Multilevel Algorithms for Acyclic Partitioning of Directed Acyclic Graphs
Julien Herrmann, Yusuf M. Özkaya, Bora Uçar, Kamer Kaya, Ümit V. Çatalyürek

To cite this version:

HAL Id: hal-02306566
https://inria.hal.science/hal-02306566
Submitted on 11 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers. L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 significant improvements 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 [13, 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 [6, Ch.9], and particularly in the computation of the Newton step for solving nonlinear systems [4, 5]. 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 [10, 11]. Other important applications of the DAGP problem include (i) fusing loops for improving temporal locality, and enabling streaming and array contractions in runtime systems [19], such as Bohrium [20]; (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 [7, 29].

Let us consider a toy example shown in Figure 1.1(a). A partition of the vertices...
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 \([2, 14]\) 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 maintain 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 \([15]\) 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 \([15, 21]\). While this is sensible, one can still try to make use of the existing undirected graph partitioning tools \([14, 16, 25, 27]\), 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 a 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

---

This manuscript is for review purposes only.
acyclic partitioning of DAGs

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 $(u_1, v_1) \cdot (u_2, v_2) \cdots (u_k, v_k)$ such that $(u_i, v_{i+1}) \in E$ for $1 \leq i < k$. A path $(u_1, v_1) \cdot (u_2, v_2) \cdots (u_k, v_k)$ is of length $k$. There is an immediate predecessor of a vertex $v$ if it is an endpoint of an incoming edge. Similarly, a successor of $v$ is a destination endpoint of an outgoing edge.

The formal problem treated in this paper is defined as follows.

The set of vertices is divided into $k$ disjoint subsets $\{V_1, \ldots, V_k\}$. 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 $V_i$ denoted by $w(V_i)$ is equal to $\sum_{u \in V_i} w_u$, which is the total vertex weight in $V_i$. 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 $u \leadsto v$ and $v' \leadsto u'$ 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 [11], which is called a convex partition. A partition is convex if for all vertex pairs $u, v$ in the same part, the vertices in any $u \leadsto v$ path 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.

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 $\varepsilon$, find an acyclic $k$-way partition $P = \{V_1, \ldots, V_k\}$ of $V$ such that
the balance constraints

\[ w(V_i) \leq (1 + \varepsilon) \frac{\sum_{v \in V} w_v}{k} \]

are satisfied for \(1 \leq i \leq k\), and the edge cut is minimized.

3. Related work. Fauzia et al. [11] 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 part reaches a specified limit, which is the cache size in their application. This implies that part sizes are not limited by the sum of the total vertex weights but is a complex function that depends on an external schedule (order) of the vertices. This differs from our problem as we limit the size of each part by the total sum of the weights of the vertices on that part.

Kernighan [17] 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 all vertices in a subset constitute a continuous block 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 [15].

Cong et al. [7] 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 [12] taking the acyclicity constraint into account. Our previous work [15] 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 [21], 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 acyclic partitioning is obtained using a topological order. To refine the partitioning, they proposed four local search heuristics which respect the balance constraint and maintain the acyclicity of the partition. Three heuristics pick a vertex and move it to an eligible part if and only if the move improves the cut. These three heuristics differ in choosing the set of eligible parts for each vertex; some are very restrictive, and some allow arbitrary target parts as long as acyclicity is maintained. The fourth heuristic tentatively realizes the moves that increase the cut in order to escape from a possible local minima. It has been reported that this heuristic delivers better results than the others. In a follow-up paper, Moreira et al. [22] 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 move-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 [21] are reported.

Previously, we had developed a multilevel partitioner [15]. 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 [23] 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. [29] present a modification of the decomposition of Cong et al. [7] 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 [2, 14] 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 [3, 14, 16, 25, 27]. 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 coarsest 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 \( \ell \), we start with a finer acyclic graph \( G_\ell \), compute a valid clustering \( C_\ell \) ensuring the acyclicity, and obtain a coarser acyclic
graph $G_{t+1}$. While our previous work [15] discussed matching based algorithms for
coarsening, we present agglomerative clustering based variants here. The new vari-
ants 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. Note that we do not make any assumptions on whether the subsets are
c connected or not.

**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 vertices of the coarse graph are the clusters in $C$. If $(u, v) \in E$ for two
vertices $u$ and $v$ that are located in different clusters of $C$ then $G_{|C}$ has an (directed)
edge from the vertex corresponding to $u$’s cluster, to the vertex corresponding to $v$’s
cluster.

**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
e 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
$p$ be an $a \sim b$ path in $G'$ such that $(w, a) \cdot p \cdot (b, w)$ is the said $w \sim w$ cycle in $G'$.
Note that $a$ can be equal to $b$ and in this case $p = \emptyset$. By the definition of the coarse
graph $G'$, $a, b \in V$ and all edges in the path $p$ are in $E \setminus \{(u, v)\}$. 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.

Hence, overall we have nine ($3 \times 3$) cases. Here, we investigate only four of them, as the
“both” conditions in (i) and (ii) can be eliminated easily by the following statements.

- (i) either $(u, a) \in E$ and $(b, u) \in E$ is impossible because otherwise, $(u, a) \cdot p \cdot (b, u)$
  would be a $u \sim u$ cycle in the original graph $G$.
- (ii) either $(v, a) \in E$ and $(b, v) \in E$ is impossible because otherwise, $(v, a) \cdot p \cdot (b, v)$
  would be a $v \sim v$ cycle in the original graph $G$.
- $(v, a) \in E$ and $(b, u) \in E$ is impossible because otherwise, $(u, v) \cdot (v, a) \cdot p \cdot (b, u)$
  would be a $u \sim u$ cycle in the original graph $G$.

Thus $(u, a) \in E$ and $(b, v) \in E$. Therefore, $(u, a) \cdot p \cdot (b, v)$ is a $u \sim v$ path in $G$
avoiding the edge $(u, v)$, which concludes the proof. \(\Box\)
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 form 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 longest 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 $\text{top}[u]$ 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 $\text{top}[u] + 1 = \text{top}[v]$, 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 $u, v \in C_i$, $|\text{top}[u] - \text{top}[v]| \leq 1$,
- 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 $\text{top}[u] \neq \text{top}[v] - 1$,

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 $t_i = \min\{\text{top}[u], u \in C_i\}$ 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 $\text{top}[u] = t_i$, or $\text{top}[u] = t_i + 1$. 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 $c$ 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, without loss of generality, the $w_i$ vertices such that $c$ is a $w_1 \sim w_1$ cycle which passes through all the $w_i$ vertices in the non-decreasing order of the indices. This also renumbers the clusters accordingly.

After renumbering the $w_i$ vertices, for every $i \in \{1, \ldots, k\}$, there is a path in $G_{[C]}$
from \( w_i \) to \( w_{i+1} \). 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 \( u_i \sim u_{i+1} \) in \( G \). Thus, \( \top[u_i] + 1 \leq \top[u_{i+1}] \). According to the second property, either there is at least one intermediate vertex between \( u_i \) and \( u_{i+1} \) and then \( \top[u_i] + 1 < \top[u_{i+1}] \); or \( \top[u_i] + 1 \neq \top[u_{i+1}] \) and then \( \top[u_i] + 1 < \top[u_{i+1}] \). Thus, in any case, \( \top[u_i] + 1 < \top[u_{i+1}] \) which can be rewritten as \( \top[u_i] < \top[u_{i+1}] - 1 \).

By definition, we know that \( t_i \leq \top[u_i] \) and \( \top[u_{i+1}] - 1 \leq t_{i+1} \). 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. \( \square \)

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 \texttt{leader} array of each vertex for the current coarsening step. Vertices with the same leader form a cluster (and will form a single vertex in the coarsened graph). For each vertex \( u \in V \), \texttt{leader}[u] is the id of the representative vertex for the cluster that will contain \( u \) after Algorithm 1. The \texttt{leader} table will be used to build the coarse graph. Any arbitrary vertex in a given cluster can be used as the leader of this cluster without impacting the rest of the algorithm. At the beginning, each vertex belongs to a singleton cluster, and \texttt{leader}[u] = u. To keep the track of trivial clusters (singleton vertices), we use an auxiliary \texttt{mark} array. The value \texttt{mark}[u] is \texttt{false} if \( u \) still belongs to a singleton cluster. Otherwise, the value is set to \texttt{true}.

For each singleton vertex \( u \), we maintain an auxiliary array \texttt{nbbadneighbors} to keep the number of non-trivial bad neighbor clusters. That is to say, the number of clusters containing a neighbor of \( u \) that would violate the second condition of Theorem 4.2 in case \( u \) was put in another cluster. Hence, if \( u \) has only one bad neighbor cluster, it can only be put into this cluster. For instance in Figure 4.1(a), at this point of the coarsening, vertex \( B \) can only be put in Cluster 1. Otherwise, if vertex \( B \) was matched with one of its other neighbors, the second condition of the theorem would be violated. Thus, if a vertex has more than one bad neighbor in different clusters, it has to stay as a singleton. For instance in Figure 4.1(b), vertex \( B \) has two bad neighbor clusters and cannot be put in any cluster without violating the second condition of Theorem 4.2. To check if there exists another bad neighbor cluster previously formed, we maintain an array \texttt{leaderbadneighbor} that keeps the representative/leader of the first bad neighbor cluster for each vertex. Initially, this value is set to minus one.

In Algorithm 1, the function \texttt{ValidNeighbors} selects the compatible neighbors of vertex \( u \), that is the neighbors in clusters that vertex \( u \) 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 \( u \), and \( u \)'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, a best neighbor, \texttt{BestNeigh}, 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 \texttt{BestNeigh}, some bookkeeping is done for the arrays related to the second condition of Theorem 4.2. More precisely, at
Lines 16–22 of Algorithm 1, the neighbors of \( u \) are informed about \( u \) joining a new cluster, potentially becoming a bad neighbor. While doing that, the algorithm skips the vertices \( v \) such that \(|\text{top}[u] - \text{top}[v]| > 1\), since \( u \) cannot form a bad neighbor cluster for such \( v \). Similarly, if the best neighbor chosen for \( u \) was not in a cluster previously, i.e., was a singleton vertex, the number of bad neighbors of its neighbors are updated (Lines 24–30).

![Fig. 4.1: Two examples of acyclic clustering.](image)

In our framework, we also implemented the version in the preliminary study [15] where the size of cluster is limited to two, meaning that it computes a matching of the vertices.

It can be easily seen that Algorithm 1 has a worst case time complexity of \( O(|V| + |E|) \). The array \( \text{top} \) is constructed in \( O(|V| + |E|) \) time, and the best, valid neighbor of a vertex \( u \) is found in \( O(|\text{Neigh}[u]|) \) 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 16 and 24.

### 4.1.2. Acyclic clustering with cycle detection

We now propose a less restrictive clustering algorithm to ensure that the acyclicity of the coarse graph is maintained. As in the previous section, we rely on the top level difference of one (or less) for all vertices in the same cluster, i.e., for any cluster \( C_i \), for all \( u, v \in C_i \), \(|\text{top}[u] - \text{top}[v]| \leq 1\). Knowing this invariant, when a new vertex is added to a cluster, a cycle-detection algorithm checks that no cycles are formed when all the clusters are contracted simultaneously. This algorithm does not traverse the entire graph by also using the fact that the top level difference within a cluster is at most one.

From the proof of Theorem 4.2, we know that with 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 the 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 used in Algorithm 2.

Let \( t \) denote the minimum top level in \( C \). When at a vertex \( u \), we normally add a successor \( y \) of \( u \) into the queue, if \(|\text{top}(y) - t| \leq 1\); if \( w \) is in the same cluster as one of its predecessors \( x \), we also add \( x \) to the queue if \(|\text{top}(x) - t| \leq 1\). This function uses markers to not to visit the same vertex multiple times, returns \( \text{true} \) if at some point in the traversal a vertex from cluster \( C \) is reached, and returns \( \text{false} \), 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.

Here, we propose different clustering strategies. These algorithms consider all the vertices in the graph, one by one, and put them in a cluster if their top level...
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 array for the coarsening

1. \( \text{top} \leftarrow \text{CompTopLevels}(G) \)
2. for \( u \in V \) do
3.   \( \text{mark}[u] \leftarrow \text{false} \) // all vertices are marked as singleton
4.   \( \text{leader}[u] \leftarrow u \)
5.   \( \text{weight}[u] \leftarrow w_u \) // keeps the total weight for each cluster
6.   \( \text{leaderbadneighbors}[u] \leftarrow -1 \)
7. for \( u \in V \) following the traversal order in input do
8.   if \( \text{mark}[u] \) then continue
9.   \( N \leftarrow \text{ValidNeighbors}(u, G, \text{leaderbadneighbors}, \text{weight}) \)
10. if \( N = \emptyset \) then continue
11. \( \text{BestNeigh} \leftarrow \text{BestNeighbour}(N) \)
12. \( \ell \leftarrow \text{leader}[\text{BestNeigh}] \)
13. \( \text{leader}[u] \leftarrow \ell \) // assign \( u \) to \( \text{BestNeigh}'s \) cluster
14. \( \text{weight}[\ell] \leftarrow \text{weight}[\ell] + w_u \) // Let the neighbors of \( u \) know that it is not a singleton anymore
15. for \( v \in \text{Neigh}[u] \) do
16.   if \( \lvert \text{top}[u] - \text{top}[v] \rvert > 1 \) then continue // \( u \) cannot form a bad cluster
17.   if \( \text{leaderbadneighbors}[v] = 0 \) then
18.     \( \text{leaderbadneighbors}[v] \leftarrow \ell \)
19.     \( \text{leaderbadneighbors}[v] \leftarrow 2 \) // mark \( v \) as unmatchable
20.   else if \( \text{nnbadneighbors}[v] = 1 \) and \( \text{leaderbadneighbors}[v] \neq \ell \) then
21.     \( \text{nnbadneighbors}[v] \leftarrow 1 \) // The first bad neighbor cluster for \( v \)
22.   \( \text{nnbadneighbors}[v] \leftarrow 2 \) // mark \( v \) as unmatchable
23. if \( \text{BestNeigh} \) was forming a singleton cluster before \( u \)'s assignment then
24.   if \( \text{mark}[\text{BestNeigh}] = \text{false} \) then
25.     for \( v \in \text{Neigh}[\text{BestNeigh}] \) do
26.       if \( \lvert \text{top}[\text{BestNeigh}] - \text{top}[v] \rvert > 1 \) then continue
27.       if \( \text{leaderbadneighbors}[v] = 0 \) then
28.         \( \text{leaderbadneighbors}[v] \leftarrow \ell \)
29.       else if \( \text{nnbadneighbors}[v] = 1 \) and \( \text{leaderbadneighbors}[v] \neq \ell \) then
30.         \( \text{nnbadneighbors}[v] \leftarrow 2 \) // mark \( v \) as unmatchable
31.     \( \text{mark}[\text{BestNeigh}] \leftarrow \text{true} \) // BestNeigh is not a singleton anymore
32.   \( \text{mark}[u] \leftarrow \text{true} \) // \( u \) is not a singleton anymore
33. return leader

Differences are at most one and if no cycles are introduced. The clustering algorithms
depending on different vertex traversal orders and priority definitions on the adjacent
edges are described in Algorithm 2. As Algorithm 1, this algorithm also returns the
leader array 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.

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 $G$

1. $top ← CompTopLevels(G)$
2. for $u ∈ V$ do
   3. $markup[u] ← false$ // if $u$’s cluster has a $v$ with $top[v] = top[u] + 1$
   4. $markdown[u] ← false$ // if $u$’s cluster has a $v$ with $top[v] = top[u] − 1$
   5. $leader[u] ← u$ // the leader vertex id for $u$’s cluster
3. for $u ∈ V$ following the traversal order in input do
4.   if $markup[u]$ or $markdown[u]$ then continue
5.   for $v ∈ Neigh[u]$ following given priority on edges do
6.       if $(|top[u] − top[v]| > 1)$ then continue // we use $|top[u] − top[v]| = 1$
7.       /* If this is a $(u, v)$ edge */
8.       if $v ∈ Succ[u]$ then
9.         /* if $markup[v]$ then continue */
10.        if DetectCycle($u, v, G, leader$) then continue
11.       leader[$u$] $←$ leader[$v$]
12.       $markdown[u] ←$ $markup[v]$ $←$ true
13.       /* If this is a $(v, u)$ edge */
14.       if $v ∈ Pred[u]$ then
15.         /* if $markdown[v]$ then continue */
16.        if DetectCycle($u, v, G, leader$) then continue
17.       leader[$u$] $←$ leader[$v$]
18.       $markdown[u] ←$ $markup[v]$ $←$ true

19. return leader

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 switches to the clustering strategy in Algorithm 1 for 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 $top[u] + 1 = top[v]$, 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 out-degree of $u$ or the in-degree 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 $G$. The acyclic bisection $(V_0, V_1)$ is such that there is no edge from the vertices in $V_1$ to those in $V_0$. 

This manuscript is for review purposes only.
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 into the heap. We separate this process into two phases. In the first phase, the key-values of the vertices in the heap are set to the weighted sum of their incoming edges, and the ties are broken in favor of the vertex closer 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 phases 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 all ancestors of the vertices in $V_0$ to $V_0$, 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 all descendants of the vertices in $V_1$ to $V_1$, 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 ways. We try all four of these choices, and return the best partition (essentially returning the best of the four choices to fix the acyclicity of $(P_0, P_1)$).

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.

This manuscript is for review purposes only.
Algorithm 3: fixAcyclicityUp

Data: Directed graph $G = (V, E)$ and a bisection $part$
Result: An acyclic bisection of $G$
1 for $u \in G$ (in reverse topological order) do
2    if $\text{part}[u] = 0$ then
3        for $v \in \text{Pred}[u]$ do
4            $\text{part}[v] \leftarrow 0$
5 return $\text{part}$

Algorithm 4: fixAcyclicityDown

Data: Directed graph $G = (V, E)$ and a bisection $part$
Result: An acyclic bisection of $G$
1 for $u \in G$ (in topological order) do
2    if $\text{part}[u] = 1$ then
3        for $v \in \text{Succ}[u]$ do
4            $\text{part}[v] \leftarrow 1$
5 return $\text{part}$

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 a cyclic partition. As previously done, 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 [12] (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

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

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 [18] (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 that moves vertices only from the maximum loaded part.

4.4. Constraint coarsening and initial partitioning. There are a number of highly successful, undirected graph partitioning libraries [16, 25, 27]. They are
Fig. 4.2: $8 \times 8$ 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 numbered 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.

not directly usable for our purposes, as the partitions can 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 numbered greater than 32 to the second part. This partition, which puts the vertices 1–32 to the first part and the rest to 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 $n^2/2 - 4n + 3$ edges for $n \geq 8$: the blue vertices in the border (excluding the corners) have one edge directed to a red vertex; the interior blue vertices have two such edges; finally, the blue vertex labeled $n^2/2$ has three such edges.

Let us also investigate the quality of the partitions from a more practical standpoint. We used MeTiS [16] as the undirected graph partitioner on a dataset of 94 matrices (their details are in Section 5). The results are given in Figure 4.3. For this preliminary experiment, we partitioned the graphs into two with the maximum allowed load imbalance $\varepsilon = 3\%$. In the experiment, for only two graphs, the output of MeTiS is acyclic, and the geometric mean of the normalized 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. A normalized edge cut value is computed by normalizing the edge cut with respect to the number of edges.

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 a little (goes to 0.0049 from 0.0045). That is why we suggest using an undirected
graph partitioner, fixing the cycles among the parts, and performing a refinement
based method for load balancing as a good (initial) partitioner.

In order to refine the initial partition in a multilevel setting, we propose a scheme
similar to the iterated multilevel algorithm used in the existing partitioners [3, 28]. In
this scheme, first a partition $P$ is obtained. Then, the coarsening phase is employed
to match (or to agglomerate) the vertices that were in the same part in $P$. After
the coarsening, an initial partitioning is freely available by using the partition $P$
on the coarsest graph. The refinement phase then can work as before. Moreira
et al. [22] use this approach for the directed graph partitioning problem. To be
more concrete, we first use an undirected graph partitioner, then fix the cycles as
discussed in Section 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 lie in any
of the parts while the overall partition is acyclic. On the other hand, if a graph is such
that its balanced acyclic partitions need to put sources in one part and the targets in
another part, then fixing acyclicity may result in moving many vertices. This in turn
will harm the edge cut found by the undirected graph partitioner.

5. Experimental evaluation. The partitioning tool presented (dagP) is imple-
mented in C/C++ programming languages. The experiments are conducted on a
computer equipped with dual 2.1 GHz, Xeon E5-2683 processors and 512GB memory.
The source code and more information is available at http://tda.gatech.edu/software/
dagP/.

We have performed an extensive evaluation of the proposed multilevel directed
Table 5.1: Instances from the Polyhedral Benchmark suite (PolyBench).

<table>
<thead>
<tr>
<th>Graph</th>
<th>Parameters</th>
<th>#vertex</th>
<th>#edge</th>
<th>max. deg.</th>
<th>avg. deg.</th>
<th>#source</th>
<th>#target</th>
</tr>
</thead>
<tbody>
<tr>
<td>2mm</td>
<td>P=10, Q=20, R=30, S=40</td>
<td>36,500</td>
<td>62,200</td>
<td>40</td>
<td>1.704</td>
<td>2100</td>
<td>400</td>
</tr>
<tr>
<td>3mm</td>
<td>P=10, Q=20, R=30, S=40, T=50</td>
<td>111,900</td>
<td>214,600</td>
<td>40</td>
<td>1.918</td>
<td>7000</td>
<td>400</td>
</tr>
<tr>
<td>adi</td>
<td>T=20, N=30</td>
<td>596,695</td>
<td>1,059,590</td>
<td>109,760</td>
<td>1.776</td>
<td>843</td>
<td>28</td>
</tr>
<tr>
<td>atax</td>
<td>M=210, N=230</td>
<td>241,730</td>
<td>385,960</td>
<td>230</td>
<td>1.597</td>
<td>48530</td>
<td>230</td>
</tr>
<tr>
<td>covariance</td>
<td>M=50, N=70</td>
<td>191,600</td>
<td>368,775</td>
<td>70</td>
<td>1.925</td>
<td>4775</td>
<td>1275</td>
</tr>
<tr>
<td>dorigen</td>
<td>P=10, Q=15, R=20</td>
<td>123,400</td>
<td>237,000</td>
<td>150</td>
<td>1.921</td>
<td>3400</td>
<td>3000</td>
</tr>
<tr>
<td>durbin</td>
<td>N=250</td>
<td>126,246</td>
<td>250,993</td>
<td>252</td>
<td>1.988</td>
<td>250</td>
<td>249</td>
</tr>
<tr>
<td>fdtd-2d</td>
<td>T=20, X=30, Y=40</td>
<td>256,479</td>
<td>436,580</td>
<td>60</td>
<td>1.702</td>
<td>3579</td>
<td>1199</td>
</tr>
<tr>
<td>gemm</td>
<td>P=60, Q=70, R=80</td>
<td>1,026,800</td>
<td>1,684,200</td>
<td>70</td>
<td>1.640</td>
<td>14600</td>
<td>4200</td>
</tr>
<tr>
<td>genver</td>
<td>N=120</td>
<td>159,480</td>
<td>259,440</td>
<td>120</td>
<td>1.627</td>
<td>250</td>
<td>120</td>
</tr>
<tr>
<td>gesumv</td>
<td>N=250</td>
<td>376,000</td>
<td>500,500</td>
<td>500</td>
<td>1.331</td>
<td>125250</td>
<td>250</td>
</tr>
<tr>
<td>heat-3d</td>
<td>T=40, N=20</td>
<td>308,480</td>
<td>491,520</td>
<td>20</td>
<td>1.593</td>
<td>1280</td>
<td>512</td>
</tr>
<tr>
<td>jacobid1</td>
<td>T=100, N=400</td>
<td>239,202</td>
<td>398,000</td>
<td>100</td>
<td>1.664</td>
<td>402</td>
<td>398</td>
</tr>
<tr>
<td>jacobid2</td>
<td>T=20, N=30</td>
<td>157,808</td>
<td>282,240</td>
<td>20</td>
<td>1.789</td>
<td>1008</td>
<td>784</td>
</tr>
<tr>
<td>ludcmp</td>
<td>N=80</td>
<td>345,520</td>
<td>676,240</td>
<td>79</td>
<td>1.963</td>
<td>6400</td>
<td>1</td>
</tr>
<tr>
<td>mvt</td>
<td>N=200</td>
<td>200,800</td>
<td>320,000</td>
<td>200</td>
<td>1.594</td>
<td>40800</td>
<td>400</td>
</tr>
<tr>
<td>seidel2d</td>
<td>M=20, N=40</td>
<td>261,520</td>
<td>490,960</td>
<td>60</td>
<td>1.877</td>
<td>1600</td>
<td>1</td>
</tr>
<tr>
<td>symm</td>
<td>M=40, N=60</td>
<td>254,020</td>
<td>440,400</td>
<td>120</td>
<td>1.734</td>
<td>5680</td>
<td>2400</td>
</tr>
<tr>
<td>syr2k</td>
<td>M=20, N=30</td>
<td>111,000</td>
<td>180,900</td>
<td>60</td>
<td>1.630</td>
<td>2100</td>
<td>900</td>
</tr>
<tr>
<td>syrk</td>
<td>M=60, N=80</td>
<td>594,480</td>
<td>975,240</td>
<td>81</td>
<td>1.640</td>
<td>8040</td>
<td>3240</td>
</tr>
<tr>
<td>trisolv</td>
<td>N=400</td>
<td>240,600</td>
<td>320,000</td>
<td>399</td>
<td>1.330</td>
<td>86600</td>
<td>1</td>
</tr>
<tr>
<td>trmm</td>
<td>M=60, N=80</td>
<td>294,570</td>
<td>571,200</td>
<td>80</td>
<td>1.939</td>
<td>6570</td>
<td>4800</td>
</tr>
</tbody>
</table>

acyclic graph partitioning method on DAG instances coming from two sources. The first set of instances is from the Polyhedral Benchmark suite (PolyBench) [26], whose parameters are listed in Table 5.1. The graphs in the Polyhedral Benchmark suite arise from various linear computation kernels. The parameters in the second column of Table 5.1 represent the size of these computation kernels. For more details, we refer the reader to the description of the Polyhedral Benchmark suite (PolyBench) [26]. The second set of instances is obtained from the matrices available in the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [8]. From this collection, we pick all the matrices satisfying the following properties: listed as binary, square, and has at least 100000 rows and at most 2 nonzeros. There were a total of 95 matrices at the time of experimentation, where two matrices (ids 1514 and 2294) having the same pattern. We discard the duplicate and use the remaining 94 matrices for experiments. For each such matrix, we take the strict upper triangular part as the associated DAG instance, whenever this part has more nonzeros than the lower triangular part; otherwise we take the strict lower triangular part. All edges have unit cost, and all vertices have unit weight.

Since the proposed heuristics have a randomized behavior (the traversals used in the coarsening and refinement heuristics are randomized), we run them 10 times for each DAG instance, and report the averages of these runs. We use performance profiles [9] to present the edge-cut results. A performance profile plot shows the probability that a specific method gives results within a factor θ 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 ε = 0.03 in (2.1) for all experiments. The vertices are unit weighted, therefore, the imbalance 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 profile chart given in Figure 5.1 shows the effect of the coarsening heuristics on the final edge cut for the whole dataset. The variants of the proposed multilevel algorithm which use different coarsening schemes are named as CoTop (Section 4.1.1), CoCyc (Section 4.1.2), and CoHyb (Section 4.1.3). Here, and in the rest of the paper, we used a randomized Depth-First topological order for the node traversal in the coarsening heuristics, since it performed better in practice. 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.

Another important characteristic to assess for a coarsening heuristic is its contraction 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. Table 5.2 gives the maximum, the average, and the standard deviation of vertex and edge weight ratios, and the average, the minimum, and the maximum number of coarsening levels observed for the two datasets. An effective coarsening heuristic should have small vertex and edge weight ratios. We see that CoCyc and CoHyb behave similarly and provide slightly better results than CoTop on both datasets. The graphs from the two datasets have different characteristics. All coarsening heuristics perform better on the PolyBench instances compared to the UFL instances: they obtain smaller ratios in the number of remaining vertices, and yield smaller edge weights. Furthermore, the maximum vertex and edge weight ratios are smaller in PolyBench instances, again with all coarsening methods. To the best of our understanding, these happen due to two reasons; (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). Based on Figure 5.1 and Table 5.2, we set CoHyb as the default coarsening heuristic, as it performs better than CoTop in terms of final edge cut, and is guaranteed to be more efficient than CoCyc in terms of run time.

5.2. Constraint coarsening and initial partitioning. We now investigate the effect of using undirected graph partitioners to obtain a more effective coarsen-
Table 5.2: The maximum, average, and standard deviation of vertex and edge weight ratios, and the average, the minimum, and the maximum number of coarsening levels for the UFL dataset on the upper half of the table, and for the PolyBench dataset on the lower half.

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 clearly separates CoHyb_C and CoHyb_CIP from CoHyb after $\theta = 1.1$. Furthermore, applying the constraint initial partitioning (on top of the constraint coarsening) brings 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.

5.3. Evaluating 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 Section 4.2.2. Both variants use the same initial partitioning approach, which utilizes MeTiS \[16\] as the undirected partitioner. The difference between UndirFix and CoHyb_CIP is the latter’s ability to refine the initial partition at multiple levels. Figure 5.3 presents this comparison. The plots show that the multilevel scheme CoHyb_CIP outperforms the single level scheme UndirFix at all appropriate ranges of $\theta$, 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. \[21\], and
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).

briefly with our previous work [15].

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 use the average edge cut value of 10 runs for CoTop and CoHyb_CIP and the average values presented in [21] for the evolutionary algorithm. As seen in the figure, the CoTop variant of the proposed multilevel approach obtains 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. The two tables just after them (Tables A.3 and A.4) give the associated balance factors. The variants CoHyb_CIP and CoTop of the proposed algorithm obtain strictly better results than the evolutionary approach in 78 and 75 instances (out of 115), respectively, when the average edge cuts are compared.
Fig. 5.4: Performance profiles for the edge cut obtained by CoHyb_CIP, CoTop, and Moreira et al.’s approach on the PolyBench dataset with $k \in \{2, 4, 8, 16, 32\}$.

As seen in the last row of Table A.2, CoHyb_CIP obtains 26% less edge cut than
the evolutionary approach on average (geometric mean) when the average cuts are
compared (0.74 vs. 1.00 in the table); when the best cuts are compared, CoHyb_CIP
obtains 48% less edge cut (0.50 vs. 0.96). Moreover, CoTop obtains 37% less edge cut
than the evolutionary approach when the average cuts are compared (0.63 vs. 1.00
in the table); when the best cuts are compared, CoTop obtains 41% less cut (0.57
vs. 0.96). In some instances (for example covariance and gemm in Table A.1 and
syrk and trmm in Table A.2), we see large differences between the average and the
best results of CoTop and CoHyb_CIP. Combined with the observation that CoHyb_CIP
yields better results in general, this suggests that the neighborhood structure can be
improved (see the notion of the strength of a neighborhood [24, Section 19.6]). All
partitions attain 3% balance.

The proposed approach with all the reported variants take about 30 minutes to
complete the whole set of experiments for this dataset, whereas the evolutionary ap-
proach 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. [21] is more com-
parable in terms of characteristics with our multilevel scheme. When we compare CoTop
with the results of the multilevel algorithm by Moreira et al., our approach provides
results that are 37% better on average and CoHyb_CIP approach provides results that
are 26% better on average, highlighting the fact that keeping the acyclicity of the
directed graph through the multilevel process is useful.

Finally, CoTop and CoHyb_CIP also outperform the previous version of our mul-
tilevel partitioner [15], which is based on a direct $k$-way partitioning scheme and
matching heuristics for the coarsening phase, by 45% and 35% on average, respec-
tively, on the same dataset.

5.5. Single commodity flow-like problem instances. In many of the in-
stances 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
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 criterion. As seen in this figure, CoHyb is the best performing variant, and 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 with the 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 Section 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.

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 is run 10 times on each graph. As expected, CoTop offers the best performance, and CoHyb offers a good trade-off between CoTop and CoCyc. An interesting remark is that these three algorithms have a better run time than the single level algorithm UndirFix. For example, on the average, CoTop is 1.44 times faster than UndirFix. This is mainly due to cost of fixing acyclicity. Undirected partitioning accounts for roughly 25% of the
Table 5.3: 13 instances from the UFL dataset with the longest coarsening times for CoTop.

<table>
<thead>
<tr>
<th>Graph</th>
<th>#vertex</th>
<th>#edge</th>
<th>Max In</th>
<th>Max Out</th>
<th>Avg Deg</th>
<th>#source</th>
<th>#target</th>
</tr>
</thead>
<tbody>
<tr>
<td>333SP</td>
<td>3,712,815</td>
<td>11,108,633</td>
<td>9</td>
<td>27</td>
<td>2.992</td>
<td>188,112</td>
<td>316,151</td>
</tr>
<tr>
<td>AS365</td>
<td>3,799,275</td>
<td>11,368,076</td>
<td>10</td>
<td>13</td>
<td>2.992</td>
<td>306,791</td>
<td>519,431</td>
</tr>
<tr>
<td>M6</td>
<td>3,501,776</td>
<td>10,501,936</td>
<td>10</td>
<td>10</td>
<td>2.999</td>
<td>280,784</td>
<td>472,230</td>
</tr>
<tr>
<td>cit-Patents</td>
<td>3,774,768</td>
<td>16,518,209</td>
<td>779</td>
<td>770</td>
<td>4.376</td>
<td>515,980</td>
<td>1,685,419</td>
</tr>
<tr>
<td>delaunay-n22</td>
<td>4,194,304</td>
<td>12,582,869</td>
<td>15</td>
<td>17</td>
<td>3</td>
<td>555,807</td>
<td>337,743</td>
</tr>
<tr>
<td>hugebubbles-0010</td>
<td>19,458,087</td>
<td>29,179,764</td>
<td>3</td>
<td>3</td>
<td>1.5</td>
<td>3,355,886</td>
<td>3,054,827</td>
</tr>
<tr>
<td>hugetrace-00020</td>
<td>16,002,413</td>
<td>23,998,813</td>
<td>3</td>
<td>3</td>
<td>1.5</td>
<td>2,514,461</td>
<td>2,407,017</td>
</tr>
<tr>
<td>hugetrace-00010</td>
<td>6,592,765</td>
<td>9,885,854</td>
<td>3</td>
<td>3</td>
<td>1.5</td>
<td>1,085,866</td>
<td>1,006,163</td>
</tr>
<tr>
<td>italy-oss</td>
<td>6,686,493</td>
<td>7,013,978</td>
<td>5</td>
<td>8</td>
<td>1.049</td>
<td>155,509</td>
<td>458,561</td>
</tr>
<tr>
<td>rgg-n=2-22-s0</td>
<td>4,194,304</td>
<td>30,359,198</td>
<td>24</td>
<td>25</td>
<td>7.238</td>
<td>3,550</td>
<td>3,576</td>
</tr>
<tr>
<td>road-usa</td>
<td>23,947,347</td>
<td>28,854,312</td>
<td>8</td>
<td>8</td>
<td>1.205</td>
<td>6,392,288</td>
<td>8,010,032</td>
</tr>
<tr>
<td>vb-edu</td>
<td>9,845,725</td>
<td>29,494,732</td>
<td>17,489</td>
<td>3841</td>
<td>2.996</td>
<td>1,489,057</td>
<td>2,794,680</td>
</tr>
<tr>
<td>wikipedia-20060925</td>
<td>2,983,494</td>
<td>26,103,026</td>
<td>74,970</td>
<td>5,844</td>
<td>8.749</td>
<td>1,406,429</td>
<td>72,744</td>
</tr>
</tbody>
</table>

Fig. 5.6: Runtimes for CoTop, CoCyc, and CoHyb variants of the proposed multilevel scheme. For each bar group, the first, second, and the third bar present the detailed runtimes of CoTop, CoCyc, and CoHyb, respectively.

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. Unlike the standard graph partitioning problem, the directions of the edges are important and the resulting partitions should have acyclic dependencies.

We proposed coarsening, initial partitioning, and refinement heuristics for the target problem. The proposed heuristics take the directions of the edges into account and maintain the acyclicity through all the multilevel hierarchy. We also proposed efficient and effective approaches to use the standard undirected graph partitioning tools in the multilevel scheme 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 a hybrid one which avoids the cycles, and in case it does not, performs a fast cycle detection (CoHyb_CIP) for the general case; (ii) a pure multilevel
scheme without constraint coarsening, using the hybrid coarsening heuristic (CoHyb)
for the cases where a number of sources need to be separated from a number of targets;
(iii) a 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 are shown to be more effective and efficient than the current state of the
art.

An avenue for the future work is 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 [22]. Plugging
in our multilevel scheme to that framework can yield significant improvements.

Acknowledgment. We thank John Gilbert for his comments on an earlier ver-

tion of this work presented at CSC’16. John suggested that we look at the spiral
ordering of the grid graph.

REFERENCES

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.


[3] U. V. Çatalyurek and C. Aykanat, PaToH: A Multilevel Hypergraph Partitioning Tool,
Version 3.0, Bilkent University, Dept. Comp. Engineering, Ankara, 06533 Turkey. PaToH


Proceedings of the 31st Annual Design Automation Conference, DAC’94, New York, NY,

Tables A.1 and A.2 the detailed edge cut results of the proposed CoTop, CoHyb, CIP

Appendix A. Detailed results on the PolyBench instances. We give in

of Moreira et al.’s evolutionary algorithm [21]. Tables A.3 and A.4 give the balance attained in the partitions. In these two tables, the average balance of the ten runs yielding the average edge cut of Tables A.1 and A.2 is reported per problem instance. The balance of the partition yielding the best edge cut of the previous tables is also given per problem instance.
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).
<table>
<thead>
<tr>
<th>Graph</th>
<th>$k$</th>
<th>Moreira et al. [21]</th>
<th>CoHyb_CIP</th>
<th>CoTop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>2</td>
<td>596</td>
<td>596</td>
<td>646</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1493</td>
<td>1492</td>
<td>1617</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>3136</td>
<td>3136</td>
<td>2845</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>6340</td>
<td>6338</td>
<td>4519</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>8923</td>
<td>8750</td>
<td>6742</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>2</td>
<td>2994</td>
<td>2991</td>
<td>4327</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5701</td>
<td>5700</td>
<td>8405</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9417</td>
<td>9416</td>
<td>14872</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16274</td>
<td>16231</td>
<td>22626</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>22181</td>
<td>21758</td>
<td>30423</td>
</tr>
<tr>
<td>lu</td>
<td>2</td>
<td>5210</td>
<td>5162</td>
<td>5351</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>13528</td>
<td>13510</td>
<td>21258</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>33307</td>
<td>33211</td>
<td>53643</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>74543</td>
<td>74006</td>
<td>105289</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>130674</td>
<td>129954</td>
<td>156187</td>
</tr>
<tr>
<td>ludcmp</td>
<td>2</td>
<td>5380</td>
<td>5337</td>
<td>5731</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14744</td>
<td>14744</td>
<td>25247</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>37228</td>
<td>37069</td>
<td>60298</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>78646</td>
<td>78467</td>
<td>106223</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>134758</td>
<td>134288</td>
<td>158619</td>
</tr>
<tr>
<td>mvt</td>
<td>2</td>
<td>24528</td>
<td>23091</td>
<td>57216</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>74386</td>
<td>73035</td>
<td>55679</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>86525</td>
<td>82221</td>
<td>62453</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>99144</td>
<td>97941</td>
<td>71650</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>105066</td>
<td>104917</td>
<td>83635</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>2</td>
<td>4991</td>
<td>4969</td>
<td>4374</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12197</td>
<td>12169</td>
<td>13177</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>21419</td>
<td>21400</td>
<td>24396</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>38222</td>
<td>38110</td>
<td>38065</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>52246</td>
<td>51531</td>
<td>58319</td>
</tr>
<tr>
<td>symm</td>
<td>2</td>
<td>94357</td>
<td>94214</td>
<td>26374</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>127497</td>
<td>126207</td>
<td>59815</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>152984</td>
<td>151168</td>
<td>91892</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>167822</td>
<td>167512</td>
<td>105418</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>174938</td>
<td>174843</td>
<td>108950</td>
</tr>
<tr>
<td>syrk</td>
<td>2</td>
<td>11098</td>
<td>3894</td>
<td>4313</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>49662</td>
<td>48021</td>
<td>12192</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>57584</td>
<td>57408</td>
<td>29194</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>59780</td>
<td>59594</td>
<td>29519</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>60502</td>
<td>60085</td>
<td>36111</td>
</tr>
<tr>
<td>trisolv</td>
<td>2</td>
<td>21802</td>
<td>218019</td>
<td>76767</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>28908</td>
<td>289088</td>
<td>72418</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>32946</td>
<td>327712</td>
<td>112236</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>352232</td>
<td>351824</td>
<td>179042</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>362016</td>
<td>359544</td>
<td>196173</td>
</tr>
<tr>
<td>trmm</td>
<td>2</td>
<td>6788</td>
<td>3549</td>
<td>367</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>43927</td>
<td>43549</td>
<td>38148</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>66148</td>
<td>65662</td>
<td>20163</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>71838</td>
<td>71447</td>
<td>20421</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>79125</td>
<td>79071</td>
<td>25279</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>138907</td>
<td>138725</td>
<td>38057</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>192752</td>
<td>191492</td>
<td>58477</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>225192</td>
<td>223529</td>
<td>92185</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>240788</td>
<td>238159</td>
<td>128838</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>246407</td>
<td>245173</td>
<td>153644</td>
</tr>
<tr>
<td>Geomean</td>
<td></td>
<td>1.00</td>
<td>0.96</td>
<td>0.74</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>CoHyb_CIP</th>
<th>CoTop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k</td>
<td>Average</td>
<td>Best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Graph</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.029</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.021</td>
<td>1.009</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.001</td>
<td>1.001</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
</tbody>
</table>

Table A.3: The partition balances for the edge cuts given in table A.1
<table>
<thead>
<tr>
<th>Graph</th>
<th>k</th>
<th>CoHyb_CIP</th>
<th>CoTop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>Best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>2</td>
<td>1.009</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.019</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.016</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.025</td>
<td>1.024</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.027</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.017</td>
<td>1.012</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.027</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.027</td>
<td>1.028</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.029</td>
<td>1.028</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>2</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.017</td>
<td>1.012</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.029</td>
<td>1.030</td>
</tr>
<tr>
<td>lu</td>
<td>2</td>
<td>1.023</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>ludcmp</td>
<td>2</td>
<td>1.020</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>mvt</td>
<td>2</td>
<td>1.021</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.025</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>seidel-2d</td>
<td>2</td>
<td>1.012</td>
<td>1.011</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.024</td>
<td>1.022</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.026</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.029</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.029</td>
<td>1.030</td>
</tr>
<tr>
<td>symm</td>
<td>2</td>
<td>1.016</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.021</td>
<td>1.019</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.027</td>
<td>1.029</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>syr2k</td>
<td>2</td>
<td>1.018</td>
<td>1.016</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.029</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>syrk</td>
<td>2</td>
<td>1.021</td>
<td>1.022</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.029</td>
<td>1.027</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>trisolv</td>
<td>2</td>
<td>1.012</td>
<td>1.021</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.026</td>
<td>1.028</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.028</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td>trmm</td>
<td>2</td>
<td>1.028</td>
<td>1.024</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.027</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1.030</td>
<td>1.030</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1.030</td>
<td>1.030</td>
</tr>
</tbody>
</table>

Table A.4: The partition balances for the edge cuts given in table A.2. The last 3 lines (Min, Average, Max) are for the whole PolyBench dataset (i.e., computed by combining this table with the previous one).