Evolutionary Multi-Level Acyclic Graph Partitioning

Orlando Moreira
Intel Corporation
Eindhoven, The Netherlands
orlando.moreira@intel.com

Merten Popp
Intel Corporation
Eindhoven, The Netherlands
merten.popp@intel.com

Christian Schulz
University of Vienna
Vienna, Austria
christian.schulz@univie.ac.at

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. This can be done by a subsequent second graph partitioning step with an additional acyclicity constraint.

We have two main contributions. First, we contribute a *multi-level* algorithm for the acyclic graph partitioning problem that 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.

We show that this can reduce the amount of communication for a real-world imaging application and thereby accelerate it by up to 5% on an embedded multiprocessor architecture. 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.

CCS CONCEPTS

• **Mathematics of computing** → **Evolutionary algorithms**; • **Software and its engineering** → *Multiprocessing / multiprogramming / multitasking*;

KEYWORDS

Graph Partitioning, Computer Vision and Imaging Applications

ACM Reference Format:
Orlando Moreira, Merten Popp, and Christian Schulz. 2018. Evolutionary Multi-Level Acyclic Graph Partitioning. In *GECCO '18: Genetic and Evolutionary Computation Conference, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205455.3205464

1 PRACTICAL MOTIVATION

Computer vision and imaging applications have high demands for computational power. However, these applications often need to run on embedded devices with severely limited compute resources and a tight thermal budget. This requires the use of specialized hardware and a programming model that allows the developers to fully utilize the compute resources.

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. To cope with memory constraints, it is necessary to break the application, which is given as a directed dataflow graph, 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. It is known that the problem is NP-complete [21] and that there is no constant factor approximation algorithm for general graphs [21]. Therefore heuristics are used in practice.

In this work, we develop a multi-level approach to enhance our previous results [21] since pure local search heuristics tend to get stuck in local optima, especially for larger graphs. We also found that our previous solution often produces unbalanced partitions and thus asymmetrical load on the multiprocessor because it does not take execution times into account.

We contribute (a) a new multi-level approach for the acyclic graph partitioning problem to better handle large graphs, (b) based on this, a coarse-grained distributed evolutionary algorithm to better escape local minima, (c) an objective function that improves load balancing on the multiprocessor architecture and (d) 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 Section 2 and then briefly introduce the notation and related work in Section 3. Our new multi-level solution is described in Section 4. We extend it with an evolutionary algorithm that provides multi-level recombination and mutation operations, as well as a novel fitness function in Section 5. The evaluation is found in Section 6. We conclude in Section 7.

2 BACKGROUND

The target applications can often be expressed as stream graphs where nodes represent tasks that process the stream data and edges denote the direction of the dataflow. Industry standards like

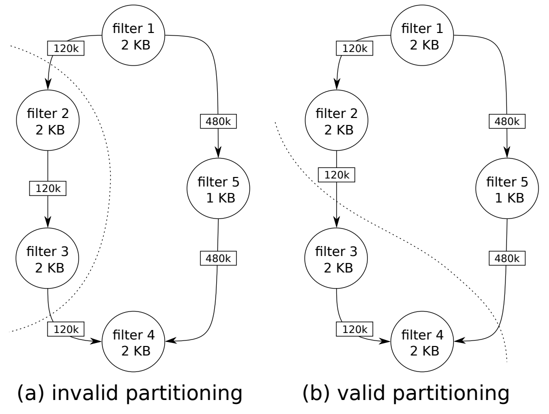

Figure 1: Partition (a) is invalid because of the bidirectional connection between gangs, (b) is valid because the quotient graph is not cyclic.

OpenVX [12] specify stream graphs as Directed Acyclic Graphs (DAG). In this work, we address the problem of mapping the nodes of a *directed acyclic stream graph* to the processors of an embedded multiprocessor. 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.

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.* This hardware is usually programmed by combining 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 [23]. Therefore a gang scheduling [9] approach is used where the kernels are divided into groups (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 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 Figure 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 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 paper, give the formal definition of the acyclic graph partitioning problem and show its relation to multiprocessor scheduling.

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.

Problem Definition. In our context, partitions have to satisfy two constraints: a balancing constraint and an acyclicity constraint. The *balancing constraint* demands that $\forall i \in \{1..k\} : c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil \frac{c(V)}{k} \rceil$ 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 $E_{ij} := \left\{(u, v) \in E : u \in V_i, v \in V_j\right\}$. 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.

Multi-level Approach. The state of the art multi-level approach to *undirected* graph partitioning consists of three main phases. In the *contraction* (coarsening) phase, the algorithm iteratively identifies a clustering and contracts the clusters. 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 clusterings are iteratively uncontracted. After uncontracting a clustering, a refinement algorithm moves nodes between blocks 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.

Relation to Scheduling.

Graph partitioning is a sub-step in our scheduling heuristic for the target hardware platform. We use a first pass of the graph partitioning heuristic with $L_{\max}$ set to the size of the program memory to find a good composition of kernels into programs with little interprocessor communication. The resulting quotient graph is then used in a second pass where $L_{\max}$ is set to the total number of processors in order to find scheduling gangs that minimize external memory transfers. In this second step the acyclicity constraint is crucially important. Note that in the first pass, this 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.

While the balancing constraint ensures that the size of the programs in a scheduling gang does not exceed the program memory size of the platform, 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 [22]. However, in our previous work, we found that our graph partitioning heuristic while optimizing edge cut occasionally makes a bad decision concerning the composition of programs. 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. In this work, we try to alleviate this problem with a fitness function in the evolutionary algorithm that improves load balancing using the estimated execution times of the programs in a gang.

After partitioning, a schedule is generated for each gang. Since partitioning is the focus of this paper, 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.

Related Work.

There has been a vast amount of research on the *undirected* graph partitioning problem so that we refer the reader to [3, 4, 28] 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 [29] 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 [17, 24, 32] or variations thereof [1] and techniques similar to algebraic multigrid, e.g. [18]. 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 [32], KaHIP [26], Metis [17] and Scotch [7]. However, none of these tools can partition directed graphs under the constraint that the quotient graph is a DAG. Very recently, Hermann et al. [13] 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 [10] for refinement. Neither the code nor detailed results per instance are available at the moment. As stated before, we integrate our previous work in [21] into a multi-level partitioner and extend it with a distributed evolutionary algorithm to better escape local minima. We further add an objective function to improve load balancing on a multiprocessor.

Gang scheduling was originally introduced to efficiently schedule parallel programs with fine-grained interactions [9]. In recent work, this concept has been applied to schedule parallel applications on virtual machines in cloud computing [30] and extended to include hard real-time tasks [11]. 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. 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 [5] or are based on graph-theoretic theorems like *max-flow min-cut* [14], with objective functions ranging from minimizing the communication cost incurred by the partitioning [5, 14] to reducing the length of the critical path in a partition [5, 16]. Due to the different nature of the problem and different objectives, a direct comparison with these approaches is not possible.

The algorithm proposed in [6] 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

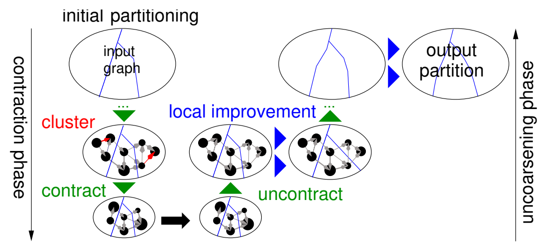

Figure 2: Depiction of the phases in the multi-level approach to graph partitioning.

evolutionary algorithm clearly outperforms this approach. Note that minimizing buffer sizes is not part of our objective.

4 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 [26] 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 local search algorithms that keep the quotient graph acyclic.

Figure 2 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 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 by reviewing the construction algorithm that we use, continue 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.

4.1 Initial Partitioning

Recall that our algorithm starts with initial partitioning on the finest level of the hierarchy. Our initial partitioning algorithm [21] creates an initial solution based on a topological ordering of the input graph and then applies a local search strategy to improve the objective of the solution while maintaining both constraints – balance and acyclicity.

More precisely, the initial partitioning algorithm computes a random topological ordering of nodes using a modified version of Kahn's algorithm [15] with randomized tie-breaking. The algorithm maintains a list S with all nodes that have indegree zero and an initially empty list T. It then repeatedly removes a random node n from both S and the graph, updates S by adding any further nodes with indegree zero and adds n to the tail of T. 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 the solution is cycle-free. In addition, the blocks are chosen such that the balance constraint is fulfilled. The initial solution 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.*

4.2 Coarsening

Our coarsening algorithms is based on the contraction of clusterings. In our approach, we use a size-constrained label propagation algorithm [19] 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. Hence, we 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.

4.3 Local Search

Recall that 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, local search refinement algorithms move nodes between block boundaries in order to improve the objective while maintaining the balancing and acyclicity constraint. We give an indepth description of the algorithms in [21] and shortly outline them here. All three algorithms identify *movable nodes* which can be moved to other blocks without violating any of the constraints. Based on a topological ordering, the first algorithm uses a sufficient condition which can be evaluated quickly to check the acyclicity constraint. Since the first heuristic can miss possible moves by solely relying upon a sufficient condition, the second heuristic maintains a quotient graph during all iterations and uses Kahn's algorithm [15] to check whether a move creates a cycle in it. The third heuristic combines the quick check for acyclicity of the first heuristic with an adapted Fiduccia-Mattheyses algorithm [10] which gives the heuristic the ability to climb out of a local minimum within a limited neighborhood.

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.

5 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 [20] 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 [2]. 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 Section 4.3 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. 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.

Miscellanea. We follow the parallelization approach of [27]: 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* [8]. We follow the description of [27] 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.

6 EXPERIMENTAL EVALUATION

System. We have implemented the algorithms described above within the KaHIP [26] framework 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 Octa-Core Intel Xeon E5-2670 processors running at 2.6 GHz with 64 GB of local memory. We use this machine in Section 6.1. *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 in Section 6.2. Henceforth, a PE is one core.

Methodology. 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. 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 ($\bar{t}$, 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 $(t, \min_{t' \leq t} \text{cut}(t'))$ into $T^I_{\min}$. Here $\min_{t' \leq t} \text{cut}(t')$ 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.

Instances. We use the algorithms under consideration on a set of instances from the Polyhedral Benchmark suite (PolyBench) [25] which have been kindly provided by Hermann et al. [13]. In addition, we use an instance of Moreira [21]. Basic properties of the instances can be found in Table 1.

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			

Table 1: Basic properties of the our instances.

6.1 Evolutionary DAG Partitioning with Cut as Objective

We will now compare the different proposed algorithms. 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. [31]. We use 16 PEs of machine A and two hours 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 two hours 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 3 shows convergence and performance plots. We provide supplementary material with detailed results per instance. To get a visual impression of the solution quality of the different algorithms, Figure 3 also presents a *performance plot* using all instances (graph, k). 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.

First of all, the performance plot in Figure 3 indicates that our evolutionary algorithm finds significantly smaller cuts than the

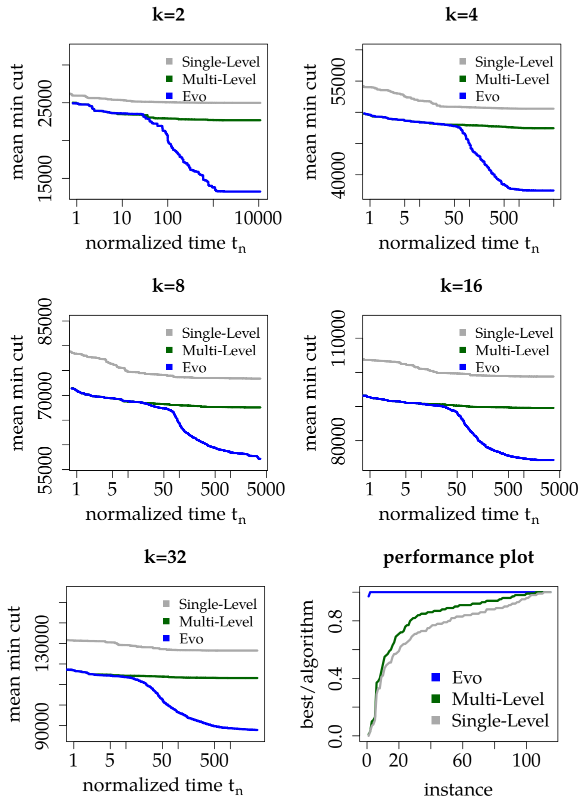

Figure 3: Convergence plots for $k \in \{2, 4, 8, 16, 32\}$ and a performance plot.

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 Figure 3). 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). The largest improvement of the evolutionary algorithm over the single- and multi-level algorithm is a factor 39 (for $k = 2$, `3mm0`). Table 2 shows how improvements are distributed

k	Multi-Level	Evolutionary
2	-11%	-53%
4	-9%	-26%
8	-8%	-22%
16	-10%	-24%
32	-11%	-30%

Table 2: Average change of best cuts compared to the state of the art single-level algorithm.

over different values of k. Interestingly, in contrast to evolutionary algorithms for the undirected graph partitioning problem, e.g. [27], 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.

6.2 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. The algorithm uses the concepts of *Gaussian pyramids* and *Laplacian pyramids* as well as a point-wise remapping function to enhance image details without creating artifacts. A detailed description of the algorithm and theoretical background is given in [23]. We model the dataflow of the filter as a DAG. 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 all algorithms (multi-level, evolutionary), the evolutionary with the fitness function set to the one described in Section 5. The time budget given to each heuristic is ten minutes. The makespans for each resulting schedule are obtained with a cycle-true compiled simulator of the hardware platform. 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. The relative improvement in makespan is compared to our previous heuristic in [21].

In this experiment, 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, the new approaches improve the makespan 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 memories. Improvements range from 1% to 5% depending on the available memory bandwidth with high improvements being seen for small memory bandwidths. For larger memory bandwidths, the improvement in makespan diminishes since the pure reduction of communication volume becomes less important.

Using our new fitness function that incorporates critical path length *increases* the makespan by 40% to 10% if the memory bandwidth is scarce (for bandwidths ranging from 1 to 3). 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, including critical path length in the fitness function improves the makespan by 3% to 33% for bandwidths ranging from 4 to 10. Hence, using the fitness function results in 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.

7 CONCLUSION

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 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 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 up to 5% 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

[1] A. Abou-Rjeili and G. Karypis. 2006. Multilevel Algorithms for Partitioning Power-Law Graphs. In *Proceedings of 20th International Parallel and Distributed Processing Symposium.*

[2] T. Bäck. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Ph.D. Dissertation.

[3] C. Bichot and P. Siarry (Eds.). 2011. *Graph Partitioning.* Wiley.

[4] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. 2014. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to app., *ArXiv:1311.3144*.

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

[6] Y. Chen and H. Zhou. 2012. Buffer minimization in pipelined SDF scheduling on multi-core platforms. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific.* IEEE, 127–132.

[7] C. Chevalier and F. Pellegrini. 2008. PT-Scotch. *Parallel Comput.* 34, 6-8 (2008), 318–331.

[8] B. Doerr and M. Fouz. 2011. Asymptotically Optimal Randomized Rumor Spreading. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming, Proceedings, Part II (LNCS),* Vol. 6756. Springer, 502–513.

[9] D. G. Feitelson and L. Rudolph. 1992. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing* 16, 4 (1992), 306–318.

[10] C. M. Fiduccia and R. M. Mattheyses. 1982. A Linear-Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Conference on Design Automation.* 175–181.

[11] J. Goossens and P. Richard. 2016. Optimal Scheduling of Periodic Gang Tasks. *Leibniz transactions on embedded systems* 3, 1 (2016), 04–1.

[12] Khronos Vision Working Group et al. 2017. The OpenVX™ Specification v1. 1. *Web:https://www.khronos.org/registry/OpenVX/specs/1.1/OpenVX_Specification_1_1.pdf* (2017).

[13] J. Herrmann, J. Kho, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. 2017. Acyclic Partitioning of Large Directed Acyclic Graphs. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing.* IEEE Press, 371–380.

[14] Y. C. Jiang and J. F. Wang. 2007. Temporal partitioning data flow graphs for dynamically reconfigurable computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 15, 12 (2007), 1351–1361.

[15] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562.

[16] C.-C. Kao. 2015. Performance-oriented partitioning for task scheduling of parallel reconfigurable architectures. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2015), 858–867.

[17] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392.

[18] H. Meyerhenke, B. Monien, and S. Schamberger. 2006. Accelerating Shape Optimizing Load Balancing for Parallel FEM Simulations by Algebraic Multigrid. In *Proceedings of 20th International Parallel and Distributed Processing Symposium.*

[19] H. Meyerhenke, P. Sanders, and C. Schulz. 2014. Partitioning Complex Networks via Size-constrained Clustering, In Proceedings of the 13th International Symposium on Experimental Algorithms. *preprint arXiv:1402.3281.*

[20] B. L. Miller and D. E. Goldberg. 1996. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Evolutionary Computation* 4, 2 (1996), 113–131.

[21] O. Moreira, M. Popp, and C. Schulz. 2017. Graph Partitioning with Acyclicity Constraints. *arXiv preprint arXiv:1704.00705* (2017).

[22] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. 2001. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 6, 2 (2001), 149–206.

[23] S. Paris, S. W. Hasinoff, and J. Kautz. 2011. Local Laplacian filters: edge-aware image processing with a Laplacian pyramid. *ACM Trans. Graph.* 30, 4 (2011), 68.

[24] François Pellegrini. 2012. Scotch and PT-scotch graph partitioning software: an overview. *Combinatorial Scientific Computing* (2012), 373–406.

[25] L. Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www.cs.ucla.edu/pouchet/software/polybench* (2012).

[26] P. Sanders and C. Schulz. 2011. Engineering Multilevel Graph Partitioning Algorithms. In *Proceedings of the 19th European Symp. on Algorithms (LNCS),* Vol. 6942. Springer, 469–480.

[27] P. Sanders and C. Schulz. 2012. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12).* 16–29.

[28] K. Schloegel, G. Karypis, and V. Kumar. 2003. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing.* 491–541.

[29] R. V. Southwell. 1935. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proc. of the Royal Society of London* 151, 872 (1935), 56–95.

[30] G. L. Stavrinides and H. D. Karatza. 2016. Scheduling Different Types of Applications in a SaaS Cloud. In *Proceedings of the 6th International Symposium on Business Modeling and Software Design (BMSD'16).* 144–151.

[31] C. Walshaw and M. Cross. 2000. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing* 22, 1 (2000), 63–80.

[32] C. Walshaw and M. Cross. 2007. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques.* 27–58.