Journal of Heuristics (2020) 26:771–799
https://doi.org/10.1007/s10732-020-09448-8

Evolutionary multi-level acyclic graph partitioning

Orlando Moreira[1] · Merten Popp[2] · Christian Schulz[3]

Received: 18 December 2018 / Revised: 7 July 2020 / Accepted: 9 July 2020 / Published online: 15 July 2020

Abstract

Directed graphs are widely used to model data flow and execution dependencies in streaming applications. This enables the utilization of graph partitioning algorithms for the problem of parallelizing execution on multiprocessor architectures under hardware resource constraints. However due to program memory restrictions in embedded multiprocessor systems, applications need to be divided into parts without cyclic dependencies. We found that this can be done by a subsequent second graph partitioning step with an additional acyclicity constraint. We have four main contributions. First, we show that this more constrained version of the graph partitioning problem is NP-complete and present linear time heuristics. We then integrate them into an existing *multi-level* graph partitioning framework to better handle large graphs. This achieves a 9% reduction of the edge cut compared to the previous single-level algorithm. Based on this, we engineer an evolutionary algorithm to *further* reduce the cut, achieving a 30% reduction on average compared to the state of the art. Finally, we integrate the partitioning heuristics into a graph compiler for an embedded multiprocessor architecture and show that this can reduce the amount of communication for a real-world imaging application and thereby accelerate it by an average of 11%. It is shown that the compiler can emit optimized code for vastly different hardware platforms using the heuristics. In addition, we demonstrate how a custom fitness function for the evolutionary algorithm can be used to optimize other objectives like load balancing if the communication volume is not predominantly important on a given hardware platform.

✉ Merten Popp
mail@merten-popp.de

Orlando Moreira
moreira.orlando@gmail.com

Christian Schulz
christian.schulz@univie.ac.at

1 GrAI Matter Labs, Eindhoven, The Netherlands

2 Braunschweig Institute of Technology, Brunswick, Germany

3 University of Vienna, Vienna, Austria

Keywords Graph partitioning · Evolutionary algorithm · Computer vision · Imaging · Embedded systems

1 Introduction

Imaging and computer vision are important elements and enabling technologies for a variety of applications ranging from consumer electronics over industrial automation to self-driving cars and have high demands for computational power. However, these applications often need to run on embedded devices with strong constraints on power and with a tight thermal budget (Wolf 2014). To cope with the high performance and low latency requirements of the application domain under these limitations, specialized hardware is needed (Wolf 2017). The downside of these specializations is that the resulting hardware platform is cumbersome to program. Therefore the task of implementing an application will not only require domain knowledge and precise understanding of the algorithm in question, but also demand profound knowledge of the hardware. Furthermore, it is a task that has to be repeated for every hardware platform the application has to run on. This is particularly not desirable for the development of imaging and vision applications for the mobile market due to the large number of different and constantly evolving platforms. One way to address this problem is by raising the level of abstraction and providing a domain-specific language for the developer that abstracts from hardware details. The opportunity here is that the developer can implement the application without targeting a specific hardware platform while the hardware vendor can implement a compiler for this language without targeting a specific use case. The challenge is that the algorithm must be expressed in a way that provides enough information to allow for an efficient execution on the target hardware platform without already prescribing a *specific* solution.

The context of this research is the development of specialized processors at Intel Corporation for advanced imaging and computer vision. In particular, our target platform is a heterogeneous multiprocessor architecture that is currently used in Intel processors. Several processors with vector units are available to exploit the abundance of data parallelism that typically exists in imaging algorithms. The architecture is designed for low power and has small local program and data memories. Since the hardware exists in different configurations with vastly different memory sizes and number of processors, we have developed a programming model in the form of a domain-specific meta-language. Developers can use the meta-language to implement imaging and vision applications by simply connecting kernels to form a directed graph. No in-depth knowledge of the hardware is required. We then provide a *graph compiler* that parses the graph and emits optimized code for the target hardware platform.

To cope with memory constraints, the compiler has to break the application into smaller blocks that are executed one after another. The quality of this partitioning has a strong impact on performance since it determines the amount of data that needs to be transferred to external memory. This will be the focus of this article. There are many existing heuristics for partitioning graphs into blocks of nodes of roughly equal size. However, our platform has the requirement that there must not be a cycle in the dependencies between the blocks because they have to be executed one after another.

The contributions of this work are (a) the identification of a new variation of the graph partitioning problem and proof of its NP-completeness, (b) design and implementation of local search heuristics for the acyclic graph partitioning problem, (c) a multi-level approach to better handle large graphs, (d) based on this, a coarse-grained distributed evolutionary algorithm to better escape local minima, (e) an objective function that improves load balancing on the multiprocessor architecture and (f) an evaluation on a large set of graphs and a real application.

Our focus is on solution quality, not algorithm running time, since these partitions are typically computed once before the application is compiled. We present all necessary background information on the application graph and hardware in Sect. 2 and then briefly introduce the notation in Sect. 3. We discuss related work in Sect. 4. The proof that the problem is NP-complete and hard to approximate is found in Sect. 5. The local search heuristics will be explained in Sect. 6. Our multi-level solution is described in Sect. 7. We extend it with an evolutionary algorithm that provides multi-level recombination and mutation operations, as well as a novel fitness function in Sect. 8. The evaluation is found in Sect. 9. We conclude in Sect. 10.

2 Background

In our programming model, applications are expressed as a directed stream graphs where nodes represent tasks that process the stream data and edges denote the direction of the dataflow. In this article, we address the problem of partitioning a *directed acyclic stream graph* to meet constraints of an embedded multiprocessor platform. The nodes of the graph are *kernels* (small, self-contained functions) annotated with code size while edges are annotated with the amount of data transferred during one execution of the application. These annotations are automatically made by our graph compiler. The programmer only writes the kernel code which he embeds into a wrapper written in our meta-language that defines how the kernel accesses data. The compiler uses this information and the size of the input images to calculate buffer sizes (program and data) as well as transfer costs.

The processors of the hardware platform have a private local data memory and a separate program memory. A direct memory access controller (DMA) is used to transfer data between the local memories and the external DDR memory of the system. Since the data memories only have a size in the order of hundreds of kilobytes they can only store a small portion of the image. Therefore the input image is divided into *tiles*.

The graph compiler will combine several kernels each into a program for one of the processors. The programs then process the tiles one after the other. However, this is only possible if the program memory size is sufficient to store all kernels. For the hardware platform under consideration it was found that this is not the case for more complex applications such as a Local Laplacian filter (Paris et al. 2011). Therefore a *gang scheduling* Feitelson and Rudolph (1992) approach is used where the kernels are divided into groups of kernels (referred to as gangs) that form smaller programs which do not violate memory constraints. Gangs are executed one after another on the hardware. After each execution, the kernels of the next gang are loaded and replace the

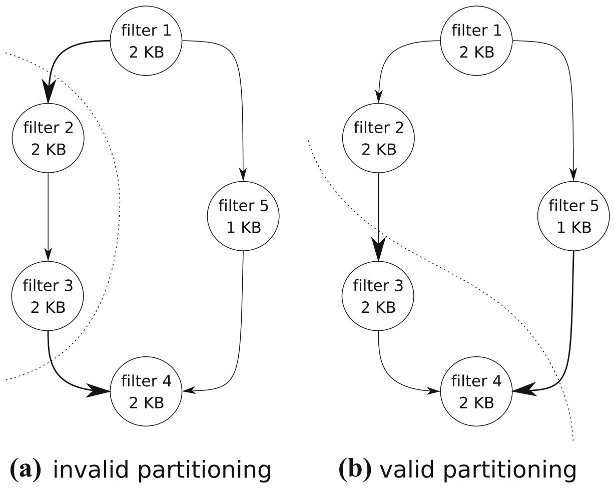

Fig. 1 Subfigure **a** shows an invalid partition with minimal edge cut, but a bidirectional connection between blocks and thus a cycle in the quotient graph. A valid partitioning with minimal edge cut is shown in **b**

current kernels in the program memory. This means that two kernels of different gangs are never resident in memory at the same time. Thus all intermediate data produced by the current gang but needed by a kernel in a later gang need to be transferred to external memory.

Data can only be consumed in the same gang where they were produced and in gangs that are scheduled at a later point in time. Therefore, a strict ordering of gangs is required where producers precede consumers. Such a partitioning is called *acyclic* because the quotient graph, which is created by contracting all nodes that are assigned to the same gang into a single node, does not contain a cycle. This does not hold for the partitioning in the left half of Fig. 1. The quotient graph is cyclic and there is no valid temporal order in which the two gangs can be executed on the platform. The right half of Fig. 1 shows a valid partitioning.

Memory transfers, especially to external memories, are expensive in terms of power and time. Thus it is crucially important how the assignment of kernels to gangs is done since it will affect the amount of data that needs to be transferred.

3 Preliminaries

We now introduce the mathematical notation used in this article, give the formal definition of the acyclic graph partitioning problem and show its relation to multiprocessor scheduling.

3.1 Basic concepts

Let $G = (V = \{0, \ldots, n-1\}, E, c, \omega)$ be a directed graph with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We are looking for *blocks* of nodes $V_1, \ldots, V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. We call a block V_i *underloaded* [*overloaded*] if $c(V_i) < L_{\max}$ [if $c(V_i) > L_{\max}$]. A *clustering* is also a partition of nodes, but k is usually not given in advance.

$N(v)$ gives the neighbors of v. If a node has a neighbor in a block different of its own block then both nodes are called *boundary nodes*. An abstract view of the partitioned graph is the so-called *quotient graph*, in which nodes represent blocks and edges are induced by connectivity between blocks. The *weighted* version of the quotient graph has node weights which are set to the weight of the corresponding block and edge weights equal to the weight of the edges that run between the respective blocks.

A matching $M \subseteq E$ is a set of edges that do not share any common nodes, i.e., the graph (V, M) has maximum degree one. *Contracting* an edge (u, v) means to replace the nodes u and v by a new node x connected to the former neighbors of u and v, as well as connecting nodes that have u and v as neighbors to x. We set $c(x) = c(u) + c(v)$ so the weight of a node in the new graph is the summed weight of the nodes it is representing in the original graph. If replacing edges of the form (u, w),(v, w) would generate two parallel edges (x, w), we insert a single edge with $\omega((x, w)) = \omega((u, w)) + \omega((v, w))$. *Uncontracting* an edge e undoes its contraction.

3.2 Problem definition

In our context, partitions have to satisfy two constraints: a balancing constraint and an acyclicity constraint. The *balancing constraint* demands that $\forall i \in \{1, \ldots, k\}$: $c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ for some imbalance parameter $\epsilon \geq 0$. The *acyclicity constraint* mandates that the quotient graph is acyclic. The objective is to minimize the total *edge cut* $\sum_{i,j} w(E_{ij})$ where $E_{ij} := \{(u, v) \in E : u \in V_i, v \in V_j\}$. The *directed graph partitioning problem with acyclic quotient graph (DGPAQ)* is then defined as finding a partition $\Pi := \{V_{1,} \ldots, V_k\}$ that satisfies both constraints while minimizing the objective function. In the *undirected* version of the problem the graph is undirected and no acyclicity constraint is given.

3.3 Multi-level approach

The multi-level approach to *undirected* graph partitioning consists of three main phases. In the *contraction* (coarsening) phase, the algorithm iteratively identifies matchings $M \subseteq E$ and contracts the edges in M. The result of the contraction is called a *level*.

Contraction should quickly reduce the size of the input graph and each computed level should reflect the global structure of the input network. Contraction is stopped when the graph is small enough to be directly partitioned. In the *refinement* phase, the matchings are iteratively uncontracted. After uncontracting a matching, a refinement algorithm moves nodes between blocks in order to improve the cut size or balance. The intuition behind this approach is that a good partition at one level will also be a good partition on the next finer level, so local search converges quickly. Moving a node on a coarse level hierarchy usually corresponds to the movement of a whole set of node movements of the finest level of the hierarchy. Intuitively, the multi-level scheme has a global view on the optimization problem on the coarse levels of the hierarchy and a very local view on the finest levels with respect to the original graph.

Local search algorithms find good solutions quickly but are more likely to get stuck in local optima. In contrast to local search algorithms, evolutionary algorithms tend to be better at searching the problem space globally. However, evolutionary algorithms lack the ability of fine tuning a solution, thus in *memetic algorithms* local search helps to improve the performance of an evolutionary algorithm (Kim et al. 2011).

3.4 Application in graph compilation

Graph partitioning is a sub-step in our graph compiler. It is used to adapt the program to different versions of our hardware platform such that is uses the hardware resources as much as possible but does not overcommit. To enable this, we map the hardware constraints of the platform to the constrains of the partitioning by setting node weights and $L_{\max}$ to appropriate values. Since the partitioning heuristics optimizes the edge cut, we can define edge weights in a way such that the heuristic optimizes a property of the partitioning that we deem beneficial for the scheduling and final performance of the application. If we set edge weights to the total amount of data transferred during one execution of the application, then reducing the edge cut will improve the memory bandwidth requirements of the application. The memory bandwidth is often the bottleneck, especially in embedded systems. A schedule that requires a large amount of transfers will neither yield a good throughput nor good energy efficiency.

Our compiler runs a first pass of the graph partitioning heuristic with the node weights set to the code size of the corresponding kernels. $L_{\max}$ is set to the size of the program memory. This first pass thus finds a good composition of kernels into programs with little interprocessor communication. The resulting quotient graph is then used in a second pass where node weights are set to 1 and $L_{\max}$ is set to the total number of processors. This finds scheduling gangs that minimize external memory transfers. The acyclicity constraint is of crucial importance in this second step. Note that in the first pass, the constraint can in principle be dropped. However, this yields programs with interdependencies that need to be scheduled in the same gang during the second pass. We found that this often leads to infeasible inputs for the second pass.

However, we also found that our graph partitioning heuristic, when only optimizing edge cut, occasionally makes a bad decision concerning the composition of gangs. Ideally, the programs in a gang all have equal execution times. If one program runs considerably longer than the other programs, the corresponding processors will be idle since the context switch is synchronized. Therefore, we try to alleviate this problem by using a fitness function in the evolutionary algorithm that considers the estimated execution times of the programs in a gang.

After partitioning, the compiler generates a schedule for each gang. Since partitioning is the focus of this article, we only give a brief outline. The scheduling heuristic is a list scheduler for a single appearance schedule (SAS). In a SAS, the code of a function is never duplicated, in particular, a kernel will never execute on more than one processor. The reason for using a SAS is the scarce program memory. List schedulers iterate over a fixed priority list of programs and start the execution if the required input data and hardware resources are available. We use a priority list sorted by the maximum length of the critical path which was calculated with estimated execution times. Since kernels perform mostly data-independent calculations, the execution time can be accurately predicted from the input size which is known from the stream graph and schedule.

4 Related work

There has been a vast amount of research on the *undirected* graph partitioning problem so that we refer the reader to Schloegel et al. (2003), Bichot and Siarry (2011), Buluç et al. (2014) for most of the material.

All general-purpose methods for this problem that are able to obtain good partitions for large real-world graphs are based on the multi-level principle. The basic idea can be traced back to multigrid solvers for systems of linear equations (Southwell 1935) but more recent practical methods are based on mostly graph theoretical aspects, in particular edge contraction and local search. For the *undirected* graph partitioning problem, there are many ways to create graph hierarchies such as matching-based schemes (Walshaw and Cross 2007; Karypis and Kumar 1998; Pellegrini 2012) or variations thereof Abou-Rjeili and Karypis (2006) and techniques similar to algebraic multigrid, e.g. Meyerhenke et al. (2006). However, as node contraction in a DAG can introduce cycles, these methods can *not* be directly applied to the DAG partitioning problem. Well-known software packages for the undirected graph partitioning problem that are based on this approach include Jostle Walshaw and Cross (2007), KaHIP Sanders and Schulz (2011), Metis Karypis and Kumar (1998) and Chevalier and Pellegrini (2008). However, none of these tools can partition directed graphs under the constraint that the quotient graph is a DAG. Very recently, Herrmann et al. (2017) presented the first multi-level partitioner for DAGs. The algorithm finds matchings such that the contracted graph remains acyclic and uses an algorithm comparable to Fiduccia–Mattheyses algorithm (Fiduccia and Mattheyses 1982) for refinement. Neither the code nor detailed results per instance are available at the moment.

Gang scheduling was originally introduced to efficiently schedule parallel programs with fine-grained interactions (Feitelson and Rudolph 1992). In recent work, this concept has been applied to schedule parallel applications on virtual machines in cloud computing (Stavrinides and Karatza 2016) and extended to include hard real-time tasks (Goossens and Richard 2016). In gang scheduling all tasks that exchange data with each other are assigned to the same gang, thus there is no communication between gangs. An important difference to our work is that the limited program memory of embedded platforms does not allow to assign all the kernels of an application to the same gang. Therefore, communication between gangs cannot be avoided, but is minimized by using graph partitioning methods.

Another application for graph partitioning algorithms that does have a constraint on cyclicity is the *temporal partitioning* in the context of reconfigurable hardware like FPGAs. These are processors with programmable logic blocks that can be reprogrammed and rewired by the user. In the case where the user wants to realize a circuit design that exceeds the physical capacities of the FPGA, the circuit netlist needs to be partitioned into partial configurations that will be realized and executed one after another. This is called temporal partitioning because any part of the circuit must be in a configuration no later than any of its outputs. It is thus comparable to the problem addressed in this article. The first algorithms for temporal partitioning worked on circuit netlists expressed as hypergraphs. Now, algorithms usually work on a behavioral level expressed as a regular directed graph. Proposed implementations include list scheduling heuristics (Cardoso and Neto 2000) or are based on graph-theoretic theorems like *max-flow min-cut* (Jiang and Wang 2007), with objective functions ranging from minimizing the communication cost incurred by the partitioning Cardoso and Neto (2000), Jiang and Wang (2007) to reducing the length of the critical path in a partition (Cardoso and Neto 2000; Kao 2015). Due to the different nature of the problem and different objectives, a direct comparison with these approaches is not possible.

The algorithm proposed in Chen and Zhou (2012) partitions a directed, acyclic dataflow graph under acyclicity constraints while minimizing buffer sizes. The authors propose an optimal algorithm with exponential complexity that becomes infeasible for larger graphs and a heuristic which iterates over perturbations of a topological order. The latter is comparable to our initial partitioning and our first refinement algorithm. We see in the evaluation that moving to a multi-level and evolutionary algorithm clearly outperforms this approach. Note that minimizing buffer sizes is not part of our objective.

5 Proof of NP-completeness

In this section, we show that the problem under consideration is NP-complete when restricted to the case $k = 2$ and $\epsilon = 0$, and also hard to approximate with a finite approximation factor for $k \geq 3$. A given solution for an instance of DGPAQ can be verified in linear time by constructing the quotient graph $\mathcal{Q}$, checking the balance constraint and checking $\mathcal{Q}$ for acyclicity. The last task can be done in linear time in the size of $\mathcal{Q}$ using Kahn's algorithm (Kahn 1962). We now reduce the subset sum problem to our problem. The proof is inspired by the reduction used in Picard and Queyranne (1980) which shows that the most balanced minimum cut problem is NP-complete.

Theorem 1 *The DGPAQ problem is NP-complete for the bi-partitioning case with* $\epsilon = 0$.

Proof We reduce the NP-complete Gary and Johnson (1979) subset sum problem to DGPAQ. The decision version of the subset problem is stated as follows: Given a set of integers $\{a_1, \ldots, a_n\}$, is there a non-empty subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$ holds? The construction of an equivalent instance of DGPAQ

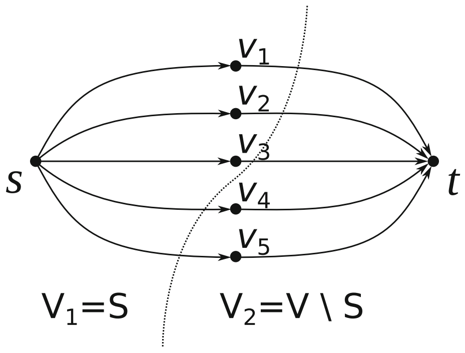

Fig. 2 Reduction: Subset sum problem is reduced to DGPAQ by creating a node for each a_i (the nodes in the center) and adding a source and sink node with edges as shown

is as follows: We construct a DAG $G = (V, E, c)$ with nodes $s, t \in V$ as well as a node $v_i \in V$ for each $i \in \{1, \dots, n\}$. Then we set $A := \sum_i 2a_i$ and define the node weights as $c(s), c(t) := A, c(v_i) := 2a_i$. Afterwards, we insert edges $(s, v_i)\ \forall i$ and $(v_i, t)\ \forall i$. The graph is a DAG -- an example topological ordering puts s first, t last and the remaining nodes at arbitrary positions in between. Figure 2 illustrates the construction. By definition, $L_{\max} = 3A/2$ for this instance of DGPAQ. Note that by construction A is divisible by 2. The construction can be done in polynomial time. Note that all balanced partitions $(S, V \backslash S)$ cut n edges, and due to the balance constraint s and t can never be in the same block. This ensures that there cannot be any edge (u, v) with $u \in V \backslash S$ and $v \in S$ and hence the quotient graph is acyclic. If the subset sum instance is a yes instance, then there is perfectly balanced bipartition and vice versa. □

The following theorem shows that it is not possible to find a finite factor approximation algorithm for our general problem where k is not a constant. The proof is a modification of the proof by Andreev and Räcke (2006) which shows this for the classical graph partitioning problem, i.e. no acyclicity constraint and for undirected inputs. Hence, we follow the proof of Andreev and Räcke (2006) closely with the difference being that the inputs that we construct are DAGs.

Theorem 2 *The directed graph partitioning problem with acyclic quotient graph has no polynomial time approximation algorithm with a finite approximation factor for* $\epsilon = 0, k \geq 3$ *unless* $P = NP$.

Proof The 3-Partition problem is defined as follows. Given $n = 3k$ integers $a_1, \dots, a_n$ and a threshold A such that $A/4 < a_i < A/2$ and $\sum_i a_i = kA$, decide whether the numbers can be partitioned into triples such that each triple adds up to A. This problem is *strongly* NP-complete (Gary and Johnson 1979), i.e. the problem remains NP-complete if all numbers a_i and A are polynomially bounded.

Now suppose we have an approximation algorithm for the directed graph partitioning problem with acyclic quotient graph for $\epsilon = 0$. We can use this algorithm to decide the 3-Partition problem with polynomially bounded numbers. To do so, we construct a graph G that contains n subgraphs. Subgraph i has a_i nodes. All weights are set to 1. We make each of the subgraphs a directed clique, i.e. all edges (u, v) with $u < v$ are inserted into the subgraph. By construction G is a DAG. This is the main difference to

Andreev and Räcke (2006) in which the subgraphs are undirected cliques. Also since all numbers are polynomially bounded, the construction takes polynomial time.

Now, if the 3-Partition instance can be solved, the k-DGPAQ problem in G can be solved without cutting any edge. Note that this solution also fulfills the acyclicity constraint. If the 3-Partition instance cannot be solved, then the optimum solution to the k-DGPAQ problem will cut at least one edge. An approximation algorithm with finite approximation factor has to differentiate between these two cases. Hence, it can solve the 3-Partition problem. □

6 Heuristic algorithms

In this section we present simple yet effective construction and local search heuristics as an initial solution for the partitioning problem. Our general approach is as follows: First create an initial solution based on a topological ordering of the input graph and then apply a local search strategy to improve the objective of the solution while maintaining both constraints. We start the section with the construction algorithm and then present different local search heuristics.

6.1 Construction algorithm

All of our local search heuristics start with an initial partitioning that fulfills both constraints, i.e. the quotient graph is acyclic and the balance constraint is satisfied. Our algorithm does this by computing a random topological ordering of the nodes using a modified version of Kahn's algorithm with randomized tie-breaking. More precisely, the algorithm initializes a list S with all nodes that have indegree zero and an empty list T. It then repeats the following steps until the list S is empty: Select a node from S uniformly at random and remove it from the list. Add the node to the tail of T. Remove all outgoing edges of the node. If this reduces the indegree of another node to zero, add it to S. When the algorithm terminates, the list T is a topological ordering of all nodes unless the graph has a cycle. Using list T, we can now derive initial solutions by dividing the graph into blocks of consecutive nodes w.r.t to the ordering. Due to the properties of the topological ordering there is no node in a block V_j that has an outgoing edge ending in a block V_i with $i < j$. Hence, the quotient graph of our solution is cycle-free. In addition, the blocks are chosen such that the balance constraint is fulfilled. There is obviously a large number of possible divisions. Our algorithm generates a balanced initial partitioning by dividing the ordering into blocks of size $\lfloor \frac{c(V)}{k} \rfloor$ or $\lceil \frac{c(V)}{k} \rceil$ uniformly at random. Since the construction algorithm is randomized, we run the heuristics ℓ times with different initial partitionings and pick the best solution afterwards.

6.2 Local search heuristics

Our local search heuristics take a given initial solution and move nodes between the blocks in order to decrease the edge cut. The reduction of the edge cut after a move

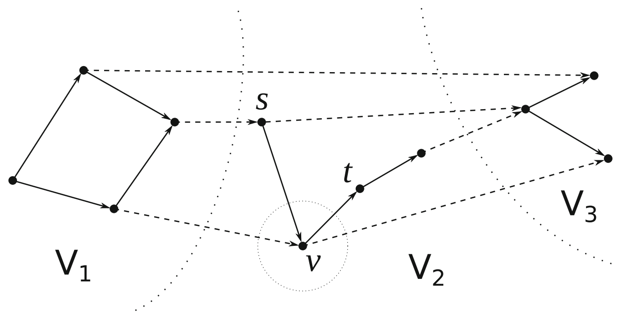

Fig. 3 A DAG divided into three blocks. Internal edges are solid, external edges are dashed. Node v is a node that has non-zero internal and external cost for both C_{in} and C_{out}. Because of $(s, v) \in E \Rightarrow C_{in}(v, 2) > 0$, the node cannot be moved to V_1. Because of $(v, t) \in E \Rightarrow C_{out}(v, 2) > 0$, the node cannot be moved to V_3 either

is called the *gain* of the move. To compute the gain when moving node v, we define two functions:

$$C_{in}(v, i) := \omega(\{(u, v) \in E : u \in V_i\})$$
$$C_{out}(v, i) := \omega(\{(v, u) \in E : u \in V_i\})$$

Roughly speaking, C_{in} is the combined weight for all edges that start in nodes of block V_i and end in v. Analogously, C_{out} is the combined weight of all edges that start in v and connect to nodes in the block V_i. If $v \in V_i$, these costs are the weights of *internal* edges. These edges will become external edges and increase the objective if we move v to a different block. If $v \in V_j$, $j \neq i$, then these costs are weights of *external* edges, which will become internal and thus reduce the edge cut if v is moved to V_i. Figure 3 shows an example of internal and external edges.

We have multiple local search heuristics that differ in the size of the local search neighborhood: *Simple Moves*, *Advanced Moves*, *Global Moves* as well as *FM moves*. We found that the heuristics can often yield better results with a different initial partitioning. In order to compare the different heuristics, we will give each heuristic the same time budget and will restart the heuristics for different initial partitionings until it is exhausted.

Simple Moves (SM) Simple moves start by picking a node v and moving it to a different block if this does not violate the constraints and improves the objective. Our simple move heuristic only considers to move a node $v \in V_i$ to adjacent blocks V_{i-1} and V_{i+1}. This is because there is a fast algorithm to check the *acyclicity constraint*. Assuming that the given solution is feasible with respect to both constraints, it is sufficient to check whether $C_{out}(v, i) = 0$ in the case that we want to move v to V_{i+1} and $C_{in}(v, i) = 0$ in the case that we want to move v to V_{i-1}. The gain of a node movement depends on the block and is calculated as:

$$\begin{cases} C_{in}(v, i-1) - C_{out}(v, i) & \text{when moving } v \text{ to } V_{i-1} \\ C_{out}(v, i+1) - C_{in}(v, i) & \text{when moving } v \text{ to } V_{i+1}. \end{cases} \quad (1)$$

A block is eligible if the move does not create a cycle and does not overload the block. In addition, the gain has to be positive or zero but the balance of the partitioning is improved. If there is such a block, we move v to it. In the case that both blocks are eligible for the move and have the same gain, the heuristic selects one uniformly at random.

We repeat the process for all nodes. Our heuristic stops if there is no node with positive gain or balance cannot be improved. Hence, our heuristic terminates when a local minimum is found with respect to the local search neighborhood defined above. Note that even though the edge cut is not *strictly* monotonically decreasing, the combination of edge cut and difference in block weight is. In one pass, the heuristic considers the in- and outgoing edges of all nodes. Thus, each edge is considered exactly twice to calculate the gain for all nodes and the complexity of the heuristic is $\mathcal{O}(m)$ per round.

Advanced Moves (AM) This algorithm increases the local search neighborhood of the Simple Moves algorithm by considering more target blocks for a move. For the node $v \in V_i$ under consideration, all incoming edges are checked to find the node $u \in V_A$ where A is maximal. Also all outgoing edges are checked to find the node $w \in V_B$ where B is minimal. Since the original partition was obtained from a topological ordering, $A \leq i \leq B$ must hold, otherwise there would be back edges in the ordering and thus it would not be a topological ordering. If $A = i = B$, then the node v has in- and outgoing edges in its own block and cannot be moved. If $A < i$, then the node can be moved to blocks preceding V_i up to and including V_A in the topological ordering without creating a cycle. This is because all incoming edges of the node will either be internal to block V_A or are forward edges starting from blocks preceding V_A. Therefore it is still a topological ordering. However, when the node is moved to a block preceding V_A, the edge starting in this block becomes a back edge and the ordering is not a topological ordering anymore. Similar, if $i < B$, the node can be moved to blocks succeeding V_i up to and including V_B. Thus moving the node to V_j with $j \in \{A, \dots, B\} \setminus \{i\}$ will preserve the topological ordering of blocks. This is a sufficient condition to ensure the acyclicity constraint and is not computationally expensive to check. However, since it is not a necessary condition, it might prevent the heuristic from testing some possible moves. The Global Moves heuristic does not have this limitation, but has a higher computational complexity.

The gain of the moves to all allowed V_j is computed with the cost functions described in the previous section as $C_{in}(v, j) - C_{out}(v, i) + C_{out}(v, j) - C_{in}(v, i)$. In each iteration, the move with the largest gain such that the constraints are maintained is selected. Tie-breaking and gains of zero are handled in the same way as in Simple Moves.

This heuristic considers each edge exactly twice in order to calculate the gain when moving the node to any other block. Afterwards, a block yielding maximal gain is selected, which can be done in time proportional to the degree of a node. Thus, the complexity of this heuristic is $\mathcal{O}(m)$.

Global Moves (GM) With this algorithm, we increase the local search neighborhood even further by considering all other blocks. Starting from the initial partition, the algorithm computes the adjacency lists of the quotient graph. Throughout the algorithm the quotient graph is kept up-to-date. When moving a node we update the adjacency information of the quotient graph and record whether a new edge has been created. If this is the case we check the quotient graph for acyclicity by using Kahn's algorithm and undo the last movement if it created a cycle.

The calculation of the gain values can be done in $\mathcal{O}(m)$ as for the other heuristics. For a node, the heuristic needs to check the acyclicity constraint for all considered moves/blocks in the worst case. Since Kahn's algorithm checks the quotient graph for acyclicity, the total complexity of this heuristic is $\mathcal{O}(m(m_Q + k))$ where m_Q is the number of edges in the quotient graph. If the quotient graph is sparse, i.e. m_Q is $\mathcal{O}(k)$, we get a complexity of $\mathcal{O}(km)$.

FM Moves (FM) This heuristic combines the quick check for acyclicity of the Advanced Moves heuristic with an adapted Fiduccia–Mattheyses algorithm Fiduccia and Mattheyses (1982) which gives the heuristic the ability to climb out of a local minimum. The initial partitioning is improved by exchanging nodes between a pair of blocks even if the gain is negative. The partition with the best objective that was seen during the pass will be returned. A pass starts with two blocks A and B, where A precedes B in the topological ordering of blocks. The algorithm will then calculate the gain for moving *enabled* boundary nodes to the other block. Using the same criterion to guarantee acyclicity as the Advanced Moves heuristic, we say that a boundary node is enabled if it is in A and does not have outgoing edges to nodes that precede B or it is in B and does not have incoming edges from nodes that follow A. The candidate moves, consisting of a gain and a node identifier, are inserted into a priority queue. The queue is a binary heap where the total order on the elements is implemented by comparing the gain of the moves and, if the gain is the same, a random number that is generated upon insertion.

In a loop that runs until the priority queue is depleted, the first move is extracted from the queue. If the selected move would overload the target block or is not enabled because it was disabled in a previous loop iteration, the heuristic continues with the next iteration. Otherwise, the move will be committed even if the gain is negative. The node is then locked, i.e. it cannot be moved again during this pass. This prevents thrashing and guarantees the termination of the algorithm. Unlike the Fiduccia–Mattheyses algorithm, a move in this scenario does not change the gain, it disables and enables other moves. For example, if a node w is moved from A to B, the heuristic will disable all nodes v in block B with $(w, v) \in E$ since they do not fulfill the condition for acyclicity anymore and moving any of them to A would introduce a back edge in the topological ordering of blocks. This does not necessarily mean that the quotient graph would become cyclic, however, assuring this would require a more expensive check like Kahn's algorithm. Note that the gain of the moves does not need to be re-calculated since w was locked and thus all nodes v will not be enabled again in this pass. On the other hand, moving w enables nodes in A if they are connected with an outgoing edge to w and if after the move they do not have other outgoing edges to blocks preceding B. The heuristic will calculate the gain for these nodes, enable and insert them into the priority queue. A move from B to A will enable and disable moves correspondingly. The loop will continue to move nodes between the blocks until the priority queue is depleted, which occurs when all nodes are either disabled or locked. Since the number of loop iterations is hard to predict due to the reinsertion of moves, it is limited to $2n/k$ which did not have a measurable impact on the quality of obtained partitionings. The best objective that was achieved in the pass is recorded. In the final step, the last moves are undone if required to reach the corresponding partitioning. This terminates the inner pass of the heuristic.

The outer pass of the heuristic will repeat the inner pass for randomly chosen pairs of blocks. At least one of these blocks has to be "active". Initially, all blocks are marked as "active". If and only if the inner pass results in movement of nodes, the two blocks will be marked as active for the next iteration. The heuristic stops if there are no more active blocks.

The overall time to compute gain values is $\mathcal{O}(m)$. We now analyze the running time for a pair of blocks. In the worst case, all nodes of both blocks are enabled in the beginning and initializing the priority queue with $2n/k$ nodes requires $O\left(\frac{n}{k}\right)$ time. Note that we cannot use a bucket priority queue, since the weights associated with the edges can be more or less arbitrarily distributed. Removing a node with the best gain from the queue takes $O\left(\log \frac{n}{k}\right)$ time. If a move is committed in an iteration, the heuristic needs to calculate the gain of adjacent nodes. However, the heuristic will never calculate the gain of a move twice during a pass. Thus the total complexity of the inner pass is $O\left(\frac{n}{k} \log \frac{n}{k}\right)$. Note that the inner pass needs to be performed for all pairs of blocks which yields overall time $O\left(m + m_{\mathcal{Q}} \frac{n}{k} \log \frac{n}{k}\right)$ per round of the algorithm, or $O\left(m + n \log \frac{n}{k}\right)$ if the quotient graph is sparse.

7 Multi-level acyclic graph partitioning

Multi-level techniques have been widely used in the field of graph partitioning for undirected graphs. We now transfer the techniques used in the KaFFPa multi-level algorithm (Sanders and Schulz 2011) to a new algorithm that is able to tackle the DAG partitioning problem. The challenge is to maintain the additional acyclicity constraint on each level. We implement algorithms that create coarser graphs without cycles and integrate the local search heuristics from the previous section that keep the quotient graph acyclic.

Figure 4 shows an overview of the algorithm. A multi-level graph partitioner has three phases: coarsening, initial partitioning and uncoarsening. We found that contracting clusterings can create coarse graphs that contain cycles and that this can make it impossible to find feasible solutions on the coarsest level of the hierarchy. Therefore, in *contrast* to classic multi-level algorithms, our algorithm starts by constructing a solution on the finest level of the hierarchy, meaning that the initial partitioning phase is moved before the coarsening phase. The larger size of the uncontracted graph is not a problem for our initial partitioning heuristic since it is a linear time algorithm. Then we continue to coarsen the graph until it has no contractable edges left. During coarsening, we transfer the solution from the finest level through the hierarchy and

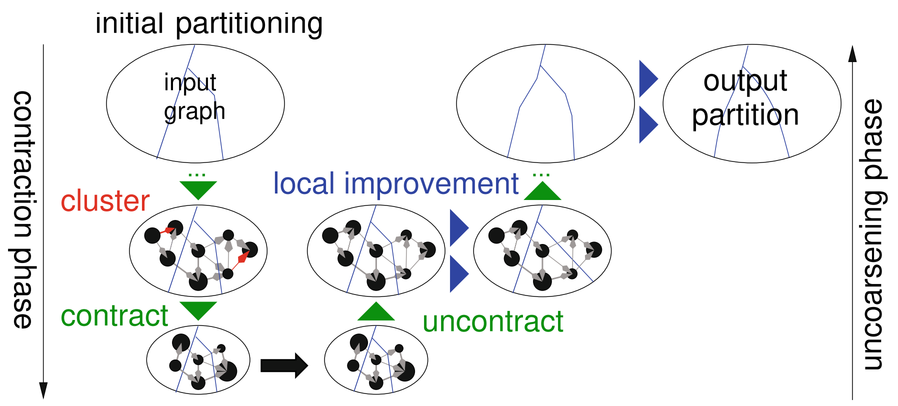

Fig. 4 Depiction of the phases in the multi-level approach to graph partitioning

use it as initial partition on the coarsest graph. As we will see later, since the partition on the finest level has been feasible, i.e. acyclic and balanced, so will be the partition that we transferred to the coarsest level. The coarser versions of the input graph may still contain cycles, but local search maintains feasibility on each level and hence, after uncoarsening is done, we obtain a feasible solution on the finest level.

The rest of the section is organized as follows. We begin with the description of the coarsening phase and then recap local search algorithms for the DAG partitioning problem that are now used within the multi-level approach.

7.1 Coarsening

Our coarsening algorithms is based on the contraction of clusterings. In our approach, we use a size-constrained label propagation algorithm (Meyerhenke et al. 2014) to compute a clustering of the graph. To compute a graph hierarchy, the clustering is contracted by replacing each cluster by a single node, and the process is repeated recursively until the graph is "small enough".

The size-constrained label propagation clustering algorithm is a very fast, near linear-time algorithm that locally optimizes the number of edges cut. Initially, each node is in its own cluster/block, i.e. the initial block ID of a node is set to its node ID. The algorithm then works in rounds. In each round, the nodes of the graph are traversed in a random order. When a node v is visited, it is *moved* to the block that has the strongest connection to v, i.e. it is moved to the cluster V_i that maximizes $\omega(\{(v,u) \mid u \in N(v) \cap V_i\})$ such that the target cluster size does not exceed a predefined bound U. Ties are broken randomly. We perform at most ℓ iterations of the algorithm instead, where ℓ is a tuning parameter. One round of the algorithm can be implemented to run in $O(n+m)$ time.

The computed clustering is contracted to obtain a coarser graph. *Contracting a clustering* works as follows: Each block of the clustering is contracted into a single node. The weight of the node is set to the sum of the weight of all nodes in the original block. There is an edge between two nodes u and v in the contracted graph if the two corresponding blocks in the clustering are adjacent to each other in G, i.e. block u and block v are connected by at least one edge. The weight of an edge (A, B) is set to the sum of the weight of edges that run between block A and block B of the clustering. Due to the way contraction is defined, a partition of the coarse graph corresponds to a partition of the finer graph with the same cut and balance. The process of computing a size-constrained clustering and contracting it is repeated recursively.

Recall that our algorithm starts with a partition on the finest level of the hierarchy. We construct this initial solution with the algorithm described in Sect. 6.1. It is then improved by a local search algorithm. Since the construction algorithm is randomized, we run the heuristics multiple times using different random seeds and pick the best solution. We call this algorithm *single-level algorithm*. We then set cut edges not to be eligible for the label propagation algorithm, i.e. cut edges of the partition will remain cut edges after contraction. That means edges that run between blocks of the given partition are not contracted. Thus the given partition can be used as a feasible initial partition of the coarsest graph. The partition on the coarsest level has the same balance and cut as the input partition. Additionally, it is also an acyclic partition of the coarsest graph. Performing coarsening by this method ensures non-decreasing partition quality, if the local search algorithm guarantees no worsening.

7.2 Uncoarsening

The refinement phase iteratively uncontracts the clusterings contracted during the first phase. Due to the way contraction is defined, a partitioning of the coarse level creates a partitioning of the finer graph with the same objective and balance, moreover, it *also* maintains the acyclicity constraint on the quotient graph. After a clustering is uncontracted, the local search refinement algorithms described in Sect. 6.2 move nodes between block boundaries in order to improve the objective while maintaining the balancing and acyclicity constraint.

After presenting our multi-level approach to handle large graphs and traverse the vast solution space more efficiently, we present an evolutionary algorithm on top of it in the next section that further improves the solution quality.

8 Evolutionary acyclic graph partitioning

Evolutionary algorithms start with a population of individuals, in our case partitions of the graph created by our multi-level algorithm using different random seeds. It then evolves the population into different populations over several rounds using recombination and mutation operations. In each round, the evolutionary algorithm uses a two-way tournament selection rule (Miller and Goldberg 1996) based on the fitness of the individuals of the population to select good individuals for recombination or mutation. Here, the fittest out of two distinct random individuals from the population is selected. We focus on a simple evolutionary scheme and generate one offspring per generation. After generation, we use an eviction rule to select a member of the population and replace it with the new offspring. In general, one has to take both, the fitness of an individual and the distance between individuals in the population, into consideration (Bäck 1996). We evict the solution that is *most similar* to the offspring among those individuals in the population that have a cut worse or equal to the cut of the offspring itself. The difference of two individuals is defined as the size of the symmetric difference between their sets of cut edges.

We now explain our multi-level recombine and mutation operators. Our recombine operator ensures that the partition quality, i.e. the edge cut, of the offspring is *at least as good as the best of both parents*. For our recombine operator, let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two individuals from the population that are used as input for our multi-level DAG partitioning algorithm. Let $\mathcal{E}$ be the set of edges that are cut edges, i.e. edges that run between two blocks, in either $\mathcal{P}_1$ *or* $\mathcal{P}_2$. All edges in $\mathcal{E}$ are blocked during the coarsening phase, i.e. they are *not* contracted during coarsening. In other words, these edges are not eligible for the clustering algorithm used during coarsening and therefore always run between clusters and not inside clusters. As before, the coarsening phase of the multi-level scheme stops when no contractable edge is left. Afterwards, we apply the better out of both input partitions w.r.t to the objective to the coarsest graph and use this as initial partitioning. We use random tie-breaking if both input individuals have the same objective value. This is possible since we did not contract any cut edge of $\mathcal{P}$. Again, due to the way coarsening is defined, this yields a feasible partition for the coarsest graph that fulfills both constraints (acyclicity and balance) if the input individuals fulfill those.

Note that due to the specialized coarsening phase and specialized initial partitioning, we obtain a high quality initial solution on a very coarse graph. Since our local search algorithms guarantee no worsening of the input partition and use random tie breaking, we can assure nondecreasing partition quality. Also *note why the combine operations work*: Local search algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few nodes. Due to the fact that our multi-level algorithms are randomized, a recombine operation performed twice using the same parents can yield a different offspring. Each time we perform a recombine operation, we choose one of the local search algorithms described in Sect. 6.2 uniformly at random.

Cross Recombine This operator recombines an individual of the population with a partition of the graph that can be from a different problem space, e.g. a k'-partition of the graph. While $\mathcal{P}_1$ is chosen using tournament selection as before, we create $\mathcal{P}_2$ in the following way. We choose k' uniformly at random in $[k/4, 4k]$ and ϵ' uniformly at random in $[\epsilon, 4\epsilon]$. We then create $\mathcal{P}_2$ (a k'-partition with a relaxed balance constraint) by using the multi-level approach. The intuition behind this is that larger imbalances reduce the cut of a partition and using a k'-partition instead of k may help us to discover cuts in the graph that otherwise are hard to discover. Hence, this yields good input partitions for our recombine operation.

Mutation We define two mutation operators. Both mutation operators use a random individual $\mathcal{P}_1$ from the current population. The first operator starts by creating a k-partition $\mathcal{P}_2$ using the multi-level scheme. It then performs a recombine operation as described above, but not using the better of both partitions on the coarsest level, but

$\mathcal{P}_2$. The second operator ensures nondecreasing quality. It basically recombines $\mathcal{P}_1$ with itself (by setting $\mathcal{P}_2 = \mathcal{P}_1$). In both cases, the resulting offspring is inserted into the population using the eviction strategy described above.

Fitness Function The fitness function is used to improve the load balancing of the target gangs. Recall that the execution of programs in a gang is synchronized. Therefore, a lower bound on the gang execution time is given by the longest execution time of a program in a gang. Pairing programs with short execution times with a single long-running program leads to a bad utilization of processors, since the processors assigned to the short-running programs are idle until all programs have finished. To avoid these situations, we use a fitness function that estimates the critical path length of the entire application by identifying the longest-running programs per gang and summing their execution times. This will result in gangs where long-running programs are paired with other long-running programs. More precisely, the input graph is annotated with execution times for each node that were obtained by profiling the corresponding kernels on our target hardware. The execution time of a program is calculated by accumulating the execution times for all firings of its contained kernels. The quality of a solution to the partitioning problem is then measured by the fitness function which is a linear combination of the obtained edge cut and the critical path length. Note, however, that the recombine and mutation operations still optimize for cuts. In the evaluation, we compare solutions that solely optimize edge cut and solutions that optimize the fitness function.

Miscellanea We follow the parallelization approach of Sanders and Schulz (2011): Each processing element (PE) has its own population and performs the same operations using different random seeds. The parallelization/communication protocol is similar to *randomized rumor spreading* (Doerr and Fouz 2011). We follow the description of Sanders and Schulz (2011) closely: A communication step is organized in rounds. In each round, a PE chooses a communication partner uniformly at random among those who did not yet receive P and sends the current best partition P of the local population. Afterwards, a PE checks if there are incoming individuals and if so inserts them into the local population using the eviction strategy described above. If P is improved, all PEs are again eligible.

9 Experimental evaluation

In this section we evaluate the performance of our algorithms. We start by presenting the systems we use for the evaluation. Then we compare the single-level heuristics on small instances with the optimal solutions and test their scalability with very large instances. Afterwards, we compare these results with the multi-level and the evolutionary algorithms. Finally, we test the impact of the proposed fitness function on the performance of a real imaging application.

9.1 Experimental protocol

9.1.1 System

We have implemented the algorithms described above using C++. All programs have been compiled using g++ 4.8.0 with full optimizations turned on (–O3 flag) and 32 bit index data types. We use two machines for our experiments: *Machine A* has two Intel Xeon E5-2670 Octa-Core processors running at 2.6 GHz with 64 GB of local memory. We use this machine in Sect. 9.3.1. *Machine B* is equipped with two Intel Xeon X5670 Hexa-Core processors (Westmere) running at a clock speed of 2.93 GHz. The machine has 128 GB main memory, 12 MB L3-Cache and 6×256 KB L2-Cache. We use this machine for the other tests. Henceforth, a PE is one core.

9.1.2 Methodology

We perform experiments on a wide range of graphs that have been used in recent literature as well as new instances. Depending on the aspect we look at, the instances vary, hence we describe them in the corresponding sections.

Generally, we mostly present two kinds of data: average values and plots that show the evolution of solution quality (*convergence plots*). In both cases we perform multiple repetitions. The number of repetitions is dependent on the test that we perform and mentioned in the corresponding sections. Average values over multiple instances are obtained as follows: For each instance (graph, k), we compute the geometric mean of the average edge cut for each instance. We now explain how we compute the convergence plots, starting with how they are computed for a single instance I: Whenever a PE creates a partition, it reports a pair (t, cut) where the timestamp t is the current elapsed time on the particular PE and *cut* refers to the cut of the partition that has been created. When performing multiple repetitions, we report average values $(\bar{t}, \text{avgcut})$ instead. After completion of the algorithm, we have P sequences of pairs (t, cut) which we now merge into one sequence. The merged sequence is sorted by the timestamp t. The resulting sequence is called T^I. Since we are interested in the evolution of the solution quality, we compute another sequence T^I_{min}. For each entry (in sorted order) in T^I we insert the entry $(t, \min_{t' \le t} \text{cut}(t'))$ into T^I_{min}. Here $\min_{t' \le t} \text{cut}(t')$ is the minimum cut that occurred until time t. N^I_{min} refers to the normalized sequence, i.e. each entry (t, cut) in T^I_{min} is replaced by (t_n, cut) where $t_n = t/t_I$ and t_I is the average time that the multi-level algorithm needs to compute a partition for the instance I.

To obtain average values over *multiple instances* we do the following: For each instance we label all entries in N^I_{min}, i.e. (t_n, cut) is replaced by (t_n, cut, I). We then merge all sequences N^I_{min} and sort by t_n. The resulting sequence is called S. The final sequence S_g presents *event based* geometric averages values. We start by computing the geometric mean cut value $\mathcal{G}$ using the first value of all N^I_{min} (over I). To obtain S_g, we sweep through S: For each entry (in sorted order) (t_n, c, I) in S we update $\mathcal{G}$, i.e. the cut value of I that took part in the computation of $\mathcal{G}$ is replaced by the new value c, and insert $(t_n, \mathcal{G})$ into S_g. Note that c can be only smaller or equal to the old cut value of I.

Lastly, we also look at *performance plots*. A curve in a performance plot for algorithm X is obtained as follows: For each instance, we calculate the ratio between the best cut obtained by any of the considered algorithms and the cut for algorithm X. These values are then sorted.

9.2 Local search heuristics

We will first evaluate the local search heuristics in isolation.

9.2.1 Comparison with optimal solutions

This section compares the results of our heuristics against the optimal solution obtained by a non-polynomial time algorithm that performs an exhaustive search. We create a set of random graphs that are close to instances from typical applications. Our generation algorithm works by consecutively adding new graph levels with a random number of nodes. Each of the new nodes is connected to a random number of nodes in previous levels. Because the application domain of this work is imaging, we use a small number of input and output nodes (between 1 and 3) which is typically the case for imaging and vision kernels (compare library of OpenVX vision functions Khronos Group 2017). Since the weight of nodes is representing the program size, we select a random value between the size of the smallest and the largest kernel in an implementation of the Local Laplacian filter for our target platform. The weight of edges is uniformly chosen between 1 and 100 to account for different sizes for intermediate buffers between the functions.

Because the following parameters have a major impact on the structure of the graph, we use two different values for each and generate 25 graphs for each of the eight resulting parameter combinations:

– The maximum size of a graph level is either set to a high value ($\sqrt{n}$) which results in a graph that can in extreme cases have $\sqrt{n}$ levels with about $\sqrt{n}$ nodes each, meaning that there is a high amount of data parallelism, and low values ($\sqrt[4]{n}$) such that the graph resembles more a long chain of nodes and thus represents the classical imaging pipeline with low data parallelism on kernel level.
– The maximum number of edges is either set to the lowest number that ensures that inner nodes have at least one incoming and one outgoing edge and that the graph is connected or to $\sqrt{n}$ per node such that the number of edges scales with the problem size. This reflects applications with few and many data dependencies between functions.
– The maximum distance in terms of node indices, over which new nodes are connected to preceding nodes in the graph, is either set to a low value that results in a graph where nodes only have incoming edges from the closest preceding levels or it is set to n which means that there is no restriction on where edges can start. The first case models application where data is short-lived and only needed for the next step in a pipeline while the second case represents scenarios with a long data lifetime.

Table 1 Each cell shows the averaged result of the heuristic for the current combination of block count k and imbalance ϵ

k	ϵ (%)	SM (%)	AM (%)	GM (%)	FM (%)
2	20	3.41	3.41	3.41	0.26
	30	11.94	11.91	11.90	0.33
	40	14.71	14.78	14.58	1.29
	50	23.32	23.36	23.04	1.21
4	20	1.89	1.27	1.33	0.74
	30	4.03	3.22	3.25	0.67
	40	5.09	3.65	3.69	0.44
	50	6.50	4.04	4.19	0.31

The value is the increase in cost compared to the optimal solution

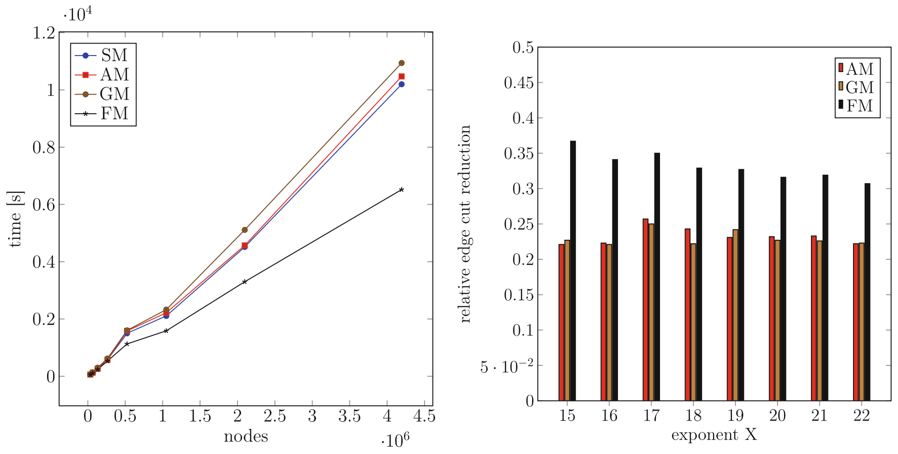

Fig. 5 Graph showing the execution time of each heuristic and the relative edge cut on directed random geometric graphs `rggX`

These 200 different problems instances were generated for problem sizes in the range of $n \in [10, \ldots, 20]$ nodes each. Table 1 shows the averaged approximation factor of the four heuristics when using a time budget of 10 ms. The results show a good approximation of the optimal solution. The quality of SM, AM and GM degrades with large ϵ since they can get trapped in a local minimum, FM moves on the other hand shows a close and consistent approximation. The heuristics generally perform better on graphs that were created with more, unconstrained edges, presumably because there are more legal moves available of which the heuristic can pick the best one. We also found that the running time for a single pass of the heuristics is consistent across the instances while it varies drastically between milliseconds and several days for the exhaustive search. This emphasizes the need for a heuristic.

9.2.2 Scalability

We now look at the scalability of our heuristics. We do this on *random geometric graphs* where nodes represent random points in the unit square and edges connect

Table 2 Basic properties of our instances

Graph	n	m	Graph	n	m
2mm0	36,500	62,200	atax	241,730	385,960
syr2k	111,000	180,900	symm	254,020	440,400
3mm0	111,900	214,600	fdtd-2d	256,479	436,580
doitgen	123,400	237,000	seidel-2d	261,520	490,960
durbin	126,246	250,993	trmm	294,570	571,200
jacobi-2d	157,808	282,240	heat-3d	308,480	491,520
gemver	159,480	259,440	lu	344,520	676,240
covariance	191,600	368,775	ludcmp	357,320	701,680
mvt	200,800	320,000	gesummv	376,000	500,500
jacobi-1d	239,202	398,000	syrk	594,480	975,240
trisolv	240,600	320,000	adi	596,695	1,059,590
gemm	1,026,800	1,684,200			

nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was chosen in order to ensure that the graph is almost connected. These graphs were taken from Bader et al. (2014) and were initially undirected. We convert them into DAGs by directing edges from smaller to larger node ids. The graph `rggX` has 2^X nodes. We vary $X \in [15, \ldots, 22]$. The allowed imbalance was set to 3% since this is one of the values used in Walshaw and Cross (2000). Figure 5 shows the averaged time required for 100 passes of each heuristic and the relative improvement in edge cut that was found for $k = 8$ by the more advanced heuristics in comparison to the Simple Moves heuristic. The figure shows a linear growth in running time of our heuristics respective to number of nodes. The worst case complexity of FM moves was shown to be superlinear since it had to be assumed that all nodes are boundary nodes, which is not the case here. In fact, that FM only considers boundary nodes appears to improve the execution time compared to the other heuristics. We conclude that our algorithms scale well to large problems.

In another small experiment, we evaluated the quality of the solution found by the initial partitioning only. As expected, the best edge cut is always a fair amount larger than the one found by the heuristics, for example 29% compared to SM for the largest random geometric graph.

9.3 Multi-level and evolutionary DAG partitioning

We now evaluate the performance of the evoluationary algorithm. We first look at the algorithm when the objective is edge cut and then study the impact on an imaging application.

Table 3 Average change of best cuts compared to the state of the art single-level algorithm

k	Multi-level (%)	Evolutionary (%)
2	− 11	− 53
4	− 9	− 26
8	− 8	− 22
16	− 10	− 24
32	− 11	− 30

9.3.1 Cut as objective

We will now compare the different proposed algorithms. We use the algorithms under consideration on a set of instances from the Polyhedral Benchmark suite (PolyBench) (Pouchet 2012) which have been kindly provided by Herrmann et al. (2017). Basic properties of the instances can be found in Table 2.

Our main objective in this section is the cut objective. In our experiments, we use the imbalance parameter $\epsilon = 3\%$ since this is one of the values used in literature benchmarks, e.g. Walshaw and Cross (2000). We use 16 PEs of machine A and 2 h of time per instance when we use the evolutionary algorithm. We parallelized repeated executions of multi- and single-level algorithms since they are embarrassingly parallel for different seeds and also gave 16 PEs and 2 h of time to each of the algorithms, i.e. all algorithms have the *same* amount of time to compute a solution. Each call of the multi-level and single-level algorithm uses one of our local search algorithms at random and a different random seed. We look at $k \in \{2, 4, 8, 16, 32\}$ and performed three repetitions per instance. Figure 6 shows convergence and performance plots. First of all, the performance plot in Fig. 6 indicates that our evolutionary algorithm finds significantly smaller cuts than the single- and multi-level scheme. Using the multi-level scheme instead of the single-level scheme already improves the result by 9% on average. This is expected since using the multi-level scheme introduces a more global view to the optimization problem and the multi-level algorithm starts from a partition created by the single-level algorithm (initialization algorithm + local search). In addition, the evolutionary algorithm always computes a better result than the single-level algorithm. This is true for the average values of the repeated runs as well as the achieved best cuts. The evolutionary algorithm computes average cuts that are 30% smaller than the ones computed by the single-level algorithm and best cuts that are 32% smaller. As anticipated, the evolutionary algorithm computes the best result in almost all cases. In three cases the best cut is equal to the multi-level, and in three other cases the result of the multi-level algorithm is better (at most 3%, e.g. for $k = 4$, `adi`). These results are due to the fact that we already use the multi-level algorithm to initialize the population of the evolutionary algorithm. In addition, after the initial population is built, the recombine and mutation operations can successfully improve the solutions in the population further and break out of local minima (see Fig. 6). Average cuts of the evolutionary algorithm are 22% smaller than the average cuts computed by the multi-level algorithm (and 25% in case of best cuts).

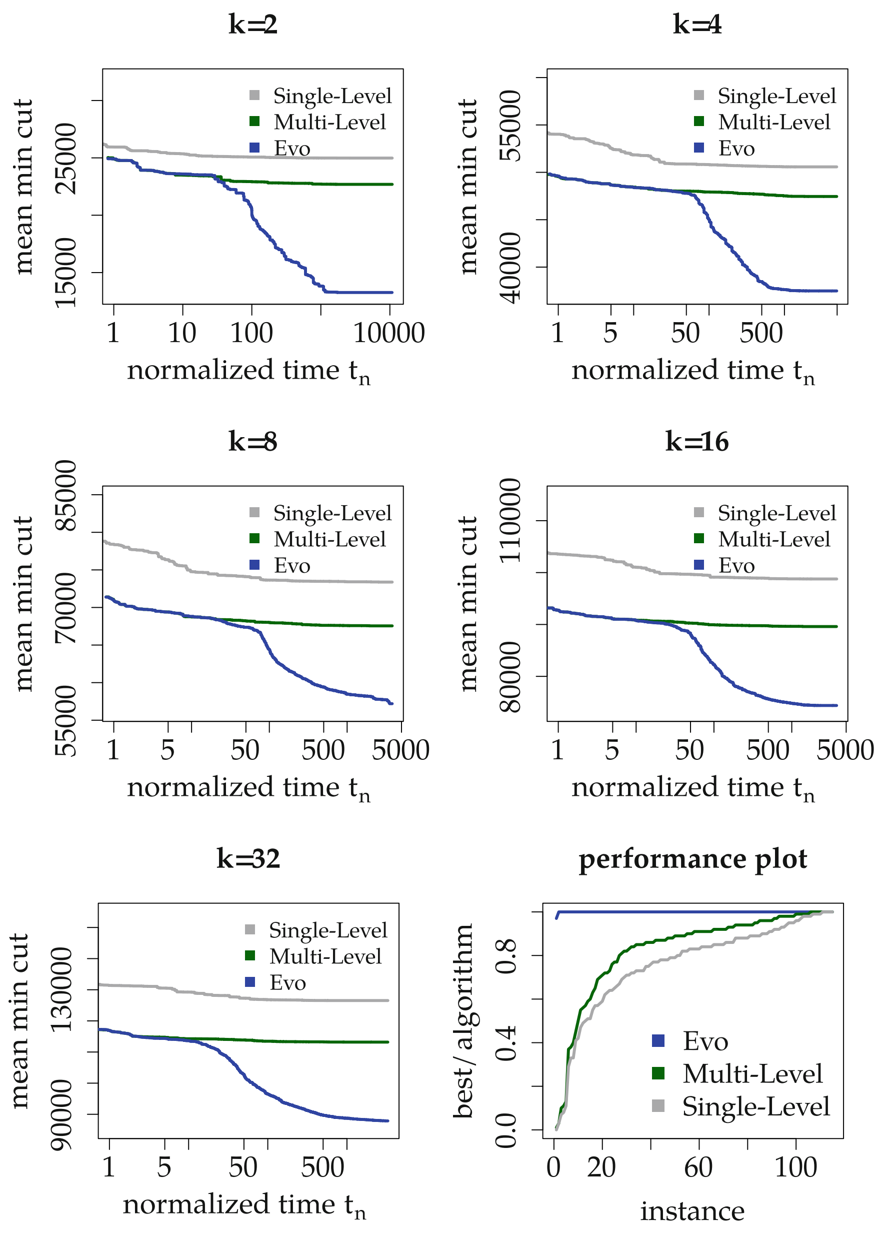

Fig. 6 Convergence plots for $k \in \{2, 4, 8, 16, 32\}$ and a performance plot

The largest improvement of the evolutionary algorithm over the single- and multi-level algorithm is a factor 39 (for $k = 2$, `3mm0`). Table 3 shows how improvements are distributed over different values of k. Interestingly, in contrast to evolutionary algorithms for the undirected graph partitioning problem, e.g. Sanders and Schulz (2011), improvements to the multi-level algorithm do not increase with increasing k. Instead, improvements more diversely spread over different values of k. We believe that the good performance of the evolutionary algorithm is due to a very fragmented search space that causes local search heuristics to easily get trapped in local minima, especially since local search algorithms maintain the feasibility on the acyclicity constraint. Due to mutation and recombine operations, our evolutionary algorithm escapes those more effectively than the multi- or single-level approach.

9.3.2 Impact on imaging application

We evaluate the impact of the improved partitioning heuristic on an advanced imaging algorithm, the *Local Laplacian filter*. The Local Laplacian filter is an edge-aware image processing filter. A detailed description of the algorithm and theoretical background is given in Paris et al. (2011). The algorithm uses concepts of *Gaussian pyramids* and *Laplacian pyramids* as well as a point-wise remapping function in order to enhance image details without creating artifacts. Nodes are annotated with program size and execution time estimate, edges with the corresponding data transfer size. The DAG has 489 nodes and 631 edges in total in our configuration. We use the single-level algorithm, the evolutionary algorithm and the evolutionary algorithm with the fitness function set to the one described in Sect. 8. The time budget given to each heuristic is 1 min. The makespans for each resulting schedule are obtained with a cycle-true compiled simulator of the hardware platform.

In the first experiment, we test how the graph partitioning heuristic can help the graph compiler to adapt the application to different versions of the hardware platform. We vary both the program and data memory size in steps of 64 KiB from 64 to 768 KiB. This should allow the graph compiler to fit more kernels into gangs. Figure 7 shows the makespan of the Local Laplacian filter averaged over 5 runs per data point when the compiler uses evolutionary algorithm in comparison to the single-level algorithm. The results in terms of edge cut as well as makespan are similar for the multi-level and the evolutionary algorithm optimizing for cuts, as the filter is fairly small. However, both of them outperform the single-level algorithm and improve the performance of the application. This is mainly because the reduction of the edge cut reduces the amount of data that needs to be transferred to external memory.

We can see that the evolutionary algorithm (solid line) *consistently* outperforms the single-level algorithm (dotted line) with a makespan that is on average 11% lower. The single-level algorithm shows high variability, with a makespan that is in one case only 0.3% larger that the evolutionary result but in three cases more than 20% larger. The evolutionary algorithm produces more stable results. The makespans appear to fall into three groups, beginning with around 6×10^7 cycles, then 4.5×10^7 cycles and finally 3×10^7 cycles. Further analysis revealed that the increasing memory sizes allows the partitioning heuristic to pack the kernels into fewer gangs. The two major drops in makespan coincide with the point where the partitioning heuristic first succeeds in

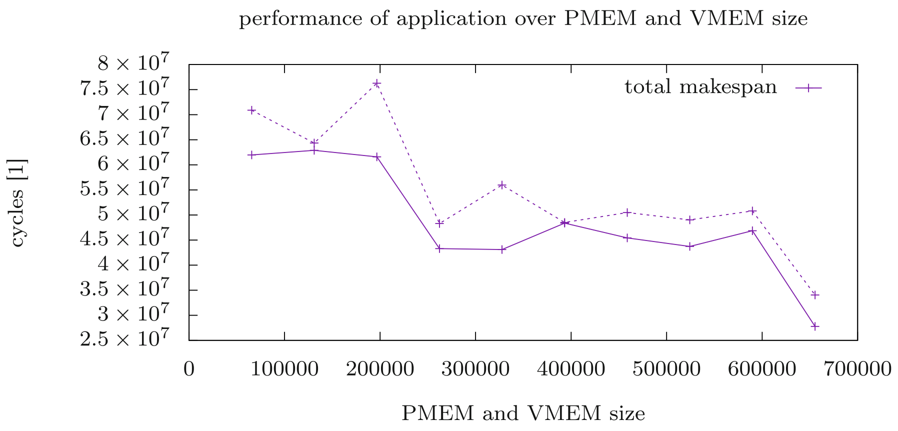

Fig. 7 Average makespan of the application over memory size

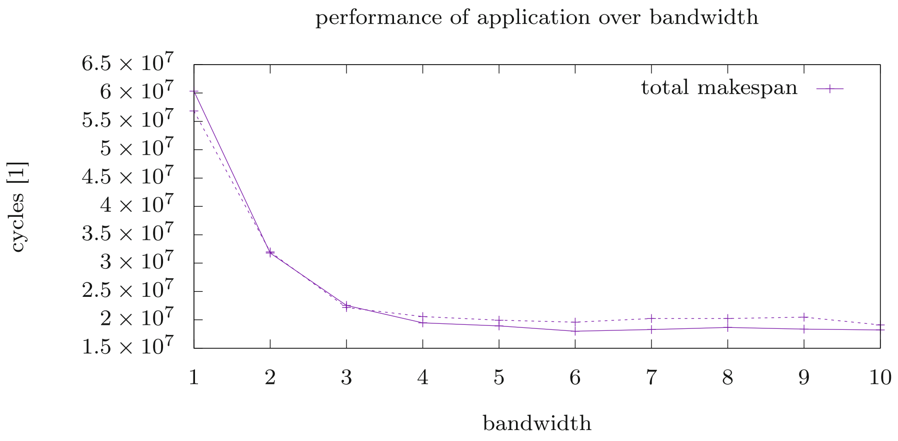

Fig. 8 Average makespan of the application over bandwidth

finding a $k' = k - 1$ partition where k is the best partition found for the previous data point.

In the second experiment, we vary the available bandwidth to external memory to assess the impact of edge cut on schedule makespan. In the following, a bandwidth of x refers to x times the bandwidth available on the real hardware. We compare the evolutionary algorithm optimizing for edge cut only with the evolutionary algorithm using our new fitness function that incorporates critical path length (Fig. 8).

The fitness function *increases* the makespan by 6% for the original bandwidth of the platform. We found that the gangs in this case are almost always memory-limited and thus reducing communication volume is predominantly important. With more bandwidth available, a break-even point follows where both approaches lead to more or less the same result. However, for bandwidths ranging from 4 to 10, including critical path length in the fitness function improves the makespan by an average of 7%. Hence, using the fitness function is a convenient way to fine-tune the heuristic for a given memory bandwidth. For hardware platforms with a scarce bandwidth, reducing the edge cut is the best. If more bandwidth is available, for example if more than one memory channel is available, one can change the factors of the linear combination to gradually reduce the impact of edge cut in favor of critical path length.

10 Conclusions

Directed graphs are widely used to model dataflow and execution dependencies in streaming applications which enables the utilization of graph partitioning algorithms for the problem of parallelizing computation for multiprocessor architectures. In this work, we designed, implemented and evaluated new heuristics that partition streaming application graphs under constraints resulting from resource restrictions in embedded hardware. In particular, we highlighted the appearance of an acyclicity constraint. We were able to show that this new version of the problem is NP-complete.

We designed and implemented heuristics that yield good approximations of the optimal solution for small problem instances. We then introduced a novel multi-level algorithm as well as the first evolutionary algorithm for the acyclic graph partitioning problem. By applying the multi-level approach, we improve the initial objective by 9%. Adding the evolutionary component yields a total reduction of 30%. Both is shown by extensive experiments over a large set of graphs. Applied to multiprocessor scheduling, this can improve the makespan by an average of 11% by limiting the communication with external memory.

Additionally, we formulated an objective function that includes load distribution and demonstrated how it can be used to tune an application for a different hardware platform. By adjusting the weights in the objective functions, the developer can easily trade communication reduction in favor of a more balanced load distribution or the other way around.

Our experiments indicate that the search space has many local minima. Hence, in future work, we want to experiment with relaxed constraints on coarser levels of the hierarchy. Other future directions of research include multi-level algorithms that directly optimize the newly introduced fitness function.

References

Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: Proceedings of 20th International Parallel and Distributed Processing Symposium (2006)

Andreev, K., Räcke, H.: Balanced graph partitioning. Theory Comput. Syst. **39**(6), 929–939 (2006)

Bäck, T.: Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms. Ph.D. Thesis (1996)

Bader, D.A., Meyerhenke, H., Sanders, P., Schulz, C., Kappes, A., Wagner, D.: Benchmarking for graph clustering and partitioning. In: Encyclopedia of Social Network Analysis and Mining (2014)

Bichot, C., Siarry, P. (eds.): Graph Partitioning. Wiley, Hoboken (2011)

Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. In: Algorithm Engineering—Selected Topics (2014). arXiv:1311.3144

Cardoso, J.M.P., Neto, H.C.: An enhanced static-list scheduling algorithm for temporal partitioning onto RPUs. In: VLSI: Systems on a Chip, pp. 485–496. Springer (2000)

Chen, Y., Zhou, H.: Buffer minimization in pipelined SDF scheduling on multi-core platforms. In: Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, pp. 127–132. IEEE (2012)
Chevalier, C., Pellegrini, F.: PT-Scotch. Parallel Comput. **34**(6–8), 318–331 (2008)
Doerr, B., Fouz, M.: Asymptotically optimal randomized rumor spreading. In: Proceedings of the 38th International Colloquium on Automata, Languages and Programming, Proceedings, Part II, LNCS, vol. 6756, pp. 502–513. Springer (2011)
Feitelson, D.G., Rudolph, L.: Gang scheduling performance benefits for fine-grain synchronization. J. Parallel Distrib. Comput. **16**(4), 306–318 (1992)
Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proceedings of the 19th Conference on Design Automation, pp. 175–181 (1982)
Gary, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness (1979)
Goossens, J., Richard, P.: Optimal Scheduling of Periodic Gang Tasks. Leibniz Trans. Embed. Syst. **3**(1), 04-1 (2016)
Herrmann, J., Kho, J., Uçar, B., Kaya, K., Çatalyürek, Ü.V.: Acyclic partitioning of large directed acyclic graphs. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 371–380. IEEE Press (2017)
Jiang, Y.C., Wang, J.F.: Temporal partitioning data flow graphs for dynamically reconfigurable computing. IEEE Trans. Very Large Scale Integr. VLSI Syst. **15**(12), 1351–1361 (2007)
Kahn, A.B.: Topological sorting of large networks. Commun. ACM **5**(11), 558–562 (1962)
Kao, C.C.: Performance-oriented partitioning for task scheduling of parallel reconfigurable architectures. IEEE Trans. Parallel Distrib. Syst. **26**(3), 858–867 (2015)
Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
Khronos Group: The OpenVX specification: vision functions. https://www.khronos.org/registry/OpenVX/specs/1.0/html/da/db6/group__group__vision__functions.html (2017)
Kim, J., Hwang, I., Kim, Y.H., Moon, B.R.: Genetic approaches for graph partitioning: a survey. In: Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO'11), pp. 473–480. ACM (2011)
Meyerhenke, H., Monien, B., Schamberger, S.: Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In: Proceedings of 20th International Parallel and Distributed Processing Symposium (2006)
Meyerhenke, H., Sanders, P., Schulz, C.: Partitioning complex networks via size-constrained clustering. In: Proceedings of the 13th International Symposium on Experimental Algorithms, LNCS. Springer (2014)
Miller, B.L., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Evol. Comput. **4**(2), 113–131 (1996)
Paris, S., Hasinoff, S.W., Kautz, J.: Local Laplacian filters: edge-aware image processing with a Laplacian pyramid. ACM Trans. Graph. **30**(4), 68 (2011)
Pellegrini, F.: Scotch and PT-scotch graph partitioning software: an overview. In: Combinatorial Scientific Computing, pp. 373–406 (2012)
Picard, J.C., Queyranne, M.: On the structure of all minimum cuts in a network and applications. Math. Program. Stud. **13**, 8–16 (1980)
Pouchet, L.: Polybench: the polyhedral benchmark suite. http://www.cs.ucla.edu/pouchet/software/polybench (2012)
Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Proceedings of the 19th European Symposium on Algorithms, LNCS, vol. 6942, pp. 469–480. Springer (2011)
Schloegel, K., Karypis, G., Kumar, V.: Graph partitioning for high performance scientific simulations. In: The Sourcebook of Parallel Computing, pp. 491–541 (2003)
Southwell, R.V.: Stress-calculation in frameworks by the method of "systematic relaxation of constraints". Proc. R. Soc. Lond. **151**(872), 56–95 (1935)
Stavrinides, G.L., Karatza, H.D.: Scheduling different types of applications in a SaaS Cloud. In: Proceedings of the 6th International Symposium on Business Modeling and Software Design (BMSD'16), pp. 144–151 (2016)
Walshaw, C., Cross, M.: Mesh partitioning: a multilevel balancing and refinement algorithm. SIAM J. Sci. Comput. **22**(1), 63–80 (2000)
Walshaw, C., Cross, M.: JOSTLE: parallel multilevel graph-partitioning software—an overview. In: Mesh Partitioning Techniques and Domain Decomposition Techniques, pp. 27–58 (2007)

Wolf, M.: Platforms and architectures for distributed smart cameras. In: Distributed Embedded Smart Cameras, pp. 3–23. Springer (2014)
Wolf, M.: Embedded computer vision. In: Handbook of Hardware/Software Codesign, pp. 1–14 (2017)