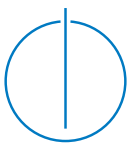
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Algorithm Engineering for Repartitioning Dynamic Graphs

Martin Schonger



DEPARTMENT OF INFORMATICS

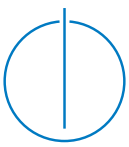
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Algorithm Engineering for Repartitioning Dynamic Graphs

Algorithm Engineering für die Repartitionierung Dynamischer Graphen

Author:	Martin Schonger
Supervisor:	Prof. Dr. Harald Räcke
Advisors:	Prof. Dr. Harald Räcke
	Univ.-Prof. Dr. Stefan Schmid
Submission Date:	15.09.2019



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2019 _____

Martin Schonger

Abstract

With an ever increasing demand for distributed computation comes the need to dynamically collocate frequently communicating parts of the workload. This corresponds to the Balanced RePartitioning (BRP) problem, which is algorithmically challenging as communication patterns are (a) rarely known in advance and (b) can change over time. Previous work has primarily considered theoretical aspects. We study the BRP from an algorithm engineering perspective. To this end, we revisit CREP, identify its bottleneck, and propose to combine it with efficient heuristics. These include the greedy approximation algorithm by Charikar as well as limiting the search for dense substructures to the 2-hop neighborhood of the latest communication request. Furthermore, we use decaying edge weights to model the temporal locality found in real network patterns. In an extensive evaluation we compare our algorithms on production-level and generated traffic traces. The proposed heuristics yield a significant improvement of both partition quality and run time.

Contents

1 Introduction 1
1.1 Notation 3
1.2 Definitions 3
1.3 Balanced RePartitioning (BRP) 4
2 Related Work 6
2.1 Network Design and Load Balancing 6
2.2 Dynamic Balanced Graph Partitioning 7
Stochastic Variant 8
Online Paging 8
Online (Re-)Matching 8
2.3 Static Balanced Graph Partitioning 9
Heuristics 9
Existing Implementations 9
2.4 Periodic Graph Partitioning 10
2.5 Verdict 10
3 Algorithms 11
3.1 CREP 11
Algorithm Definition 11
An Intuition Why CREP Makes Sense 12
On the Proof of CREP's Competitive Ratio 12
3.2 CREP Revisited 15
The α-LSP 15
Analysis of CREP with a Suboptimal Algorithm for α-LSP . 17
Finding a Target Partition. 17

3.3 CREP with Charikar (NAIVE) . . . 18
Charikar's Algorithm . . . 18
Analysis of the Modified Charikar . . . 18
Approximating the α-LSP with Charikar . . . 21
3.4 Charikar on Connected Components (CC) . . . 22
3.5 Charikar on the Hop-Neighborhood (HOP) . . . 23
3.6 Charikar and Greedy Exploration (GREEDY) . . . 24
3.7 Decaying Edge Weights . . . 24
4 Implementation Notes . . . 26
4.1 Modular BRP Framework . . . 26
4.2 Efficiently Maintaining Communication Components . . . 26
4.3 Charikar . . . 27
4.4 Zero-Overhead Aging . . . 27
5 Experimental Evaluation . . . 27
5.1 Dataset . . . 28
5.2 Default Settings and Baseline . . . 28
5.3 Derivation of HOP + Aging . . . 29
5.4 An Overview . . . 30
5.5 What Influence does α Have? . . . 33
5.6 The Effects of Aging . . . 35
5.7 1, 2, 3 or 4 Hops? . . . 37
5.8 Varying the Partition Size . . . 39
5.9 HOP on a Large Trace with Low Structure . . . 42
5.10 Results . . . 42
6 Conclusion and Future Work . . . 43

1 Introduction

In the advent of big data there is an ever increasing demand for distributed computation. This inevitably leads to higher communication volume and increased network traffic. As communication patterns in data centers are known to feature temporal and spacial structure, i.e. they are often sparse, skewed, bursty, and exhibit locality [11, 63], the overall communication cost may be reduced by dynamically collocating frequently interacting workloads onto the same server or rack.

This problem can be modeled in the setting of graph partitioning and as such was named *Balanced RePartitioning (BRP)* problem [5]. We are given a set of n vertices together with a sequence of pairwise communication requests and we have to maintain a partitioning of the vertices into l parts of size $k = n/l$ each. Inter-partition requests incur a cost of 1, while intra-partition communication does not incur any cost. The partitioning can be altered at any time a cost of $\alpha \geq 1$ per vertex move. Our main objective is the joint minimization of inter-partition communication and repartitioning costs. Intuitively, frequently communicating vertices should be assigned to the same partition. When the communication pattern shifts, a repartitioning may be required.

The static offline variant of the BRP, i.e. where no migration is allowed and the entire request sequence is known in advance, corresponds to the l-balanced graph partitioning problem. It is not only NP-complete but also inapproximable within any finite factor, unless P = NP [4]. This lets us believe that the BRP problem is algorithmically challenging as well.

Today, only very little is known about efficient algorithms for dynamic balanced graph repartitioning. Existing work mainly revolves around theoretical algorithms that aim to minimize the worst-case competitive ratio, but which come with the disadvantage of having a high run time. Indeed, these algorithms have only been analyzed theoretically and no implementation exists. The most relevant work on this matter is due to Avin *et al.* [5, 8]. They prove a lower bound of k on the competitive ratio for any deterministic algorithm with augmentation $\delta < l$. Second, they propose the *CREP (Component-based REPartitioning)* algorithm, which is $\mathcal{O}((1 + 1/\epsilon) \cdot k \cdot \log k)$-competitive, $\epsilon \geq 1/k$. On a high level, this algorithm dynamically groups frequently interacting vertices into communication components and keeps all vertices of a single component in the same partition.

While CREP exhibits desirable theoretical properties, we identify a serious bottleneck from an algorithm engineering perspective: the repeated computation of a cardinality maximal subgraph having density at least α, subsequently referred to as α-LSP. The problem is NP-hard (see Section 3.2) and, to our knowledge, has not yet been studied in this form. We are not aware of any (exact) algorithm for it. Thus, we propose efficient approximations that work well in practice.

As a basis, we combine CREP with a greedy approximation algorithm by Charikar [16], which has originally been introduced for 2-approximating the Densest Subgraph Problem (DSP). This allows for a first implementation of an algorithm for the BRP problem. However, it is still too slow and results in high communication cost. While we show that Charikar's 2-approximation guarantee

carries over to the Heaviest Subgraph Problem (HSP), it does not hold for the α-LSP.

Assuming an exact solution X of the subproblem can be computed, we find that during the execution of CREP X is always a subgraph of the connected component of the current request. Therefore, we restrict Charikar to the connected component of the most recent request and obtain better run times as well as greatly reduced total costs. From our extensive work with Charikar we have the impression, that it generally performs worse on disconnected inputs for our purpose of approximating the α-LSP.

We achieve a further speedup by introducing a hop heuristic, i.e. restricting Charikar to the 2-hop neighborhood of the current request. Hereby, the number of vertices explored at each iteration is reduced, resulting in an even greater locality behavior of the algorithm.

Finally, motivated by the temporal structure of real world network traffic, we propose decaying edge weights to model a constrained time window of interest. Our lazy implementation does not generate any significant overhead in complexity. This heuristic consistently improves the partitioning quality.

Run time and partition quality of the algorithms and heuristics are compared in an extensive evaluation with two network traces: an NS2 simulation-based trace (pFabric [3]) and a production-level trace (HPC [17]). There we also explore the influence of various parameters, both problem- (e.g. α) and algorithm-specific (e.g. number of hops). The parameter space of aging is studied as well.

Contributions

- We present the first efficient algorithm for the balanced repartitioning problem. Assuming an algorithm engineering perspective, we revisit CREP and identify its bottleneck as the repeated computation of a largest subgraph having density at least α. Therefore, we propose to combine it with an approximation algorithm by Charikar.
- We find that the theoretical weakness of this combination lies in the (dis-) connectedness of inputs for Charikar and propose various heuristics to address it. These heuristics exploit the fact that real network traces feature temporal and spacial locality. Most notably, we restrict the area considered for migration to the 2-hop neighborhood of the most recent request.
- By gradually decaying edge weights we further address dynamically shifting communication patterns of production-level traces. This approach can be realized in a lazy fashion with minimal overhead.
- In an extensive evaluation of the proposed algorithms based on synthetic and real network traffic we validate that our heuristics yield performance improvements in terms of partition quality and run time. We find that HOPage is up to 5 times faster and 2.5 times better (in terms of communication costs) than NAIVE. Moreover, we shed some light on the influence of both problem- and algorithm-specific parameters on quality and speed.

1.1 Notation

This section provides an overview of some notations and conventions used throughout the thesis. By $G = (V, E)$ we denote a graph with vertex set $V = \{ v_i \mid i \in [n], n \in \mathbb{N}_0 \}$ and edge set $E = \{ e_j \mid j \in [m], m \in \mathbb{N}_0 \}$. For an undirected graph we have $e_j = \{v_a, v_b\}$, whereas for a directed graph we write $e_j = (v_a, v_b)$. Unless specified otherwise, a graph is assumed to be undirected. Additionally, functions $w\colon E \to \mathbb{R}$ and/or $\mathcal{W}\colon V \to \mathbb{R}$ are used for assigning values/weights to edges and vertices respectively. When it is not already clear from the context, these functions are related to a graph by writing $G = (V, E[, w][, \mathcal{W}])$. In general, we assume $w(E) \subseteq \mathbb{N}_0$ as well as $w(V) \subseteq \mathbb{N}_0$. If w is omitted, the graph is called unweighted, otherwise it is called weighted. In the former case the following can be implicitly assumed: $\forall e \in E\colon w(e) = 1$. For the sake of convenience we extend w to graphs: $w(G) \coloneqq \sum_{e \in E} w(e)$. Observe that when comparing directed and undirected graphs the weight value of the undirected graphs has to be multiplied by 2.

Let $G = (V, E, w)$ be an undirected, weighted graph and $X \subseteq V$ a vertex set. Two standard properties of X are the average degree $\deg_{avg}(X) \coloneqq \frac{2 \cdot w(X)}{|X|}$ and the edge density $\rho(X) \coloneqq \frac{2 \cdot w(X)}{|X| \cdot (|X|-1)}$, [48]. In this work we use a slightly different notion of density, used in the CREP algorithm [5]. It is defined as $\rho_c(X) \coloneqq \frac{w(X)}{|X|-1}$ and we simply call it *CREP-density*. While ρ exhibits the linearity property, ρ_c does not. Particularly, ρ_c is not additive. This fact is crucial for the proof of the theoretical guarantees of CREP and also the reason any approximation of the α-LSP (as defined in section 3.2) will generally fail to produce solutions with guarantees. We often simply use the term density for ρ_c when the context is clear.

Let $X \subseteq V$ be a subset of vertices. By $G_X = (X, E_X[, w_X][, \mathcal{W}_X])$ we denote the subgraph induced by the vertices in X. Formally, $E_X \coloneqq \{ \{v_a, v_b\} \mid v_a, v_b \in X \}$, $w_X \coloneqq w|_X$ and $\mathcal{W}_X \coloneqq \mathcal{W}|_X$. The functions w and $\mathcal{W}$ carry over naturally. We often simply write X instead of G_X. As a consequence, we also apply classical set operations on vertex sets and are ultimately interested in the subgraph induced by the modified vertex set.

A partition of a graph $\mathcal{P} = \{ S_t \subseteq V \mid t \in [l] \}$ is a set of pairwise disjoint vertex sets such that $\dot{\bigcup}_{t \in [l]} S_t = V$. We overload the notion of $\mathcal{P}$ as a function $\mathcal{P}\colon V \to [l]$, which assigns vertices to their partition. A partition naturally induces a cut $C = (S_t \mid t \in [l])$, with the corresponding cut-set being $\mathcal{E} = \{ \{u, v\} \mid \exists i \neq j\colon u \in S_i, v \in S_j \}$. When dealing with cuts we are typically interested in minimizing/maximizing the value/weight of the cut, $val(C) = \sum_{e \in \mathcal{E}} w(e)$.

By $\mathbf{1}_n$ we denote the $n \times 1$ column vector having all n elements equal to 1. If the dimensions are clear from context, we simply write $\mathbf{1}$. Let further $\overrightarrow{k}_n$ be the $n \times 1$ column vector having all elements equal to k.

1.2 Definitions

Optimization problem, [14]. We are given a set of instances $\mathbb{I}$, corresponding to all possible instantiations of the problem definition. For a specific instance

or input $I \in \mathbb{I}$ let $\mathcal{I}$ denote the set of feasible solutions, where each feasible solution is associated with a value: *cost* or *profit*. The solution to the minimization formulation is defined as $I^* = \text{argmin}_{i \in \mathcal{I}} \, cost(i)$. Let the corresponding cost be $\text{OPT}(I) = cost(I^*)$. Analogously, for the maximization formulation we have $I^* = \text{argmax}_{i \in \mathcal{I}} \, profit(i)$.

Approximation Algorithm, [14]. Let $c \geq 1$. Given an (intractable) minimization problem $\mathbb{I}$, an algorithm ALG is called a c-approximation if $\forall I \in \mathbb{I}$: $\text{ALG}(I) \leq c \cdot \text{OPT}(I)$, where $\text{ALG}(I)$ denotes the *cost* of the solution computed by ALG. Similarly, ALG is said to be a c-approximation for a maximization problem if $\forall I \in \mathbb{I}$: $\text{ALG}(I) \geq \text{OPT}(I)/c$, where $\text{ALG}(I)$ is the *profit* of the computed solution. Additionally, we require that an approximation algorithm runs in time polynomial in the input parameters.

Online Computation and Competitive Analysis, [13, 14]. In the (traditional) offline setting an algorithm has access to the entire input before computing a solution. In this thesis we are in the realm of online algorithms, which are presented with sequential batches of the input and are required to provide a (temporary) solution after each batch. We emphasize that, at any point in time, the algorithm does not have access to future batches and cannot modify computed solutions from the past. The performance of an online algorithm is usually measured by comparing it to the optimal offline algorithm OPT. Let $c \geq 1$. An online algorithm ALG is c-competitive if there is a constant α such that for all finite inputs I,

$$\text{ALG}(I) \leq c \cdot \text{OPT}(I) + \alpha,$$

where $\text{ALG}(I)$ is the total cost incurred by running ALG on the input I. Note that α does not depend on I. ALG is called *strictly* c-competitive if $\alpha \leq 0$. The inclusion of the additive constant α allows the factor c to be independent of initial conditions. If an online algorithm is strictly c-competitive and runs in polynomial time, it can be viewed as a c-approximation algorithm. The competitive ratio (CR) of an algorithm ALG is defined as the infimum over the set of all values c such that ALG is c-competitive. It is denoted as $\mathcal{R}(\text{ALG})$. We further call an algorithm *competitive* if its competitive ratio is independent of the input I. Conducting a competitive analysis for an algorithm ALG (for a minimization problem) can be viewed as a worst case analysis involving an adversary seeking to maximize ALG's cost by carefully constructing an input sequence for ALG.

While there are no performance requirements in the definition of competitiveness of online algorithms, in practice we are interested in efficient algorithms. Computation within polynomial (in the input parameters) time is desirable.

1.3 Balanced RePartitioning (BRP)

The Balanced RePartitioning problem, first introduced by Avin, Bienkowski, Loukas, Pacut, and Schmid [5], lies at the core of this thesis.

In short, we need to maintain a perfectly balanced partitioning of a set of communicating vertices. Inter-partition communication incurs a cost of 1 per request, while intra-partition communication does not incur any cost. At any point in time, vertices may be migrated between partitions at a cost of $\alpha \geq 1$ per vertex move. The objective is then to balance migration costs with the corresponding saved communication costs. For example, collocating a set of frequently communicating vertices seems intuitive, while migrating a vertex to some partition where there are located no or only a few future communication partners would, depending on the specific value of α, probably not pay off.

More formally, let $V = \{v_0, \ldots, v_{n-1}\}$ be our vertex set. We write i for v_i when the context is clear. The time interval of interest is $\mathcal{T}_0 = \{0, \ldots, t_{end}\}$, $t_{end} \in (\mathbb{N}_0 \cup \infty)$. We further define the shorthand notation $\mathcal{T} = \mathcal{T}_0 \setminus \{0\}$. Let l be the number of required partitions of size $k = n/l$ each. We denote the partitioning at time t as $\mathcal{P}_t = \{S_t^0, \ldots, S_t^{l-1}\}$. Similarly to the vertices, we write i instead of S_t^i when the context permits to do so. The initial assignment of vertices to partitions is $\mathcal{P}_0$, where we generally assume a uniform distribution of the vertices across all partitions. At each point in time $t \in \mathcal{T}$ we are required to serve a communication request $r_t = (i, j)$ from source vertex v_i to target vertex v_j. The entire sequence of requests is denoted as σ. Immediately before serving a request r_t, the partitioning $\mathcal{P}_{t-1}$ may be updated to $\mathcal{P}_t$. Moving a single vertex to another partition costs $\alpha \geq 1$. Then, r_t is served and incurs a cost of 0 if $\mathcal{P}_t(i) = \mathcal{P}_t(j)$, and 1 otherwise.

Partition size augmentation. It is common to allow the online algorithm to place more than n/l elements in a single partition. The amount of additional space is specified by the parameter $\delta \geq 1$: $k' = \delta \cdot (n/l)$. Such an algorithm is called δ-augmented. This augmented online algorithm is then compared to the optimal offline algorithm without augmentation.

Hardness results. The static offline version of the BRP, where no migrations are allowed and σ is known in advance, corresponds to the NP-complete balanced graph partitioning problem [4]. This suggests that the BRP is also algorithmically challenging. It has been proven [8] that no deterministic online algorithm can attain a competitive ratio smaller than k, even with an arbitrary amount of augmentation $\delta < l$. The constraint on δ ensures that the algorithm cannot place all $n = l \cdot k$ nodes on the same cluster.

Offline BRP We now formalize the offline variant of the BRP. Offline in the sense that the entire request sequence is known beforehand. Let the partitioning at time t be encoded as the matrix $M_t \in \{0,1\}^{n \times l}$ where $(M_t)_{ia} = 1$ if vertex i is assigned to partition a, and $(M_t)_{ia} = 0$ otherwise. For M_t to be valid each row must sum to 1, i.e. $M_t \mathbf{1}_l = \mathbf{1}_n$, and each column must sum to k, i.e. $M_t^T \mathbf{1}_n = \overrightarrow{k}_l$. $\mathcal{M}$ is the set of all matrices that encode a perfectly balanced partitioning. Let a request $r_t = (i, j)$ be encoded as the vector $r_t \in \{-1, 0, 1\}^n$ where $(r_t)_i = -1$, $(r_t)_j = 1$ and $(r_t)_s = 0$ for $s \neq i, j$. Equation 1 depicts the main optimization

objective of the offline BRP. An exact algorithm for it (in theory) will later be denoted as OPT.

$$\begin{aligned} \underset{(M_t \mid t\in\mathcal{T}) \in \mathcal{M}^{|\mathcal{T}|}}{\operatorname{argmax}} \quad & \frac{1}{2}\sum_{t\in\mathcal{T}}(M_t^T r_t)^T(M_t^T r_t) \\ & + \frac{\alpha}{2}\sum_{t\in\mathcal{T}} \mathbf{1}_n^T \cdot ((M_t - M_{t-1}) \circ (M_t - M_{t-1})) \cdot \mathbf{1}_k \qquad (1) \\ \text{s.t.} \quad & \forall t \in \mathcal{T}\colon M_t \cdot \mathbf{1}_l = \mathbf{1}_n \\ & \forall t \in \mathcal{T}\colon M_t^T \cdot \mathbf{1}_n = \overrightarrow{k}_l \end{aligned}$$

While formulating the offline variant as a single optimization problem makes sense, the online version must rather be seen as a series of inter-dependent cost minimization problems. The question, whether it permits the definition of a single optimization problem, remains open.

2 Related Work

With applications becoming increasingly data-centric (e.g. web services such as search engines or social networks [44], training of machine learning models [38], or scientific simulations [42]) comes the need to distribute workloads across multiple compute instances, inevitably increasing the network communication volume. While local communication is very fast and often unavoidable, non-local communication can significantly impact execution time and hence should be reduced.

2.1 Network Design and Load Balancing

A lot of research has been devoted to designing static network topologies and to achieve balanced utilization. In terms of topology, especially for HPC applications [46], a key metric is the bisection bandwidth, i.e. the minimal total inter-partition bandwidth when arbitrarily bisecting the network. To obtain a high bisection bandwidth, data centers usually rely on models such as trees [19, 58], expanders [59] or more specific ones (e.g. VL2 [25] or DCell [26]). On top of that, different load balancing mechanisms have been proposed to optimize metrics like throughput, latency, and robustness [63]. This state-of-the-art combination results in efficient, reliable routing.

Another approach to reduce communication cost is via demand-aware networks. These networks are optimized toward the workload they serve, either by statically mapping the workload to network resources [49, 50] or in a dynamic fashion. Despite being a fundamental problem, relatively little is known about demand-aware and reconfigurable networks. Avin and Schmid [10] only recently initiated the study of this field from an algorithmic point of view, proposing a taxonomy and a formal model. These theoretical considerations are facilitated through networking hardware and software becoming increasingly flexible and al-

lowing for fast reconfiguration [37, 40, 45, 62]. The authors of [10] later developed ReNet, a statically optimal self-adjusting network [9].

However, while all above techniques aim at optimizing network utilization and are therefore absolutely vital, none of them solves the problem of dynamically minimizing non-local communication. Consider the hierarchical structure of a data center for example. There we have multiple processing cores within a single server, multiple servers within a rack, and the racks are again grouped in PODs. The above methods (1) specify the network topology with worst-case communication patterns in mind, (2) ensure that redundant paths are utilized uniformly, (3) require the communication volume to be specified in advance and then compute a statically optimal mapping of the workload onto the physical units, or (4) implement the network in a flexible manner such that it can adapt or quickly be adapted to the current communication pattern. What is missing is the dynamic collocation of workload-pieces that are frequently communicating, such that these pieces are assigned to compute cores in the same server or rack.

2.2 Dynamic Balanced Graph Partitioning

This task of dynamically (re-)mapping workload-pieces onto processing units, thereby jointly minimizing the total communication and migration cost, has, to the best of our knowledge, only been studied by Avin *et al.* [5, 8]. They consider the problem in the setting of graph partitioning, where communication endpoints correspond to vertices and communication volume is modeled as edge weights. The authors formulate the novel *Balanced RePartitioning (BRP)* problem. We only give a brief and informal description here, see Section 1.3 for details. Given n vertices we have to maintain a partitioning with l parts each of size $k = n/l$, thus a perfectly balanced partitioning. Over time, pairwise communication requests encoded as edge weight updates arrive and have to be served. A request entails a cost of 0 if the two endpoints reside in the same partition at the time of serving the request, and a cost of 1 otherwise. At any point in time, vertices may be moved between partitions at a cost of $\alpha \geq 1$ per move. While an offline algorithm has complete knowledge of the entire request sequence, an online algorithm is only presented with one request at a time and thereupon required to return an updated partitioning. Online algorithms are often allowed to use additional space per partition, since this simplifies their analysis. Such an algorithm is called δ-augmented if the partition size is $\delta \cdot k$. Avin *et al.* derive a lower bound of k on the competitive ratio of any deterministic online algorithm with $\delta < l$ augmentation for the BRP. As their main contribution, they propose the algorithm CREP. With augmentation at least $2 + \epsilon$, CREP attains a competitive ratio of $\mathcal{O}((1 + 1/\epsilon) \cdot k \cdot \log k)$, $\epsilon \geq 1/k$. The idea behind CREP is to identify dense structures in the graph and to migrate the corresponding nodes into the same partition once a certain threshold is reached. Over time, nodes are grouped in size-constrained communication components to reflect high communication value.

The local and dynamic characteristic of CREP is valid as communication patterns in data centers are known to feature structure. The corresponding traffic matrices tend to be sparse, skewed, bursty, and to exhibit locality [11, 63]. Avin *et*

al. [7] define the notion of trace complexity, based on entropy and randomization. Thereby, they distinguish between temporal and non-temporal complexity. From this perspective, CREP should perform best on traces of high temporal structure and simultaneously low non-temporal structure.

Stochastic Variant

While we study the general BRP problem, where request patterns can change arbitrarily over time, [6, 27] deal with the more restricted *learning variant*. They assume an underlying distribution of the requests that is unknown initially and has to be learned in an incremental fashion. The communication pattern is revealed edge-by-edge by an adversary, whose objective is to maximize the total cost of the online algorithm. They further assume that the communication network permits a perfectly balanced partition with a cut value of 0. Thus, at some point in time enough information has been revealed such that no future node migrations are necessary. Intuitively, the formation of connected components in the graph directly corresponds to the behavior of a disjoint-set data structure. The authors give a distributed, exponential-time, $\mathcal{O}((l \log l \log n)/\epsilon)$-competitive online algorithm for the problem, assuming a partition size of $(1+\epsilon)n/l$, where $\epsilon \in (0, 0.5)$. Under the same capacity constraint relaxation, they present a polynomial-time algorithm that attains a competitive ratio of $\mathcal{O}((l^2 \log l \log n)/\epsilon^2)$. Finally, a lower bound of $\Omega(1/\epsilon + \log n)$ on the competitive ratio of any deterministic online algorithm (with augmentation at least $\epsilon n/l$) is proven. This is substantially better than what can possibly be achieved in the more general model. However, due to the nature of real world network traffic, where communication patterns are mostly temporary, it is important to study the latter.

Online Paging

In the specific case of $l = 2$ the BRP problem can be seen as an online paging (or online caching) problem [21], where requests for items need to be served from a cache of finite capacity. Avin *et al.* give a reduction of online paging (with cache size $k-1$) to the more general BRP. Thereby they transfer the lower bound of $k-1$ on the competitive ratio of any deterministic algorithm for the paging problem to the BRP with 2 partitions. Note that in the BRP we can, apart from collocating nodes and serving requests locally at cost 0, serve requests remotely at cost 1. Thus, online paging with bypassing [1, 2, 41] better reflects our problem at hand. Intuitively, bypassing the cache directly corresponds to serving a request remotely.

Online (Re-)Matching

For $k = 2$, l arbitrary and no augmentation the BRP corresponds to an online version of the classical maximum matching problem. Vertices are considered matched if and only if they are assigned to the same partition. Note that in standard maximum matching the objective is to maximize the total weight of matched edges. This is, however, equivalent to minimizing the total weight of unmatched edges, i.e. the cut value, which is exactly the objective of the BRP problem. The authors

of [5] show that any deterministic algorithm for this problem has a competitive ratio of at least 3.

2.3 Static Balanced Graph Partitioning

The static offline version of the BRP is known as the l-balanced graph partitioning problem. Here, no migration is allowed and the algorithm has knowledge of the entire request sequence beforehand. This problem is NP-complete. Andreev and Räcke [4] proved that, unless P=NP, it cannot even be approximated within any finite factor for $k \geq 3$. An $\mathcal{O}(\log^{3/2} n)$-approximation can be obtained in exponential time (in l) [35]. For $n/l = 2$ the static problem resembles the above mentioned maximum matching problem, which is solvable in polynomial time [18]. For $l = 2$ we recover the NP-hard minimum bisection problem [23], for which an approximation ratio of $\mathcal{O}(\log n)$ [47] is the best result so far.

Due to the hardness of the l-balanced graph partitioning problem, a bicriteria form is often considered: the (l, δ)-balanced graph partitioning problem. It resembles the static offline BRP with augmentation. Given a graph we seek a partition with l pieces, each of size at most $\nu \cdot (n/l)$, such that the value of the induced cut is minimized. The current best approximation ratios are $\mathcal{O}(\sqrt{\log n \cdot \log l})$ for $\delta \geq 2$ [36] and $\mathcal{O}(\log n)$ for $\delta > 1$ [20].

Heuristics

While the approximation algorithms for graph partitioning mentioned above are of high theoretical importance, state-of-the-art graph partitioning implementations usually rely on heuristics [15]. One very successful heuristic for large graphs is the multilevel graph partitioning approach [53]. First, the input graph is hierarchically coarsened into a series of smaller graphs. Thereby, it is important to maintain the invariant that cuts in the coarser graphs reflect cuts in the finer graphs. On coarsest graph an initial partitioning of high quality is computed. Note that on this level, the graphs are small enough so that expensive algorithms can be used for obtaining the initial partitioning. In the third and last phase the results from the coarser levels are propagated to the finer graphs. There are two stages per level-up: after mapping the solution from the coarser level to the next-higher level, local heuristics are used to improve the partitioning.

Existing Implementations

State-of-the-art graph partitioning libraries based on the multilevel approach include METIS [30, 31] and Jostle [60, 61], which both implement coarsening via edge-contractions based on matchings (METIS for example uses a technique called Sorted Heavy Edge Matching) and methods due to [22, 33] for local improvement. The more recent KaFFPa [53] employs a new sequential approach with flow-based local improvement. KaFFPaE [52, 54] is yet another variant which integrates KaFFPa into an evolutionary strategy. What sets KaFFPaE apart from METIS

and Jostle is that it allows to enforce strict balance constraints and thus is able to guarantee the feasibility of the output partition.

We can employ these methods to obtain a reference for the partition quality on static graphs, i.e. *after* having observed an entire sequence of requests. Specifically, we choose METIS due to its widespread use and low execution times.

2.4 Periodic Graph Partitioning

Different strategies have also been proposed for the setting of load balancing by repartitioning, where a trade-off between partition quality and migration volume is involved. There are two basic methodologies: scratch-remap (compute a new partition from scratch and then compute an optimal mapping with respect to the previous partition) and the rebalance approach (remove vertices from overflowing partitions as long as such exist). With both techniques having their drawbacks, Schloegel *et al.* combine them in the ParMETIS implementation [32, 56, 57], a distributed version of METIS. Another implementation for the setting of load balancing is the parallel version of Jostle [61]. Although these heuristics cannot provide exact performance guarantees, they are quite effective in practice [36]. What unites all existing implementations for repartitioning is that they are intended to being applied periodically. This can be a reasonable assumption for e.g. numerical simulations [42, 43]. We are in the setting of fully dynamic repartitioning, i.e. a short sequence of requests might introduce a future dense community and thus request-by-request decisions are required. Nonetheless, we experimented with ParMETIS' V3_AdaptiveRepart routine in this more challenging setting. The trade-off behavior (edge cut value vs. migration effort) is specified by a single parameter, the ITR factor. It describes the ratio between inter-partition communication cost and vertex redistribution cost.

When testing with different configurations and combinations thereof (e.g. varying alpha, resetting the edge weights at certain points, decaying edge weights, only calling the routine upon requests between different partitions, dynamically adapting the ITR factor, etc.) we obtained very inconsistent results compared to our algorithm HOPage. While the quality of the partitioning was at times comparable and for one specific setting of α even improved on the pFabric trace, V3_AdaptiveRepart performed a lot worse than HOPage on the HPC trace where it produced very high migration costs.

We therefore suggest that these existing algorithms for periodic rebalancing are not suitable for tackling the BRP on general communication patterns without further modifications. The reason may be that they consider the entire graph at each execution and are oblivious to the location of the most recent request.

2.5 Verdict

To the best of our knowledge, apart from theoretical considerations there exist no implementations of online algorithms for approximating the general BRP problem. We implement an efficient version of CREP and propose heuristics to further

improve its running time and partition quality. Finally, we explore the possibility of decaying edge weights as a means to prevent unfavorable migrations.

3 Algorithms

In this section we first describe CREP, the algorithm upon which our work builds. We discuss its bottleneck, the repeated computation of the α-LSP, in more detail and also study the effect of approximating this essential subproblem. We then turn to describing our heuristics and analyze their worst-case complexity.

To be consistent with the actual implementation of the algorithms, we follow a modular approach in their theoretical description. Thus, after the subsection on CREP we limit the scope to efficient algorithms for the α-LSP. These can then simply be plugged into our framework for the BRP problem.

3.1 CREP

Avin *et al.* [5] proposed CREP as one of the first algorithms for the balanced repartitioning problem. It is particularly appealing because of the proven worst-case performance guarantee. With $(2+\epsilon)$-augmentation CREP attains a competitive ratio of $\mathcal{O}((1+\frac{1}{\epsilon}) \cdot k \cdot \log k)$, where $\epsilon \geq 1/k$. With only a logarithmic factor difference to the lower bound of k it is reasonable to base our work on this algorithm.

In short, CREP maintains a graph structure where vertices correspond to communication endpoints and edge weights represent the number of paid inter-partition requests. Vertices are further grouped into communication components, each of size at most k. Intuitively, recently and frequently interacting vertices should belong to the same component. CREP then maintains the natural invariant that all nodes of a component must be assigned to the same partition. At any time, the algorithm serves the current request, updates graph and components, and eventually decides on moving vertices between partitions.

Algorithm Definition

In addition to the partitioning $\mathcal{P}$ required by the problem definition of Balanced RePartitioning, CREP groups recently and frequently interacting nodes into logical, size-constrained communication components $\mathcal{C}$. This second-order partitioning can be viewed as a refinement of the first-order partitioning in the sense that every element of $\mathcal{C}$ is a subset of some element of $\mathcal{P}$. Thus, all nodes from one component are placed in the same partition. More formally,

$$\forall i, j \in V : \mathcal{C}(i) = \mathcal{C}(j) \implies \mathcal{P}(i) = \mathcal{P}(j).$$

In the following, we assume an augmentation of $2+\epsilon$. CREP starts from n singleton components. At any time $t \geq 1$, CREP processes a request r_t between two vertices i and j. If both endpoints are currently in the same partition, the request is served at a cost of 0 and CREP continues to the next request. Otherwise, if $\mathcal{P}(i) \neq \mathcal{P}(j)$, the following steps are performed:

1. Increase the edge weight $w(i, j)$ by 1.
2. Find a mergeable component set S of maximal cardinality. A non-trivial component set $S = \{ c_1, \ldots, c_{|S|} \}$ is called *mergeable* if $w(S) \geq (|S| - 1) \cdot \alpha$, where $w(S)$ denotes the total weight of edges that run between two components of S. If such a set S exists, the contained components are merged into one new component c. Thereby, the weights of all edges within S are reset to 0. This merging procedure is carried out as $|S| - 1$ pairwise merge actions.
3. If c contains more than k vertices, it is split into singleton components and CREP continues to the next request. Otherwise, CREP assigns all vertices of c to the same partition. Consider a pairwise merge action of the components c_a and c_b. Without loss of generality let $|c_a| \leq |c_b|$. If the $\mathcal{P}(c_b)$ has at least $|c_a|$ free capacity, all vertices of c_a are moved to $\mathcal{P}(c_b)$. If this is not the case, the vertices of both, c_a and c_b are moved to some partition with at most k vertices. Note that such a partition always exists due to the augmentation of $2 + \epsilon$ (averaging argument), and that $|c_a \cup c_b| <= k$ since c has not been split into singletons. Then, CREP continues to the next request.

The pseudo code for CREP with $(2 + \epsilon)$-augmentation is summarized in Algorithm 1. There, we already included our modifications discussed in Section 3.2.

An Intuition Why CREP Makes Sense

Intuitively, CREP merges a cardinality-maximal set of components into a single new component when the contained vertices communicated roughly as much as collocating all these vertices costs in the worst case, i.e. every contained vertex needs to be moved. Thus, we wait just long enough before triggering migrations so that the migration costs never exceed the communication costs by more than a constant factor.

In numbers, we merge a component set S as soon as the density (more specifically ρ_c) of the induced subgraph reaches α. At this point, the components belonging to S have exchanged $\alpha \cdot (|S| - 1) \cong \alpha \cdot |S|$ paid communication requests. This is exactly as much as migrating $|S|$ vertices costs. And since this holds for each component of S recursively, the total migration costs never exceed the total communication costs by more than a factor of 2. Therefore, (1) CREP does intuitively the right thing by being careful not to migrate too much, and (2) it is sufficient to only consider communication costs in the analysis.

On the Proof of CREP's Competitive Ratio

Let σ be an arbitrary input sequence of requests. We denote the cost of OPT on σ as OPT(σ). Similarly, we write CREP(σ) for CREP's cost on σ. In order to compute a complexity class for the competitive ratio of CREP, two bounds are needed: an upper bound on CREP(σ) as well as a lower bound for OPT(σ). For the latter we will essentially get the sum of the sizes of deleted components, while the former boils down to the sum of the sizes of merged components times $\log k$.

Algorithm 1: CREP with $(2+\epsilon)$-augmentation

input: number of distinct vertices n, number of partitions l, partition size k, augmentation δ

Construct graph $G = (V, E, w)$, where $w(u, v) \coloneqq 0$ for all $\{u, v\} \in \binom{V}{2}$. Define Φ such that it maps each vertex v to its corresponding singleton component: $\Phi(v) \coloneqq \phi_v$.

for *each new request* $r_t = \{ u_t, v_t \}$ **do**
  Let $\phi_u = \Phi(u_t)$ and $\phi_v = \Phi(v_t)$.
  if $\phi_u \neq \phi_v$ **then**
    $w(u_t, v_t) \coloneqq w(u_t, v_t) + 1$
    $w(v_t, u_t) \coloneqq w(u_t, v_t)$
  end
  `/* Let com(`X`) denote the total weight of edges between vertices contained in` X`. */`
  Let $X = \{ \phi_i \mid i \in [z] \}$ be the largest cardinality set of components with $\text{com}(X) \geq (|X| - 1) \cdot \alpha$.
  if $|X| > 1$ **then**
    `// Let vol(`X`) denote the total number of vertices in` X`.`
    if $vol(X) \leq k$ **then** `// Merge components in` X`.`
      Let $\phi_X = \cup_{i \in [z]} \phi_i$.
      `// Reset intra-component edge weights.`
      For all $u, v \in \phi_X, u \neq v$ set $w(u, v) \coloneqq 0$.
      `// Collocate vertices in` ϕ_X
      `/* Let used(`y`) denote the used space in partition` y`, and let usedby(`y, c`) denote the space used by vertices in component` c `in partition` y`. */`
      Let Ψ be the set of partitions which contain at least one vertex of ϕ_X. Let $Y \subseteq \Psi$ be the set of partitions such that for $y \in Y$ the following holds: $\text{used}(y) - \text{usedby}(y, \phi_X) + |\phi_X| \leq \delta \cdot k$.
      if $Y \neq \emptyset$ **then**
        Let $s \in Y$ be the partition containing the most vertices of ϕ_X.
        Migrate all vertices of ϕ_X, which are not already in s, to s.
      else
        Migrate ϕ_X to a partition s with maximum free capacity.
      end
    else `// Split components in` X `into singleton components.`
      Split every $\phi_i \in X$ into $\text{vol}(\phi_i)$ singleton components. Thus, for each $v \in \phi_i$ set $\Phi(v) \coloneqq \phi_v$ and delete ϕ_i.
    end
  end
end

Lower bound on OPT(σ). We start with the more interesting lower bound on OPT(σ). When OPT migrates it pays α and we are done. Thus, what can we conclude when OPT does not migrate but CREP does? Intuitively, we ultimately look at the event of a component being deleted. Assume, for example, that a merge resulted in a component c of size $z \cdot k \gg k$. In this case c must span at least z partitions in the solution of OPT. By the definition of the CREP-density ρ_c the component c is guaranteed to be well connected and thus we know that OPT pays a minimum amount of communication cost when it does not migrate.

Because of its importance, we elaborate on that last statement. It is based on the fact that the CREP-density, defined as $\rho_c(S) = w(S)/(|S|-1)$, is *not* additive. This is due to the additive constant of -1 in the denominator. It implies the following: Let $G_1, \ldots, G_q$ be vertex-disjoint subgraphs with $\rho_c(G_i) \leq \alpha$. When their union $\bigcup_i G_i$ has a CREP-density of at least α, then there must be a certain minimum weight between the subgraphs G_i. This intuitive fact is essential for Lemma 1.

Before we can continue to a more formal discussion of the lower bound on OPT(σ), we need to introduce some additional notation. For a component c let $\mathcal{F}(c)$ represent its *component epoch*. $\mathcal{F}(c)$ contains all (time t, vertex v) pairs such that at time t the vertex v belongs to c and at that time c is a top-level component (i.e. it is not part of another component). Let $S(c)$ denote the set of components that were merged to create component c at time $\tau(c)$.

At the core of the proof idea of the lower bound on OPT(σ) lies Lemma 4 in [5]. Due to its importance for subsequent sections we restate it in Lemma 1 together with a proof. There we need the fact that at any time during the execution of CREP no non-trivial component set S can have $\rho_c(S) > \alpha$. In particular, a mergeable component set always has $\rho_c = \alpha$ and there can never exist a mergeable component set directly after CREP performed merges.

Lemma 1. *Fix any component c and partition $S(c)$ into a set of $g \geq 2$ disjoint component sets $S_1, S_2, \ldots, S_g$. The number of communication requests in $\mathcal{F}(c)$ that are between sets S_i is at least $(g-1) \cdot \alpha$.*

Proof. We know that the number of communication requests between the component sets S_i that CREP had to pay for is $w(S(c)) - \sum_{i=1}^{g} w(S_i)$. Further, $\rho_c(S(c)) = \alpha \iff w(S(c)) = (|S(c)| - 1) \cdot \alpha$ and $\rho_c(S_i) \leq \alpha \iff w(S_i) \leq (|S_i| - 1) \cdot \alpha$. This gives

$$w(S(c)) - \sum_{i=1}^{g} w(S_i) \geq (|S(c)| - 1) \cdot \alpha - \sum_{i=1}^{g} (|S_i| - 1) \cdot \alpha = (g-1) \cdot \alpha.$$

□

Next, we relate the costs of OPT to CREP's component structure. Therefore, we initially restrict ourselves to a single component c. Let OPT(c) denote OPT's costs that stem from requests between two vertices contained in $\mathcal{F}(c)$. We look at the set $S(c)$ and how the vertices contained in the components therein are distributed among the partitions in the *solution of OPT*. By applying Lemma 1

we get a lower bound on the number of communication requests that are between different partitions and thus are paid for by OPT. Lemma 5 in [5] formalizes this.

Finally, we can bring $\mathrm{OPT}(\sigma)$ into the equation. Consider how any component c, that was eventually deleted by CREP, was created. This process can be modeled as a tree with singleton components as leaves and c as the root. Inner nodes correspond to intermediate components that were created at some point by merging the child nodes. We can now apply Lemma 5 of [5] to each node in this tree and, using the fact that $|c| > k$, arrive at $\mathrm{OPT}(\sigma) \geq \sum_{c \in \mathrm{DEL}(\sigma)} |c|/(2k) \cdot \alpha$, where $\mathrm{DEL}(\sigma)$ denotes the set of components that were deleted by CREP. [5] formalizes this argument in Lemma 6 therein.

As a side note consider the following: The way in which Lemma 6 in [5] is formulated suggests that the size of deleted components can be much larger than k. Otherwise we could bound $|c|$ for any component c by some constant times k and would in turn be able to bound $\mathrm{OPT}(\sigma)$ by $\mathcal{O}(|\mathrm{DEL}(\sigma)|) \cdot \alpha$. However, this idea remains to be proved or disproved.

Upper bound on CREP(σ). To bound $\mathrm{CREP}(\sigma)$ from above, we separately consider the communication cost $\mathrm{CREP}_r(\sigma)$ and the migration cost $\mathrm{CREP}_m(\sigma)$.

The former can be expressed roughly as the number of merge actions (that are not immediately followed by a delete operation) times α, see Lemma 7 in [5]. To bound $\mathrm{CREP}_m(\sigma)$ assume without loss of generality that all merge actions are binary. Then it can be upper bounded by the sum of the sizes of the smaller component being merged over all merge actions. See Lemma 8 in [5] for details.

3.2 CREP Revisited

In this section we discuss CREP's bottleneck, the repeated computation α-LSP, in more detail. We also study the theoretical implications of a suboptimal solution for this problem. Various approaches for approximating this part of CREP are considered. Finally, we look at the computation of a suitable target partition in the case of a merge action.

The α-LSP

We define the subproblem of finding a cardinality-maximal subgraph with CREP-density at least α as the *α-Largest Subgraph Problem (LSP)*. This problem has to be solved each time CREP serves an inter-partition request r_t. It is easy to see that, in the case the α-LSP has a non-empty solution S_t at time t, both endpoints of r_t are contained in S_t. Another interesting fact is that S_t is always connected, which turns out to be of central importance for our proposed heuristics. We prove this observation in Lemma 2.

Lemma 2. *During the execution of CREP, a mergeable component set always induces a connected subgraph.*

Proof. This property follows from a simple proof by contradiction and the fact that the CREP-density ρ_c is not additive. If, at some point, there existed a mergeable

set of components S that is not connected, then at least one connected subset of S would be mergeable. This subset, however, would have been classified as mergeable at an earlier point in time. Thus, during the execution of CREP in its original form there never exists a non-connected mergeable component set. □

The α-LSP is NP-hard. This follows from a simple reduction of the NP-complete [34] densest-at-least-k subgraph problem. For an unweighted graph, there are polynomially many possible values for the density. We could solve the α-LSP for each of these values and, among the computed subgraphs having at least k vertices, return the densest one as solution for the densest-at-least-k subgraph problem. Another related problem is the densest subgraph problem (DSP), which is solvable in polynomial time [24]. Charikar [16] propose a simple greedy algorithm that achieves a 2-approximation for the DSP

What sets these related problems apart from the α-LSP is that they mainly revolve around finding a substructure with maximum density, while we ask for a *largest* substructure over a certain density threshold. We are not aware of any previous work that studied the α-LSP in this form.

How to approximate the α-LSP? The α-LSP needs to be solved repeatedly and, assuming there is exclusively inter-partition communication, possibly even every request. Therefore, an efficient algorithm or heuristic for approximating this problem is vital. Our objective shifts from computing the largest mergeable subgraph to finding a substructure that is *large and alpha-dense*. We emphasize the last part: the density requirement for a component set to be merged remains intact.

A number of different approaches have been collected, most of which originally serve the purpose of finding and/or maintaining dense and/or well-connected regions in a graph. We highlight three of them:

- Charikar's greedy algorithm [16]: remove vertices in increasing order of their degree and return the densest encountered subgraph. See Section 3.3 for details.
- Maintain a k-core decomposition over time [55]: upon an edge update (insertion/removal) identify the set of vertices which could possibly switch to a different k-core and then iterate these vertices. A downside of this approach is that it is only defined for unweighted graphs.
- Keep track of connected components by using a combination of a spanning forest (e.g. a forest of Euler tour trees) and a Cutset structure (for finding an alternative connecting edge e' when some edge e in a tree is deleted) [28]. This strategy is difficult to implement, especially in the light of changing vertices (c.f. merges and deletions of communication components).

None of these has been proposed to specifically solve our problem. Ultimately, we selected Charikar due to its simplicity and linear complexity in the graph size.

Analysis of CREP with a Suboptimal Algorithm for α-LSP

In this section we discuss the consequences of employing an approximation algorithm for the α-LSP. More specifically, we consider any algorithm ALG that exhibits some approximation guarantee on the density of the returned subgraph.

Assume that ALG is μ-approximate with respect to the density, where $\mu > 1$. Let G be a graph, and $G_1, \ldots, G_z \subsetneq G$ be disjoint and *disconnected* subgraphs with $|G_i| < k$ and $\rho_c(G_i) \in (\alpha, \mu \cdot \alpha)$. Let further $\mathcal{G} = \dot{\bigcup}_{i \in [z]} G_i$ and $\rho_c(\mathcal{G}) \geq \alpha$. Then the following scenario is possible: the subgraphs G_i are created over time and ALG does not find any them as $\rho_c(G_i)/\mu < \alpha$. ALG may, however, eventually return $\mathcal{G}$ as a mergeable subgraph with density $\geq \alpha$. This constitutes a problem because $\mathcal{G}$ is not well connected. In fact, there are no edges at all between the different subgraphs G_i. Consequently, as $|G_i| < k$, we cannot attribute any cost to the optimal offline algorithm OPT. The lower bound on OPT can hence not be extended for the combination of CREP with an arbitrary approximation algorithm for the α-LSP.

Intuitively, this tells us is that for CREP to achieve any guarantee on the competitive ratio, we need the assurance that a to-be-merged component set is always well connected.

On the other hand, the upper bound on CREP(σ) is not affected. This is due to merge actions still only being triggered when the corresponding component set has a CREP-density of at least α.

Finding a Target Partition.

CREP's original definition specifies migration only for the binary case, i.e. two components and being merged. See Section 3.1 for details. While this is sufficient for the theoretical analysis, we need to define a migration strategy when $g > 2$ components are merged.

Let $S_1, \ldots, S_g$ be the components being merged into a new component $S = \dot{\bigcup}_{i \in [g]} S_i$. Let $u(y)$ denote the used space in partition y, and let $u(y, c)$ denote the space used by vertices of component c in partition y. Let $\Psi \subseteq \mathcal{P}$ be the set of partitions which contain at least one vertex of S. Let $Y \subseteq \Psi$ be the set of partitions such that for $y \in Y$ the following holds: $u(y) - u(y, S) + |S| \leq \delta \cdot k$. If $Y \neq \emptyset$, let $s \in Y$ be the partition containing the most vertices of S. Else, let s be a partition with maximum free capacity. We migrate all vertices of S, which are not already in s, to s.

Another aspect worth mentioning is the possibility of eviction. In this scenario the algorithm would be allowed to relocate not only the to-be-collocated vertices but also a limited number of vertices from the target partition. Thus, it would have to strike a balance between (1) the optimality of the target partition with respect to the vertices that are to be placed there, and (2) the effort of freeing enough space on the target partition. While we have briefly explored the potential of such a more involved method (using e.g. a scoring function that considers the number of vertices requiring relocation, the total weight of edges becoming inter-partition edges, as well as the total eviction cost), the results were moderate and

this area of potential improvement remains to be studied.

3.3 CREP with Charikar (NAIVE)

We describe our first naive approach of making CREP more efficient. Because of its simplicity, we use the algorithm of Charikar [16] to approximate the α-LSP.

In short, whenever an inter-partition request is processed, we increase the corresponding edge weight and execute Algorithm 3 on the *entire* graph. If a subgraph $X \neq \emptyset$ is returned, we merge the components contained in X and move the corresponding vertices into the same partition. This strategy is henceforth called NAIVE.

Charikar's Algorithm

Charikar proposed an intuitive greedy algorithm for 2-approximating the densest subgraph problem. Starting from the entire graph, for which the densest subgraph is to be approximated, the algorithm repeatedly removes the vertex with the smallest degree, breaking ties arbitrarily. When a vertex is removed, the degrees of its neighbors are updated accordingly. Once all vertices have been removed, out of the $|V|$ considered subgraphs Charikar returns the densest one.

We extended Charikar to weighted graphs (see Algorithm 2), and show in Lemma 3 that the approximation guarantee still holds in this generalized setting. However, since we are not interested in finding the heaviest subgraph, but instead seek the largest subgraph with $\rho_c \geq \alpha$, we further modify Algorithm 2 to return as soon as the current density ρ_c is at least α. If no mergeable subgraph is found, an empty graph is returned. This final version of Charikar is outlined in a Algorithm 3. We prove in Lemma 4 that substituting ρ_c for ρ does not alter the approximation guarantee.

Algorithm 2: Charikar for HSP

input: graph $G = (V, E)$

```
Let n ← |V|, X ← G and Y ← X.
while X ≠ ∅ do
    Let v ← argmin_{u∈X} deg(u)
    X ← X \ {v}
    if ρ(X) < ρ(Y) then
        Y ← X
    end
end
return Y
```

Analysis of the Modified Charikar

Assume that there exists some subgraph $S \subseteq X$ with density $\rho(S) \geq \alpha$. Algorithm 3 does not necessarily return S, nor any other subgraph with density at

Algorithm 3: Charikar for α-LSP

input: graph $G = (V, E)$, migration cost α

Let $n \leftarrow |V|$ and $X \leftarrow G$.
while $X \neq \emptyset$ ***and*** $\rho_c(X) < \alpha$ **do**
  Let $v \leftarrow \text{argmin}_{u \in X} \deg(u)$
  $X \leftarrow X \setminus \{v\}$
end
return X

least α. By Lemma 3 we are only guaranteed to get some subgraph $Y \subseteq X$ with $\rho(Y) \geq \alpha/2$. Furthermore, the algorithm does not exhibit any guarantee on the size of the returned subgraph Y compared to the size of the optimal solution X^*. The example in Figure 1 illustrates this.

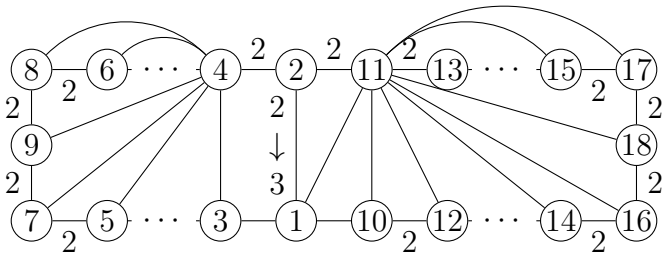


Figure 1: An example where Charikar returns a subgraph of suboptimal size. Let $\alpha = 3$. This graph depicts a snapshot of the communication network that can actually occur in practice. Let the number of vertices to left of 1 and 2 be a and the number of vertices to the right of 1 and 2 be $b > a$. Both, a and b are uneven. We consider the point in time where $w(1, 2)$ is increased from 2 to 3. There, we can assume without loss of generality that Charikar first removes vertex 10. Eventually, the subgraph consisting of all vertices and edges to the left of 1 and 2 (inclusive) would be returned. However, the subgraph to the right of 1 and 2 (inclusive) also has $\rho_c = \alpha$ and is arbitrarily larger.

Lemma 3. *Algorithm 2 achieves a 2-approximation for the Heaviest Subgraph Problem in undirected graphs. There is no constant $c < 2$ such that Algorithm 2 achieves a c-approximation.*

Proof. We adopt the proof from [34]. Let X^* denote the optimal solution of the HSP and let $\rho(X^*) = d_{OPT}$ be its density.

Intermediate corollary: For every vertex $v \in X^*$ it holds that $\deg(v) \geq d_{OPT}$. Proof of intermediate corollary: Let X^* and d_{OPT} be defined as before. Then, by

the optimality of X^*, we get for any $u \in X^*$:

$$\begin{aligned} d_{OPT} = \frac{w(X^*)}{|X^*|} &\geq \frac{w(X^*) - \deg_{X^*}(u)}{|X^*| - 1} \\ \frac{w(X^*) \cdot (|X^*| - 1)}{|X^*|} &\geq w(X^*) - \deg_{X^*}(u) \\ \deg_{X^*}(u) &\geq w(X^*) \cdot (1 - \frac{|X^*| - 1}{|X^*|}) \\ &= w(X^*) \cdot \frac{|X^*| - |X^*| + 1}{|X^*|} \\ &= \frac{w(X^*)}{|X^*|} = d_{OPT}. \end{aligned}$$

This concludes the proof of the intermediate corollary.

Let X be the subgraph just before Algorithm 2 removes the first vertex $w \in X^*$. By the definition of the algorithm and the intermediate corollary, all vertices in X have degree $\geq d_{OPT}$. Thus, we get:

$$\rho(X) \geq \frac{\frac{1}{2}|X| \cdot d_{OPT}}{|X|} = \frac{1}{2}\, d_{OPT}.$$

Therefore, Algorithm 2 finds a subgraph with density at least $\frac{1}{2}\, d_{OPT}$.

To prove that the bound of 2 is tight, we restate the example given in [34] in a more precise manner. Let $G_1 = (V_a \dot\cup V_b, E)$ be a complete bipartite graph with $|V_a| = d$ and $|V_b| = D$. Let G_2 be the disjoint union of D complete graphs $K^i_{d+1}, i \in \{1, \ldots, D\}$. All vertices of G_2 have degree d, and vertices of G_1 have degrees d or D. Fix d and let $D \to \infty$. Further, let $G \coloneqq G_1 \dot\cup G_2$ and execute Algorithm 2 on G. We may assume that the algorithm first removes all vertices of degree d from G_1. The density of the densest subgraph we encounter during the execution of Algorithm 2 approaches $\frac{d+3}{d+2} \cdot \frac{d}{2}$ as $D \to \infty$. However, the density of G_1 approaches d as $D \to \infty$. In fact, G_1 is the optimal solution. □

Lemma 4. *Substitute edge density $\rho(X)$ with CREP-density $\rho_c(X) \coloneqq \frac{w(X)}{|X|-1}$ in Algorithm 2. Then there exists a constant $c < 2$ such that the modified algorithm achieves a c-approximation for the HSP.*

Proof. Let X^* denote the optimal solution of the HSP and let $\rho_c(X^*) = d_{c,OPT}$ be its density. We further assume $|X^*| \geq 2$. Similarly to the proof of Lemma 3, we first show the following intermediate corollary: For every vertex $v \in X^*$ it holds that $\deg(v) \geq d_{c,OPT}$.

Proof of intermediate corollary: Let X^* and $d_{c,OPT}$ be defined as before. Then,

by the optimality of X^*, we get for any $u \in X^*$:

$$\begin{aligned}
d_{c,OPT} = \frac{w(X^*)}{|X^*|-1} &\geq \frac{w(X^*) - \deg_{X^*}(u)}{|X^*|-2} \\
\frac{w(X^*)\cdot(|X^*|-2)}{|X^*|-1} &\geq w(X^*) - \deg_{X^*}(u) \\
\deg_{X^*}(u) &\geq w(X^*) - w(X^*)\cdot\frac{(|X^*|-1)-1}{|X^*|-1} \\
&= w(X^*)\cdot(1-1+\frac{1}{|X^*|-1}) \\
&= \frac{w(X^*)}{|X^*|-1} = d_{c,OPT}.
\end{aligned}$$

This concludes the proof of the intermediate corollary.

Let X be the subgraph just before the modified algorithm removes the first vertex $w \in X^*$. By the definition of the algorithm and the intermediate corollary, all vertices in X have degree $\geq d_{c,OPT}$. Thus, we get:

$$\begin{aligned}
\rho_c(X) &\geq \frac{\frac{1}{2}|X|\cdot d_{c,OPT}}{|X|-1} \\
&= \frac{\frac{1}{2}(|X|-1)\cdot d_{c,OPT}}{|X|-1} + \frac{\frac{1}{2}\,d_{c,OPT}}{|X|-1} \\
&= \frac{1}{2}\,d_{c,OPT} + \underbrace{\frac{1}{2}\,\frac{d_{c,OPT}}{|X|-1}}_{>0} > \frac{1}{2}\,d_{c,OPT}.
\end{aligned}$$

□

Approximating the α-LSP with Charikar

While Section 3.2 highlights the conceptual problem of any approximation algorithm for the α-LSP, a specific algorithm such as Charikar might nonetheless exhibit theoretical guarantees when applied repeatedly during the execution of CREP.

However, we construct the following example to show that this is not the case here. Let $\alpha = 6$ and $k = 4$. Initially, the graph corresponding to the communication network consists of two disconnected subgraphs X and Y. X has 3 vertices u, v, w which are connected in a circle by 3 edges of weight 4, Y has 7 vertices $z_1, \ldots, z_7$ which are connected in a circle by 7 edges of weight 5. We further assume that all nodes are singleton components. For the sake of completeness, let $\mathcal{P}_0 = \{\,\{v, z_1, z_4, z_6\}, \{w, z_2, z_5\}, \{u, z_3, z_7\}\,\}$ be the initial partitioning fulfilling the invariant that each edge has both endpoints in different partitions. Hence, at time $t = 0$ we have $\rho_{c,0}(X \cup Y) = 47/9 < \alpha$, $\rho_{c,0}(Y) = 35/6 < \alpha$ and $\rho_{c,0}(X) = 12/2 = \alpha$. For all other subsets of vertices and their induced subgraph $\rho_{c,0} < \alpha$. Thus, only X is mergeable at this time. Charikar does not find it, however, as it

starts by removing vertices from X since they have a degree smaller than that of any vertex in Y. We consider the following request sequence σ:

$$\sigma_i = (u, v),\ i \in [4]$$
$$\sigma_5 = (u, w)$$
$$\sigma_6 = (v, w)$$

The first five requests only affect the respective edge weights and densities, but not the fact that X is the only mergeable subgraph and that Charikar does not find it. When the edge weight corresponding to the sixth request is increased to 5 the situation becomes interesting. While $X \cup Y$ is still not mergeable ($\rho_{c,6}(X \cup Y) = 53/6 < \alpha$), X is not the only mergeable subgraph. Note that we added exactly 6 weight to X and thus can take any vertex $z \in Y$ to obtain $\rho_{c,6}(X \cup \{z\}) = 18/3 = \alpha$. Consider Charikars execution. At this point in time w is not trivially the first node to be removed by Charikar. With the minimal degree in the network being 10, one of the nodes $w, z_1, \ldots, z_7$ must be removed first. Assume without loss of generality that z_1 is selected, and that the remaining nodes of $Y \setminus \{z_1\}$ are removed in increasing order of their index. Charikar then finds the mergeable subgraph $X \cup \{z_7\}$.

In the setting of Lemma 1 we may choose $c = \{u, v, w, z_7\}$, $g = 2$, $S_1 = \{\{u\}, \{v\}, \{w\}\}$, and $S_2 = \{\{z_7\}\}$. However, since $com(S_1, S_2) = 0$, no cost can be attributed to OPT. Hence, we face the problem of merged component sets not being well connected, as discussed in Section 3.2.

3.4 Charikar on Connected Components (CC)

We identify two theoretical weaknesses in the NAIVE strategy discussed in Section 3.3. The first stems from the fact that at each call to Charikar all n vertices have to be processed. By the definition of Charikar, initially all considered nodes have to be ordered with respect to their degree. Additionally, upon removing a vertex, its neighbors' degrees are changed and thus the remaining vertices have to be brought back in order.

The second weakness is more conceptual. While we cannot prove it, we have the intuition that, when trying to solve the α-LSP, Charikar performs especially bad on disconnected graphs. Together with the fact, that an exact CREP always merges connected component sets (see Lemma 2), this seems like a suboptimal situation.

Therefore, as opposed to the NAIVE strategy, we restrict the execution of Charikar to the connected component of the most recent request. This modification only affects line 8 in Algorithm 1. The routine for constructing the connected component with a single vertex as seed is depicted in Algorithm 4.

Worst-case complexity of Algorithm 4. In the worst case G is a complete graph with every vertex having an unweighted degree of $n-1$. Thus, G constitutes a single connected component and all n vertices and $(n \cdot (n-1))/2$ edges are

Algorithm 4: explore_CC

input : graph G, edge $e = (u, v)$
output: graph X

```
1 Initialize graph X with the vertex u.
2 while there exists a vertex a ∈ G \ X that is reachable from any vertex in
    X do
3 |   Add a to X.
4 |   Add all edges from G to X that connect a with vertices in X.
5 end
```

processed. Assuming a constant effort per instruction, we get a complexity of $(n \cdot (n-1))/2 \in \mathcal{O}(n^2)$.

3.5 Charikar on the Hop-Neighborhood (HOP)

From a theoretical perspective, the CC strategy from Section 3.4 is already a major improvement over NAIVE. However, as the connected components can potentially become very large, we propose a different heuristic for restricting the relevant substructure where Charikar is executed upon: the h-hop neighborhood of the most recent request. We define the variant with $h = 2$ as HOP.

Upon receiving an inter-partition request r_t, we construct the immediate neighborhood such that no vertex is no further than h hops from the edge corresponding to r_t. Thereby, the hop-distance of an edge $e = (i, j)$ to a vertex v is defined as the minimum of the hop-distances between i and v, and j and v. The corresponding routine is outlined in Algorithm 5. Similarly to CC, only line 8 in Algorithm 1 requires modification.

While it is still possible for the h-hop neighborhood to become large, it is intuitively less likely than for the connected component to do so.

Algorithm 5: explore_HOP

input : number of hops h, graph G, edge $e = (u, v)$
output: graph X

```
1  Initialize graph X with the vertex u.
2  Initialize queue q with u and v.
3  while q not empty do
4  |   Pop a as the first element in q.
5  |   Add a to X.
6  |   Add all edges from G to X that connect a with vertices in X.
7  |   if a is reachable from e in less than h hops then
8  |   |   Add each direct neighbor of a in G \ X, that is not in q, to q.
9  |   end
10 end
```

Worst-case complexity of Algorithm 5. In the worst case G is a complete graph with every vertex having an unweighted degree of $n-1$. Then the `while` loop is entered n times and in each iteration all $n-1$ edges of a are checked (and some also added to the result graph). Assuming a constant effort per instruction, we get a complexity of $(n-1) \cdot (n-1) \in \mathcal{O}(n^2)$.

3.6 Charikar and Greedy Exploration (GREEDY)

In this section we study a third approach, called GREEDY, for reducing the size of the subgraph passed on to Charikar. As opposed to CC and HOP, we now put a strict limit on the number of vertices considered for merging. Let this limit be denoted as g. For GREEDY we generally set $g = 2 \cdot k$.

Let r_t be an inter-partition request, and let . We initialize the subgraph S with the edge e corresponding to r_t and its two incident vertices. Then, from the set of vertices directly connected to S, we repeatedly add the vertex of maximum degree (with respect to the entire graph) until one of the following conditions is fulfilled: (1) there are no more vertices left in the connected component of e, that have not already been added to S, or (2) S encompasses g vertices. Charikar is then called on this constructed subgraph S. Algorithm 6 depicts the greedy exploration procedure. Only line 8 in Algorithm 1 requires modification to incorporate the GREEDY heuristic.

The strict limit on the subgraph size may improve the run time in practice. However, as the algorithm requires to order elements, there will be some form of priority queue involved. This could negatively impact the run time in the case of large connected components, since a lot of vertices would have to be ordered.

Furthermore, the resulting subgraph S may not represent a local neighborhood. It could potentially be a set of vertices connected in a line. In contrast, HOP assures that the distance of any two points in the computed neighborhood is no greater than $2h$. Thus, it enforces more locality. Due to this shortcoming, GREEDY might in general be less effective than HOP.

Worst-case complexity of Algorithm 6. In the worst case G is a complete graph with every vertex having an unweighted degree of $n-1$. Then all vertices are inserted into the priority queue at a cost of $O(\log n)$ each. The `while` loop is entered h times and in each iteration all $n-1$ edges of a are checked (and some also added to the result graph). Assuming a constant effort per instruction, we get a complexity of $n \cdot \mathcal{O}(\log n) + h \cdot (n-1) \in \mathcal{O}(n \log n + hn)$.

3.7 Decaying Edge Weights

The heuristics discussed in the previous sections aim to directly reduce the initial size of the component set considered for merging. In addition, we propose to combine them with the concept of decaying edge weights. We call this heuristic *aging*. Hereby, more recent requests are considered to be of higher importance when making merge decisions. There are two motivations behind this approach:

Algorithm 6: explore_GREEDY

input : max number of vertices g, graph G, edge $e = (u, v)$
output: graph X

```
1  Initialize graph X with the vertex u.
2  Initialize max priority queue pq with u and v, both having key ∞.
3  while pq not empty and |X| < g do
4  |  Pop a as the first element in pq.
5  |  Add a to X.
6  |  Add all edges from G to X that connect a with vertices in X.
7  |  if a is reachable from e in less than h hops then
8  |  |  Add each direct neighbor of a in G \ X, that is not in pq, to pq
   |  |  with the degree in G as key.
9  |  end
10 end
```

1. The more obvious one stems from the fact that we aim to obtain better results in practice and that real network traces feature temporal locality. Restricting our view of the communication graph to the more recent history directly exploits this fact.

2. Additionally, we would like to avoid unnecessary migrations. When the algorithm is in a good state for the current communication pattern it should remain there, even if there are short-lived disturbances. However, these anomalies might be just long enough that our algorithm would decide to migrate. Aging could mitigate this behavior by keeping track of the current weight of components.

To control the strength of aging we define two parameters: the aging factor $\gamma \in (0, 1]$ and the time warp $\lambda \geq 1$. Intuitively, edge weights are decreased every λ requests by multiplication with γ.

More formally, we keep a separate clock t_a for aging that is incremented every λ requests. We need two values per edge e: a weight e_w as well as the time of its last modification e_t (relative to t_a). The value e_w corresponds to the actual weight of e at time e_t. The *aged weight value* of e can then be obtained as

$$w(e) = e_w \cdot \gamma^{t_a - e_t}.$$

This approach does *not* require updating the weight of every edge at every increment of t_a. The standard weight function w is simply extended to implicitly take the two values e_w and e_t as input. Then, when the weight of an edge is accessed for reading or updating, we use the modified weight function and obtain the aged value. In the case of an edge weight update, e.g. upon receiving a new request, we additionally set e_t to the current value of t_a.

Remarks. We refer the reader to Section 6 for a brief discussion of adaptive aging parameters. We also considered the special variant with $\lambda = 1$, i.e. continuous aging as opposed to periodical. However, in order to obtain results comparable to a periodical decay, the parameter γ has to be set very close to 1. Thus, it is more intuitive to focus on the general variant with both, γ and λ, variable.

4 Implementation Notes

In this section we elaborate on the most notable aspects of our implementation. All algorithm-specific code is written in compliance with the C++17 standard. It was designed with multi-platform support in mind. The main algorithm was tested on Debian 9 (gcc 8.3.0) as well as Windows 10 (MSVC 2019). All dependencies are managed via CMake. The source code is available at `https://github.com/martinschonger/aerdg`.

4.1 Modular BRP Framework

We chose a modular approach to facilitate later extension and addition of other strategies for certain parts of the problem. In its essence, our framework consists of three components. The base module (`brp_alg`) provides a basic structure for the other modules and handles user interaction. It also houses meta logic for decaying edge weights. The update module (`comp_mig_alg` and `explore_neighborhood`) decides, based on the latest request, whether a set of communication components is to be merged and also computes which vertices to collocate. CC, HOP and GREEDY belong to this module. Lastly, the partition module (`part_alg`) manages the assignment of vertices to partitions. It is further responsible for performing actual migrations, which includes determining a suitable target partition.

4.2 Efficiently Maintaining Communication Components

To keep track of the communication components we use a custom implementation of a union-find (UF) data structure. While a standard disjoint-set structure fulfills the requirement of constant-time find queries and merges, it provides no means efficiently iterate all elements of a specific component. Therefore, we combine UF it with the concept of doubly-linked lists. In addition to the `parent` and `size` arrays we add the two arrays `next` and `prev`, which store references to the next and previous element in the respective component. This does not impact the complexity of find and merge, but enables linear-time iteration of individual components.

When a component is deleted we are required to keep all edges with only one endpoint in the component. Therefore, we maintain two graphs: the `base_graph` and the `meta_graph`. The vertices of the former graph correspond to physical nodes, while the vertices of the latter represent communication components. Maintaining two graphs allows us to copy and restore edges from the `base_graph` to the `meta_graph` upon deletion of a communication component.

4.3 Charikar

We used the default priority queue from the STL, which is based on a binary heap. Charikar requires a lot of key changes, one per processed edge. As keys only decrease during a single execution, we decided to simply reinsert updated neighbors instead of using a decrease key operation.

As opposed to newly initializing the priority queue for each execution of Charikar, we have also considered maintaining the priority queue across executions. However, with this approach we would still have to extract and insert every node at most calls to Charikar. This is because there are more Charikar executions which do not find a mergeable set compared to the executions that find one.

Apart from the STL priority queue we experimented with two additional variants. First, motivated by the good theoretical complexity of fibonacci heaps, we deployed Boost's [12] `fibonacci_heap` implementation. Second, we implemented a custom bucket queue [39]. Our motivation in this case was that node degrees only decrease during an execution of Charikar, a property which favors the bucket queue. However, both alternatives performed worse than the default priority queue.

4.4 Zero-Overhead Aging

In order to keep the overhead introduced by aging to a minimum, we implemented it in a lazy fashion. We use two global counters, one is increased with every new request and the other every λ requests. For each value that requires decay two properties are stored: the time of its last modification and the aged value at that time. When such value is accessed, either for reading or writing, the aged value is computed based on the current time, the time of its last modification and the actual value back then. In the case of modification, first the actual value at the current time is computed, and then the change is applied.

5 Experimental Evaluation

In addition to the theoretical considerations in Section 3, we empirically evaluate our algorithms on different network traces. We not only compare the proposed heuristics with each other, but also shed some light on the influence of various parameters on the partition quality as well as the run time.

While we started off with traces from different Facebook datacenters [51], these turned out to be unsuitable for optimizing our algorithms due to their lack of temporal structure and only moderate non-temporal structure. Therefore, we mainly use two traces from pFabric [3] and HPC [17] in this section.

All experiments were performed on a Laptop with an Intel i7-8550U CPU and 16.0 GB of Memory, running Microsoft Windows 10 Pro with WSL Debian 9.

5.1 Dataset

As shown in Table 1, our data set consists of 3 traces. Each represents a different type of network traffic. Before we can accurately describe the traces, we need to introduce the terms *temporal and non-temporal structure* and *temporal and non-temporal complexity*. Temporal structure and complexity reflect the burstiness of the traffic pattern. A high burstiness corresponds to high temporal structure and low temporal complexity. Non-temporal structure and complexity give information about the skewiness of the network traffic. A uniform distribution corresponds to a high temporal complexity and a low temporal structure.

The pFabric trace was generated by a NS2 simulation with 50 percent network load [3]. It has rather high temporal and very low non-temporal structure. HPC represents a MPI trace of an exascale multigrid application on a high performance compute cluster [17]. This trace comes with medium temporal and non-temporal structure. The third entry corresponds to a rack-level trace from a Facebook data center running hadoop. Partly due to being sampled, it features almost zero temporal structure. The non-temporal complexity is about the same as for HPC. Note that while pFabric and HPC have 144 and 1,024 distinct communication endpoints respectively, the Facebook data has 27,358 endpoints. This, together with its low structure and the resulting few migrations, are the main factors why the algorithms have much higher run time on this trace and also the reason we only include one experiment with this traffic data.

The temporal and non-temporal complexity of the different traces is best compared in the form of a complexity map. We refer the reader to Figure 2 in [7], which includes all above traces.

Type	Trace	Sources	Dest.	Distinct vertices	Requests
pFabric [3]	5	144	144	144	30,000,000
HPC [17]	multigrid	1,024	1,024	1,024	17,947,800
Facebook [51]	hadoop	365	27,358	27,358	302,265,398

Table 1: Trace meta data

5.2 Default Settings and Baseline

For most experiments, we use $\alpha = 6.0$, $k = 32$, $augm = 2.1$, and either $\gamma = 1.0$ and $\lambda = 1$, or $\gamma = 0.7$ and $\lambda = 400$. Every experiment is performed 10 times with z requests each. Two consecutive runs are non-overlapping and z depends on the specific setting. Unless specified otherwise, we use $z = 500{,}000$ for pFabric and $z = 100{,}000$ for HPC. We always skipped the first 1,500,000 requests, regardless of the specific trace. For example, when $z = 100{,}000$, the starting points for the 10 runs are $1.5M, 1.6M, 1.7M, \ldots, 2.4M$.

As a dynamic baseline we use our NAIVE algorithm due to the lack of a better alternative, e.g. an (expensive) exact algorithm for the BRP. We also compare

the algorithms against the static partitioning produced by METIS' state-of-the-art PartGraphRecursive [30, 31] routine, which is based on multilevel recursive bisection. The time complexity of this implementation is $\mathcal{O}((n+m) \cdot \log k)$ [29].

While we experimented with ParMETIS' V3_AdaptiveRepart [32, 56, 57] routine, as mentioned in section 2, this did not yield consistent results and is hence not used as baseline.

5.3 Derivation of HOP + Aging

In this section we illustrate the process of engineering our HOP algorithm with aging. Starting from CREP in its original form, we add improvements step-by-step and show their effects on run time and partition quality.

NAIVE. CREP requires solving the α-LSP upon every inter-partition request and is therefore presumed impractical. Due to its simplicity, we initially employ Charikar with the goal to approximate this problem within feasible time. Unfortunately, we cannot compare NAIVE's partition quality with that of the original CREP since we are not aware of an exact solver for the α-LSP.

CC. We have seen in the theoretical discussion in Section 3 that the NAIVE approach, where Charikar is called on the entire graph, is suboptimal. This is mainly because (1) the solution of the α-LSP is always connected in the setting of CREP, and (2) Charikar seems to perform particularly bad on disconnected inputs for our purpose. We directly address both aspects by restricting the execution of Charikar on the connected component of the most recent request.

While this change yields a drastic reduction of the communication costs on pFabric, the migration costs on both pFabric and HPC are unacceptably high (higher than the respective communication costs). See columns 2 and 3 in Figures 2(a) and 4(a). The run time on pFabric is reduced by a factor of at least 2 (Figure 2(a)). On HPC, CC's run time is slightly higher than that of NAIVE (Figure 5(a)). We conclude that, at least run time-wise, we are going in the right direction but there is still room for improvement.

HOP. We suspect that the connected components can become quite large and that the run time can be greatly reduced by further restricting the input for Charikar. Hence, we propose the more resilient heuristic HOP, which limits the initial subgraph considered for merging to the 2-hop neighborhood of the latest request.

Figures 3(a) and 5(a) confirm that this heuristic does indeed consistently improve the run time compared to CC, with the effect being stronger on the HPC trace. However, HOP migrates way too much on the pFabric data set. Interestingly, the communication costs of CC and HOP are nearly the same, indicating that HOP's many migrations are not helpful (Figure 3(a)). Thus, next we primarily need to address partition quality rather than efficiency.

Aging. High and unjustifiable migration costs might be attributed to an incorrect view of the communication network. It seems as though we are making decisions on the wrong ground truth. While the BRP, per se, does not incorporate the notion of a fixed but unknown underlying request distribution, the structure of production-level network traffic (see Section 2.2) suggests that this assumption does not accurately reflect reality. However, presuming a fixed distribution over the entire request sequence would be too inflexible either. Instead, a continuously changing distribution should be expected. We propose aging as a means to restrict our view of the communication network to the most recent period of time and, thus, implicitly model a dynamic request distribution.

The effects on the partition quality are impressive. We compare the left and right plots in Figures 2 and 4. Aging with $\gamma = 0.7$ and $\lambda = 400$ reduces the migration costs of HOP on pFabric by a factor of more than 5, and on HPC by nearly a factor of 3. Interestingly, the communication costs on pFabric are virtually unaffected, which confirms our hypothesis of needless migrations from the previous paragraph on HOP. Communication costs on HPC are only increased by 20 percent. These improvements come with a worse run time (Figures 3 and 5). We suspect that this is due to fewer merge actions being performed and, hence, fewer edges are removed from the graph, which means more edges tend to be processed at each inter-partition request.

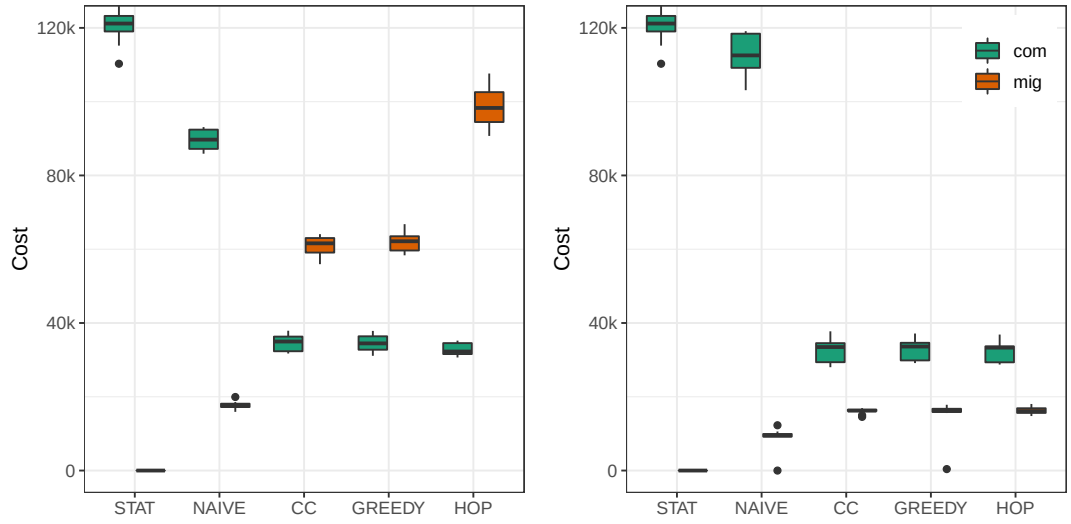
We conclude that the combination of HOP with moderate aging (short: HOPage) unites exceptional partition quality and attractive run times, compared to the other configurations, which generally only perform well in at most one of the two categories.

5.4 An Overview

In the above sections we compare two algorithms at time and eventually arrive at HOPage. Now we put the algorithms STAT, NAIVE, CC, GREEDY, and HOP into perspective by comparing them on pFabric and HPC both with and without aging. Therefore, we consider Figures 2, 4, 3, and 5.

We look at the costs first. One of the most interesting observations is the following. On both traces, independent of aging, the algorithms CC, GREEDY and HOP have nearly identical communication costs. In the setting without aging, the corresponding migration costs differ slightly, while aging, apart from decreasing them significantly, seems to smooth them out as well. This may indicate that (1) many migrations are unnecessary, and (2) the considered neighborhoods feature a very similar *core structure*. By core structure we refer to the area of the neighborhood that is very close to the most recent request. As the subgraphs themselves are unlikely to be of equal size in all three cases (this is supported by the high run time of CC on HPC shown in Figure 5(b)), it also suggests that Charikar's execution is more desirable on connected structures: vertices far from the latest request are removed first.

Figures 2 and 4 also beautifully highlight the trace characteristics. This can be seen by looking at STAT in the leftmost column. The high cost on pFabric stems from low non-temporal structure. Lower cost on HPC is due to it being skewed



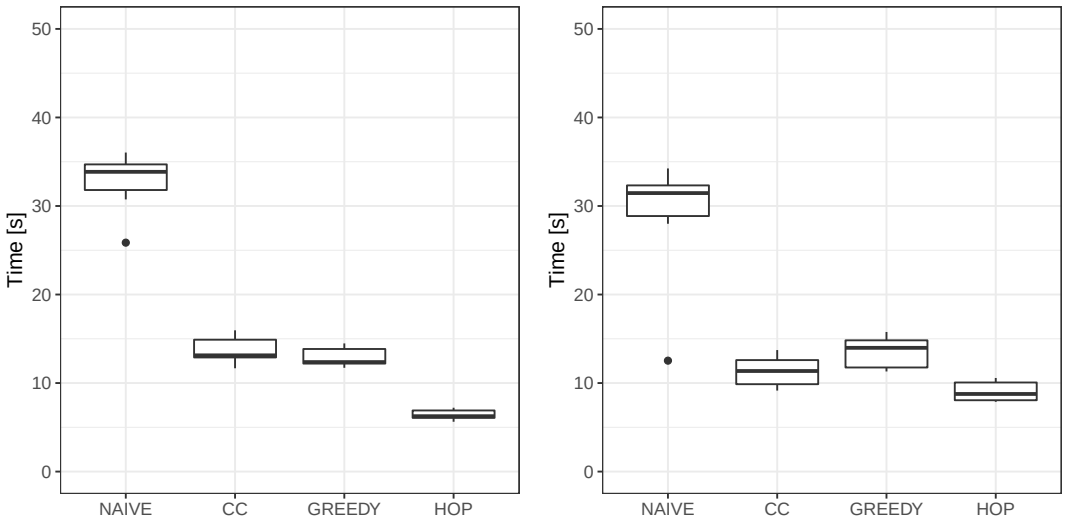
(a) Without weight decay (b) With weight decay ($\gamma = 0.7, \lambda = 400$)

Figure 2: Communication and migration costs of the four algorithms NAIVE, CC, GREEDY and HOP on the pFabric trace.

and of moderate temporal complexity. Together with the fact that pFabric has very high temporal structure, these properties explain why the performance of our algorithms is better on pFabric.

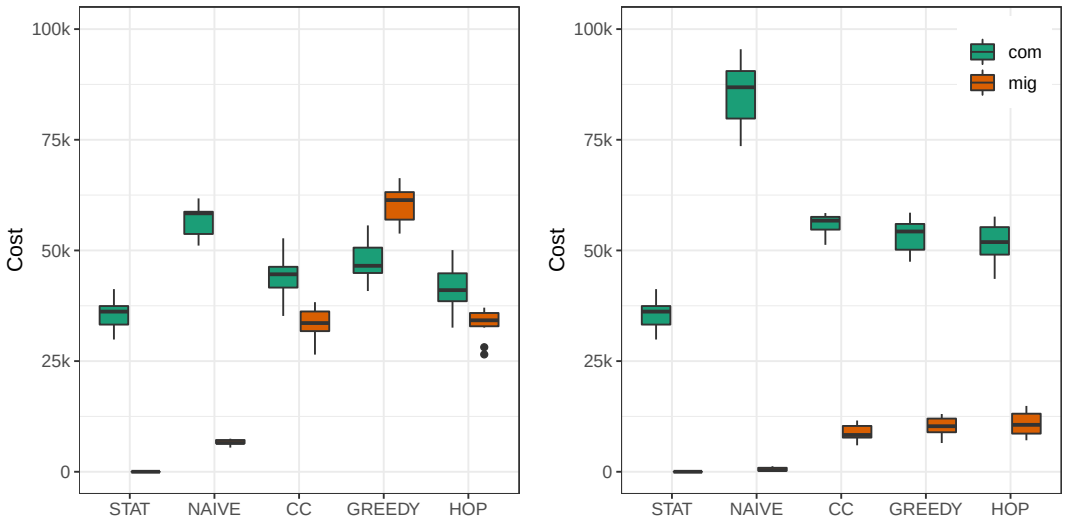
Run time-wise, the overall results on both traces are as expected: the more we restrict the search space for Charikar, the lower the run time (Figures 3 and 5). However, there is one exception. The run time of CC is higher than that of NAIVE on HPC. As the search space of CC is at most that of NAIVE and the costs of CC are significantly lower than for NAIVE, a possible explanation is that the exploration procedure does add some overhead.

Further, note that NAIVE performs consistently worse than the other three algorithms. Its communication costs and run times are considerably higher. Therefore, we restrict ourselves to the comparison of CC and HOP in subsequent sections. GREEDY is not further considered due to reasons discussed in Section 3.6.



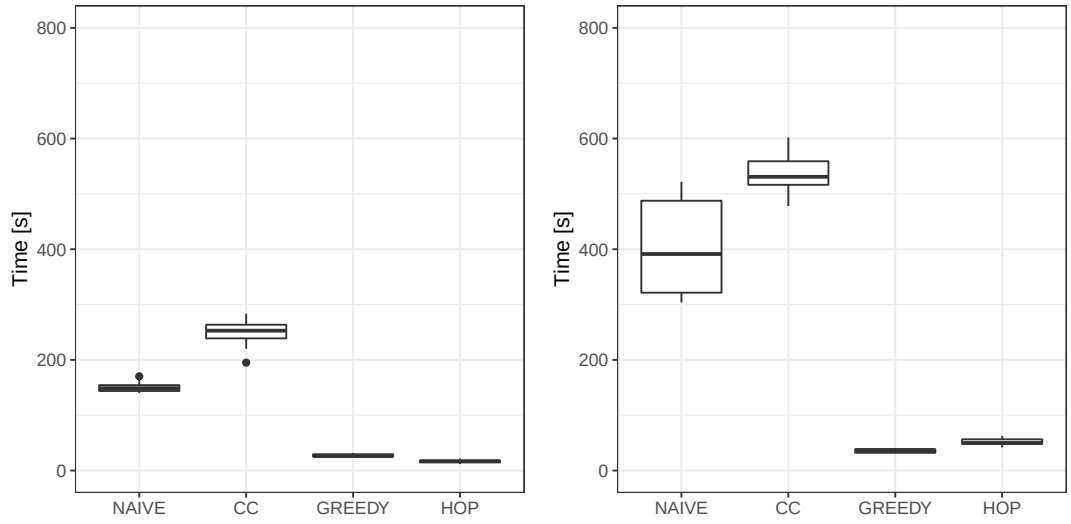
(a) Without weight decay (b) With weight decay ($\gamma = 0.7, \lambda = 400$)

Figure 3: Run times of the four algorithms NAIVE, CC, GREEDY and HOP on the pFabric trace.



(a) Without weight decay (b) With weight decay ($\gamma = 0.7, \lambda = 400$)

Figure 4: Communication and migration costs of the four algorithms NAIVE, CC, GREEDY and HOP on the HPC trace.



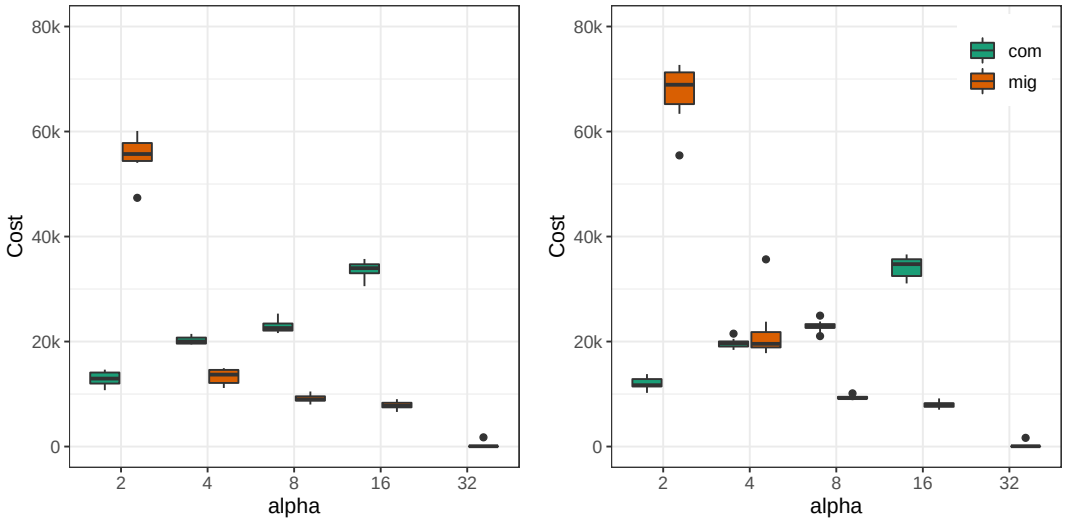
(a) Without weight decay (b) With weight decay ($\gamma = 0.7, \lambda = 400$)

Figure 5: Run times of the four algorithms NAIVE, CC, GREEDY and HOP on the HPC trace.

5.5 What Influence does α Have?

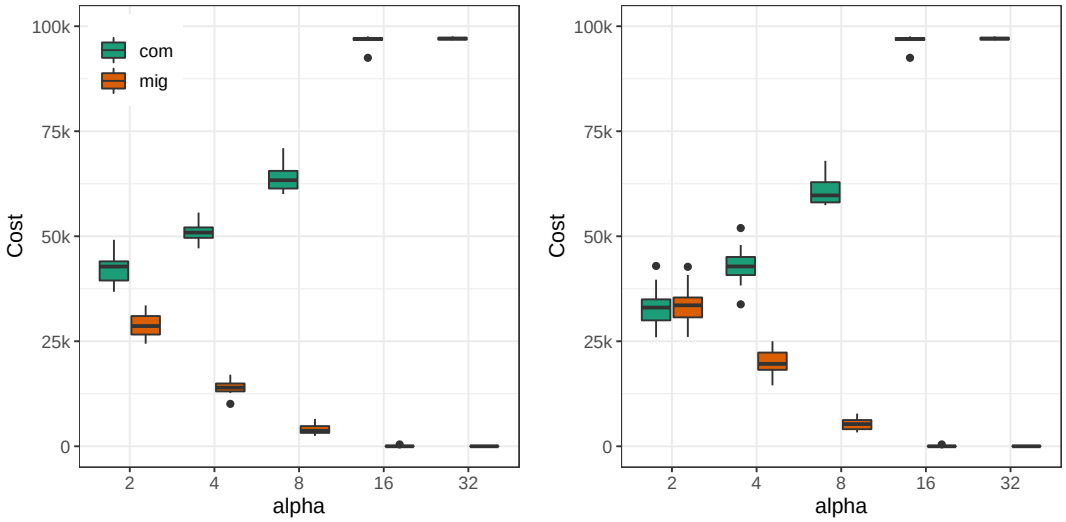
In this section we study the effects of varying α, a central problem-specific parameter. Consider Figures 6 and 7. As expected, increasing alpha results in fewer migrations for CC and HOP on both traces: a larger alpha means a higher threshold for component sets to become mergeable. On pFabric, the greatest reduction in migration costs occurs when moving from $\alpha = 2$ to $\alpha = 4$. At $\alpha = 32$, both algorithms do not perform any migrations on pFabric, whereas this phenomenon can be observed already for $\alpha = 16$ in the case of HPC.

Interestingly, increasing α does barely affect the run time on pFabric (except $\alpha = 32$). This might be due to the great locality of the trace, resulting in consistently feasible neighborhood sizes. For HPC on the other hand, run time increases steadily with α for CC, but increases only slightly for HOP. We can attribute this to the different structural properties of HPC and therefore larger connected components. Also note that for the HPC trace we only used 100,000 requests (compared to 300,000 for pFabric), but the run times are already significantly higher.



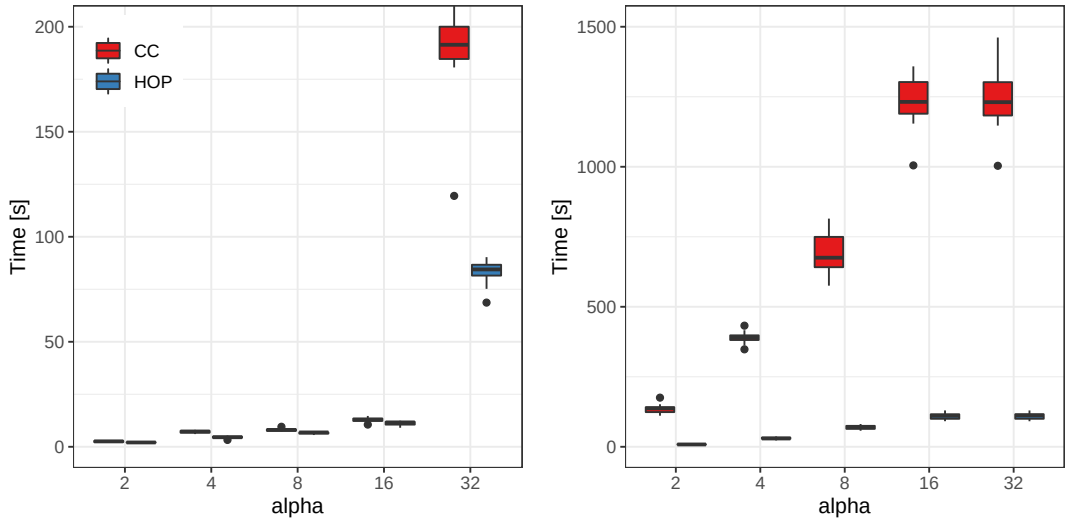
(a) CC (b) HOP

Figure 6: Communication and migration costs of CC and HOP on the pFabric trace for different values of α. Weight decay is set to $\gamma = 0.7$ and $\lambda = 400$. $z = 300{,}000$. For $\alpha = 32$ the median communication cost of CC and HOP maxed out at around 230k.



(a) CC (b) HOP

Figure 7: Communication and migration costs of CC and HOP on the HPC trace for different values of α. Weight decay is set to $\gamma = 0.7$ and $\lambda = 400$.



(a) pFabric, $z = 300{,}000$ (b) HPC

Figure 8: Run times of CC and HOP for different values of α. Weight decay is set to $\gamma = 0.7$ and $\lambda = 400$.

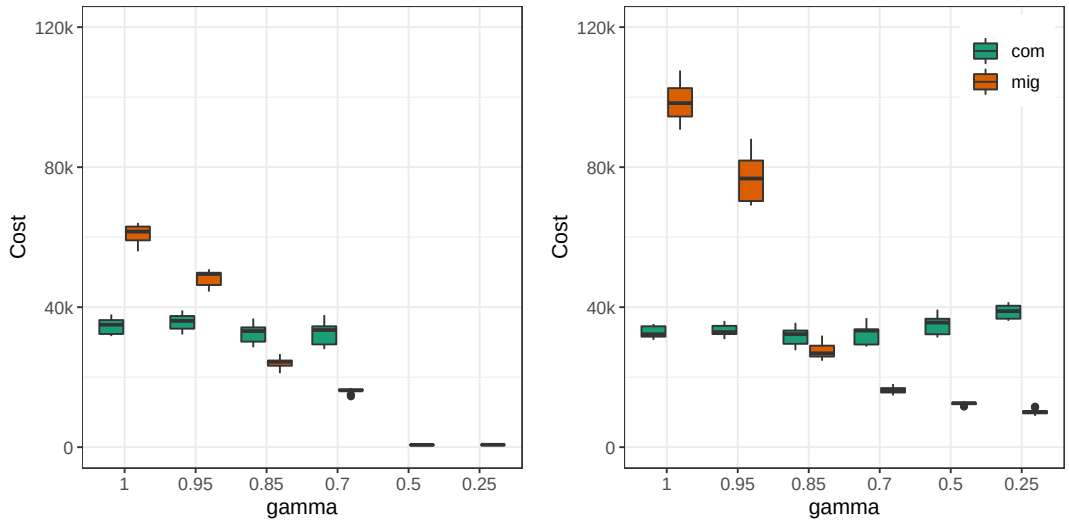
5.6 The Effects of Aging

Up to now we only considered the settings of (1) zero aging, and (2) aging with $\gamma = 0.7$ and $\lambda = 400$. Thus, the effects of varying the decay parameters remain to be explored. For the sake of clarity, we study the influence of γ and λ separately.

Varying gamma. We consider Figures 9 and 10. On both traces, increasing γ reduces the number of migrations, up to the point where the algorithms do not migrate at all (this can be observed for CC on pFabric with $\gamma \in \{0.5, 0.25\}$ for example). This is precisely what we expected. However, especially on pFabric, the migration costs decrease at a higher rate than the communication costs increase. In fact, the latter are only moderately affected when varying γ, but again, only up to a tipping point where the algorithms do not migrate at all. CC appears to reach this state sooner, somewhere between $\gamma = 0.7$ and $\gamma = 0.5$. The costs for both algorithms are nearly identical on HPC.

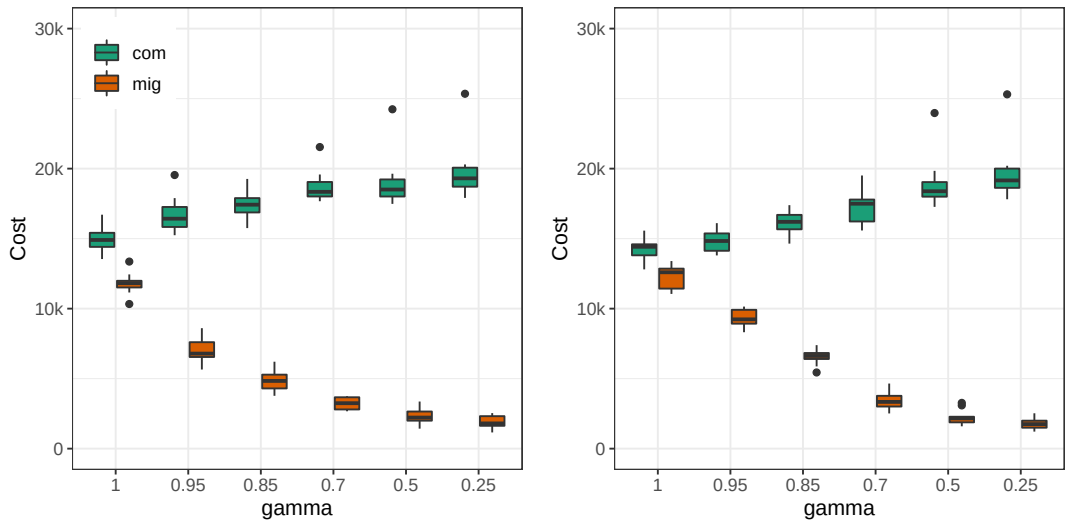
The run times of CC and HOP generally increase with γ (Figure 11). This could be due to the existence of more edge artifacts at larger γ. Edges with very low weight are not likely to be removed and thus clutter up the graph. On HPC, they increase stronger for CC than for HOP. Also, up to the tipping point between $\gamma = 0.7$ and $\gamma = 0.5$, the run time of CC seems to slightly decrease on pFabric. The reason behind this trend is unclear.

Note that, especially in Figure 9, setting γ at around 0.8 or 0.7 appears to achieve a good trade-off between communication and migration costs. This is one of the motivations behind our default choice of $\gamma = 0.7$ (together with $\lambda = 400$).



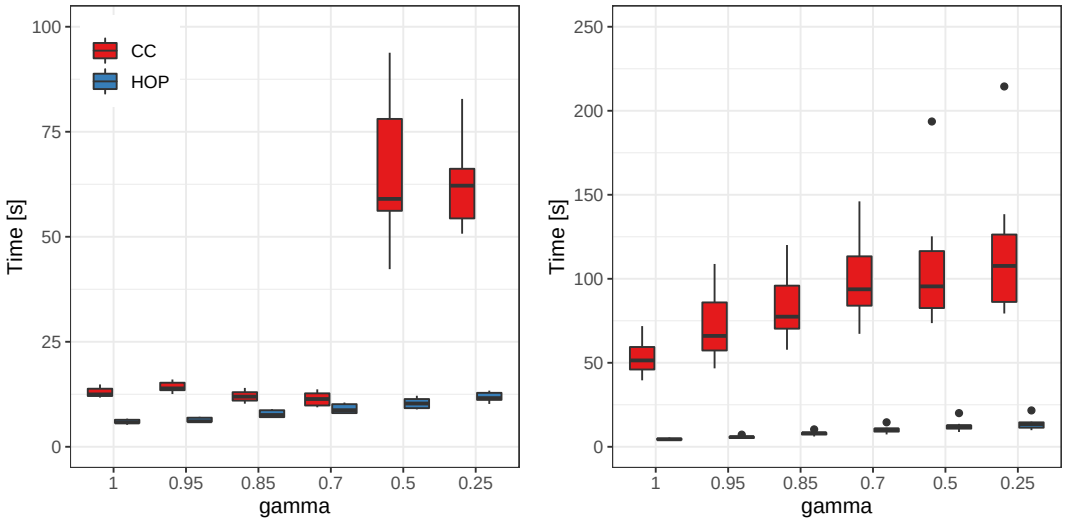
(a) CC (b) HOP

Figure 9: Communication and migration costs of CC and HOP on the pFabric trace for different values of γ. $\lambda = 400$. For $\gamma \in \{0.5, 0.25\}$ the median communication cost of CC and HOP maxed out at around 260k.



(a) CC (b) HOP

Figure 10: Communication and migration costs of CC and HOP on the HPC trace for different values of γ. $\lambda = 400$, $z = 30{,}000$.



(a) pFabric (b) HPC, $z = 30{,}000$

Figure 11: Run times of CC and HOP for different values of γ. $\lambda = 400$.

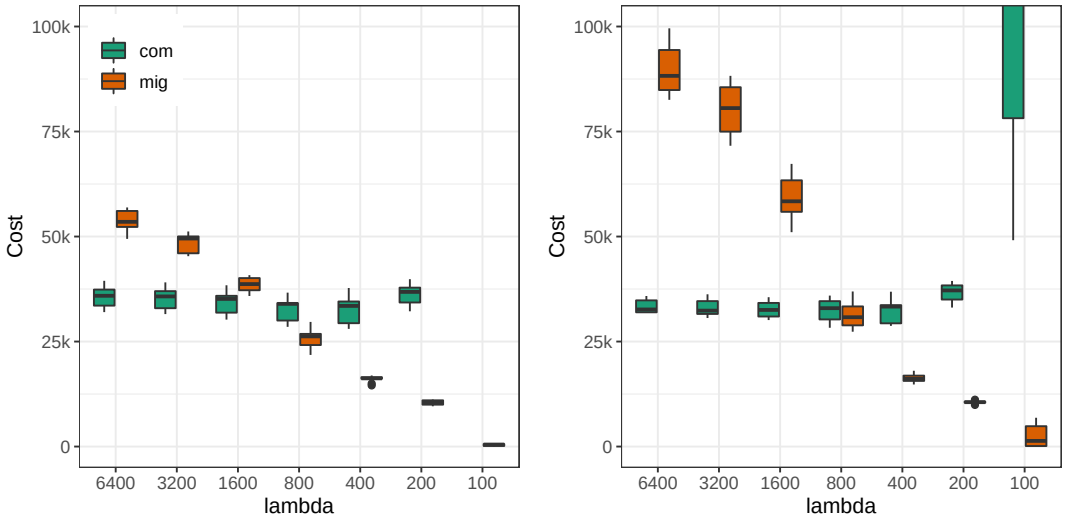
Varying lambda. Setting $\gamma = 0.7$ and varying λ, we observe results very similar to the ones above where we varied γ: (1) with increasing λ the algorithms migrate less, (2) there exists a tipping point from which onward no migrations are performed (between $\lambda = 200$ and $\lambda = 100$ for both algorithms), and (3) the migration costs decrease at a much higher rate than the communication costs increase. Furthermore, we observe unexplainable initially declining run times of CC on pFabric. See Figures 12, 13 and 14 for details.

When looking at Figure 12 we see that our default choice of $\lambda = 400$ is reasonable (in combination with $\gamma = 0.7$.

5.7 1, 2, 3 or 4 Hops?

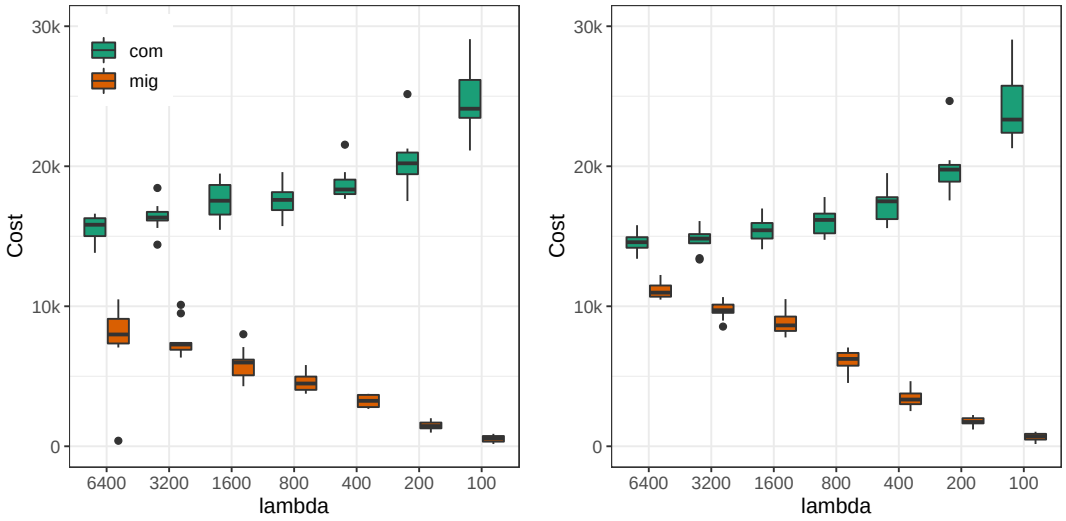
In all other experiments, whenever HOP is used it is configured with $h = 2$. Thus, the 2-hop neighborhood of the most recent request is passed as input to Charikar, which in turn decides whether to merge a subset thereof. What happens when we restrict the considered neighborhood even further by setting $h = 1$? Or, what if we allowed more than 2 hops?

The effects on the partition quality are depicted in Figure 15. We observe that the communication costs remain surprisingly constant when increasing the number of hops, independent of the trace. Taking a closer look, the communication costs are slightly increasing, whereas the migration costs decrease minimally. Thus, it seems that the more we increase h, the more careful our algorithm becomes when making merge decisions. We further note that the 1-hop neighborhood does not seem to contain enough information to facilitate educated merges. In conclusion, the overall trend of Figure 15 was expected as we recover CC with $h \to \infty$.



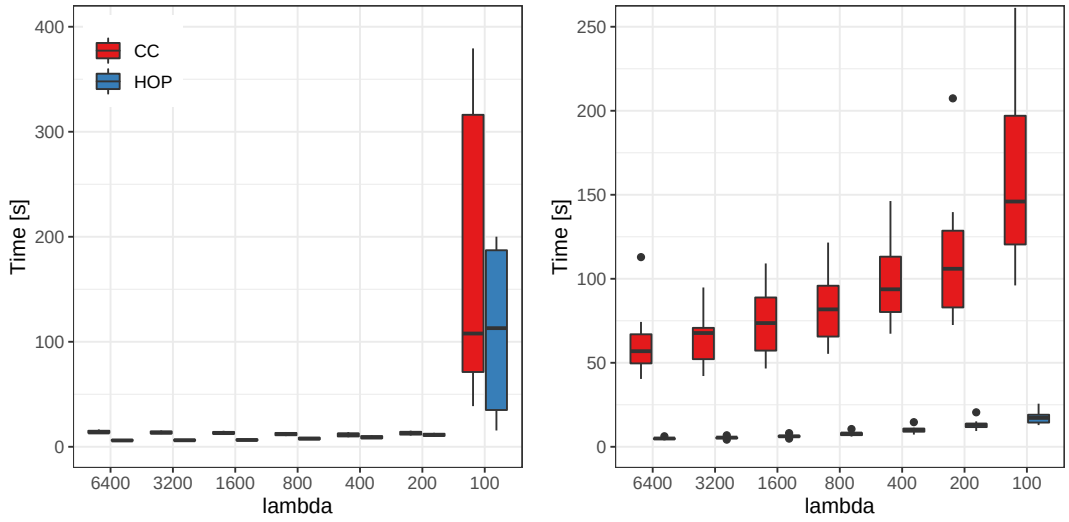
(a) CC (b) HOP

Figure 12: Communication and migration costs of CC and HOP on the pFabric trace for different values of λ. $\gamma = 0.7$. For $\lambda = 100$ the median communication cost of CC maxed out at around 280k, and the cost of HOP varied greatly from 80k to 360k.



(a) CC (b) HOP

Figure 13: Communication and migration costs of CC and HOP on the HPC trace for different values of λ. $\gamma = 0.7$, $z = 30{,}000$.



(a) pFabric (b) HPC, $z = 30{,}000$

Figure 14: Run times of CC and HOP for different values of λ. $\gamma = 0.7$. For $\lambda = 100$ the run time of CC varied from 75 to 320 (median 100), and that of HOP varied from 30 to 180 (median 110).

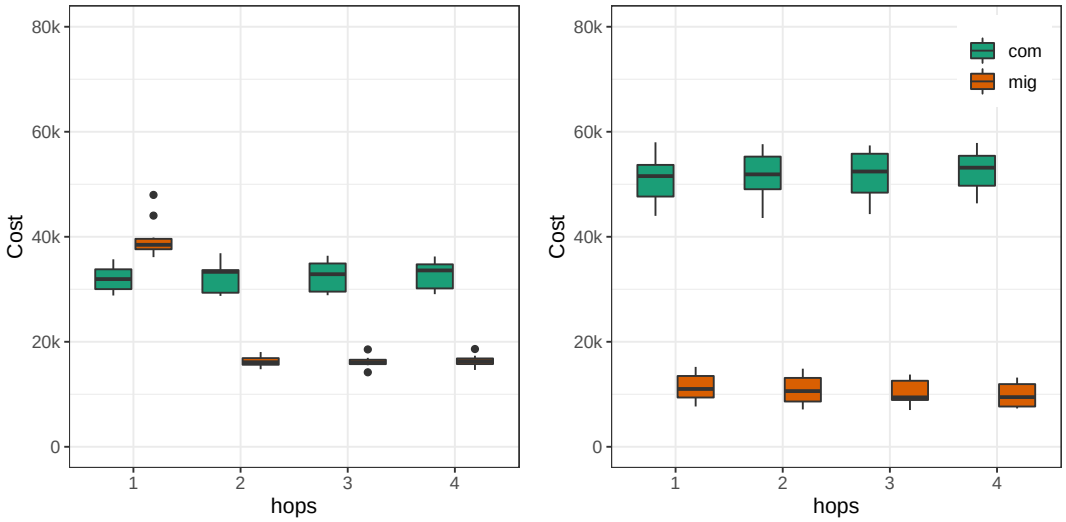
An increase of the run time with larger values for h is not surprising, either (see Figure 16). The plateauing of the run time in Figure 16(a) indicates that the diameter of connected components in pFabric is generally not much larger than 8. No such indication is present for HPC.

Our default setting of $h = 2$ can be justified with Figures 15 and 16. The 2-hop neighborhood strikes a balance between partition quality and efficiency. If speed is of utmost importance, a value of $h = 1$ might also be considered. We refer the reader to the discussion in Section 6 for a different approach to further restricting the neighborhood size.

5.8 Varying the Partition Size

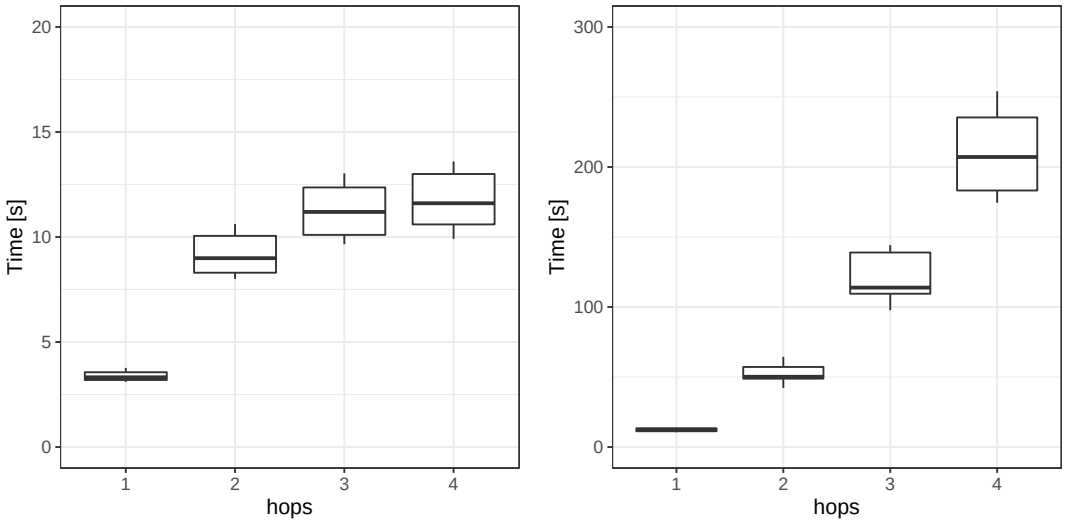
Here we shed some light on the influence of the partition size k on the behavior of our algorithms. As mentioned in Section 2, the case $k = 2$ corresponds to an online maximum matching problem.

Naturally, as k grows, the communication costs decrease (see Figures 17 and 18). However, while k is doubled at each step, our algorithms do not automatically induce half the communication costs. The decrease still seems to roughly happen linearly within most of the parameter space. A general increase of the migration costs was expected, as a larger partition size corresponds to a higher threshold for deleting a component. Thus, communication components can become larger. Interestingly, the migration costs on pFabric are barely influenced by k, showing only a minimal increase. This could stem from the characteristic of the trace itself, but remains to be studied.



(a) pFabric (b) HPC

Figure 15: Communication and migration costs of HOP for different values of h. $\gamma = 0.7$, $\lambda = 400$.

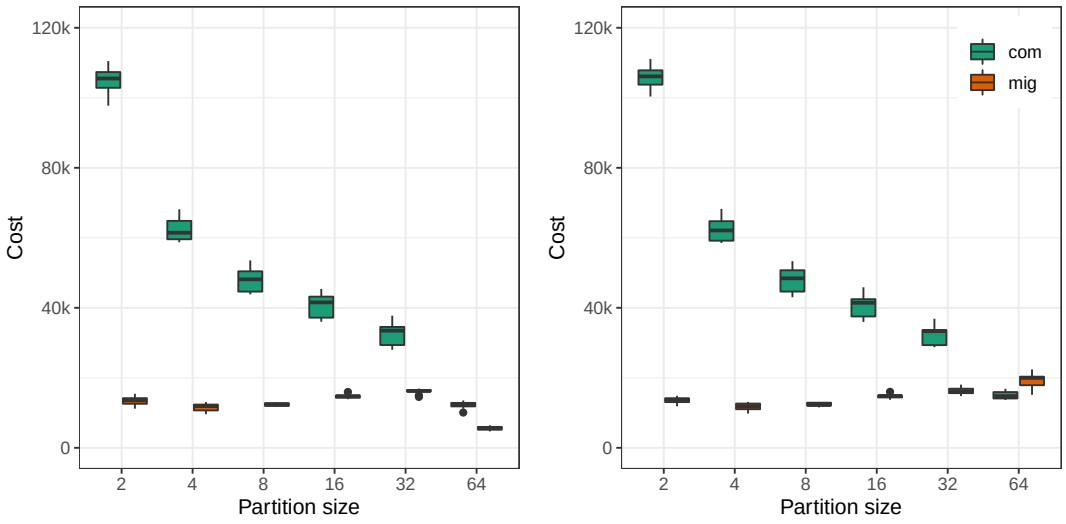


(a) pFabric (b) HPC

Figure 16: Run times of HOP for different values of h. $\gamma = 0.7$, $\lambda = 400$.

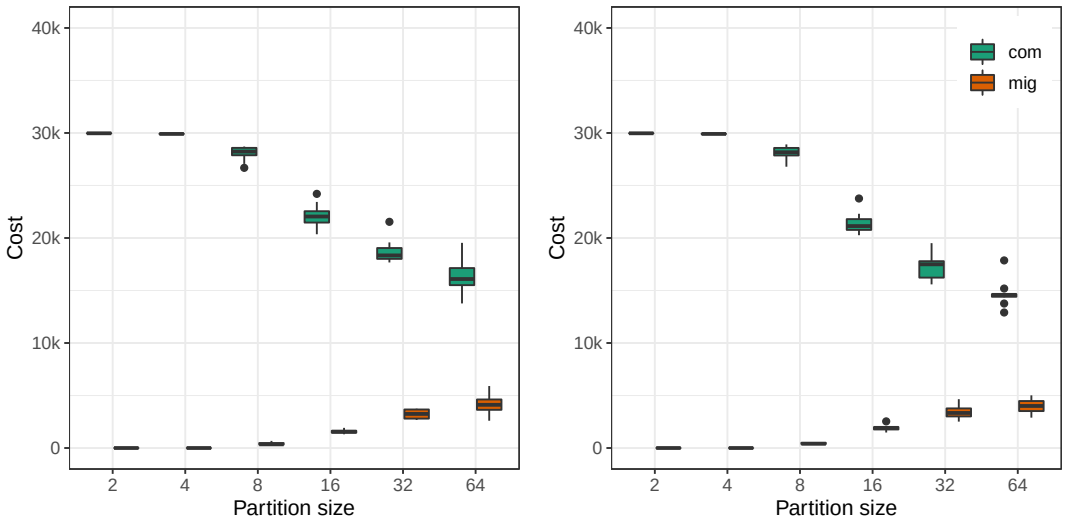
We notice two further phenomena regarding the costs. First, there appears to be a jump of the migration costs between $k = 32$ and $k = 64$ of CC on pFabric. Second, Figure 18 indicates that the communication pattern of HPC mainly features clusters of size greater than 4.

The run time of CC and HOP improves with larger values for k. This was expected as a larger partition size means less inter-partition requests and, thus, fewer neighborhood explorations and fewer calls to Charikar.



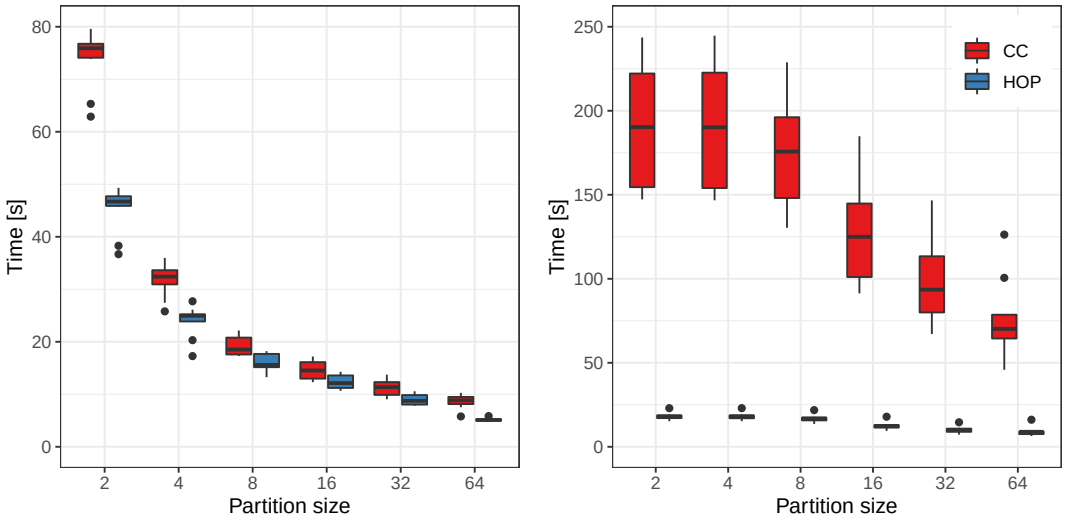
(a) CC (b) HOP

Figure 17: Communication and migration costs of CC and HOP on the pFabric trace for different values of the partition size k. $\gamma = 0.7$, $\lambda = 400$.



(a) CC (b) HOP

Figure 18: Communication and migration costs of CC and HOP on the HPC trace for different values of the partition size k. $\gamma = 0.7$, $\lambda = 400$, $z = 30{,}000$.



(a) pFabric (b) HPC, $z = 30{,}000$

Figure 19: Run times of CC and HOP for different values of the partition size k. $\gamma = 0.7$, $\lambda = 400$.

5.9 HOP on a Large Trace with Low Structure

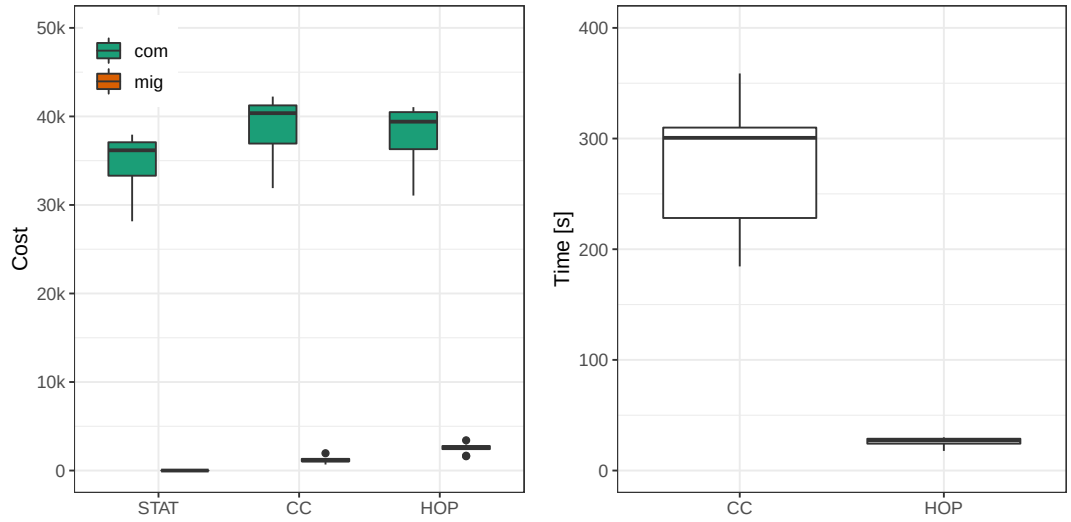
In this section we evaluate our algorithms on the Facebook trace, which exhibits significantly less structure than pFabric and HPC. It has more than 27,000 possible distinct endpoints. Therefore, in Figure 20(a), we observe only very little migration. HOP performs better than CC but cannot beat the static partitioning in terms of costs. The run time of HOP is by a factor of 10 lower compared to CC.

We conclude, that our proposed heuristic is able to improve the results also for less-structured data, particularly with regard to the run time. However, as was expected, traces with little structure appear to constitute less ideal inputs for the BRP.

5.10 Results

Overall, the experiments confirm our theoretical findings and intuitions. Restricting the vertex set considered for merging to the connected neighborhood of the most recent request does indeed result in better partition quality, i.e. lower total cost. While calling Charikar on entire connected components is impractical, combining the greedy algorithm with our HOP heuristic yields promising run times. It seems as though HOP strikes a balance between size restriction and representation capacity.

Single heuristics on their own give improvements only for specific parameter configurations. The experiments show that rather a combination thereof can consistently achieve convincing performance. Our most advanced algorithm, HOPage, constitutes an example for a promising combination. It unites good partition qual-



(a) Communication and migration costs

(b) Run times

Figure 20: CC and HOP on the Facebook trace without aging. $\alpha = 2.0$, $z = 50{,}000$.

ity and attractive run times. For reasonable settings of the parameters, HOPage performs consistently better than or comparable to CC.

As expected, the performance of the algorithms very strongly depends on the characteristics of the trace. We find that there are trace- and algorithm-specific limits on the feasible region for individual parameters, such as α or λ. Based on our current insight, we recommend setting $\gamma = 0.7$ and $\lambda = 400$ as a rule of thumb. However, the question whether there exists a combination that exhibits solid performance on the majority of traces remains open.

6 Conclusion and Future Work

Based on our fundamental understanding of the interplay of CREP, α-LSP and Charikar regarding locality and connectedness of substructures considered for merging, as well as the efficient computation of such, we conclude that combining CREP with Charikar and some heuristic conceptually similar to our HOP strategy is a very promising direction of research for approaching the dynamic balanced graph partitioning problem.

While our algorithms and results are already significant, we identify various areas of potential improvement in future research. Apart from these, the following list also includes interesting properties we consider worth exploring.

- We see much potential in exploring further the combination of HOP and GREEDY. Thereby we would set a first-order limit of 2 or 3 hops, and then greedily add vertices from this initial neighborhood to obtain a second-order subset. This final subset is then passed to Charikar. With this approach we would combine the enforced locality of HOP with the strict size limit of

GREEDY. An essential ingredient for this algorithm to work efficiently is a smart implementation of the greedy logic.

- A fundamental challenge of our algorithm is how to set the parameters right. It would be interesting to study the potential of adaptive aging parameters. In this case, the algorithm could select suitable values for γ and λ based on the current temporal structure of the communication pattern. Depending on factors such as temporal spacing between dense structures and the intra- and inter-connectedness of these structures. However, this seems algorithmically challenging.

- For real-world scenarios lowering the threshold for mergeable component sets could be advantageous. Similarly to the previous idea, this threshold could be set in a dynamic fashion. A related idea is to maintain and incorporate component weights into merge decisions. This could potentially limit unfavorable migrations.

- What remains open is the behavior of our algorithm in a setting with less than $2 + \epsilon$ augmentation. Early experiments have shown that reducing augmentation to below 1.6 results in HOP having problems to find a target partition with sufficient free capacity.

- GREEDY considers vertices in the order of their degree with respect to the entire graph. Alternatively, they could be visited by their degree with respect to the current subgraph under construction. While we have tried to implement this approach, further research is necessary.

- Sometimes, when an optimal target partition cannot be selected because of a predicted small capacity violation, it might be rewarding to consider the eviction of vertices. However, in a setting of changing vertex sets (cf. component merges and deletions) this turned out to be rather complex. See also Section 3.2.

- Implementation-wise, we have early ideas of a parallel realization of our algorithm. Multiple requests would be processed at the same time. All we need to check is whether the set of partitions containing at least one vertex of the mergeable component sets intersect. If yes, synchronization is required. Else, we can continue to the migration step. If two parallel processes select the same target partition, we solve this conflict by diverting the process handling the later request.

- As Avin *et al.* mentioned in [8], randomized algorithms for the BRP constitute an open problem. For example, we might be able to speed up the exploration of the local neighborhood by selecting vertices with probability proportional to their weighted degree, as opposed to sorting them by their degree.

- Instead of defining the notion of a mergeable component set with the CREP-density ρ_c, the well-studied min-cut could be used. In this setting, a subgraph is said to be mergeable if its min-cut lies above a certain threshold. The greatest advantage of this approach is that we need not necessarily find the largest such structure at the first try. Rather, we can solve it in a cascading manner. This is due to the nature of approximations for the min-cut. In fact, this approach may even allow us to preserve some guarantees from CREP.

References

[1] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke, "An o (log k)-competitive algorithm for generalized caching," *ACM Transactions on Algorithms*, vol. 15, no. 1, pp. 1–18, 2019, ISSN: 15496325. DOI: 10.1145/3280826.

[2] S. Albers, "New results on web caching with request reordering," *Algorithmica*, vol. 58, no. 2, pp. 461–477, 2010, ISSN: 0178-4617. DOI: 10.1007/s00453-008-9276-x.

[3] M. Alizadeh *et al.*, "Pfabric," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013, ISSN: 01464833. DOI: 10.1145/2534169.2486031.

[4] K. Andreev and H. Räcke, "Balanced graph partitioning," *Theory of Computing Systems*, vol. 39, no. 6, pp. 929–939, 2006, ISSN: 1432-4350. DOI: 10.1007/s00224-006-1350-7.

[5] C. Avin, M. Bienkowski, A. Loukas, M. Pacut, and S. Schmid, *Dynamic balanced graph partitioning*, 2015. [Online]. Available: http://arxiv.org/pdf/1511.02074v4.

[6] C. Avin, L. Cohen, M. Parham, and S. Schmid, "Competitive clustering of stochastic communication patterns on a ring," *Computing*, vol. 101, no. 9, pp. 1369–1390, 2019, ISSN: 0010-485X. DOI: 10.1007/s00607-018-0666-x.

[7] C. Avin, M. Ghobadi, C. Griner, and S. Schmid, *Measuring the complexity of packet traces*, 2019. [Online]. Available: http://arxiv.org/pdf/1905.08339v1.

[8] C. Avin, A. Loukas, M. Pacut, and S. Schmid, "Online balanced repartitioning," in *Distributed Computing*, ser. Lecture Notes in Computer Science, C. Gavoille and D. Ilcinkas, Eds., vol. 9888, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 243–256, ISBN: 978-3-662-53425-0. DOI: 10.1007/978-3-662-53426-7_18.

[9] C. Avin and S. Schmid, *Renets: Toward statically optimal self-adjusting networks*, 2019. [Online]. Available: http://arxiv.org/pdf/1904.03263v1.

[10] C. Avin and S. Schmid, *Toward demand-aware networking: A theory for self-adjusting networks*, 2018. [Online]. Available: http://arxiv.org/pdf/1807.02935v1.

[11] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th annual conference on Internet measurement - IMC '10*, M. Allman, Ed., New York, New York, USA: ACM Press, 2010, p. 267, ISBN: 9781450304832. DOI: 10.1145/1879141.1879175.

[12] Boost, *Boost c++ libraries*, 2019. [Online]. Available: https://www.boost.org/ (visited on 09/09/2019).

[13] A. Borodin and R. El-Yaniv, *Online computation and competitive analysis*. Cambridge: Cambridge University Press, 1998, ISBN: 0-521-56392-5.

[14] N. Buchbinder and J. Naor, "The design of competitive online algorithms via a primal—dual approach," *Foundations and Trends® in Theoretical Computer Science*, vol. 3, no. 2–3, pp. 93–263, 2007, ISSN: 1551-305X. DOI: 10.1561/0400000024.

[15] A. Buluc, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent advances in graph partitioning*, 2013. [Online]. Available: http://arxiv.org/pdf/1311.3144v3.

[16] M. Charikar, "Greedy approximation algorithms for finding dense components in a graph," in *Approximation Algorithms for Combinatorial Optimization*, ser. Lecture Notes in Computer Science, G. Goos, J. Hartmanis, J. van Leeuwen, K. Jansen, and S. Khuller, Eds., vol. 1913, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 84–95, ISBN: 978-3-540-67996-7. DOI: 10.1007/3-540-44436-X_10.

[17] U. S. DOE, *Characterization of the doe mini-apps*, 2016. [Online]. Available: https://portal.nersc.gov/project/CAL/doe-miniapps.htm.

[18] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965, ISSN: 0008-414X. DOI: 10.4153/CJM-1965-045-4.

[19] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*, V. Bahl, D. Wetherall, S. Savage, and I. Stoica, Eds., New York, New York, USA: ACM Press, 2008, p. 63, ISBN: 9781605581750. DOI: 10.1145/1402958.1402967.

[20] A. E. Feldmann and L. Foschini, "Balanced partitions of trees and applications," *Algorithmica*, vol. 71, no. 2, pp. 354–376, 2015, ISSN: 0178-4617. DOI: 10.1007/s00453-013-9802-3.

[21] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, "Competitive paging algorithms," *Journal of Algorithms*, vol. 12, no. 4, pp. 685–699, 1991, ISSN: 01966774. DOI: 10.1016/0196-6774(91)90041-V.

[22] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *19th Design Automation Conference*, IEEE, 6/14/1982 - 6/16/1982, pp. 175–181, ISBN: 0-89791-020-6. DOI: 10.1109/DAC.1982.1585498.

[23] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing - STOC '74*, R. L. Constable, R. W. Ritchie, J. W. Carlyle, and M. A. Harrison, Eds., New York, New York, USA: ACM Press, 1974, pp. 47–63. DOI: 10.1145/800119.803884.

[24] A. V. Goldberg, "Finding a maximum density subgraph," 1984.

[25] A. Greenberg *et al.*, "Vl2," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication - SIGCOMM '09*, P. Rodriguez, E. Biersack, K. Papagiannaki, and L. Rizzo, Eds., New York, New York, USA: ACM Press, 2009, p. 51, ISBN: 9781605585949. DOI: 10.1145/1592568.1592576.

[26] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "Dcell," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication - SIGCOMM '08*, V. Bahl, D. Wetherall, S. Savage, and I. Stoica, Eds., New York, New York, USA: ACM Press, 2008, p. 75, ISBN: 9781605581750. DOI: 10.1145/1402958.1402968.

[27] M. Henzinger, S. Neumann, and S. Schmid, *Efficient distributed workload (re-)embedding*, 2019. [Online]. Available: http://arxiv.org/pdf/1904.05474v1.

[28] B. M. Kapron, V. King, and B. Mountjoy, "Dynamic graph connectivity in polylogarithmic worst case time," in *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, S. Khanna, Ed., Philadelphia, PA: Society for Industrial and Applied Mathematics, 2013, pp. 1131–1142, ISBN: 978-1-61197-251-1. DOI: 10.1137/1.9781611973105.81.

[29] G. Karypis, *Complexity of pmetis and kmetis algorithms*, 2007. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/node/419 (visited on 09/08/2019).

[30] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[31] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, 1998, ISSN: 07437315. DOI: 10.1006/jpdc.1997.1404.

[32] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing '96*, B. Clayton, Ed., New York, New York, USA: ACM Press, 1996, 35–es, ISBN: 0897918541. DOI: 10.1145/369028.369103.

[33] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970, ISSN: 00058580. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[34] S. Khuller and B. Saha, "On finding dense subgraphs," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. Nikoletseas, and W. Thomas, Eds., vol. 5555, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 597–608, ISBN: 978-3-642-02926-4. DOI: 10.1007/978-3-642-02927-1_50.

[35] R. Krauthgamer and U. Feige, "A polylogarithmic approximation of the minimum bisection," *SIAM Review*, vol. 48, no. 1, pp. 99–130, 2006, ISSN: 0036-1445. DOI: 10.1137/050640904.

[36] R. Krauthgamer, J. Naor, and R. Schwartz, "Partitioning graphs into balanced components," in *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, C. Mathieu, Ed., Philadelphia, PA: Society for Industrial and Applied Mathematics, 2009, pp. 942–949, ISBN: 978-0-89871-680-1. DOI: 10.1137/1.9781611973068.102.

[37] D. Kreutz, F. M. V. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, *Software-defined networking: A comprehensive survey*, 2014. [Online]. Available: http://arxiv.org/pdf/1406.0440v3.

[38] M. Li *et al.*, "Scaling distributed machine learning with the parameter server," in *11th 5USENIX6 Symposium on Operating Systems Design and Implementation (5OSDI6 14)*, 2014, pp. 583–598.

[39] K. Mehlhorn and P. Sanders, *Algorithms and data structures: The basic toolbox*. Berlin: Springer, 2008, ISBN: 9783540779773.

[40] W. M. Mellette *et al.*, "Rotornet," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*, Unknown, Ed., New York, New York, USA: ACM Press, 2017, pp. 267–280, ISBN: 9781450346535. DOI: 10.1145/3098822.3098838.

[41] M. Mendel and S. S. Seiden, "Online companion caching," *Theoretical Computer Science*, vol. 324, no. 2-3, pp. 183–200, 2004, ISSN: 03043975. DOI: 10.1016/j.tcs.2004.05.015.

[42] H. Meyerhenke, "Dynamic load balancing for parallel numerical simulations based on repartitioning with disturbed diffusion," in *2009 15th International Conference on Parallel and Distributed Systems*, IEEE, 12/8/2009 - 12/11/2009, pp. 150–157, ISBN: 978-1-4244-5788-5. DOI: 10.1109/ICPADS.2009.114.

[43] H. Meyerhenke, "Shape optimizing load balancing for mpi-parallel adaptive numerical simulations," *Graph Partitioning and Graph Clustering*, vol. 588, p. 67, 2012.

[44] M. Noormohammadpour and C. S. Raghavendra, "Datacenter traffic control: Understanding techniques and tradeoffs," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1492–1525, 2018. DOI: 10.1109/COMST.2017.2782753.

[45] G. Porter *et al.*, "Integrating microsecond circuit switching into the data center," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 447–458, 2013, ISSN: 01464833. DOI: 10.1145/2534169.2486007.

[46] B. Prisacari, G. Rodriguez, C. Minkenberg, and T. Hoefler, "Bandwidth-optimal all-to-all exchanges in fat tree networks," in *Proceedings of the 27th international ACM conference on International conference on supercomputing - ICS '13*, A. D. Malony, M. Nemirovsky, and S. Midkiff, Eds., New York, New York, USA: ACM Press, 2013, p. 139, ISBN: 9781450321303. DOI: 10.1145/2464996.2465434.

[47] H. Räcke, "Optimal hierarchical decompositions for congestion minimization in networks," in *Proceedings of the fourtieth annual ACM symposium on Theory of computing - STOC 08*, R. Ladner and C. Dwork, Eds., New York, New York, USA: ACM Press, 2008, p. 255, ISBN: 9781605580470. DOI: 10.1145/1374376.1374415.

[48] P. Ronhovde, R. K. Darst, D. R. Reichman, and Z. Nussinov, *An edge density definition of overlapping and weighted graph communities*, 2013. [Online]. Available: http://arxiv.org/pdf/1301.3120v1.

[49] M. Rost and S. Schmid, "Charting the complexity landscape of virtual network embeddings," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, IEEE, 5/14/2018 - 5/16/2018, pp. 1–9, ISBN: 978-3-903176-08-9. DOI: 10.23919/IFIPNetworking.2018.8696604.

[50] M. Rost and S. Schmid, "Virtual network embedding approximations: Leveraging randomized rounding," in *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, IEEE, 5/14/2018 - 5/16/2018, pp. 1–9, ISBN: 978-3-903176-08-9. DOI: 10.23919/IFIPNetworking.2018.8696623.

[51] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 5, pp. 123–137, 2015, ISSN: 01464833. DOI: 10.1145/2829988.2787472.

[52] P. Sanders and C. Schulz, "Distributed evolutionary graph partitioning," in *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, D. A. Bader and D. Mutzel, Eds., Philadelphia, PA: Society for Industrial and Applied Mathematics, 2012, pp. 16–29, ISBN: 978-1-61197-212-2. DOI: 10.1137/1.9781611972924.2.

[53] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Algorithms – ESA 2011*, ser. Lecture Notes in Computer Science, C. Demetrescu and M. M. Halldórsson, Eds., vol. 6942, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 469–480, ISBN: 978-3-642-23718-8. DOI: 10.1007/978-3-642-23719-5_40.

[54] P. Sanders and C. Schulz, "Think locally, act globally: Highly balanced graph partitioning," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, D. Hutchison *et al.*, Eds., vol. 7933, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 164–175, ISBN: 978-3-642-38526-1. DOI: 10.1007/978-3-642-38527-8_16.

[55] A. E. Saríyüce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and Ü. V. Çatalyürek, "Streaming algorithms for k-core decomposition," *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433–444, 2013, ISSN: 21508097. DOI: 10.14778/2536336.2536344.

[56] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *ACM/IEEE SC 2000 Conference (SC'00)*, IEEE, 11/4/2000 - 11/10/2000, p. 59, ISBN: 0-7803-9802-5. DOI: 10.1109/SC.2000.10035.

[57] K. Schloegel, G. Karypis, and V. Kumar, "Multilevel diffusion schemes for repartitioning of adaptive meshes," *Journal of Parallel and Distributed Computing*, vol. 47, no. 2, pp. 109–124, 1997, ISSN: 07437315. DOI: 10.1006/jpdc.1997.1410.

[58] A. Singla, "Fat-free topologies," in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks - HotNets '16*, B. Ford, A. C. Snoeren, and E. Zegura, Eds., New York, New York, USA: ACM Press, 2016, pp. 64–70, ISBN: 9781450346610. DOI: 10.1145/3005745.3005747.

[59] A. Valadarsky, M. Dinitz, and M. Schapira, "Xpander," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks - HotNets-XIV*, J. de Oliveira, J. Smith, K. Argyraki, and P. Levis, Eds., New York, New York, USA: ACM Press, 2015, pp. 1–7, ISBN: 9781450340472. DOI: 10.1145/2834050.2834059.

[60] C. Walshaw and M. Cross, "Mesh partitioning: A multilevel balancing and refinement algorithm," *SIAM Journal on scientific Computing*, vol. 22, no. 1, pp. 63–80, 2000, ISSN: 1064-8275. DOI: 10.1137/S1064827598337373.

[61] C. Walshaw and M. Cross, "Jostle: Parallel multilevel graph-partitioning software–an overview," *Mesh partitioning techniques and domain decomposition techniques*, pp. 27–58, 2007.

[62] A. Yazidi, H. Abdi, and B. Feng, "Data center traffic scheduling with hot-cold link detection capabilities," in *Proceedings of the 2018 Conference on Research in Adaptive and Convergent Systems - RACS '18*, C.-C. Hung and L. B. Said, Eds., New York, New York, USA: ACM Press, 2018, pp. 268–275, ISBN: 9781450358859. DOI: 10.1145/3264746.3264797.

[63] J. Zhang, F. R. Yu, S. Wang, T. Huang, Z. Liu, and Y. Liu, "Load balancing in data center networks: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 3, pp. 2324–2352, 2018. DOI: 10.1109/COMST.2018.2816042.