based on Theorem 4.2 is described in Algorithm 1. This heuristic visits all vertices in an order, and adds the visited vertex to a cluster, if certain criteria are met; if not, the vertex stays as a singleton. When visiting a singleton vertex, the clusters of its in-neighbors and out-neighbors are investigated, and the best (according to an objective value) among those meeting the criterion described in Theorem 4.2 is selected.

Algorithm 1 returns the *leader* table of each vertex for the current coarsening step. Vertices with the same leader form a cluster (and will be a single vertex in the coarsened graph). At the beginning of the execution, each vertex is its own leader. Throughout the execution, for each vertex, we maintain the number of *bad neighbors*, that is to say, the number of its neighbors that would contradict the second condition of Theorem 4.2 if this vertex was put in a cluster. When considering a vertex with only one *bad neighbor*, or with several *bad neighbors* but all in the same cluster, this vertex can only be put in this cluster. For instance in Figure 4.1(a), at this point of the coarsening, vertex B can only be put in cluster 1. If vertex B was matched with one of its other neighbors, the second condition of Theorem 4.2 would be violated. If a vertex has more that one *bad neighbor* in different clusters, it has to stay as a singleton in order not to violate the second condition of Theorem 4.2. For instance in Figure 4.1(b), vertex B cannot be put in any cluster without violating the second condition of Theorem 4.2. In Algorithm 1, the function *ValidNeighbors* selects the *compatible neighbors* of vertex v, that is the neighbors in clusters that vertex v can join. This selection is based on the top level difference (to respect the first condition of Theorem 4.2), the number of *bad neighbors* of v and v's neighbors (to respect the second condition of Theorem 4.2), and the size limitation (we do not want a cluster to be bigger than 10% of the total weight of the graph). Then, the best neighbor according to an objective value, such as the edge cost, is selected. After setting the leader of vertex u to the same value as the leader of its best neighbor, some bookkeeping is done for the arrays related to the second condition of Theorem 4.2. More precisely, at Lines 15–20 of Algorithm 1, the neighbors of u are informed about u joining to a new cluster, and potentially becoming a bad neighbor. Similarly, if the best neighbor chosen for u was not in a cluster previously, the number of *bad neighbors* of its neighbors are updated (Lines 22–27).

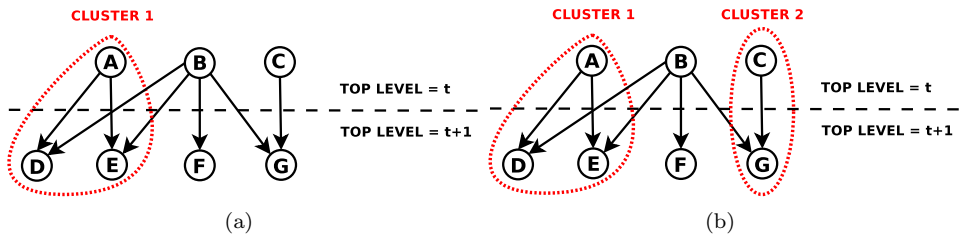


Fig. 4.1: Two examples of acyclic clustering.

We tried two traversal orders of the vertices (random vertex traversal and depth-first topological traversal) and two priority orders for the adjacent edges (random edge ordering and an ordering with non-increasing costs). We also tried a version where the size of cluster is limited by two, meaning that we actually compute a matching of the vertices–this is what we had in the preliminary study [12].

It can be easily seen that Algorithm 1 has a worst case time complexity of $O(|V|+|E|)$. The array `top` is constructed in $O(|V|+|E|)$ time, and the best, valid neighbor of a vertex u is found in $O(|\texttt{Neigh}[u]|)$ time. The neighbors of a vertex are visited at most once to keep the arrays related to the second condition of Theorem 4.2 up to date at Lines 15 and 22.

Algorithm 1: Clustering with forbidden edges

Data: Directed graph $G = (V, E)$, a traversal order of the vertices in V, a priority on edges
Result: The *leader* table for the coarsening

```
1  top ← CompTopLevels(G)
2  for u ∈ V do
3      mark[u] ← false
4      leader[u] ← u
5      weight[u] ← w_u
6      nbbadneighbors[u] ← 0
7      leaderbadneighbors[u] ← −1
8  for u ∈ V following the traversal order in input do
9      if mark[u] then continue
10     N ← ValidNeighbors(u, G, nbbadneighbors, leaderbadneighbors, weight)
11     if N = ∅ then continue
12     BestNeigh ← BestNeighbour(N)
13     leader[u] ← leader[BestNeigh]
14     weight[leader[u]] ← weight[leader[u]] + w_u
15     for v ∈ Neigh[u] do
16         if |top[u] − top[v]| > 1 then continue
17         if nbbadneighbors[v] = 0 then
18             nbbadneighbors[v] ← 1
19             leaderbadneighbors[v] ← leader[u]
20         else if nbbadneighbors[v] = 1 and leaderbadneighbors[v] ≠ leader[u]
               then nbbadneighbors[v] ← 2
21     if mark[BestNeigh] = false then
22         for v ∈ Neigh[BestNeigh] do
23             if |top[BestNeigh] − top[v]| > 1 then continue
24             if nbbadneighbors[v] = 0 then
25                 nbbadneighbors[v] ← 1
26                 leaderbadneighbors[v] ← leader[BestNeigh]
27             else if nbbadneighbors[v] = 1 and
                   leaderbadneighbors[v] ≠ leader[BestNeigh] then
                   nbbadneighbors[v] ← 2
28     mark[u] ← true
29     mark[BestNeigh] ← true
30 return C
```

4.1.2. Acyclic clustering with cycle detection. We now propose a less restrictive clustering algorithm to ensure that the coarse graph is acyclic. As in the

previous section, we rely on the top level difference of one (or less) for all vertices in the same cluster. The algorithm, then, checks dynamically that there will be no cycles when all the clusters are contracted simultaneously, each time we consider the addition of a vertex to a given cluster. This is done via a local cycle detection algorithm which, to avoid traversing the entire graph, uses the fact that in each cluster, the top level difference is at most one.

From the proof of Theorem 4.2, we know that with such a feasible clustering, if adding a vertex to a cluster whose vertices' top level values are t and $t+1$ creates a cycle in the contracted graph, then this cycle goes through only vertices with top level values t or $t+1$. Thus, when considering the addition of a vertex u to a cluster C containing v, we check potential cycle formations by traversing the graph starting from u in a breadth-first manner in the *DetectCycle* function in Algorithm 2. Let t denote the minimum top level in C. When at a vertex w, we normally add a successor y of w into the queue, if $|\texttt{top}(y) - t| \leq 1$; if w is in the same cluster as one if its predecessors x, we also add x to the queue if $|\texttt{top}(x) - t| \leq 1$. This function uses markers to not to visit the same vertex multiple times, and returns *true* if at some point in the traversal a vertex from cluster C is reached, and returns *false* otherwise. In the worst-case, this cycle detection algorithm completes a full graph traversal but in practice, it stops quickly and does not introduce a significant overhead.

Same as for Algorithm 1, we propose different clustering strategies. These algorithms consider all the edges in the graph, one by one, and put them in a cluster if the top level difference is at most one and if no cycles are detected. The clustering algorithms depending on different vertex traversal orders and priority definitions on the adjacent edges are described in Algorithm 2. Same as for Algorithm 1, it returns the *leader* table of each vertex for the current coarsening step. When a vertex is put in a cluster with top level values t and $t+1$, its *markup* (respectively *markdown*) value is set to *true* if its top level value is t (respectively $t+1$). Since the worst case complexity of the cycle detection is $O(|V|+|E|)$, the worst case complexity of Algorithm 2 is $O(|V|(|V|+|E|))$. However, the cycle detection stops quickly in practice and the behavior of Algorithm 2 is closer to $O(|V|+|E|)$ as described in Subsection 5.6.

4.1.3. Hybrid acyclic clustering. The cycle detection based algorithm can suffer from quadratic run time for vertices with large in-degrees or out-degrees. To avoid this, we design a hybrid acyclic clustering which uses the clustering strategy described in Algorithm 2 by default and uses the clustering strategy described in Algorithm 1 in the neighborhood of large degree vertices. We define a limit on the degree of a vertex (typically $\sqrt{|V|}/10$) for calling it *large degree*. When considering an edge (u, v) where $\texttt{top}[u] + 1 = \texttt{top}[v]$, if the degrees of u and v do not exceed the limit, we use the cycle detection algorithm to determine if we can contract the edge. Otherwise, if the outdegree of u or the indegree of v is too large, the edge will be contracted if Algorithm 1 allows so. The complexity of this algorithm is in between those of Algorithm 1 and Algorithm 2 and will likely avoid the quadratic behavior in practice (if not, the degree parameter can be adapted).

4.2. Initial partitioning. After the coarsening phase, we compute an initial acyclic partitioning of the coarsest graph. We present two heuristics. One of them is akin to the greedy graph growing method used in the standard graph/hypergraph partitioning methods. The second one uses an undirected partitioning and then fixes the acyclicity of the partitions. Throughout this section, we use (V_0, V_1) to denote the bisection of the vertices of the coarsest graph G. We aim at returning an acyclic bisection (V_0, V_1) of the coarsest graph such that there is no edge from vertices in V_1

Algorithm 2: Clustering with cycle detection

```
Data: Directed graph G = (V, E), a traversal order of the vertices in V, a priority
      on edges
Result: A feasible clustering C of G
top ← CompTopLevels(G)
for u ∈ V do
    markup[u] ← false
    markdown[u] ← false
    leader[u] ← u
for u ∈ V following the traversal order in input do
    if markup[u] and markdown[u] then continue
    for v ∈ Neigh[u] following given priority on edges do
        if (|top[u] − top[v]| > 1) then continue
        if v ∈ Succ[u] then
            if markup[v] then continue
            if DetectCycle(u, v, G, leader) then continue
            leader[u] ← leader[v]
            markup[u] ← markdown[v] ← true
        if v ∈ Pred[u] then
            if markdown[v] then continue
            if DetectCycle(u, v, G, leader) then continue
            leader[u] ← leader[v]
            markdown[u] ← markdup[v] ← true
return leader
```

to vertices in V_0.

4.2.1. Greedy directed graph growing. One approach to compute a bisection of a directed graph is to design a greedy algorithm that moves vertices from one part to another using local information. Greedy algorithms have shown to be effective for initial partitioning in multilevel schemes in the undirected case. We start with all vertices in V_1 and replace vertices towards V_0 by using heaps. At any time, the vertices that can be moved to V_0 are in the heap. These vertices are those whose all in-neighbors are in V_0. Initially only the sources are in the heap, and when all the in-neighbors of a vertex v are moved to the first part, v is inserted to the heap. We separate this process into two phases. In the first phase, the key-values of the vertices in the heap is equal to the weighted sum of their incoming edges, and the ties are broken in favor of the vertex which is closest to the first vertex moved. The first phase continues until the first part has more than 0.9 of the maximum allowed weight (modulo the maximum weight of a vertex). In the second phase, the actual gain of a vertex is used. This gain is equal to the sum of the weights of the incoming edges minus the sum of the weights of the outgoing edges. In this phase, the ties are broken in favor of the heavier vertices. The second phase stops, as soon as the required balance is obtained. The reason that we separated this heuristic into two is that at the beginning, the gains are of no importance, and the more vertices become movable the more flexibility the heuristic has. Yet, towards the end, parts are fairly balanced, and using actual gains can help keeping the cut small.

Since the order of the parts is important, we also reverse the roles of the parts, and the directions of the edges. That is, we put all vertices in V_0, and move the vertices one by one to V_1, when all out-neighbors of a vertex have been moved to V_1.

The proposed greedy directed graph growing heuristic returns the best of the these two alternatives.

4.2.2. Undirected bisection and fixing acyclicity. In this heuristic, we partition the coarsest graph as if it were undirected and then move the vertices from one part to another in case the partition was not acyclic. Let (P_0, P_1) denote the (not necessarily acyclic) bisection of the coarsest graph treated as if it were undirected.

The proposed approach designates arbitrarily P_0 as V_0 and P_1 as V_1. One way to fix the cycle is to move to V_0 all ancestors of the vertices in V_0, thereby guaranteeing that there is no edge from vertices in V_1 to vertices in V_0, making the bisection (V_0, V_1) acyclic. We do these moves in a reverse topological order, as shown in Algorithm 3. Another way to fix the acyclicity is to move to V_1 all descendants of the vertices in V_1, again guaranteeing an acyclic partition. We do these moves in a topological order, as shown in Algorithm 4. We then fix the possible unbalance with a refinement algorithm.

Note that we can also initially designate P_1 as V_0 and P_0 as V_1, and again use Algorithms 3 and 4 to fix a potential cycle in two different manners. We try all of these alternatives and return the best of the partitions (essentially returning the best of four different alternatives to fix the acyclicity of (P_0, P_1)).

Algorithm 3: fixAcyclicityUp

```
Data: Directed graph G = (V, E) and a bisection part
Result: An acyclic bisection of G
for u ∈ G (in reverse topological order) do
    if part[u] = 0 then
        for v ∈ Pred[u] do
            part[v] ← 0
return part
```

Algorithm 4: fixAcyclicityDown

```
Data: Directed graph G = (V, E) and a bisection part
Result: An acyclic bisection of G
for u ∈ G (in topological order) do
    if part[u] = 1 then
        for v ∈ Succ[u] do
            part[v] ← 1
return part
```

4.3. Refinement. This phase projects the partition obtained for a coarse graph to the next, finer one and refines the partition by vertex moves. As in the standard refinement methods, the proposed heuristic is applied in a number of passes. Within a pass, we repeatedly select the vertex with the maximum move gain among those that can be moved. We tentatively realize this move if the move maintains or improves the balance. Then the most profitable prefix of vertex moves are realized at the end of the pass. As usual, we allow the vertices move only once in a pass; therefore once a vertex is moved, it is not eligible to move again during the same pass. We use heaps with the gain of moves as the key value, where we keep only movable vertices. We

call a vertex *movable*, if moving it to the other part does not create cyclic partition. As previously, we use the notation (V_0, V_1) to designate the acyclic bisection with no edge from vertices in V_1 to vertices in V_0. This means that for a vertex to move from part V_0 to part V_1, one of the two conditions should be met (i) either all its out-neighbors should be in V_1; (ii) or the vertex has no out-neighbors at all. Similarly, for a vertex to move from part V_1 to part V_0, one of the two conditions should be met (i) either all its in-neighbors should be in V_0; (ii) or the vertex has no in-neighbors at all. This is in a sense the adaptation of boundary Fiduccia-Mattheyses [10] (FM) to directed graphs, where the boundary corresponds to the movable vertices. The notion of movability being more restrictive, results in an important simplification with respect to the undirected case. The gain of moving a vertex v from V_0 to V_1 is

$$\sum_{u \in \texttt{Succ}[v]} w(v,u) - \sum_{u \in \texttt{Pred}[v]} w(u,v) \ , \tag{4.1}$$

and the negative of this value when moving it from V_1 to V_0. This means that the gain of vertices are static: once a vertex is inserted in the heap with the key value (4.1), it is never updated. A move could render some vertices unmovable; if they were in the heap, then they should be deleted. Therefore, the heap data structure needs to support insert, delete, and extract max operations only.

We have also implemented a swapping based refinement heuristic akin to the boundary Kernighan-Lin [**?**] (KL), and another one moving vertices only from the maximum loaded part. For graphs with unit weight vertices, we suggest using the boundary FM, and for others we suggest using one pass of boundary KL followed by one pass of the boundary FM from the maximum loaded part.

4.4. Constraint coarsening and initial partitioning. There are a number of highly successful graph partitioning libraries [13, 20, 22]. They are not directly usable for our purposes, as the partitions could be cyclic. Fixing such partitions, by moving vertices to break the cyclic dependencies among the parts, can increase the edge cut dramatically (with respect to the undirected cut). Consider for example, the $n \times n$ grid graph, where the vertices are integer positions for $i = 1, \ldots n$ and $j = 1, \ldots, n$ and a vertex at (i, j) is connected to (i', j') when $|i - i'| = 1$ or $|j - j'| = 1$, but not both. There is an acyclic orientation of this graph, called spiral ordering, as described in Figure 4.2 for $n = 8$. This spiral ordering defines a total order. When the directions of the edges are ignored, we can have a bisection with perfect balance by cutting only $n = 8$ edges with a vertical line. This partition is cyclic; and it can be made acyclic by putting all vertices whose number greater than 32 to the second part. This partition, which puts the vertices 1–32 to the first part and the rest in the second part, is the unique acyclic bisection with perfect balance for the associated directed acyclic graph. The edge cut in the directed version is 35 as seen in the figure (gray edges). In general one has to cut $n^2 - 4n + 3$ edges for $n \geq 8$, as among the blue vertices in the border (excluding the corners) have one edge directed to a red vertex; those in the interior, except the one labeled $n^2/2$ have two such edges; the vertex labeled $n^2/2$ has three such edges.

Since the theoretical analysis shows pessimistic results for a constructed case, let us also investigate some results from a practical stand point. We used MeTiS [13] as the undirected graph partitioner on a dataset of 94 matrices (details are in Section 5) in Figure 4.3. For this preliminary experiments, we partitioned the graphs into two with maximum allowed load imbalance of 3% (i.e., $\varepsilon = 3\%$). In these tests, in only two graphs, the output of MeTiS was acyclic, and the geometric mean of the normalized

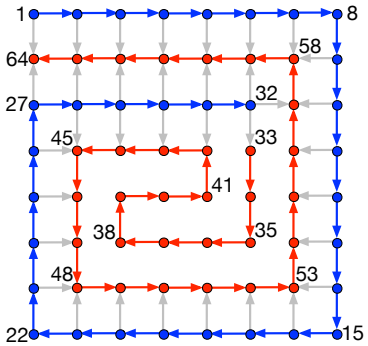
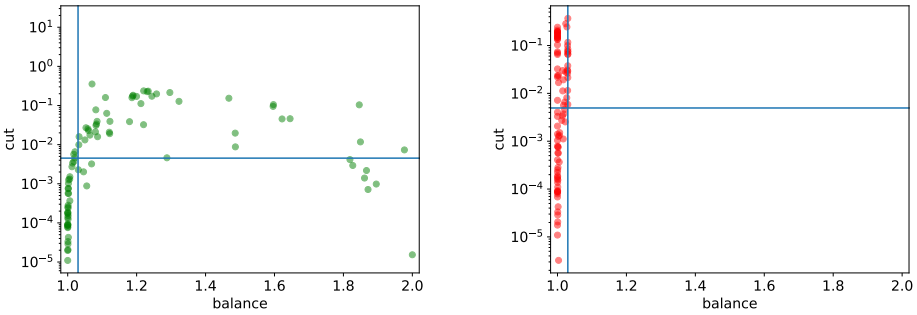


Fig. 4.2: 8×8 grid graph whose vertices are ordered in a spiral way; a few of the vertices are labeled with their number. All edges are oriented from a lower numbered vertex to a higher ordered one. There is a unique bipartition with 32 vertices in each side. The edges defining the total order are shown in red and blue, except the one from 32 to 33; the cut edges are shown in gray; other internal edges are not shown.



(a) Undirected partitioning and fixing cycles

(b) Undirected partitioning, fixing cycles, and balancing

Fig. 4.3: Normalized edge cut (normalized with respect to the number of edges), and the balance obtained after using an undirected graph partitioner and fixing the cycles (left), and after ensuring balance with refinement (right).

edge cut is 0.0012. Figure 4.3(a) shows the normalized edge cut and the load imbalance after fixing the cycles, while Figure 4.3(b) shows the two measurements after meeting the balance criteria. In both figures, the horizontal lines mark the geometric mean of the normalized edge cuts, and the vertical lines mark the 3% imbalance ratio. In Figure 4.3(a), there are 37 instances in which the load balance after fixing the cycles is feasible. The geometric mean of the normalized edge cuts in this subfigure is 0.0045, while in the other subfigure it is 0.0049. Fixing the cycles increases the edge cut with respect to an undirected partitioning, but not catastrophically (only by $0.0045/0.0012 = 3.75$ times in these experiments), and achieving balance after this step increases the cut only little (goes to 0.0049 from 0.0045). That is why we suggest the method of using an undirected graph partitioner, fixing the cycles among the parts,

and a refinement based method for load balancing as a good (initial) partitioner.

In order to be able to refine the initial partition in a multilevel setting we propose a scheme similar to the *iterated multilevel algorithm* used in previous partitioners [2, **?**]. In this scheme, first a partition P is obtained. Then, the coarsening phase is constrained to match or agglomerate vertices that were in the same part in P. After this phase, an initial partitioning is freely available by using the partition P on the coarsest graph. The refinement phase then can work as is before. To be more concrete, we first use an undirected graph partitioner, then fix the cycles as discussed in Subsection 4.2.2, and then refine this acyclic partition for balance with the proposed refinement heuristics in Subsection 4.3. We then use this acyclic partition for constraint coarsening and initial partitioning. We expect this scheme to be successful in graphs with many sources and targets where the sources and targets can be in different parts while the overall partition is acyclic. On the other hand, if a number of sources need to be separated from a number of targets, fixing acyclicity may result in moving all vertices in a single part.

5. Experimental evaluation. We have performed an extensive evaluation of the proposed multilevel directed acyclic graph partitioning method on DAG instances coming from two sources. The first set of instances come from the Polyhedral Benchmark suite (PolyBench) [21], whose parameters are listed in Table 5.1. The second set of instances are obtained from the matrices available in the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [**?**]. From this collection, we pick all matrices satisfying the following properties: listed as binary, square, and has at least 100000 rows and at most 2^{26} nonzeros. There were a total of 95 matrices at the time of experimentation, where two matrices (ids 1514 and 2294) had the same pattern. We discarded the duplicate and used the 94 matrices for experiments. For each such matrix, we took the strict upper triangular part as the associated DAG instance, whenever this part has more nonzeros than the lower triangular part; otherwise we took the lower triangular part. All edges have unit cost, and all vertices have unit weight. The experiments were conducted on computers equipped with dual 2.1 GHz Xeon E5-2683 processors and 512GB memory.

Since the proposed heuristics have randomized behavior, we run them 10 times for each DAG instance, and report the averages of these runs. We use performance profiles [7] to present the edge cut results. The performance profile plot helps compare different methods for the number of cut edges. A plot shows on the y-axis the probability that a specific method gives results which are within θ, shown in the x-axis, of the best edge cut obtained by any of the methods compared in the plot. Hence, the higher and closer a plot to the y-axis, the better the method is.

We set the load imbalance parameter $\varepsilon = 0.03$ in (2.1) for all experiments. The vertices are unit weighted, therefore, balance is rarely an issue for a move based partitioner.

5.1. Coarsening evaluation. We first evaluate the proposed coarsening heuristics. The aim is to find an effective one to set as a default coarsening heuristic.

The performance profiles of Figure 5.1 show the effect of coarsening heuristics on the final edge cut for the whole dataset. The proposed multilevel algorithm using different coarsening schemes are names as `CoTop` (Subsection 4.1.1), `CoCyc` (Subsection 4.1.2), and `CoHyb` (Subsection 4.1.3). In Figure 5.1, we see that `CoCyc` and `CoHyb` behave similarly; this is expected as not all graphs have vertices with large degrees. From this figure, we conclude that in general the coarsening heuristics `CoHyb` and `CoCyc` are more helpful than `CoTop` in reducing the edge cut.

Graph	Parameters	*#vertex*	*#edge*	max. deg.	avg. deg.	#source	#target
`2mm`	P=10, Q=20, R=30, S=40	36,500	62,200	40	1.704	2100	400
`3mm`	P=10, Q=20, R=30, S=40, T=50	111,900	214,600	40	1.918	3900	400
`adi`	T=20, N=30	596,695	1,059,590	109,760	1.776	843	28
`atax`	M=210, N=230	241,730	385,960	230	1.597	48530	230
`covariance`	M=50, N=70	191,600	368,775	70	1.925	4775	1275
`doitgen`	P=10, Q=15, R=20	123,400	237,000	150	1.921	3400	3000
`durbin`	N=250	126,246	250,993	252	1.988	250	249
`fdtd-2d`	T=20, X=30, Y=40	256,479	436,580	60	1.702	3579	1199
`gemm`	P=60, Q=70, R=80	1,026,800	1,684,200	70	1.640	14600	4200
`gemver`	N=120	159,480	259,440	120	1.627	15360	120
`gesummv`	N=250	376,000	500,500	500	1.331	125250	250
`heat-3d`	T=40, N=20	308,480	491,520	20	1.593	1280	512
`jacobi-1d`	T=100, N=400	239,202	398,000	100	1.664	402	398
`jacobi-2d`	T=20, N=30	157,808	282,240	20	1.789	1008	784
`lu`	N=80	344,520	676,240	79	1.963	6400	1
`ludcmp`	N=80	357,320	701,680	80	1.964	6480	1
`mvt`	N=200	200,800	320,000	200	1.594	40800	400
`seidel-2d`	M=20, N=40	261,520	490,960	60	1.877	1600	1
`symm`	M=40, N=60	254,020	440,400	120	1.734	5680	2400
`syr2k`	M=20, N=30	111,000	180,900	60	1.630	2100	900
`syrk`	M=60, N=80	594,480	975,240	81	1.640	8040	3240
`trisolv`	N=400	240,600	320,000	399	1.330	80600	1
`trmm`	M=60, N=80	294,570	571,200	80	1.939	6570	4800

Table 5.1: Instances from the Polyhedral Benchmark suite (PolyBench).

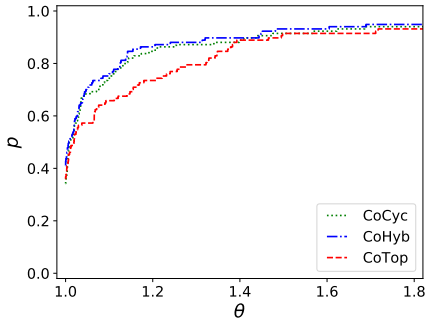


Fig. 5.1: Performance profiles for the edge cut obtained by the proposed multilevel algorithm using three difference coarsening heuristics.

Another important characteristic to assess for coarsening heuristics is its contracting efficiency. It is important that the coarsening phase does not stop too early and that the coarsest graph is small enough to be partitioned efficiently in the initial partitioning phase. Table 5.2 gives the average ratio of the number of vertices of the coarsest graph compared to the original one (under the column header Vertex Ratio) for both datasets separately. The table also gives the ratio of the total weight of the edges to the original one (under the column header Edge Weight Ratio), and the number of coarsening levels needed to achieve these ratios. An effective coarsening heuristic should have small ratios. Again, we see that `CoCyc` and `CoHyb` behave similarly and provide slightly better results that `CoTop` on both datasets. The graphs

Algorithm	Vertex Ratio (%)		Edge Weight Ratio (%)		Level		
	Avg	Max	Avg	Max	Avg	Max	Min
CoTop	1.28	46.72	26.16	87.00	12.45	17.0	2
CoCyc	1.22	47.29	26.22	87.90	12.74	17.6	2
CoHyb	1.22	46.70	26.29	87.00	12.69	17.7	2
CoTop	1.30	8.50	25.67	47.60	7.44	11.8	4
CoCyc	0.04	4.10	24.96	37.00	8.37	12.0	5
CoHyb	0.05	3.60	24.81	39.00	8.46	11.9	5

Table 5.2: Max and average vertex, edge weight ratios and number of coarsening iterations for UFL dataset on the upper half of the table, and for the PolyBench dataset on the lower half.

from the two datasets have different characteristics. All coarsening heuristics perform better on the PolyBench instances than on the UFL instances: they obtain smaller ratios in the number of remaining vertices, and yield smaller edge weights. Furthermore, the maximum vertex and edge ratios are smaller in PolyBench instances, again with all coarsening methods. To the best of our understanding, two related reasons for this observation are (i) the average degree in the UFL instances is larger than that of the PolyBench instances (3.63 vs. 1.72); (ii) the ratio of the total number of source and target vertices to the total number of vertices is again larger in the UFL instances (0.13 vs 0.03). From Figure 5.1 and Table 5.2, we set `CoHyb` as the default coarsening heuristic, as it performs better in terms of final edge cut, behaves better than `CoTop`, and is guaranteed to be more run time efficient than `CoCyc`.

5.2. Constraint coarsening and initial partitioning. We now investigate the effect of using undirected graph partitioners to obtain more effective coarsening and initial partitions as explained in Subsection 4.4. We compare three variants of the proposed multilevel scheme. All of them use the refinement described in Subsection 4.3 in the uncoarsening phase.

- `CoHyb`: this variant uses the hybrid coarsening heuristic described in Subsection 4.1.3 and the greedy directed graph growing heuristic described in Subsection 4.2.1 in the initial partitioning phase. This method does not use constraint coarsening.
- `CoHyb_C`: this variant uses an acyclic partition of the finest graph obtained as outlined in Subsection 4.2.2 to guide the hybrid coarsening heuristic as described in Subsection 4.4, and uses the greedy directed graph growing heuristic in the initial partitioning phase.
- `CoHyb_CIP`: this variant uses the same constraint coarsening heuristic as the previous method, but inherits the fixed acyclic partition of the finest graph as the initial partitioning.

The comparison of these three variants are given in Figure 5.2 for the whole dataset. From Figure 5.2, we see that using the constraint coarsening is always helpful with respect to not using them. This shows with a clear separation of `CoHyb_C` and `CoHyb_CIP` from `CoHyb` after $\theta = 1.1$. Furthermore, applying the constraint initial partitioning (on top of the constraint coarsening) bring tangible improvements.

In the light of the experiments presented here, we suggest the variant `CoHyb_CIP` for general problem instances, as this has clear advantages over others in our dataset.

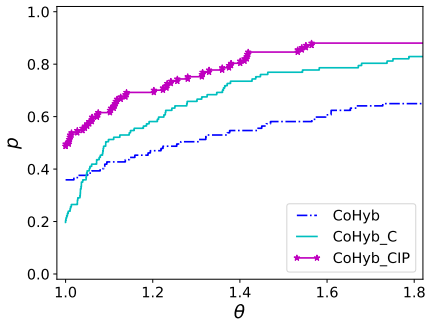


Fig. 5.2: Performance profiles for the edge cut obtained by the proposed multilevel algorithm using the constraint coarsening and partitioning (`CoHyb_CIP`), using the constraint coarsening and the greedy directed graph growing (`CoHyb_C`), and the best identified approach without constraints (`CoHyb`).

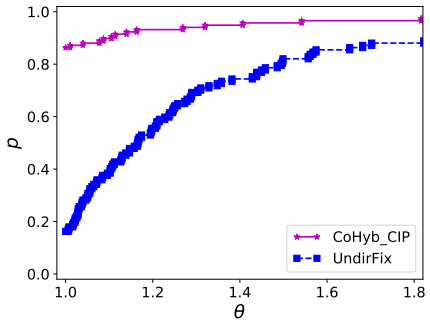


Fig. 5.3: Performance profiles for the edge cut obtained by the proposed multilevel approach using the constraint coarsening and partitioning (`CoHyb_CIP`) and using the same approach without coarsening (`UndirFix`).

5.3. Evaluation `CoHyb_CIP` with respect to a single level algorithm. We compare `CoHyb_CIP` (the variant of the proposed approach with constraint coarsening and initial partitioning) with a single level algorithm that uses an undirected graph partitioning, fixes the acyclicity, and refines the partitions. This last variant is denoted as `UndirFix`, and it is the algorithm described in Subsection 4.2.2. Both variants use the same initial partition, which utilizes MeTiS [13] as undirected partitioner, and the difference between `UndirFix` and `CoHyb_CIP` is the latter's ability to refine that initial partition at multiple levels. Figure 5.3 presents this comparison on the experimental dataset. The plots show that the multilevel scheme `CoHyb_CIP` outperforms the single level scheme `UndirFix` at all appropriate ranges of θ, attesting to the importance of the multilevel scheme.

5.4. Comparison with existing work. Here we compare our approach with the evolutionary graph partitioning approach developed by Moreira et al. [18], and briefly with our previous work [12].

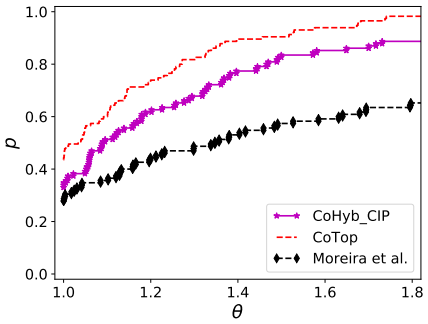


Fig. 5.4: Performance profiles for the edge cut obtained by `CoHyb_CIP`, `CoTop`, and Moreira et al.'s results on the PolyBench dataset with $k \in \{2, 4, 8, 16, 32\}$.

Figure 5.4 shows how `CoHyb_CIP` and `CoTop` compare with the evolutionary approach in terms of the edge cut on the 23 graphs of the PolyBench dataset, for the number of partitions $k \in \{2, 4, 8, 16, 32\}$. We used the average edge cut value out of 10 for `CoTop` and `CoHyb_CIP` and the average value presented in [18] for the evolutionary algorithm. The `CoTop` variant of the proposed multilevel approach provides the best results on this specific dataset (all variants of the proposed approach outperform the evolutionary approach).

Tables A.1 and A.2 show the average and best edge cuts found by `CoHyb_CIP` and `CoTop` variants of our partitioner and the evolutionary approach on the PolyBench dataset. Both variants of the proposed algorithm, `CoHyb_CIP` and `CoTop`, obtain strictly better results in 74 instances out of 115 compared to the evolutionary approach. On average (geometric mean), `CoHyb_CIP` outperforms the evolutionary approach by 45% when the average cuts are compared; when the best cuts are compared, `CoHyb_CIP` obtains 53% lower cuts. Moreover, `CoTop` outperforms the evolutionary approach by 59% when the average cuts are compared; when the best cuts are compared, `CoTop` obtains 71% lower cuts.

Also, note that the proposed approach with all the reported variants took about 30 minutes to complete the whole set of experiments for this dataset, whereas the evolutionary approach is much more compute-intensive, as it has to run the multilevel partitioning algorithm numerous times to create and update the population of partitions for the evolutionary algorithm. The multilevel approach of Moreira et al. [18] is more comparable in terms of characteristics with out multilevel scheme. When we compare `CoHyb_CIP` with the results of the multilevel algorithm by Moreira et al., our approach provides results that are 87% better on average, highlighting the fact that keeping the acyclicity of the directed graph through the multilevel process is useful.

Finally, `CoHyb_CIP` also outperforms the previous version of our multilevel partitioner [12], which was based on a direct k-way partitioning scheme and matching heuristics for the coarsening phase, by 63% on average on the same dataset.

5.5. Single commodity flow-like problem instances. In many of the instances of our dataset, graphs have many source and target vertices. We investigate how our algorithm performs on problems where all source vertices should be in a given part, and all target vertices should be in the other part, while also achieving balance. This is a problem close to the maximum flow problem, where we want to find the

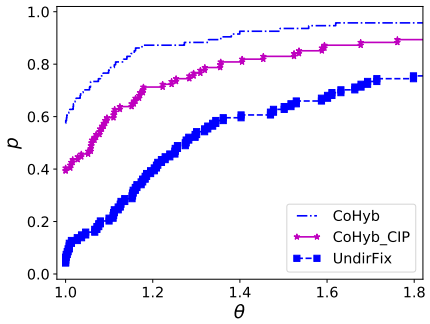


Fig. 5.5: Edge cut for `CoHyb`, `CoHyb_CIP` and `UndirFix` for single source, single target graph dataset, average of 5 runs.

maximum flow (or minimum cut) from the sources to the targets with balance on part weights. Furthermore, addressing this problem also provides a setting for solving partitioning problems with fixed vertices.

For these experiments, we used the UFL dataset. We discarded all isolated vertices, added to each graph a source vertex S (with an edge from S to all source vertices of the original graph with a cost equal to the number of edges) and target vertex T (with an edge from all target vertices of the original graph to T with a cost equal to the number of edges). A feasible partition should avoid cutting these edges, and separate all sources from the targets.

The performance profiles of `CoHyb`, `CoHyb_CIP` and `UndirFix` are given in Figure 5.5 with the edge cut as the evaluation criteria. As seen in this figure, `CoHyb` is the best performing variant, and the `UndirFix` is the worst performing variant. This is interesting as in the general setting, we saw a reverse relation. The variant `CoHyb_CIP` performs in the middle, as it combines the other two.

5.6. Runtime performance. We now assess the runtime performance of the proposed algorithms. Figure 5.6 shows the runtime comparison and distribution for 13 graphs of our dataset among those that took longest coarsening time for the `CoTop` variant. A description of these 13 graphs can be found in Table 5.3. In Figure 5.6, each graph has three bars representing the runtime for the multilevel algorithm using the coarsening heuristics described in Subsection 4.1: `CoTop`, `CoCyc`, and `CoHyb`. We can see that the run time performance of the three coarsening heuristics are similar. This means that, the cycle detection function in `CoCyc` does not introduce a large overhead, as stated in Subsection 4.1.2. Most of the time, `CoCyc` has a bit longer run time than `CoTop`, and `CoHyb` offers a good tradeoff. Note that in Figure 5.6 the computation time of the initial partitioning is negligible compared to that of the coarsening and uncoarsening phases, which means that the graphs have been efficiently contracted during the coarsening phase.

The performance profile in Figure 5.7 shows the comparison of the five variants of the proposed multilevel scheme and the single level scheme on the whole dataset. Each algorithm has been run 10 times on each graphs. As expected, `CoTop` offers the best performance and `CoHyb` offers a good tradeoff between `CoTop` and `CoHyb`. An interesting remark is that these three algorithms have a better run time than the single level algorithm `UndirFix`. Finally, the variants of the multilevel algorithm

Graph	#vertex	#edge	Max In	Max Out	Avg Deg	#source	#target
333SP	3,712,815	11,108,633	9	27	2.992	188,112	316,151
AS365	3,799,275	11,368,076	10	13	2.992	306,791	519,431
M6	3,501,776	10,501,936	10	10	2.999	280,784	472,230
cit-Patents	3,774,768	16,518,209	779	770	4.376	515,980	1,685,419
delaunay-n22	4,194,304	12,582,869	15	17	3	555,807	337,743
hugebubbles-00010	19,458,087	29,179,764	3	3	1.5	3,355,886	3,054,827
hugetrace-00020	16,002,413	23,998,813	3	3	1.5	2,514,461	2,407,017
hugetric-00010	6,592,765	9,885,854	3	3	1.5	1,085,866	1,006,163
italy-osm	6,686,493	7,013,978	5	8	1.049	155,509	458,561
rgg-n-2-22-s0	4,194,304	30,359,198	24	25	7.238	3,550	3,576
road-usa	23,947,347	28,854,312	8	8	1.205	6,392,288	8,010,032
wb-edu	9,845,725	29,494,732	17,489	3841	2.996	1,489,057	2,794,680

Table 5.3: 13 instances from the UFL dataset.

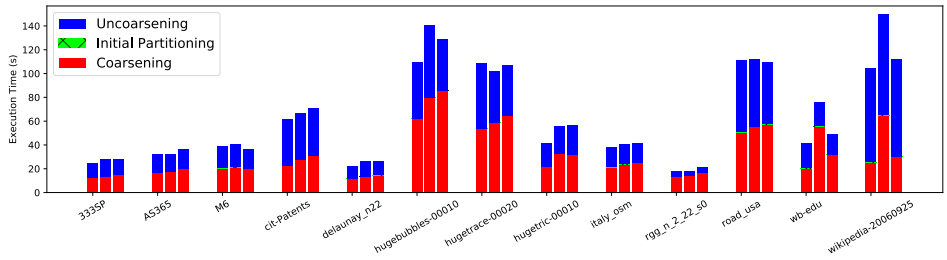


Fig. 5.6: Runtimes for `CoTop`, `CoCyc`, and `CoHyb` variants of the proposed multilevel scheme. For each graph, the first, second, and the third bar represents the detailed runtime of `CoTop`, `CoCyc`, and `CoHyb`, respectively.

using constraint coarsening heuristics provide satisfying run time performance with respect to the others.

6. Conclusion. We proposed a multilevel approach for acyclic partitioning of directed acyclic graphs. This problem is close to the standard graph partitioning in that the aim is to partition the vertices into a number of parts while minimizing the edge cut and meeting a balance criterion on the part weights. Different from the standard graph partitioning problem, the directions of the edges is important and the resulting partitions should have acyclic dependencies.

We proposed coarsening, initial partitioning, and refinement heuristics for the target problem. The proposed heuristics maintain the acyclicity of the input graphs, take advantage of the directions of the edges, and maintains the acyclicity all through the multilevel hierarchy. We also proposed effective ways to use the standard undirected graph partitioning methods in the multilevel scheme in the form of constraints for coarsening and initial partitioning. We performed a large set of experiments on a dataset with graphs having different characteristics and evaluated different combinations of the proposed heuristics. Our experiments suggested (i) the use of constraint coarsening and initial partitioning, where the main coarsening heuristic is the one that hybridizes the fast cycle detection and the one that avoids the possibility (`CoHyb_CIP`) for the general case; (ii) pure multilevel scheme, without constraint coarsening, using the hybrid coarsening heuristic (`CoHyb`) for the cases where a number of sources need

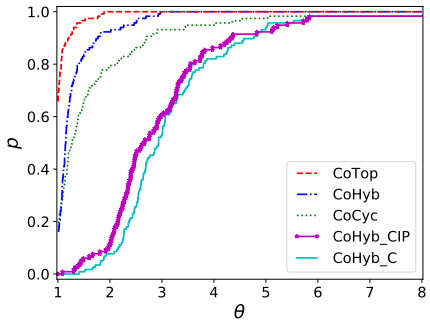


Fig. 5.7: Runtime performance for `CoCyc`, `CoHyb`, `CoTop`, `CoHyb_C`, `CoHyb_CIP` and `UndirFix` on the whole dataset, average of 10 runs.

to be separated from a number of targets; (iii) pure multilevel scheme, without constraint coarsening, using the fast coarsening algorithm (`CoTop`) for the cases where the degrees of the vertices are small. All three approaches were shown to be more effective and efficient than the current state of the art.

Future work includes, applying the proposed multilevel scheme in real life applications that are based on task-graphs. This requires a scheduling step to be applied after the proposed partitioning scheme, which needs further investigations. A recent work uses a multilevel algorithm for recombination and mutation [17]. Plugging in our multilevel scheme to that framework could possibly lead to improvements.

Acknowledgement. We thank John Gilbert for his comments on an earlier version of this work presented at CSC'16. John suggested us to look at the spiral ordering of the grid graph.

REFERENCES

[1] K. Agrawal, J. T. Fineman, J. Krage, C. E. Leiserson, and S. Toledo, *Cache-conscious scheduling of streaming applications*, in Proc. Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, New York, NY, USA, 2012, ACM, pp. 236–245.

[2] Ü. V. Çatalyürek and C. Aykanat, *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Dept. Comp. Engineering, Ankara, 06533 Turkey. PaToH is available at http://cc.gatech.edu/~umit/software.html, 1999.

[3] T. F. Coleman and W. Xu, *Parallelism in structured Newton computations*, in Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 2007, pp. 295–302.

[4] T. F. Coleman and W. Xu, *Fast (structured) Newton computations*, SIAM Journal on Scientific Computing, 31 (2009), pp. 1175–1191.

[5] T. F. Coleman and W. Xu, *Automatic Differentiation in MATLAB using ADMAT with Applications*, SIAM, 2016.

[6] J. Cong, Z. Li, and R. Bagrodia, *Acyclic multi-way partitioning of Boolean networks*, in Proceedings of the 31st Annual Design Automation Conference, DAC'94, New York, NY, USA, 1994, ACM, pp. 670–675.

[7] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, Mathematical programming, 91 (2002), pp. 201–213.

[8] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, *On characterizing the data access complexity of programs*, SIGPLAN Not., 50 (2015), pp. 567–580.

[9] N. Fauzia, V. Elango, M. Ravishankar, J. Ramanujam, F. Rastello, A. Rountev, L.-N.

Pouchet, and P. Sadayappan, *Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential*, ACM Trans. Archit. Code Optim., 10 (2013), pp. 53:1–53:29.
[10] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Design Automation, 1982. 19th Conference on, IEEE, 1982, pp. 175–181.
[11] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.
[12] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek, *Acyclic partitioning of large directed acyclic graphs*, in Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID, Madrid, Spain, May 2017, pp. 371–380.
[13] G. Karypis and V. Kumar, *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Cent., Minneapolis, 1998.
[14] B. W. Kernighan, *Optimal sequential partitions of graphs*, J. ACM, 18 (1971), pp. 34–40.
[15] M. R. B. Kristensen, S. A. F. Lund, T. Blum, and J. Avery, *Fusion of parallel array operations*, in Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, New York, NY, USA, 2016, ACM, pp. 71–85.
[16] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, *Bohrium: A virtual machine approach to portable parallelism*, in Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IPDPSW '14, Washington, DC, USA, 2014, IEEE Computer Society, pp. 312–321.
[17] O. Moreira, M. Popp, and C. Schulz, *Evolutionary acyclic graph partitioning*, CoRR, abs/1709.08563 (2017).
[18] O. Moreira, M. Popp, and C. Schulz, *Graph partitioning with acyclicity constraints*, CoRR, abs/1704.00705 (2017), http://arxiv.org/abs/1704.00705.
[19] J. Nossack and E. Pesch, *A branch-and-bound algorithm for the acyclic partitioning problem*, Computers & Operations Research, 41 (2014), pp. 174–184.
[20] F. Pellegrini, *SCOTCH 5.1 User's Guide*, Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2008.
[21] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, URL: http://web.cse.ohio-state.edu/ pouchet/software/polybench/, (2012).
[22] P. Sanders and C. Schulz, *Engineering multilevel graph partitioning algorithms*, in Algorithms – ESA 2011: 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings, C. Demetrescu and M. M. Halldórsson, eds., Berlin, Heidelberg, 2011, Springer Berlin Heidelberg, pp. 469–480.
[23] E. S. H. Wong, E. F. Y. Young, and W. K. Mak, *Clustering based acyclic multi-way partitioning*, in Proceedings of the 13th ACM Great Lakes Symposium on VLSI, GLSVLSI '03, New York, NY, USA, 2003, ACM, pp. 203–206.

Appendix A. Detailed results on the PolyBench instances. We give in Tables A.1 and A.2 the detailed edge cut results of the proposed `CoTop`, `CoHyb_CIP` and of Moreira et al.'s evolutionary algorithm [18].

Graph	k	Moreira et al. [18]		CoHyb_CIP		CoTop	
		Average	Best	Average	Best	Average	Best
2mm	2	200	200	200	200	200	200
	4	947	930	3642	3010	2113	1900
	8	7181	6604	8948	7144	4815	3495
	16	13330	13092	12760	11807	11815	10590
	32	14583	14321	15305	15023	16058	15463
3mm	2	1000	1000	1989	1989	1000	1000
	4	38722	37899	9175	4437	9904	8655
	8	58129	49559	15613	10891	26365	18132
	16	64384	60127	36765	32497	37510	32153
	32	62279	58190	48187	45412	45978	43853
adi	2	134945	134675	141852	138832	141795	138462
	4	284666	283892	215174	213720	215195	214573
	8	290823	290672	256740	255941	256818	256037
	16	326963	326923	283323	282321	282229	280700
	32	370876	370413	306591	304429	306017	303736
atax	2	47826	47424	40108	40108	39876	39876
	4	82397	76245	45733	45733	48645	48645
	8	113410	111051	50624	49967	51512	50336
	16	127687	125146	58570	56009	59499	57583
	32	132092	130854	68942	65562	67744	62741
covariance	2	66520	66445	42747	42555	41530	8589
	4	84626	84213	66948	52958	57043	26008
	8	103710	102425	82689	75031	92575	57092
	16	125816	123276	88714	86217	111433	86453
	32	142214	137905	97605	93414	132870	122577
doitgen	2	43807	42208	35955	35697	5947	5947
	4	72115	71072	65190	63567	37781	36807
	8	76977	75114	74101	67028	52772	48982
	16	84203	77436	80880	76337	66369	64359
	32	94135	92739	81772	76302	75147	72595
durbin	2	12997	12997	12997	12997	12997	12997
	4	21641	21641	21566	21566	21566	21566
	8	27571	27571	27520	27520	27520	27520
	16	32865	32865	32912	32912	32912	32912
	32	39726	39725	39827	39826	39827	39826
fdtd-2d	2	5494	5494	5689	5656	6064	5907
	4	15100	15099	15362	14957	16995	16796
	8	33087	32355	28656	27256	35417	34000
	16	35714	35239	39578	38899	44082	42959
	32	43961	42507	49651	48060	53661	51943
gemm	2	383084	382433	23734	23046	40624	5303
	4	507250	500526	56274	40231	54406	46677
	8	578951	575004	230630	200190	144359	96059
	16	615342	613373	285368	244160	278602	253790
	32	626472	623271	357692	330762	336214	304041
gemver	2	29349	29270	23871	21032	20913	20913
	4	49361	49229	39376	38210	40283	40174
	8	68163	67094	54276	51947	55326	52798
	16	78115	75596	57632	54698	60167	57063
	32	85331	84865	71160	70105	73612	71218
gesummv	2	1666	500	1189	1189	500	500
	4	98542	94493	4927	4927	11361	11188
	8	101533	98982	65979	64195	9698	9344
	16	112064	104866	73137	69742	36447	27633
	32	117752	114812	84357	80473	44601	40868
heat-3d	2	8695	8684	9392	9137	9426	9322
	4	14592	14592	16431	15951	16760	16451
	8	20608	20608	26004	25293	26099	25473
	16	31615	31500	42233	41713	42111	41560
	32	51963	50758	67117	65641	70621	69908

Table A.1: Comparing the edge cuts obtained by CoHyb_CIP and CoTop with those obtained by the evolutionary algorithm of Moreira et al. on the Polyhedral Benchmark Suite (first set of results).

Graph	k	Moreira et al. [18]		CoHyb_CIP		CoTop	
		Average	Best	Average	Best	Average	Best
jacobi-1d	2	596	596	762	697	678	660
	4	1493	1492	1957	1764	1791	1743
	8	3136	3136	3550	3111	3447	3226
	16	6340	6338	6527	5792	4968	4834
	32	8923	8750	9585	8826	6872	6557
jacobi-2d	2	2994	2991	3855	3732	3425	3291
	4	5701	5700	8496	8240	7253	7063
	8	9417	9416	15535	14940	13157	13021
	16	16274	16231	24653	23830	21589	20799
	32	22181	21758	31002	30368	29384	28377
lu	2	5210	5162	5433	5429	6088	6041
	4	13528	13510	38795	25403	23504	16312
	8	33307	33211	61152	49162	57977	52436
	16	74543	74006	112974	98149	108552	97596
	32	130674	129954	169037	159692	161763	149943
ludcmp	2	5380	5337	9134	9131	5407	5339
	4	14744	14744	29942	23678	24661	22003
	8	37228	37069	60989	54468	62607	53560
	16	78646	78467	112550	104933	107778	97619
	32	134758	134288	169868	156114	165703	150499
mvt	2	24528	23091	43074	37884	21040	19792
	4	74386	73035	53838	46610	37075	35043
	8	86525	82221	67303	54091	46303	41871
	16	99144	97941	75180	73479	55237	52003
	32	105066	104917	73786	71213	62563	58177
seidel-2d	2	4991	4969	4888	4868	4725	4568
	4	12197	12169	12176	11767	11908	11471
	8	21419	21400	22402	21714	22136	21532
	16	38222	38110	40642	39849	40059	39586
	32	52246	51531	62336	60397	58816	57590
symm	2	94357	94214	44469	37752	43706	43463
	4	127497	126207	74940	71427	86077	75470
	8	152984	151168	97393	91629	116995	111479
	16	167822	167512	111192	106452	134147	128078
	32	174938	174843	118951	115138	145787	143436
syr2k	2	11098	3894	15827	11439	16299	16259
	4	49662	48021	26167	22863	22363	19433
	8	57584	57408	31886	27811	29271	27730
	16	59780	59594	36365	30664	32530	31431
	32	60502	60085	38420	36442	37127	35287
syrk	2	219263	218019	21308	5853	14027	10031
	4	289509	289088	103973	46064	60411	52200
	8	329466	327712	145919	120364	118824	106794
	16	354223	351824	220278	189386	188733	175327
	32	362016	359544	257402	228839	227868	215512
trisolv	2	6788	3549	336	336	336	336
	4	43927	43549	828	828	828	828
	8	66148	65662	2156	2156	2156	2156
	16	71838	71447	6057	5871	6057	5871
	32	79125	79071	13489	13031	13600	13253
trmm	2	138937	138725	3440	3440	23860	3440
	4	192752	191492	42499	30019	70459	29741
	8	225192	223529	122686	115208	131586	112423
	16	240788	238159	158768	150211	156921	146885
	32	246407	245173	170057	164285	172070	163735
Geomean		1.00	0.96	0.69	0.63	0.63	0.56

Table A.2: Comparing the edge cuts obtained by CoHyb_CIP and CoTop with those obtained by the evolutionary algorithm of Moreira et al. on the Polyhedral Benchmark Suite (second set of results). The last line (Geomean) is for the whole PolyBench dataset (i.e., computed by combining this table with the previous one), where the performance of the algorithms are normalized with respect to the average values shown under the column Moreira et al.