Penalized Graph Partitioning for Static and Dynamic Load Balancing

Tim Kiefer, Dirk Habich$^{(\boxtimes)}$, and Wolfgang Lehner

Dresden Database Systems Group, Technische Universität Dresden, Dresden, Germany
{tim.kiefer,dirk.habich,wolfgang.lehner}@tu-dresden.de

Abstract. With ubiquitous parallel architectures, the importance of optimally distributed and thereby balanced work is unprecedented. To tackle this challenge, graph partitioning algorithms have been successfully applied in various application areas. However, there is a mismatch between solutions found by classic graph partitioning and the behavior of many real hardware systems. Graph partitioning assumes that individual vertex weights add up to partition weights (here, referred to as *linear graph partitioning*). This implies that performance scales linearly with the number of tasks. In reality, performance does usually not scale linearly with the amount of work due to contention on various resources. We address this mismatch with our novel *penalized graph partitioning* approach in this paper. Furthermore, we experimentally evaluate the applicability and scalability of our method.

1 Introduction

Modeling problems as graphs and balancing the load of corresponding distributed algorithms by means of graph partitioning has numerous applications in scientific computing [10,22,25]. Balanced min-cut (hyper)graph partitioning is appealing because it balances the load while at the same time minimizing communication costs. In recent years, graph partitioning was successfully used in other areas like data management as well [6,9,23]. Taking data management systems as an example, the possible applications for graph partitioning range from high-level database-as-a-service architectures [1,19] to low-level parallelism found in modern multi-socket-multi-core systems [17]. With more parallel architectures being used, the problem of optimally balancing work gains importance.

However, there is a mismatch between solutions found by classic graph partitioning and the behavior of many real hardware systems. Graph partitioning assumes that individual vertex weights add up to partition weights (here, referred to as *linear graph partitioning*). In the context of distributed systems, the assumption implies that performance scales linearly with the number of tasks. In reality however, performance does usually not scale linearly with the amount of work due to contention on hardware [3], operating system [18], or application resources [20]. We address this mismatch with *penalized graph partitioning*, a special case of non-linear graph partitioning, in this paper. The result is a load

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 146–158, 2016.
DOI: 10.1007/978-3-319-43659-3_11

balancing algorithm that shares the advantages of classic graph partitioning and that is at the same time considering the non-linear performance of real systems.

1.1 Penalized Performance Model

In this paper, we consider distributed systems where multiple (heterogeneous) tasks are executed concurrently on the various nodes. In the simplest case, loads induced by tasks are combined by summing them up to derive a node's global load. This method is referred to as the *linear model* as it models an ideal system where performance scales linearly with the amount of work that needs to be done. However, in practice, performance often depends on all kinds of workload parameters like request rates, request types, and the concurrent execution of requests. Contention on resources caused by concurrent execution may lead to performance that does not scale linearly with the amount of work. Therefore, we propose to use a *non-linear* model to combine the individual loads. To grasp the general behavior of complex systems, we assume a simplified *penalized* resource consumption model, which is a combination of the linear model and a (possibly non-linear) penalty function. Up to a certain load or degree of parallelism, the linear usage assumption often holds because the system is then underutilized and sufficient resources are available. However, when a certain load level is reached, contention occurs and the performance does not scale linearly beyond this load level. The penalty function is used to account for the contention.

While we acknowledge that modeling real systems is a challenging problem in itself, we assume here that the model, i.e., the penalty function, is given. Depending on the actual system, low-level and application-level experiments may be necessary to find a sufficiently accurate system model.

1.2 Motivating Example

To demonstrate the potential of penalized graph partitioning in presence of non-linear resources, we perform a synthetic partitioning experiment. To run the experiment, we generate a workload graph that contains 1000 heterogeneous tasks with weights following a Zipf distribution.[1] Each task in the workload graph is communicating with 0 to 10 other tasks (again Zipf distributed). To model a system, we use an exponential penalty function and assume that the underlying resource can execute 16 parallel tasks before the penalty grows with the square of the cardinality due to contention (Fig. 1a).

The workload in this experiment is partitioned into 32 balanced partitions using a standard graph partitioning library. Afterward, to estimate the actual load for each node, the penalty function is applied to each partition based on the partition cardinality (Fig. 1b). The resulting partition weights are compared to a second partitioning of the graph that was generated by our novel penalized graph partitioning algorithm (Fig. 1c).

[1] Comparable workloads can be found in actual systems, e.g., database-as-a-service systems [24].

The unmodified partitioning algorithm, which is unaware of the contention, tries to balance the load. The resulting relative weights show that the node with the highest partition weight receives 3.1 times the load of the node with the lowest partition weight. In contrast, the penalized partitioning algorithm leads to partition weights, and hence node utilizations, that are balanced within a tolerance of 3 %.

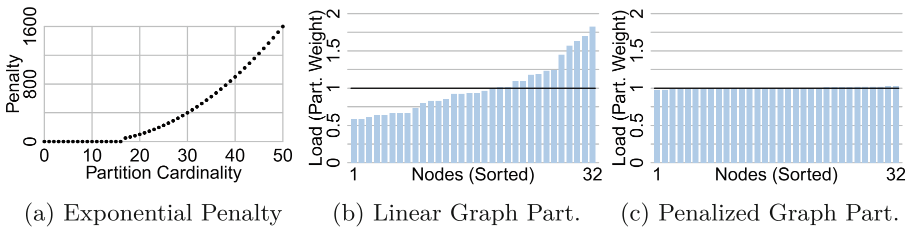


Fig. 1. Partitioning experiment (loads normalized to average)

1.3 Related Work

Graph partitioning has been a topic of interest in the scientific computing community at least since the late 1990s. Early works on the multilevel graph partitioning paradigm [13] led to many papers about variations and extensions of the balanced min-cut partitioning problem, e.g., about multi-constraint partitioning [14], incremental update strategies [11], or heterogeneous infrastructures [21]. A rather recent book and a survey provide excellent overviews of the results in the field [2,4]. To the best of our knowledge, we are the first to consider penalized, i.e., non-linear, graph partitioning.

In recent years, graph partitioning was successfully used in data management applications as well [6,9,23]. These applications will most likely benefit from penalized graph partitioning due to the complex and often heterogeneous tasks and the ever-present contention on bottleneck resources.

1.4 Contributions

Our main contribution in this paper is a load balancing algorithm based on penalized graph partitioning. In detail, we recap the basics of graph partitioning (Sect. 2) before we introduce our novel method to partition graphs with penalized partition weights, i.e., vertex weights that do not sum up linearly to partition weights (Sect. 3). Thereby, we also propose an extension to the penalized graph partitioning algorithm to deal with dynamic workloads. Our experimental evaluation shows the applicability and scalability of penalized graph partitioning in Sect. 4 before we conclude the paper in Sect. 5.

2 Graph Partitioning

Given an undirected, weighted graph, the balanced k-way min-cut graph partitioning problem (GPP) refers to finding a k-way partitioning of the graph such that the total edge cut is minimized and the partitions are balanced within a given tolerance. The following definitions are used to formalize the problem and to describe its solution heuristics in detail. In this paper, we limit ourselves to graphs with a single weight per vertex. Without restriction, penalized graph partitioning works with multiple vertex weights as well (e.g., based on [14]).

Let $G = (V, E, w_V, w_E)$ be an *undirected, weighted graph* with a set of vertices V, a set of edges E, and weight functions w_V and w_E. Vertex and edge weights are positive real numbers: $w_V\colon V \to \mathbb{R}_{>0}$ and $w_E\colon E \to \mathbb{R}_{>0}$. The weight functions are naturally extended to sets of vertices and edges:

$$w_V(V') := \sum_{v \in V'} w_V(v) \text{ for } V' \subseteq V \quad \text{and} \quad w_E(E') := \sum_{e \in E'} w_E(e) \text{ for } E' \subseteq E.$$

Let $\Pi = (V_1, \ldots, V_k)$ be a *partitioning* of V into k partitions $V_1, \ldots, V_k$ such that: $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for all $i \neq j$. Given a partitioning, an edge that connects partitions is called a *cut edge* and E_c is the set of all cut edges in a graph. The objective of the GPP is to minimize the *total cut* $w_E(E_c)$, i.e., the aggregated weight of all cut edges.

A *balance constraint* demands that all partitions have about equal weights. Let μ be the average partition weight: $\mu := w_V(V)/k$. For a balanced graph partitioning it must hold that $\forall i \in \{1, \ldots, k\}\colon w_V(V_i) \leq (1 + \epsilon) \cdot \mu$, where $\epsilon \in \mathbb{R}_{\geq 0}$ is a given imbalance parameter to specify a tolerable degree of imbalance (depending on the application).

2.1 Partitioning Algorithm

Partitioning a graph into k partitions of roughly equal size such that the total cut is minimized is NP-complete [12]. Heuristics, especially the multilevel partitioning framework [4,13], are used in practice to solve the problem.

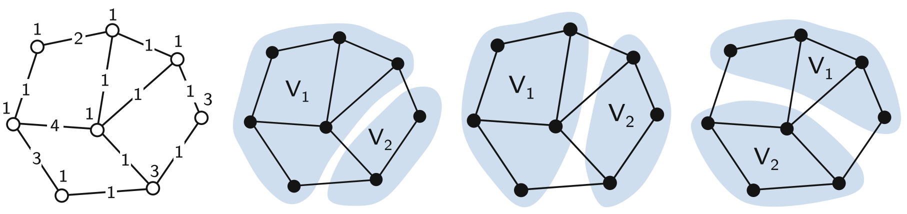
The multilevel graph partitioning framework consists of three phases: (1) *coarsening* the graph, (2) finding an *initial partitioning* of the coarse graph, and (3) *uncoarsening* the graph and projecting the coarse solution to the finer graphs. In the **coarsening phase**, a series of smaller graphs is derived from the input graph. Coarsening is commonly implemented by contracting a subset of vertices and replacing it with a single vertex. Parallel edges are replaced by a single edge with the accumulated weight of the parallel edges. Contracting vertices like this implies that a balanced partitioning on the coarse level represents a balanced partitioning on the fine level with the same total cut. Different strategies exist to select vertices to be contracted. Finding a matching is a tradeoff between using heavy edges (and hence reducing the final cut) and keeping uniform vertex weights (and hence improving partition balance). The coarsening ends when the coarsest graph is sufficiently small to be initially partitioned.

Different algorithms exist to find an **initial partitioning** [4]. Methods for the initial partitioning are either based on direct k-way partitioning or on recursive bisection. A simple but effective method to find an initial partitioning is greedy graph growing. A random start vertex is grown using breadth-first search, adding the vertex that increases the total cut the least in each step. The search is stopped as soon as half of the total vertex weight is assigned to the growing partition. Because the quality of the bisection strongly depends on the randomly selected start vertex, multiple iterations with different starts are used and the best solution is kept. The k-way extension of graph-growing starts with k random vertices and grows them in turns.

The initial partitioning is **uncoarsened** by repeatedly assigning previously contracted vertices to the same partition. Each extraction of vertices is followed by a **refinement step** to improve the total cut or the balance of the partitions. For instance, local vertex swapping is a refinement metaheuristic that can be parametrized with different strategies to select vertices to move [8,15,16].

3 Penalized Graph Partitioning

The idea of our penalized graph partitioning is to introduce a *penalized partition weight* and to modify the graph partitioning problem accordingly. We define the resulting problem as the *Penalized Graph Partitioning Problem* (P-GPP). Figure 2 shows an example graph with vertex and edge weights denoted in Fig. 2a. Solving the GPP leads to the partitioning with the total cut of 3 shown in Fig. 2b. When the cardinality of a partition is penalized linearly, the solution of the P-GPP having a total cut of 4 is shown in Fig. 2c. However, when the penalty of a partition grows with the square of the partition cardinality, the partitioning with the total cut of 4 shown in Fig. 2d is the solution to the P-GPP. The partitioning obviously depends on the performance model, i.e., the given penalty function.



(a) V/E Weights (b) No Penalty (c) Linear Penalty (d) Square Penalty

Fig. 2. Example of graph partitionings with different penalty functions

3.1 Prerequisites

Let $G = (V, E, w_V, w_E)$ be an undirected, weighted graph as in Sect. 2. Furthermore, let p be a positive, monotonically increasing penalty function that penalizes a partition weight based on the partition cardinality:

$$p\colon \mathbb{N} \to \mathbb{R}_{\geq 0} \text{ with } p(n_1) \leq p(n_2) \text{ for } n_1 \leq n_2.$$

The vertex weight function is extended to sets $V' \subseteq V$ such that it incorporates the penalty:

$$w_V(V') := \sum_{v \in V'} w_V(v) + p(|V'|).$$

The example partitioning in Fig. 2c uses a linear penalty function, i.e., $p(|V|) := |V|$. Accordingly, using the definition, the partition weights are

$$w_V(V_1) = \sum_{v \in V_1} w_V(v) + p(|V_1|) = 5 + 5 = 10 \text{ and}$$
$$w_V(V_2) = \sum_{v \in V_2} w_V(v) + p(|V_2|) = 7 + 3 = 10.$$

The example partitioning in Fig. 2d uses a square penalty function, i.e., $p(|V|) := |V|^2$. Accordingly, the partition weights are $w_V(V_1) = w_V(V_2) = 22$.

Adding penalties to partition weights invalidates some of the assumptions made in the GPP and its solution algorithms. Most fundamentally, the combined weight of two or more partitions is not equal to the weight of a partition containing all the vertices. Using the definition and two partitions V_1 and V_2:

$$w_V(V_1 \cup V_2) = w_V(V_1) + w_V(V_2) + p(|V_1 \cup V_2|) - p(|V_1|) - p(|V_2|).$$

For arbitrary penalty functions we must assume that $p(|V_1 \cup V_2|) \neq p(|V_1|) + p(|V_2|)$. It follows that in general $w_V(V_1 \cup V_2) \neq w_V(V_1) + w_V(V_2)$. Hence, the total weight of all vertices is in general not equal to the total weight of all partitions. We therefore introduce the following definitions of the two weights. Given a graph and a partitioning, the *total vertex weight* w_V is the penalized weight of all vertices, i.e.,

$$w_V := \sum_{v \in V} w_V(v) + p(|V|).$$

The *total partition weight* w_Π on the other hand is the sum of the weights of all partitions, i.e.,

$$w_\Pi := \sum_{i=1}^{k} w_V(V_i).$$

Consider the example partitioning in Fig. 2d; using the definition, $w_V = 12 + 64 = 76$ and $w_\Pi = 22 + 22 = 44$.

It follows that the total partition weight w_Π of the graph is not constant but depends on the partitioning, specifically the cardinalities of the partitions. This observation has implications in all steps of the graph partitioning algorithm, e.g., the balance constraint has to use the average total partition weight $\mu := w_\Pi / k$ instead of the average total vertex weight.

3.2 Penalized Graph Partitioning Algorithm (Static Case)

We propose modifications of the multilevel graph partitioning algorithm to solve the P-GPP. First, we describe two basic operations that need to reflect partition penalties. Then, we will detail the necessary modifications to the three building blocks of the multilevel graph partitioning framework.

During graph partitioning and refinement, it is often necessary to move a vertex between partitions or to merge partitions. For the sake of computational efficiency, the weights of the resulting partitions should be computed incrementally instead of from scratch.

Operation 1. *When a vertex v is moved from partition V_1 to partition V_2, the partition weights of the resulting partitions $V_1' := V_1 \setminus v$ and $V_2' := V_2 \cup v$ are as follows:*

$$\begin{aligned} w_V(V_1') &= w_V(V_1 \setminus v) = w_V(V_1) - w_V(v) - p(|V_1|) + p(|V_1| - 1) \text{ and} \\ w_V(V_2') &= w_V(V_2 \cup v) = w_V(V_2) + w_V(v) - p(|V_2|) + p(|V_2| + 1). \end{aligned}$$

Operation 2. *When two partitions V_1 and V_2 are combined, the partition weight of the resulting partition $V' := V_1 \cup V_2$ can be calculated as follows:*

$$w_V(V') = w_V(V_1) + w_V(V_2) + p(|V_1| + |V_2|) - p(|V_1|) - p(|V_2|).$$

To **coarsen** the graph, a matching of vertices has to be determined and vertices have to be contracted accordingly. The heuristics introduced in Sect. 2.1 can be used to coarsen a graph with penalized partition weights. However, the vertex weight of the contracted vertex has to correctly incorporate the penalty to ensure that a balanced partitioning of the coarse graph will lead to a balanced partitioning during the uncoarsening steps. Therefore, contracted vertices are treated like partitions themselves and the weight of a contracted vertex is calculated as in Operation 2.

We use a modified version of recursive bisection and greedy region growing to find an **initial k-way partitioning** of graphs with penalized partition weights. In the region growing algorithm, moving a vertex between partitions has to use Operation 1 to calculate the resulting partition weights. Moreover, the stop condition of the region growing algorithm has to be modified to account for the new balance constraint. In the original formulation, the algorithm stopped when the growing partition reached at least half of the total vertex weight. To achieve balanced partitions and because the total vertex weight is in general not equal to the total partition weight, the latter has to be used in the stop condition. Furthermore, since the total partition weight depends on the partitioning it repeatedly has to be recalculated after vertices have been moved, again using Operation 1.

The penalties have to be considered during the **uncoarsening and refinement** of the graph. Similar to the modifications of the region growing algorithm, the local vertex swapping method has to use Operation 1 whenever a vertex is moved between partitions. Furthermore, when vertex swapping is used to balance a partitioning, the modified balance constraint has to be used. This implies that stop conditions and checks use the total partition weight instead of the total vertex weight. Since the total partition weight depends on the partitioning, it has to be recalculated after a vertex has been moved (Operation 1).

3.3 Incrementally Updating the Partitioning (Dynamic Case)

With dynamic workloads, the partitioning needs to be periodically re-evaluated to ensure balanced partitions and an optimal total cut. Updating the partitioning after changes is a tradeoff between the quality of the new partitioning and the migration costs induced by implementing the new partitioning.

The problem of incrementally updating a partitioning is known as dynamic load balancing or repartitioning and is a well studied problem for the original graph partitioning problem [5,7]. In this paper, we adapt an existing hybrid update strategy for penalized graph partitioning and show in our experimental evaluation that it performs well in the presence of penalized partition weights. Whenever the graph changes such that the balance constraint is violated, balancing and refinement steps based on local vertex swapping try to move vertices such that the partitioning is balanced again. If no balanced partitioning can be found using the local search strategy, the graph is partitioned from scratch and the new partitioning is mapped to the previous partitioning such that the migration cost is minimized. To prevent the total cut in the graph from slowly deteriorating, a new partitioning is computed in the background after a certain number of local refinement operations (even when the partitioning is still balanced). The new partitioning replaces the current one only if the new total cut justifies the migration overhead.

4 Experimental Evaluation

METIS is a set of programs for graph partitioning and related tasks based on multilevel recursive bisection, multilevel k-way partitioning, and multi-constraint partitioning.[2] We modified METIS (v5.1) to support the penalized graph partitioning methods proposed in this paper (we denote the resulting tool PenMETIS). Our modifications are based on the serial version of METIS but can also be incorporated in the parallel version of METIS in the future.

4.1 Scalability Experiments

In this section, we evaluate the overhead that penalized partition weights introduce in the partitioning process. Furthermore, we investigate how penalized graph partitioning scales with the size of the graph and the number of partitions. We use a linear penalty function and example graphs from the Walshaw Benchmark [26] to analyze penalized graph partitioning. The corresponding Graph Partitioning Archive[3] contains 34 graphs from applications such as finite element computation, matrix computation, and VLSI design. The largest graph (`auto`) contains 448695 vertices and 3314611 edges and can be considered large in the context of workload graphs.

Penalized Partitioning Overhead. In this experiment, we investigate the overhead of penalized partition weights. Figure 3 shows the absolute partitioning times for all benchmark graphs using METIS and PenMETIS.[4] The figure shows that penalized partitioning introduces only a small overhead. More specifically, PenMETIS takes on average 28 % (42 ms) more time than METIS.

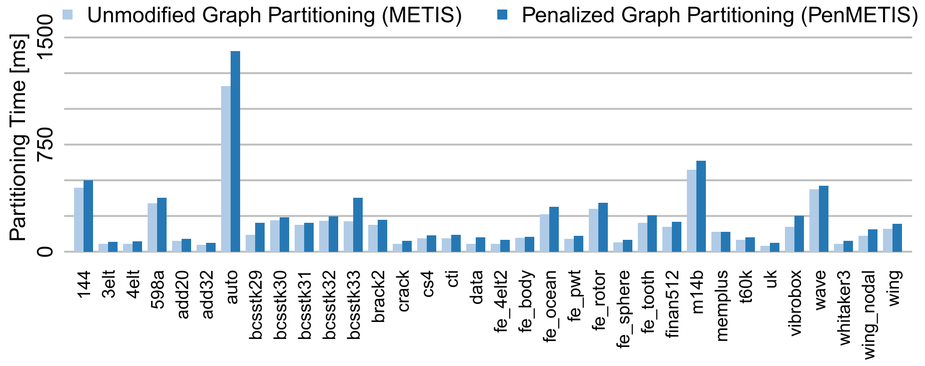


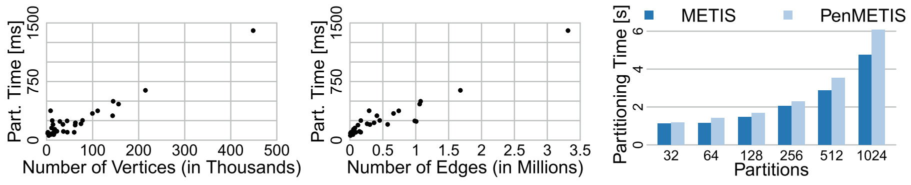
Fig. 3. Partitioning time comparison (64 partitions, 3 % imbalance) (Color figure online)

[2] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[3] http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/.

[4] We use a fairly moderate AMD Opteron (Istanbul) CPU running at 2.6GHz for this experiment. As mentioned before, METIS and PenMETIS run single-threaded.

Scalability with Graph Size. Figures 4a and b show the execution times of PenMETIS charted by the number of vertices and by the number of edges. The charts indicate that the graph partitioning algorithms scale linearly with both parameters.



(a) Scalability with $|V|$ (b) Scalability with $|E|$ (c) Scalability with k

Fig. 4. Execution times of PenMETIS depending on the number of vertices $|V|$, edges $|E|$ (64 partitions, 3 % imbalance), and partitions k (graph `auto`) (Color figure online)

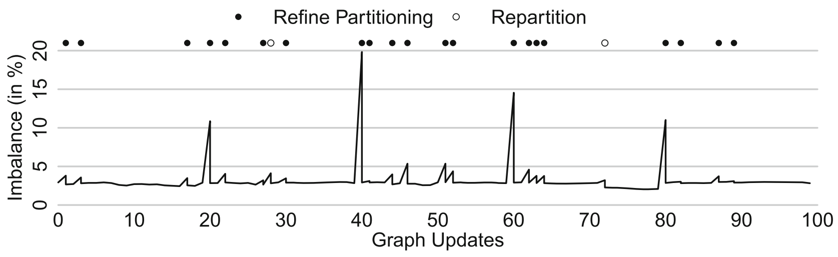
Scalability with Partition Count. In a second scalability experiment, we investigate how penalized graph partitioning scales with the number of partitions. In Fig. 4c, we show partitioning times for METIS and PenMETIS for the largest benchmark graph (`auto`) and various partition counts. Beyond 64 partitions, the partitioning time scales linearly with the number of partitions.

4.2 Incremental Update Experiment

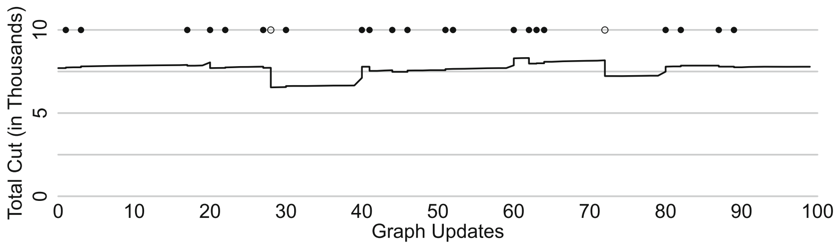
In this experiment, we evaluate the ability of PenMETIS to react to changes in the workload. We start our experiment with the previously introduced synthetic workload graph containing 1000 vertices and the same exponential penalty function (see Sect. 1.2). We additionally generate random edge weights (between 1 and 100) to get a more realistic evaluation of the total cut. The workload graph is initially partitioned into 32 partitions with an imbalance parameter of 3 %.

To simulate a changing workload, we define two workload graph modifications. A *minor change* is implemented by updating the vertex and edge weights of 1 % of all vertices and all edges (randomly selected). A *major change* is implemented by updating the vertex and edge weights of 10 % of all vertices and all edges. The complete experiment consists of 100 workload changes where one major change follows after every 19 minor changes. Figure 5 shows the results.

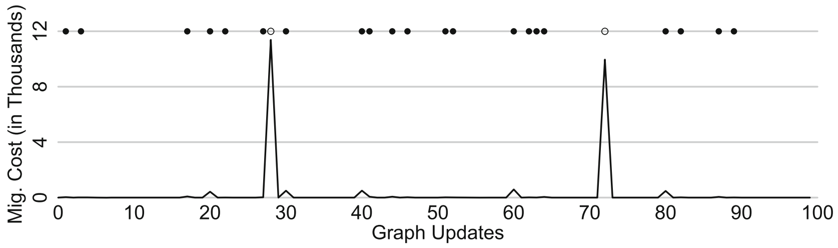
After each workload change, the current partitioning is evaluated against the new workload graph. The update mechanism is triggered when the balance constraint is violated. The update strategy first tries to regain a balanced partitioning by using local refinement strategies. A complete repartitioning is only triggered when the local refinement fails. In addition, the update strategy repartitions the workload graph in the background after every ten changes. However, the new partitioning is only implemented when it leads to a total cut that is



(a) Imbalance of the Graph (Before and After Refinements/Repartitionings)



(b) Total Cut of the Graph (Before and After Refinements/Repartitionings)



(c) Migration Cost for the Refinements/Repartitionings

Fig. 5. Incremental Update Experiment (32 Partitions, 3 % Imbalance)

more than 10 % better than the old cut. The evolution of the graph imbalance and the total cut are summarized in Figs. 5a and b. The results show that minor changes eventually and major changes always lead to violations of the balance constraint. However, in many cases (21 out of 23 in the experiment) the local refinement algorithm is able to regain a balanced partitioning. A complete repartitioning is triggered only in two cases, which in both cases leads to considerably better total cuts.

We report the sum of all vertex weights of vertices that are moved between partitions as the total migration cost for an update (Fig. 5c). The figure shows that partitioning the workload graph from scratch causes considerably higher migration costs than refining an existing partitioning.

5 Conclusion

In this paper, we presented penalized graph partitioning, a special case of non-linear graph partitioning. An experimental evaluation showed the applicability and scalability of penalized graph partitioning as a load balancing mechanism in the presence of non-linear performance due to contention on resources.

We believe that penalized graph partitioning is a versatile method that can be applied to many distributed systems. We showed that existing extensions for basic graph partitioning, specifically for dynamic repartitioning, can be applied to penalized graph partitioning as well. In the future, we will present results to show that the same holds for other extensions that deal with, e.g., multiple resources, heterogeneous infrastructures, or partial allocations. We will also show that the idea of penalized graph partitioning can be generalized to arbitrary non-linear performance models.

Acknowledgments. This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence *Center for Advancing Electronics Dresden* (Orchestration Path) and under the DFG project LE 1416/22-1.

References

1. Amazon. Amazon Relational Database Service (2015)
2. Bichot, C.-E., Siarry, P. (eds.): Graph Partitioning. Wiley, Hoboken (2011)
3. Blagodurov, S., Zhuravlev, S., Fedorova, A.: Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst. **28**(4), 8 (2010)
4. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent Advances in Graph Partitioning. preprint: Computing Research Repository (2013)
5. Catalyurek, U.V., et al.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: IPDPS (2007)
6. Curino, C., Jones, E.P.C., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. In: VLDB (2010)
7. Devine, K.D., Boman, E.G., Heaphy, R.T., Hendrickson, B.A.: New challenges in dynamic load balancing. Appl. Numer. Math. **52**, 133–152 (2005)
8. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: DAC (1982)
9. Golab, L., Hadjieleftheriou, M., Karloff, H., Saha, B.: Distributed data placement to minimize communication costs via graph partitioning. In: SSDBM (2014)
10. Hendrickson, B., Kolda, T.G.: Graph partitioning models for parallel computing. Parallel Comput. **26**(12), 1519–1534 (2000)
11. Hendrickson, B., Leland, R., Van Driessche, R.: Enhancing data locality by using terminal propagation. In: HICSS (1996)
12. Hyafil, L., Rivest, R.L.: Graph Partitioning and Constructing Optimal Decision Trees are Polynomial Complete Problems. Technical report, IRIA (1973)
13. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: SC (1995)
14. Karypis, G., Kumar, V.: Multilevel Algorithms for Multi-Constraint Graph Partitioning. Technical report, University of Minnesota (1998)
15. Karypis, G., Kumar, V.: Multilevel k-way partitioning scheme for irregular graphs. J. Parallel Distrib. Comput. **48**(1), 71–95 (1998)

16. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Syst. Tech. J. **49**(2), 291–307 (1970)
17. Kissinger, T., et al.: ERIS: a NUMA-aware in-memory storage engine for analytical workloads. In: ADMS (2014)
18. Li, C., Ding, C., Shen, K.: Quantifying the cost of context switch. In: ExpCS (2007)
19. Microsoft. Microsoft Windows Azure (2015)
20. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. In: VLDB (2010)
21. Pellegrini, F.: Static mapping by dual recursive bipartitioning of process and architecture graphs. In: SHPCC (1994)
22. Pothen, A.: Graph Partitioning Algorithms with Applications to Scientific Computing. Technical report, Old Dominion University (1997)
23. Quamar, A., Kumar, K.A., Deshpande, A.: SWORD: scalable workload-aware data placement for transactional workloads. In: EDBT (2013)
24. Schaffner, J., et al.: RTP: Robust tenant placement for elastic in-memory database clusters. In: SIGMOD (2013)
25. Schloegel, K., Karypis, G., Kumar, V.: Graph partitioning for dynamic. adaptive and multi-phase scientific simulations. In: CLUSTER (2001)
26. Soper, A.J., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. J. Global Optim. **29**(2), 225–241 (2004)