Acyclic Partitioning of Large Directed Acyclic Graphs

Julien Herrmann*, Jonathan Kho*, Bora Uçar†, Kamer Kaya‡ and Ümit V. Çatalyürek*
*School of Computational Science and Engineering
Georgia Institute of Technology, Atlanta, Georgia 30332–0250
e-mails: julien.herrmann@cc.gatech.edu, jkho3@gatech.edu, umit@gatech.edu
†CNRS and LIP (UMR 5668 CNRS, ENS Lyon, UCB Lyon 1, Inria) ENS Lyon, France
e-mail: bora.ucar@ens-lyon.fr
‡Sabancı University, Istanbul, Turkey
The Ohio State University, Columbus, OH, USA
e-mail: kaya@sabanciuniv.edu

***Abstract*—Finding a good partition of a computational directed acyclic graph associated with an algorithm can help find an execution pattern improving data locality, conduct an analysis of data movement, and expose parallel steps. 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 and develop a direct k-way partitioning scheme. To the best of our knowledge, no such scheme exists in the literature. 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, the total volume of communication between the parts, and the critical path latencies. We use the solution returned by well-known undirected graph partitioners as a baseline to evaluate our acyclic partitioner, knowing that the space of solution is more restricted in our problem. The experiments are run on large graphs arising from linear algebra applications.**

***Keywords*-directed graph; acyclic partitioning; multilevel partitioning;**

I. Introduction

We investigate the problem of partitioning directed acyclic graphs for task mapping in parallel systems to improve the parallel execution time. To the best of our knowledge, there has been no work on directed acyclic graph partitioning for this purpose. When the underlying model is a directed graph, usually it is converted to an undirected graph, and a traditional, undirected graph partitioning approach is used.

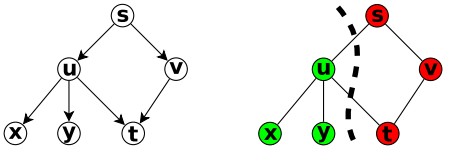
Load balancing and mapping is only one part of the parallel execution. A full execution needs a complete schedule that obeys the dependencies [1]. One can still use undirected graph partitioning to reduce the communication among processors, however, overlooking dependencies may force to create schedules for which the critical execution path creates a cut among the parts, hence extra communication latency.

Here we show that a special class of partitioning, namely *acyclic partitioning* is needed to more accurately solve the problem of reducing the parallel execution time. Informally, an acyclic partitioning is a partitioning of vertices into parts where there are no cycles among parts, when the inter-part edges are considered (formal definition will be given in the next section). There are some heuristics for this purpose. We propose the first of its class, a multilevel directed acyclic partitioning method.

The directed acyclic graph (DAG) partitioning problem appears in many applications. At the beginning, we were motivated by the characterization of the parallel data movement complexity and dynamic analysis of the data locality potential [2], [3]. As the latency and energy gap increases among the hierarchical layers of the modern computers, it is crucial to understand the *data movement complexity* of an algorithm, instead of its time complexity. However, this new form of complexity is not well characterized and harder to measure: it depends on code transformations and the architectural parameters such as the fast memory (registers/caches) capacity. A thorough understanding is important to reveal the possible performance improvements beyond the current compiler optimizations, and to select the most suitable implementation for a specific architecture. Closer to our objective of optimizing the parallel execution time, another formulation of the DAG partitioning problem arises in exposing parallelism in automatic differentiation [4, Ch.9], and in general, in the computation of the Newton step for solving nonlinear systems [5]. Other important applications of the DAG partitioning problem include (i) fusing loops for improving temporal locality, and enabling streaming and array contractions in runtime systems [6], such as Bohrium [7]; (ii) analysis of cache efficient execution of streaming applications on uniprocessors [8].

Let us consider a toy example which maps six atomic tasks to two processors. Figure 1a displays the directed graph with six vertices and five edges corresponding to task-dependencies; an edge (u, v) implies that v depends on u. If the edge orientations are removed and the computation is modeled with an undirected graph, the best balanced partition with a 3/3 vertex split has two edges in the cut (all other balanced partitions have three or more edges in the cut)

as shown in Fig. 1b. However, when the edge orientations are considered, the two parts formed by the best undirected partition become inter-dependent in a cyclic way as shown in Fig. 1c. Hence, when they are mapped to different processors, either on the same node or distributed nodes, the communication uses L3 cache instead of L1 and this cyclic dependency will increase the execution time due to the latency incurred by the extra communication among the processors. All three-vertex paths (i.e., $(s,u,x),(s,u,y),(s,u,t)$, and (s,v,t)) in this toy example are the code's critical paths. Let us investigate this using the L1 and L3 cache latencies of i7-4770 CPU[1] which are 4 and 36 cycles, respectively. The execution is penalized by these latencies due to the intra- and inter-part edges. For the performance evaluation, each task is assumed to be performed in a single cycle. Hence, for the best undirected partition, executing the (s,u,t) path will incur a latency of 75. Yet, for the acyclic partition in Fig. 1d even though the edge cut is three, the total latency of this path (and also (s,v,t) and (s,u,y) paths too) will be 43.



(a) A toy graph (b) Undirected partitioning

(c) Undirected partitioning with directed edges (d) Acyclic partitioning of the directed graph

Figure 1: **a)** A toy example with six tasks and six dependencies, **b)** a valid 2-way partitioning of an undirected graph, **c)** a non-acyclic partitioning when edges are oriented, **d)** an acyclic partitioning of the same directed graph.

The rest of the paper is organized as follows: Section II introduces the notation and background on directed acyclic graph partitioning and Section III briefly surveys the existing literature. The proposed multilevel partitioning heuristics are proposed in Section IV. Section V presents the experimental results and Section VI concludes the paper.

II. Preliminaries and Notation

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

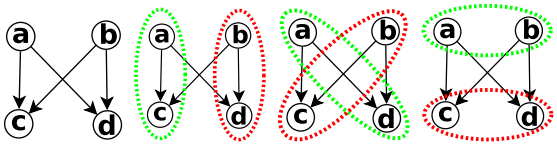
The path $u \rightsquigarrow v$ represents a dependency of v to u. We say that the edge (u,v) is *redundant* if there exists another $u \rightsquigarrow v$ path in the graph. That is when we remove a redundant (u,v) edge, u remains to be connected to v, and hence, the dependency information is preserved. We use $\texttt{Pred}[v] = \{u|(u,v) \in E\}$ to represent the (immediate) predecessors of a vertex v, and $\texttt{Succ}[v] = \{u|(v,u) \in E\}$ to represent the (immediate) successors of v. Every vertex u has a weight denoted by w_u and every edge $(u,v) \in E$ has a weight 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 $1 \le i \le k$ is equal to $\sum_{u\in V_i} w_u$ which is the total vertex weight in that part. Given a partition, an edge is called a *cut edge* if its endpoints are in different parts. In practice, a *constraint*, e.g., balance, lower and/or upper bound, on the part weights accompanies the problem with an *objective function* based on the weights of the cut edges. 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 $1 \le i \ne j \le k$.

There is a related definition in the literature [3], 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.

Figure 2 shows that the definitions of an acyclic partition and a convex partition are not equivalent. Indeed, for the toy graph in Figure 2a, there are three possible balanced partition shown in Figs. 2b, 2c, and 2d. They are all convex but only that in Fig. 2d is acyclic.



(a) Graph (b) Cyclic & convex (c) Cyclic & convex (d) Acyclic & convex

Figure 2: A toy example (left), two cyclic and convex partitionings, and an acyclic partitioning (right).

When there is an upper bound on the part weights and another upper bound on the sum of the cut edge weights, it is shown that deciding on the existence of a k-way acyclic partition is NP-complete [9]. Several other objectives can be taken into account, such as the volume of data communication (by counting only the number of outputs

[1] http://www.7-cpu.com/cpu/Haswell.html

that have to be sent from a partition set to another), or the longest path in the partitioned graph. Our heuristics will be evaluated with these three metrics in Section V. The formal problem treated in this paper is defined as follows.

Definition 2.2 (DAG partitioning problem): Given a DAG G, and an upper bound B, find an acyclic k-way partition $P = \{V_1, \ldots, V_k\}$ such that the weight of each part is no larger than B and the edge cut is minimized.

III. Related Work

Fauzia et al. [3] 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. We implement this heuristic with the same limit on part sizes to address our acyclic partitioning problem and use this heuristic in Section V-B.

Kernighan [10] provided 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 running time proportional to the number of edges in the graph, if the part weights are taken as constant. Although, Kernighan's algorithm guarantees that the upper bound and the lower bound on the weights of the parts are met, there is no guarantee on the final number of parts. In order to use this algorithm, as a heuristic, we modify it to obtain the desired number of parts. This modified version is used in our multilevel heuristic (Section IV-B) where a topological order of the vertices is computed and used as a total order.

Cong et al. [11] describe a Fiduccia-Mattheyses (FM)-based acyclic multi-way partitioning algorithm for boolean networks. They generate an initial acyclic partitioning 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. A detailed description of this heuristic is given in Section IV-C, since we use it as a refinement technique in our multilevel partitioner.

Other related work on acyclic partitioning of directed graphs include (i) an exact, branch-and-bound algorithm by Nossack and Pesch [13] which works on the integer programming formulation of the acyclic partitioning problem. This solution is, of course, too costly to be used in practice; (ii) Wong et.al. [14] present an acyclic multi-way partitioning heuristic with a process similar to the multilevel scheme where they use clustering and adopt Cong et al.'s approach later.

IV. Directed Multilevel Graph Partitioning

We propose a new multilevel tool for obtaining acyclic partitions of directed graphs. Multilevel frameworks became de-facto standard for solving graph and hypergraph partitioning problems efficiently, hence used by almost all, if not all, of the current state-of-the-art partitioning tools [15], [16], [17]. Similar to the other multilevel graph frameworks, our algorithm has three phases, a **coarsening** phase aiming to reduce the number of vertices by clustering them, the **initial partitioning** of the coarsened graph and the **uncoarsening** phase where the initial solution to the finer graphs is projected and refined iteratively until a solution for the original graph is obtained.

A. Coarsening

In this phase, we obtain smaller acyclic graphs by combining the vertices until a minimum vertex count is reached or the reduction in the number of vertices is lower than a threshold. At each level ℓ, we start with a finer acyclic graph G_ℓ, we compute a valid matching $\mathcal{M}_\ell$ ensuring the acyclicity, and obtain a coarser acyclic graph $G_{\ell+1}$. However, unlike in the undirected model, not all the 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 an edge is contractible (its end points are matchable), if unifying the two endpoints does not create a cycle. To maintain acyclicity, we propose a novel and efficient mechanism to check if an edge is contractible or not based on a precomputed topological ordering of the vertices.

*Definition 4.1 (***Matching):** A matching of a DAG $G = (V, E)$ is a subset of edges without common vertices.

*Definition 4.2 (***Coarse Graph):** Given a DAG $G = (V, E)$ and a matching M of G, we let $G_{|M}$ denote the coarse graph created by contracting the edges of M.

*Definition 4.3 (***Feasible Matching):** A feasible matching M of a DAG $G = (V, E)$ is a matching such that $G_{|M}$ is acyclic.

Theorem 1: Given a DAG $G = (V, E)$ with $u, v \in V$ and $(u, v) \in E$. Then $G_{|\{(u,v)\}}$ is acyclic if and only if there is no path from u to v in G avoiding the edge (u, v).

Proof: Let $G = (V, E)$ be a DAG with $u, v \in V$ and $(u, v) \in E$. Let $G' = (V', E') = G_{|\{(u,v)\}}$ be the coarse graph obtained when merging u and v. Let 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 obviously all the edges of this path are also in

G' and the corresponding path in G' goes from w to w, which creates a cycle in the coarse graph.

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 and $b \in V'$ be the last one. Let $\mathbf{p}$ be an $a \rightsquigarrow b$ path in G' such that $(w,a)\cdot\mathbf{p}\cdot(b,w)$ is a $w \rightsquigarrow w$ 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 $E\backslash\{(u,v)\}$. Moreover, either $(u,a) \in E$ or $(v,a) \in E$, and either $(b,u) \in E$ or $(b,v) \in E$:

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

■

At each step of the coarsening phase, we want to find a matching M ensuring that the coarsened graph will be acyclic. For running time complexity reasons, we rely only on static information while searching for a feasible matching. Let $\text{TOPL}(v)$ be the level of a vertex v in a topological ordering of $G = (V,E)$. In Theorem 2, we give the necessary and sufficient conditions for a matching to be feasible.

Theorem 2 (Correctness of the proposed coarsening): Let $G = (V,E)$ be a DAG and $M = \{(u_1,v_1),\ldots,(u_k,v_k)\}$ a matching such that:

- $\forall i \in \{1,\ldots,k\}$, $\text{TOPL}(v_i) = \text{TOPL}(u_i) + 1$, or $\texttt{Succ}[u_i] = \{v_i\}$, or $\texttt{Pred}[v_i] = \{u_i\}$,
- $\forall i \neq j \in \{1,\ldots,k\}$, either $(u_i,v_j) \notin E$ or $\text{TOPL}(u_i) \neq \text{TOPL}(v_j) + 1$.

Then, the coarse graph $G_{|M}$ is acyclic.

Proof: Let us assume (for the sake of contradiction) that there is a matching with the same properties above, and the coarsened graph has a cycle. We pick $M = \{(u_1,v_1),\ldots,(u_k,v_k)\}$ a minimal cardinality one. Let w_i be the merged vertex in the coarsened graph $G_{|M}$ obtained by merging u_i and v_i for all $\{1,\ldots,k\}$. By assumption, there is a cycle in $G_{|M}$. Let us consider $\mathbf{c}$ a minimum length cycle in $G_{|M}$. This cycle passes through all the w_i vertices. Otherwise, there would be a smaller cardinality matching with the properties above and creating a cycle in the coarsened graph, contradicting the minimal cardinality of M. 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 for the indices.

After the reordering, for every $i \in \{1,\ldots,k\}$, there is a path in $G_{|M}$ 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, either **1)** there is a $u_i \rightsquigarrow u_{i+1}$ path in G; or **2)** there is a $u_i \rightsquigarrow v_{i+1}$ path in G; or **3)** there is a $v_i \rightsquigarrow u_{i+1}$ path in G; or **4)** there is a $v_i \rightsquigarrow v_{i+1}$ path in G.

Let us assume that there exists an $i_0 \in \{1,\ldots,k\}$ such that there is a path from $w_{i_0-1} = \{u_{i_0-1}; v_{i_0-1}\}$ to u_{i_0} and another path from u_{i_0} to $w_{i_0+1} = \{u_{i_0+1}; v_{i_0+1}\}$ in G. Then there is a $w_{i_0-1} \rightsquigarrow w_{i_0+1}$ path in the coarsened graph obtained by merging all the endpoints except u_{i_0} and v_{i_0}. Hence, the matching $M\backslash\{(u_{i_0},v_{i_0})\}$ also has the same properties and forms a cycle in the coarsened graph, which contradicts the minimal cardinality assumption on M.

A similar contradiction arises if we assume that there exist two paths $w_{i_0-1} \rightsquigarrow v_{i_0}$ and $v_{i_0} \rightsquigarrow w_{i_0-1}$ in G (or $w_{i_0-1} \rightsquigarrow u_{i_0}$ and $v_{i_0} \rightsquigarrow w_{i_0-1}$ in G). Thus, for every $i \in \{1,\ldots,k\}$, there is a $u_i \rightsquigarrow v_{i+1}$ path in G and, since there is no path from v_i to v_{i+1}, vertex u_i has to have another successor than v_i. Similarly, since there is no path from u_i to u_{i+1}, vertex v_{i+1} has to have another predecessor than u_{i+1}. According to the first matching property, for every $i \in \{1,\ldots,k\}$, $\text{TOPL}(u_i) + 1 = \text{TOPL}(v_i)$.

Since there is a path from u_i to v_{i+1}, $\text{TOPL}(u_i) + 1 \leq \text{TOPL}(v_{i+1})$. According to the second property, either there is at least an intermediate vertex between u_i and v_{i+1} and then $\text{TOPL}(u_i)+1 < \text{TOPL}(v_{i+1})$; or $\text{TOPL}(u_i) + 1 \neq \text{TOPL}(v_{i+1})$ and then $\text{TOPL}(u_i) + 1 < \text{TOPL}(v_{i+1})$. Thus, in any case, for every $i \in \{1,\ldots,k\}$, $\text{TOPL}(v_i) < \text{TOPL}(v_{i+1})$, which leads the self-contradicting statement $\text{TOPL}(v_1) < \text{TOPL}(v_{k+1}) = \text{TOPL}(v_1)$ and concludes the proof.

■

We propose different matching algorithms ensuring that all the graphs obtained in the multilevel hierarchy are acyclic. These algorithms consider all the edges in the graph, one by one, and put them in the matching if they respect the properties of Theorem 2. The traversal order for the edges is based on a vertex traversal order and a priority on adjacent edges. The matching algorithms (depending on different vertex traversal orders and priority definitions on the adjacent edges) are described in Algorithm 1. Since, we can compute the TOPL value of all vertices in $O(|V|+|E|)$ time, the overall complexity of Algorithm 1 is $O(|V|+|E|)$. 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 traversal and traversal in the non-increasing order of their weights).

Algorithm 1: CompMatching

Data: Directed graph $G = (V, E)$, a traversal order of the vertices in V, a priority on edges
Result: A feasible matching M of G

```
match ← ∅
top ← CompTopLevels(G)
for u ∈ V do mark[u] ← false

for u ∈ V following the traversal order in input do
    if mark[u] then continue

    for v ∈ Pred[u] ∪ Succ[u] following given priority on edges do
        if mark[v] then continue

        if (top[u] ≠ top[v] − 1) and (|Pred[v]| ≠ 1) and (|Succ[u]| ≠ 1) then continue

        if v ∈ Pred[u] then
            M ← M ∪ {(v, u)}
            for w ∈ Succ[v] do
                if top[v] = top[w] − 1 then
                    mark[v] ← false
        else
            M ← M ∪ {(u, v)}
            for w ∈ Succ[u] do
                if top[u] = top[w] − 1 then
                    mark[w] ← false
        mark[u] ← mark[v] ← true
return M
```

B. Initial Partitioning

After the coarsening phase, we compute an initial acyclic partitioning of the coarsened graph. To do that, we present different heuristics based on existing algorithms in the literature. Since the number of edges in the coarsest graph is relatively small, it is a good practice to try different initial partitioning algorithms and pick the best solution.

1) Kernighan's Algorithm: To compute an initial partitioning of the coarsest graph, we first topologically order the vertices of this final graph. Then we use Kernighan's algorithm [10] to generate an optimal partition based on this topological ordering. Since our multilevel partitioner finds an acyclic partition with exactly k parts while respecting an upper bound constraint on their weights, B, we slightly modified the dynamic programming formulation given in Kernighan's algorithm. This new dynamic program has a linear execution time in the number of edges times k, if the part weights are considered as constant. To avoid having this complexity, we use the following heuristic as initial partitioning. We first run the original Kernighan's algorithm with an upper bound equal to B and a lower bound equal to $W - (k-1) \times B$ where W is the total weight of the graph. By doing this, there is still no guarantee on the number of parts, but we reduced the space of solutions without removing any solution with k parts, thus increasing our chances to find a solution with k parts. If it fails to find a solution with k parts (which means that for these given upper bound and lower bound, and this topological order of the vertices, the best partition does not have k parts), we run our modified version of the Kernighan's algorithm ensuring a number of partition parts equal to k.

2) Greedy Partitioning: We propose another initial acyclic partitioning heuristic (for initial partitioning) using a greedy algorithm filling the parts one by one with the current best *eligible vertex* among the *free vertices*. At any given time, a *free vertex* is a vertex that has not been put in a part yet. An *eligible vertex* is a free vertex without predecessors or whose predecessors are all not free. For each eligible vertex u, we define $\text{gain}_i(u)$ as the decrease in the edge cut when putting it into part V_i. Thus, the best eligible vertex is the one with the largest gain for the part we are currently filling. The gains can be computed as shown in Algorithm 3.

Algorithm 2: CompGain

Data: Directed graph $G = (V, E)$, vertex u, partition part, and destination part d
Result: Gain of moving vertex u to part d

```
gain ← 0
for v ∈ Pred[u] do
    if part[u] = part[v] then
        gain ← gain − c_{v,u}
    else if part[v] = d then
        gain ← gain + c_{v,u}
for v ∈ Succ[u] do
    if part[u] = part[v] then
        gain ← gain − c_{v,u}
    else if part[v] = d then
        gain ← gain + c_{v,u}
return gain
```

C. Refinement/Uncoarsening

During the uncoarsening level corresponding to the ℓ-th coarsening level, we project the partition $\mathcal{P}_{\ell+1}$ obtained for $G_{\ell+1}$ to G_ℓ. Then we refine it by using an FM-like, move-based direct k-way refinement algorithm. In the undirected case, the refinement process computes the $k-1$ gains for each vertex, i.e., the decrease in the edge-cut when the vertex is moved to other part. The move with the highest gain is performed, and the corresponding vertex is marked so that it will not be moved again in this refinement pass. However, in the directed case, the best move can violate the acyclicity condition. In the refinement algorithm proposed by Cong et al. [11], a safety check is performed before moving a vertex. If the move violates acyclicity, it is not performed, and the algorithm considers the next best move. We propose a variant of this algorithm with a better

Algorithm 3: Greedy Partitioning

Data: Directed graph $G = (V, E)$ and number of parts k
Result: An acyclic partition part of G

lb $\leftarrow 0.9 \times \frac{|V|}{k}$
$\forall i \in \{1, ..., k\}, V_i \leftarrow \emptyset$; $\forall u \in V$, free$[u] \leftarrow true$
for $i \in \{1, ..., k-1\}$ **do**
  set $\leftarrow \{u \in V,$ such as Pred$[u] = \emptyset$ or $\forall v \in$ Pred$[u]$, free$[v] = false\}$
  for $u \in$ set **do**
    gain$_i[u] \leftarrow$ CompGain$(G, u,$ part$, i)$
  heap $\leftarrow$ Max-heap associated to gain$_i$
  while $|V_i| < lb$ **do**
    $u \leftarrow$ Extract max from heap
    $V_i \leftarrow V_i \cup \{u\}$
    free$[u] \leftarrow false$
    for $v \in$ Succ$[u]$ **do**
      $ready \leftarrow true$
      for $w \in$ Pred$[v]$ **do**
        if free$[w] = true$ **then** $ready \leftarrow false$
      if $ready$ **then**
        gain$_i[v] \leftarrow$ CompGain$(G, v,$ part$, i)$
        Insert v in heap
return $\{V_1, ..., V_k\}$

complexity which traverses the vertices one by one and perform the best feasible move for each.

1) Acyclic k-way refinement: The proposed heuristic relies on the cheap verification of a particular move not creating a cycle in the quotient graph. In short, a quotient graph is created at the beginning of the refinement pass and updated through the heuristic. By keeping the weights of the edges in the quotient graph, we can maintain it in time $\mathcal{O}(degree(u))$ after moving a vertex u. Checking if a given move will create a cycle can be performed in $\mathcal{O}(k^2)$ time by first trying to topologically sort the parts based on the updated quotient graph. If the topological sort fails, it means that the updated quotient graph has a cycle and hence, the move cannot be performed.

If the best move for a given vertex creates a cycle in the quotient graph, we perform its best feasible move if its gain is not smaller than the second best gain for this vertex. With this approach, we may perform a feasible move even if it is not the best one available while avoiding moves with insufficient gains. Algorithm 4 describes the refinement heuristic.

2) Topological refinement: We also design a new FM-like, move-based direct k-way refinement algorithm that ensures the acyclicity of the partition, based on a topological order of the parts.

Definitions: Given an acyclic partition $\mathcal{P} = \{V_1, \ldots, V_k\}$, we compute a topological order among parts that will be maintained during the refinement process. For two parts V_i and V_j, we write $V_i \prec V_j$ if V_i comes before V_j in that topological order. For simplicity, we assume that the parts

Algorithm 4: Acyclic k-way Refinement

Data: Directed graph $G = (V, E)$, partition part
Result: Refined partition part

for $u \in V$ **do**
  copy$[u] \leftarrow$ part$[u]$
  moved$[u] \leftarrow false$
$ec \leftarrow$ CompEdgeCut$(G,$ part$)$
$ecmin \leftarrow ec$
$QG \leftarrow$ BuildQuotientGraph$(G, part)$
for $u \in V$ *and* $k \in \{1, ..., k\}$ **do**
  $gain[u][k] \leftarrow$ CompGain$(G, u,$ part$, k)$
for $u \in V$ **do** maxgain$[u] \leftarrow \max_{i=1..k}\{$gain$[u][i]\}$
heap $\leftarrow$ Max-heap associated to maxgain
$idx, ecidx \leftarrow 0$
while heap *not empty* **do**
  $u \leftarrow$ Extract max from heap
  parts $\leftarrow$ Sorted$(\{1...k\}, key =$ gain$[u][\cdot])$
  $i \leftarrow 1$
  $k \leftarrow$ parts$[i]$
  while CreateCycle$(QG, u, k,$ part$)$ *and* gain$[u][k] \geq$ gain$[u][$parts$[2]]$ **do**
    $i \leftarrow i + 1$
    $k \leftarrow$ parts$[i]$
  moves$[idx] \leftarrow u$
  $idx \leftarrow idx + 1$
  part$[u] \leftarrow k$
  UpdateQuotientGraph(QG)
  $ec \leftarrow ec -$ gain$[u]$
  if $ec < ecmin$ **then**
    $ecmin \leftarrow ec$
    $ecidx \leftarrow idx$
  for $v \in$ Pred$[u] \cup$ Succ$[u]$ **do**
    if *not* moved$[v]$ **then** Update$($heap$,$ gain$, v)$
for $i = idx - 1$ **downto** $ecidx$ **do**
  part$[$moves$[i]] =$ copy$[$moves$[i]]$
return part

are renumbered such that $V_1 \prec V_2 \prec \cdots \prec V_k$. Let a vertex be an *incoming boundary vertex* if it has no incoming edge, or if all its incoming neighbours are in another part. Let a vertex be an *outgoing boundary vertex* if it has no outgoing edge or if all its outgoing neighbours are in another partition. Finally, let a vertex be a *boundary vertex* if it is an incoming boundary vertex and/or an outgoing boundary vertex. If a vertex is not a boundary vertex, it cannot be moved without violating the acyclicity condition. For each incoming boundary vertex u in the part V_u, let V^u_{max} be the largest part index (according to $\prec$) of its incoming neighbours. To keep the topological order among the parts, the vertex u can thus be moved only to the parts V_i such that $V^u_{max} \preceq V_i \prec V_u$. Since V^u_{max} is the only one of these parts hosting a neighbor of u, the vertex u can only be moved to the part V^u_{max}. Similarly, for each outgoing boundary vertex u, we define V^u_{min} the minimum partition of its outgoing neighbours according the $\prec$ relation. To maintain

the topological order among the parts, the vertex u can thus be moved only to the parts V_i such that $V_u \prec V_i \preceq V^u_{min}$. For the same reason as before, u can only be moved to the part V^u_{min}. There are rare cases where u is both an incoming and outgoing boundary vertex at the same time. Such vertices can be moved to either V^u_{max} or V^u_{min} but we will just consider the part with the largest gain. Hence, to maintain the topological order among the parts, we use a single eligible part for each boundary vertex.

Move selection heuristic: Our proposed refinement algorithm maintains a list of the boundary vertices at each step, the single part they can be moved to, and the gain for this move. These gains are stored in a heap to easily retrieve the maximum gain at each refinement step and to move the vertex with the largest gain. We use a tie-breaking scheme which chooses the move that will result in the smallest maximum size for a part. To avoid being stuck in a local minimum, we move every vertex that have not been moved yet even if its gain is negative. At the end, we roll back to the best partition observed. This refinement algorithm is described in Algorithm 6.

Algorithm 5: Update(heap, moveto, gain, u)

Data: Number of parts k, acyclic partition part, heap of boundary vertices heap, moveto, gain, vertex u to update
Result: Update heap, moveto, and gain

if $\texttt{Pred}[u] = \emptyset$ **then**
  $max \leftarrow \max(\texttt{part}[u] - 1, 1)$
else
  $max \leftarrow \max\{\texttt{part}[v] \text{ for } v \in \texttt{Pred}[u]\}$
if $\texttt{Succ}[u] = \emptyset$ **then**
  $min \leftarrow \max(\texttt{part}[u] + 1, k)$
else
  $min \leftarrow \min\{\texttt{part}[v] \text{ for } v \in \texttt{Succ}[u]\}$
if $(max \neq \texttt{part}[u])$ and $(min = \texttt{part}[u])$ **then**
  Put u in heap
  $\texttt{moveto}[u] \leftarrow max$
  $\texttt{gain}[u] \leftarrow \text{CompGain}(G, u, \texttt{part}, max)$
if $(min \neq \texttt{part}[u])$ and $(max = \texttt{part}[u])$ **then**
  Put u in heap
  $\texttt{moveto}[u] \leftarrow min$
  $gain[u] \leftarrow \text{CompGain}(G, u, \texttt{part}, min)$
if $(min \neq \texttt{part}[v])$ and $(max \neq \texttt{part}[u])$ **then**
  Put u in heap
  $gain1 \leftarrow \text{CompGain}(G, u, \texttt{part}, min)$
  $gain2 \leftarrow \text{CompGain}(G, u, \texttt{part}, max)$
  if $gain1 > gain2$ **then**
    $\texttt{moveto}[u] \leftarrow min$
    $\texttt{gain}[u] \leftarrow gain1$
  else
    $\texttt{moveto}[v] \leftarrow max$
    $\texttt{gain}[v] \leftarrow gain2$

Algorithm 6: Topological Refinement

Data: Directed graph $G = (V, E)$, partition part
Result: Refined partition part

for $u \in V$ **do** $\texttt{copy}[u] \leftarrow \texttt{part}[u]$
$ec \leftarrow \text{CompEdgeCut}(G, \texttt{part})$
$ecmin \leftarrow ec$
$\texttt{moveto}, \texttt{gain}, \texttt{moved}, \texttt{moves} \leftarrow []$
$\texttt{heap} \leftarrow$ Empty max-heap associated to gain
for $u \in V$ **do**
  Update(heap, moveto, gain, u)
  $\texttt{moved}[u] \leftarrow false$
$idx, ecidx \leftarrow 0$
while heap *not empty* **do**
  $u \leftarrow$ Extract max from heap
  $\texttt{moves}[idx] = u$
  $idx \leftarrow idx + 1$
  $\texttt{part}[u] \leftarrow \texttt{moveto}[u]$
  $ec \leftarrow ec - \texttt{gain}[u]$
  if $ec < ecmin$ **then**
    $ecmin \leftarrow ec$
    $ecidx \leftarrow idx$
  for $v \in \texttt{Pred}[u] \cup \texttt{Succ}[u]$ **do**
    if *not* $\texttt{moved}[v]$ **then**
      Update(heap, moveto, gain, v)
for $i = idx - 1$ **downto** $ecidx$ **do**
  $\texttt{part}[\texttt{moves}[i]] = \texttt{copy}[\texttt{moves}[i]]$
return part

V. Experimental Evaluation

A. Experimental Setup

We have performed an extensive evaluation of the proposed multilevel directed graph acyclic partitioning method (dagP) on a set of instances from the Polyhedral Benchmark suite (PolyBench) [18]. The experiments were conducted on computers equipped with dual 2.4 GHz Xeon E5-2680 processors and 128GB memory.

We have performed two different sets of experiments. In the first set, we aimed to evaluate the merits of different options of the proposed dagP method. In the second set of experiments, we investigated the effectiveness of dagP in comparison to other partitioning methods. The options that were varied include the coarsening matching order, the coarsening matching method, the initial partitioning method, and the refinement method. The names of the heuristics in the figures follow the pattern **Order-Match-Initial Partition-Refinement** where:

- **Order** defines the traversal order of the vertices during the coarsening phase described in Section IV-A: When it is 'Rand' (respectively 'Top'), the matching algorithm (Algorithm 1) traverses the vertices in a random order (respectively in a depth-first topological order).
- **Match** defines the order while traversing the adjacent edges in the matching algorithm: in 'Rand', the adjacent edges are traversed in a random order; in 'HEM' (resp.

Graph	Parameters	$\#vertex$	$\#edge$	out-deg.	deg.
2mm	P=10, Q=20, R=30, S=40	36,500	62,200	40	1.704
3mm	P=10, Q=20, R=30, S=40, T=50	111,900	214,600	40	1.918
adi	T=20, N=30	596,695	1,059,590	109,760	1.776
atax	M=210, N=230	241,730	385,960	230	1.597
covariance	M=50, N=70	191,600	368,775	70	1.925
doitgen	P=10, Q=15, R=20	123,400	237,000	150	1.921
durbin	N=250	126,246	250,993	252	1.988
fdtd-2d	T=20, X=30, Y=40	256,479	436,580	60	1.702
gemm	P=60, Q=70, R=80	1,026,800	1,684,200	70	1.640
gemver	N=120	159,480	259,440	120	1.627
gesummv	N=250	376,000	500,500	500	1.331
heat-3d	T=40, N=20	308,480	491,520	20	1.593
jacobi-1d	T=100, N=400	239,202	398,000	100	1.664
jacobi-2d	T=20, N=30	157,808	282,240	20	1.789
lu	N=80	344,520	676,240	79	1.963
ludcmp	N=80	357,320	701,680	80	1.964
mvt	N=200	200,800	320,000	200	1.594
seidel-2d	M=20, N=40	261,520	490,960	60	1.877
symm	M=40, N=60	254,020	440,400	120	1.734
syr2k	M=20, N=30	111,000	180,900	60	1.630
syrk	M=60, N=80	594,480	975,240	81	1.640
trisolv	N=400	240,600	320,000	399	1.330
trmm	M=60, N=80	294,570	571,200	80	1.939

Table I: Instances from the Polyhedral Benchmark Suite.

'NWHEM'), the adjacent edges are traversed in the non-increasing order of their weights (resp. in the non-increasing order of their weight divided by the weight of the corresponding adjacent vertex).

- **Initial Partition** defines which algorithm described in Section IV-B will be used to partition the coarsest graph. These algorithms are run multiple times, and the best solution is picked as the initial partition. When the value is 'Kern' (resp. 'GP'), we run the Kernighan algorithm (resp. the Greedy Partitioning) six times. When the value is 'Ker+GP', we run Kernighan and Greedy Partitioning three times each.
- **Refinement** defines which refinement heuristic described in Section IV-C will be used during the uncoarsening phase. When its value is 'AcycKWay' (resp. 'TopRef') we use the acyclic k-way refinement heuristic (resp. the topological refinement heuristic).

We use performance profiles [19] to present the results. In the first set of experiments, we compared the total number of edge cuts and the partitioning time for varying options of the proposed dagP method. We use the ratio of a performance indicator to the best of all the dagP options as the performance metric and call it τ. A point (τ, f) in the profile means that in f fraction of the test cases, the performance indicator of the corresponding algorithm is at most τ times worse than the best algorithm's performance indicator. Hence, the closer to the y-axis is the better the option combination is. These experiments are performed using $K \in \{2, 4, 8, 16, 32\}$, and on the 23 benchmarks seen in Table I. For each graph in the Benchmark set, we run each heuristic 10 times with different random seeds.

B. Experimental Results

The first performance profile in Fig. 3a uses the total edge cut as the performance indicator. As the figure shows, using a depth-first topological ordering for coarsening in dagP (first three Top-X-X-X variants in the figure) generally performs the best. In particular, when this coarsening and both of the Kernighan and Greedy Partitioning algorithms have been used, the variant obtains at most $1.3\times$ of the best result for about 90% of the instances.

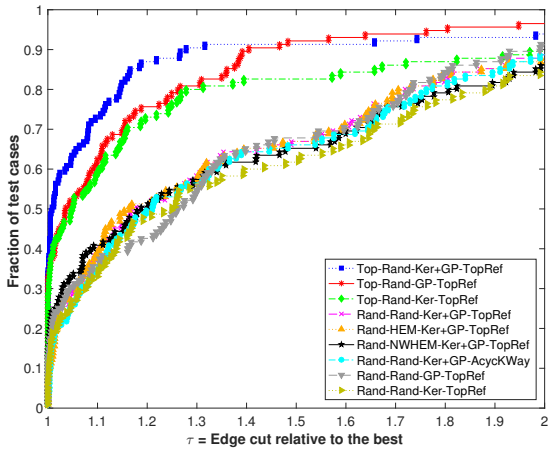
The second performance profile in Fig. 3b uses the partitioning time as the performance indicator. Here, the dagP variants with different options perform similarly. However, as the figure shows, Algorithm 4 (acyclic k-way refinement) degrades the performance of dagP. The Top-X-X-X variants, which produce partitions with small edge cut values, are slower when the Kernighan's algorithm is used during the initial partitioning phase. Although they are only slower at most $1.6\times$ than the fastest one for 70% of the benchmark instances, for most of the remaining 30% they are more than three times slower. This implies a performance bottleneck for this option combination. However, as shown in Fig. 3a, this combination also yields high-quality partitions.

The second set of performance profiles starting from Fig. 4a displays the comparison of three DAG partitioning methods. Since the partition is required to preserve an acyclic dependency structure, we limit our analysis to only algorithms that produce a DAG partitioning. Figure 4a uses the total edge cut as the performance indicator. The dagP variant with the best edge-cut performance, Top-Rand-Ker+GP-TopRef, is chosen for this set of experiments. As the figure shows, this variant is the best one for about 70% of the benchmark instances and is within $1.1\times$ of the best method in about 90% of them.

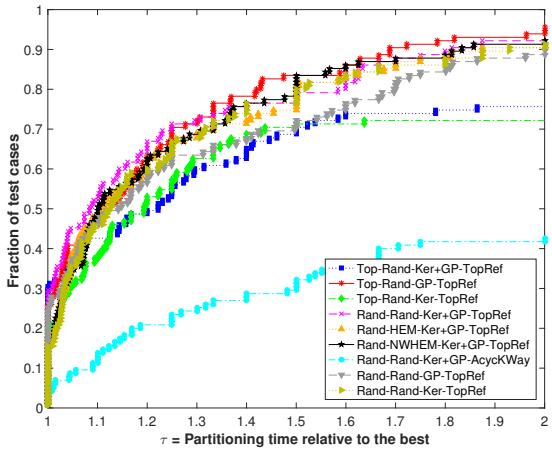
The performance profile in Fig. 4b uses the partitioning time as the indicator. We did not optimize Fauzia et al.'s algorithm, hence, it is excluded from the figure. Thanks to the coarsening phase of the multilevel approach, which reduces the number of vertices and subsequently the search space, the proposed method performs the best in almost 100% of the benchmark instances. On the other hand, Kernighan's algorithm is more than $20\times$ slower than dagP for about 50% of these instances.

Figure 5a uses the *total communication volume* as the performance indicator. The results are similar to that of the edge cut profile, since the two metrics are closely related. The difference is that for the total communication volume, multiple edges from a single vertex to vertices in another part is counted as one. Hence, unlike the edge cut, the number of edges is not important for this metric which models many real-life applications better. For this performance indicator, dagP performs the best in about 70% of benchmark instances and within the $1.2\times$ of the best in about 95% of them.

Figure 5b uses the critical path length or the latency of the



(a) Total edge cut

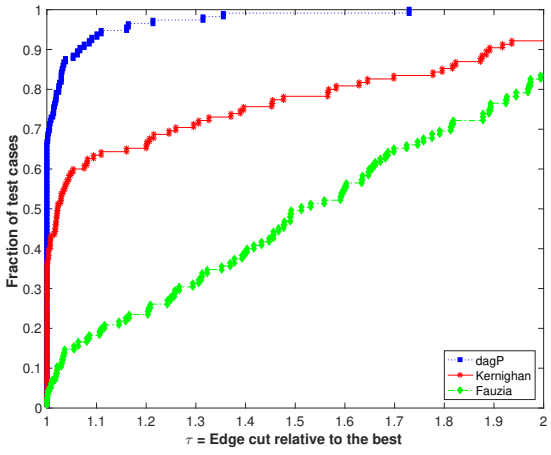


(b) Partitioning time

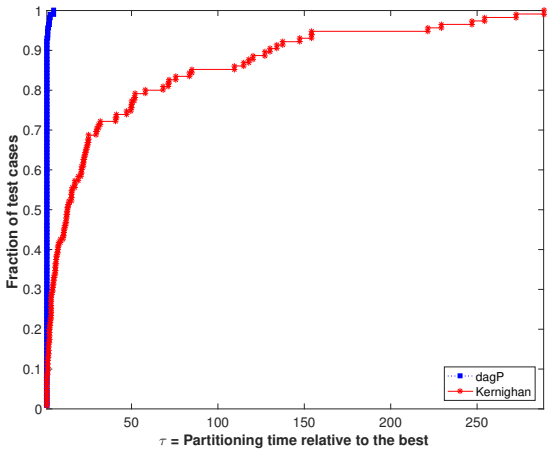
Figure 3: Performance profiles for dagP variants with different options: edge cut and partitioning time are used as two performance indicators. To generate the variants, Rand-Rand-Ker+GP-TopRef is used as the base variant and a few options are changed. We observed that AcycKWay does not improve the edge cut compared to the base variant, hence it is used in only one. On the other hand, as the figure shows, when a topological vertex traversal order is employed in the coarsening, the edge cut becomes smaller.

benchmark for performance profiles. The critical path length is defined as the length of the longest series of nodes and edges starting from an input node and ending at an output node. For this evaluation, we assumed that edges that cross partitions have a weight of 11 nanoseconds representing the latency to the L3 cache and edges within the same partition have a weight of 1 nanosecond representing the L1 cache. Each node also has a latency of 1 to model task executions. Even though dagP is not optimized for minimizing the critical path length, we present the comparison to get an idea of its partitions latency characteristics. The best latency is obtained by Kernighan's algorithm, Fauzia et al.'s algorithm, and dagP for 85%, 50%, and 45% of the instances. That being said, dagP produces partitions with critical path latencies that are at most $1.25\times$ worse than the best one for about 95% of the benchmark instances.

In summary, our approach is the first algorithm for the directed acyclic graph partitioning problem that exploits multilevel partitioning. Additionally, it finds partitions with better edge cut and communication volume than previous proposed algorithms in a fraction of the time.



(a) Edge cuts

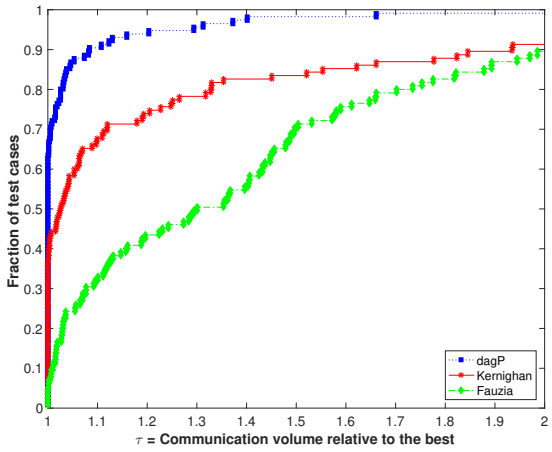


(b) Partitioning time

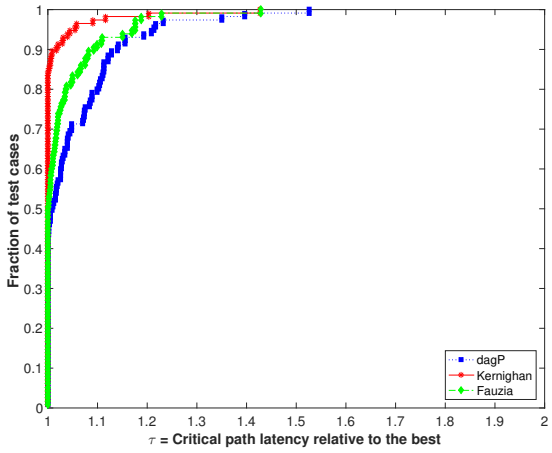
Figure 4: Performance profiles comparing three partitioning methods using edge cut and partitioning time as the performance metrics. For both metrics, dagP is better than Kernighan's and Fauzia et al.'s algorithms.

VI. Conclusion

We investigated the problem of partitioning directed acyclic graphs for task mapping in parallel systems. To the best of our knowledge, we proposed the first multilevel partitioning tool specialized for this purpose. Experiments



(a) Communication volume



(b) Critical path

Figure 5: Performance profiles comparing three partitioning methods based on total communication volume and critical path length. Critical path length is the length of the longest series of nodes and edges starting from an input node and ending at an output node. The cut edges that cross partitions have a weight of 11 nanoseconds to model L3 cache latency, and the intra-part edges have a weight of 1 nanosecond to model L1 cache. Each vertex also has a latency of 1 to model task executions.

on various graphs from linear algebra applications confirmed that dagP is much faster than similar tools and algorithms in the literature and can produce high quality partitions that will better exploit the parallelism by reducing the communication among the processors/nodes.

Future work includes, applying the proposed dagP method to real DAG execution and see the improvements on performance that can be achieved. This requires a scheduling step to be applied after dagP, which needs further investigations.

REFERENCES

[1] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[2] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "On characterizing the data access complexity of programs," *SIGPLAN Not.*, vol. 50, no. 1, pp. 567–580, Jan. 2015.

[3] 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.*, vol. 10, no. 4, pp. 53:1–53:29, Dec. 2013.

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

[5] ——, "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.

[6] 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: ACM, 2016, pp. 71–85.

[7] 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*, ser. IPDPSW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 312–321.

[8] 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*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 236–245.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[10] B. W. Kernighan, "Optimal sequential partitions of graphs," *J. ACM*, vol. 18, no. 1, pp. 34–40, Jan. 1971.

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

[12] 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.

[13] J. Nossack and E. Pesch, "A branch-and-bound algorithm for the acyclic partitioning problem," *Computers & Operations Research*, vol. 41, pp. 174–184, 2014.

[14] 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*, ser. GLSVLSI '03. New York, NY, USA: ACM, 2003, pp. 203–206.

[15] 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://bmi.osu.edu/ umit/software.htm, 1999.

[16] 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.

[17] F. Pellegrini, *SCOTCH 5.1 User's Guide*, Laboratoire Bordelais de Recherche en Informatique (LaBRI), 2008.

[18] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://web.cse.ohio-state.edu/ pouchet/software/polybench/*, 2012.

[19] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical programming*, vol. 91, no. 2, pp. 201–213, 2002.