Online Balanced Repartitioning of Dynamic Communication Patterns in Polynomial Time

Tobias Michael Forner
Technical University of Munich, Germany

Harald Räcke
Technical University of Munich, Germany

Stefan Schmid
University of Vienna, Austria

Abstract

This paper revisits the online balanced repartitioning problem (introduced by Avin et al. at DISC 2016) which asks for a scheduler that dynamically collocates frequently communicating nodes, in order to reduce communication costs while minimizing migrations in distributed systems. More specifically, communication requests arrive online and need to be served, either remotely across different servers, or locally within a server at no cost; before serving a request, the online scheduler can change the mapping of nodes to servers, i.e., migrate nodes, at cost α per node move. Avin et al. presented a deterministic $O(n \log n)$-competitive algorithm in a research augmentation model, which is optimal up to a logarithmic factor; however, their algorithm has the drawback that it relies on expensive repartitioning operations which result in a super-polynomial runtime. Our main contribution is a different deterministic algorithm which achieves the same competitive ratio, but runs in polynomial time. This algorithm is analyzed both analytically and empirically.

2012 ACM Subject Classification Theory of computation → Online algorithms; Computer systems organization → Distributed architectures

Keywords and phrases Online algorithms, graph partitioning, migration, competitive analysis

Acknowledgements Research supported by ERC Consolidator grant agreement No. 864228 (AdjustNet).

1 Introduction

Most distributed systems critically rely on an efficient interconnecting communication network. With the increasing scale of these systems, the network traffic often grows accordingly: applications related to distributed machine learning, batch processing, or scale-out databases, spend a considerable fraction of their runtime shuffling data [?]. An interesting method to improve the efficiency in these systems is to exploit their resource allocation flexibilities: many distributed systems are highly virtualized today and support to relocate (or migrate) communication partners (e.g. virtual machines). By collocating two frequently communicating nodes on the same server, slow and costly inter-server communication can be reduced. However, relocations also come at a cost, and the number of migrations should be kept low accordingly.

This paper revisits the online balanced re-partitioning problem [5] which models the tradeoff between the benefits and the costs of dynamic relocations. The goal is to design an algorithm which maintains, at any time, a mapping of n communication nodes (virtual machines) to ℓ servers of fixed equal size k; in the absence of augmentation, $n = \ell k$, **Stefan: please check:** whereas in the variant with augmentation, the server size is increased by a factor $1/\epsilon$ for some $\epsilon > 0$. The communication pattern can be seen as a dynamic graph, from which communication requests arrive in an online manner; in other words, the online algorithm does not have prior knowledge of future communication requests. The goal is to strike a balance between the benefits and the costs of migrations. More specifically, the cost model is as follows: if a communication request is served remotely, i.e., between nodes mapped to different servers, it incurs a communication cost of 1; communication requests between nodes located on the same server are free of cost. Before the cost for the current request is payed, an algorithm has the option to migrate nodes at a cost of α for each node move.

The problem is known to combine different aspects of classic online problems such as **rent or buy** [10] (should a request be served remotely at a small cost (rent) or should the communicating nodes be collocated at a large cost up-front (buy)?), or **paging** (which existing node to "evict" if two nodes are to be collocated on the same server?). Indeed, in some respect the problem can be seen as a symmetric version of caching: two nodes can be "cached" together on any server.

1.1 Contributions

Our main result is a deterministic online algorithm pCrep for the dynamic balanced graph partitioning problem which achieves a competitive ratio of $O(k \log k)$ and runs in *polynomial time*, for a constant augmentation.

A $O(k \log k)$-competitive algorithm was already given by Avin et al. in [5], together with an almost tight lower bound $\Omega(k)$. However, the algorithm relies on expensive repartitioning which results in a super-polynomial runtime.

We not only evaluate our algorithm analytically but also in simulations, based on real datacenter workloads. To this end, we also perform algorithm engineering and make slight adaptations of our algorithms in order to improve their results. We will make our implementation publicly available as open source, together with this paper.

1.2 Preliminaries

Let us introduce some definitions and notations that will be used throughout the paper. We define a graph $G = (V, E, w)$ where V is the set of vertices, E the set of (undirected) edges, and $w : E \to \mathbb{N}$ assigns each edge an (integer) weight. Given a graph $G = (V, E, w)$, we define an *(edge) cut* of G as a pair of two disjoint subsets X, Y of V such that $X \cup Y = V$. The value of this cut is the sum of the weight of edges between nodes from X and Y, i.e. $\sum_{e=\{u,v\}\in E: u\in X, v\in Y} w(e)$ is the value of the cut (X, Y). Note that such a cut can also be defined by the set of the edges connecting X and Y that are cut. We call a cut a *minimum (edge) cut* of G if it is one of the cuts with minimum value.

The *connectivity* of a graph G is equal to the value of a minimum edge cut of G. This definition will be used in order to define the communication components our algorithm maintains as these are subsets of V which induce subgraphs of high connectivity. We explain the concept of components in greater detail later.

Furthermore we define the term (s,t)*-cut* as a cut (X, Y) for which $s \in X$ and $Y \in X$, i.e. a (s,t)-cut separates the nodes s and t in G. Then a *minimum* (s,t)*-cut* is a (s,t)-cut of minimum value. Note that a minimum (s,t)-cut is not necessarily a minimum cut.

Finally we call an injective function $m : X \to Y$ a *mapping of X to Y*. We use this terminology for example when we talk about the assignment of nodes to servers.

Stefan: fixme: Herr Forner, bitte entfernen Sie CoreDel von den Figures, und nennen Sie CREP-Adj zu pCREP um.

1.3 Organization

The remainder of this paper is organized as follows. **Stefan: fixme: Herr Forner, bitte hier einfach ganz kurz ein Satz pro Section, siehe andere Paper, ohne details und citations.**

We first introduce general definitions and notation in **??** which we will use throughout this thesis. We then formally define the Dynamic Balanced Graph Partitioning (DBGP) problem in Section 2. In Section 7 we discuss previous work related to the DBGP problem. After that we describe two algorithmic ideas, denoted by pCrep and Crep-Core, which are inspired by prior work ([2, 5]) and the respective implications and challenges for their competitive analysis. In Section 4 we show the competitive ratio of $O(2/\epsilon \cdot k \log k)$ for the algorithm pCrep. Then we discuss our implementation of pCrep and show that the algorithm can be implemented in polynomial time in Section 5. After that we empirically evaluate our algorithms by comparing their results with those of existing implementations in Section 6. We summarize the results of this thesis in Section 8. Finally we describe avenues for future work in **??**.

2 Model

We consider the problem of maintaining a partitioning of a set of $n = k \cdot \ell$ nodes (e.g., processes or virtual machines) that communicate with each other, into ℓ clusters (e.g., servers or racks) of size k each, while minimizing both the cost due to communication and due to node migrations. More formally we are given **Stefan: Herr Forner, bitte ersetzen Sie ueberall l mit ℓ.** l servers $V_0, ..., V_{l-1}$, each with capacity k and an initial perfect mapping of $n = k \cdot l$ nodes to the l servers, i.e. each server is assigned exactly k nodes. An input sequence $\sigma = (u_1, v_1), (u_2, v_2), ...(u_i, v_i), ...$ describes the sequence of communication requests: the pair (u_t, v_t) represents a communication request between the nodes u_t and v_t arriving at time t. At time t the algorithm is allowed to perform node migrations at a cost of $\alpha > 1$ per move. After the migration step, the algorithm pays cost 1 if u_t and v_t are mapped to different servers and does not pay any cost otherwise. Note that an algorithm may also choose to perform no migrations at all. This problem was introduced by Avin et al. [5] (DISC 2016) and is known as online balanced re-partitioning.

We are in the realm of competitive analysis and as a result we compare an online algorithm Onl to the optimal offline algorithm Opt. Onl only learns of the requests in the input sequence σ as they happen and as a result only knows about the partial sequence $(u_1, v_1), ..., (u_t, v_t)$ at time t whereas Opt has perfect knowledge of the complete sequence σ at all times.

The goal is to design an online algorithm Onl with a good competitive ratio with regard to Opt defined as follows. An online algorithm Onl is ρ-competitive if there exists a constant β such that

$$\text{Onl}(\sigma) \leq \rho \cdot \text{Opt}(\sigma) + \beta \; \forall \sigma$$

where $\text{Onl}(\sigma)$ and $\text{Opt}(\sigma)$ denote the cost of serving input sequence σ of Onl and Opt respectively.

We consider a model with augmentation (as in prior work [5]), and allow the online algorithm to use larger capacities per cluster. In particular, the online algorithm is allowed to assign $\delta \cdot n/k$ nodes to each cluster where $\delta > 1$. This augmented online algorithm is then compared with the optimal offline algorithm Opt which is not allowed to use any augmentation.

3 Algorithmic Ideas

We first present the main algorithmic ideas underlying our approach, before presenting the polynomial-time implementation later in this paper. In particular, our algorithm, pCREP,

here i am

Stefan: shall we say that only one algo has competitive ratio? and the other is for comparison? also in contribution?

Both algorithms CREP-CORE and
pCREP share a similar idea: to compute a second-order partitioning of the communication nodes into *communication components* which represent node-induced sub-graphs of the original communication graph given by the requests from the input sequence σ. **Stefan: todo: shall we say server or cluster throughout this paper?** Initially each node forms a singleton component, but as the input sequence σ is revealed to the algorithms new communication patterns unfold. The algorithm keeps track of these patterns by maintaining a graph in which the nodes represent the actual communication nodes and the weighted edges represent the number of communication requests between nodes that were part of different components at the time of the request, i.e. for edge $e = \{u, v\}$, $w(e)$ represents the number of paid communication requests between u and v. We say that a communication request between nodes u and v is *paid* if the nodes are located on different servers at the time of the request. Both algorithms merge a set S of components into a new component C if the connectivity of the component graph induced by the components in S is at least α. After each edge insertion the algorithm checks whether there exists a new component set S with $|S| > 1$ which fulfills this requirement.

If after any request and the insertion of the resulting edge the algorithm discovers a new subset S of nodes whose induced subgraph has connectivity at least α and which is of cardinality at most k, it merges the components that form this set into one new component and collocates all the nodes in the resulting set on a single server. The algorithm reserves additional space $\min\{\lfloor \epsilon \cdot |C| \rfloor, k - |C|\}$ for each component on the server it is currently located on. Note that the additional reservation may be zero for components smaller than $1/\epsilon$. This reservation guarantees that nodes are not migrated too often for the analysis to work. This also limits the total space a component can use to a maximum of k. This makes sense as a component whose size exceeds k is never merged but deleted instead. To this end the algorithms keep track of the reservations for each component.

Both algorithms use augmentation $2 + \epsilon$ in order to guarantee that the collocation of such component sets of at most k individual communication nodes is always possible without moving a node not in C. This guarantees by an averaging argument that there is always at least one cluster with capacity at least k, which a newly merged component can be moved to.

If the subset has cardinality greater than k the resulting component is deleted. The definition of this deletion process is the main difference between our algorithms which we discuss in the following subsections. The common part of both algorithms is also summarized in the form of pseudocode in Algorithm 1. Note that the subroutine *delete(Y)* of a component set Y is different for each of the algorithms. In the pseudocode description we denote the reservation of a component C by $res(C)$ and the current server it is mapped to by $serv(C)$. The free capacity of a server i is denoted by $cap(i)$.

We also describe the particular challenges each approach entails when it comes to the competitive analysis in their respective subsections.

The main differentiating factor of these algorithms compared to prior work [2, 5] is that we merge once a component set reaches connectivity α, while prior approaches do so once the

Algorithm 1 Crep

```
Initialize an empty graph on $n$ nodes
turn each of the $n$ nodes into a singleton component
for all $r = \{u, v\} \in \sigma$ do
    if $comp(v) \neq comp(u)$ then
        $w(\{u, v\}) \leftarrow w(\{u, v\}) + 1$
    end if
    if $\exists$ component set $X$ with connectivity at least $\alpha$ and $|X| > 1$ and $nodes(X) \leq k$ then
        mergeAndRes($X$)
    end if
    if $\exists$ component set $Y$ with connectivity at least $\alpha$ and $nodes(Y) > k$ then
        delete($Y$) //to be specified later
    end if
end for
```

Algorithm 2 mergeAndRes(X)

```
for all $C \in X$ do
    $cap(serv(C)) \leftarrow cap(serv(C)) + res(C)$
end for
$N \leftarrow collocate(X)$ //moves all components from $X$ to the same server as described
//$N$ contains the newly created component
if $|N| > 2/\epsilon$ then
    //reserve additional space
    $res(N) \leftarrow \min\{\lfloor \epsilon \cdot |N| \rfloor, k - |N|\}$
    $cap(serv(N)) \leftarrow cap(serv(N)) - res(N)$
end if
```

component set reaches a certain density threshold. More specifically earlier algorithms merge a component set S once it fulfills $w(S) \geq (|S| - 1) \cdot \alpha$ where $w(S)$ denotes the cumulative weight of the edges between nodes contained in the components of S. The similarities are especially apparent when comparing the respective lemmas and properties that are used in order to bound the weight between partitions of mergeable component sets. These are Lemma 4.3 in [2], Property 3 in [5] and Lemma 6 in this paper. Our connectivity-based approach however allows us to achieve a polynomial runtime.

Now we describe the differences in the deletion steps of Crep-Core and pCrep and present the implications for their respective competitive analysis.

3.1 pCrep

The first algorithm, pCrep, resets all the edges contained in the deleted component C and also resets the weights of edges *adjacent* to C, i.e. all edges $e = \{u, v\}$ are reset to zero if u or v were contained in component C at the time of its deletion. The deletion method is also described in pseudocode in Algorithm 3.

The idea for the competitive analysis will be to relate the cost of both Opt and pCrep to the deleted components in the solution of pCrep. The fact that pCrep also resets adjacent edges means that we can uniquely identify requests with the deleted component whose deletion led to the reset of the corresponding edge weights to zero. This means that we do not have to face the problems we discussed in the previous section on Crep-Core. However

Algorithm 3 delete(Y) of pCREP

```
for all $e = \{u, v\} \in E$ do
  if $u \in Y$ or $v \in Y$ then
    $w(e) \leftarrow 0$
  end if
end for
for all $C \in Y$ do
  $cap(serv(C)) \leftarrow cap(serv(C)) + res(C)$
  $res(C) \leftarrow 0$
end for
```

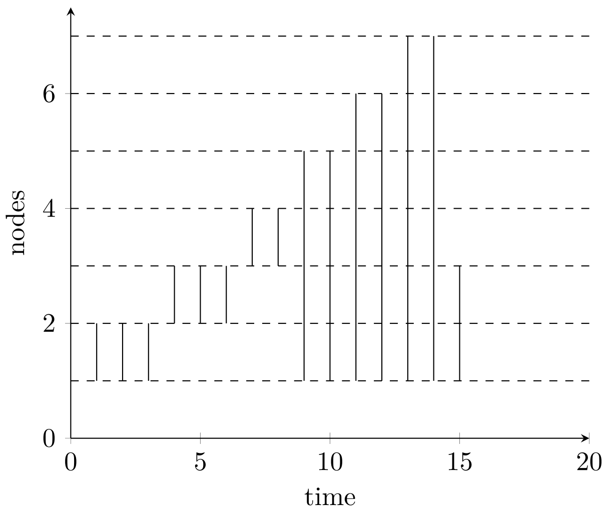

Figure 1 illustration for the pCREP approach

the question remains whether the deletion of adjacent edges is valid, i.e. whether we can still preserve a good competitive ratio for this approach.

3.2 Crep-Core

In the CREP-CORE algorithm, all edges $e = \{u, v\}$ are reset to zero if both u and v were contained in component C at the time of its deletion. This approach appears in pseudocode in Algorithm 4. This algorithm is a heuritic, and we will mainly consider it as a baseline in simulation.

4 Competitive Analysis

We now present a competitive analysis of the pCREP algorithm, and will simply call this algorithm CREP from now on. We analyze the competitive ratio of CREP with augmentation $(2+\epsilon)$ and show in Theorem 15 that CREP is $O(2/\epsilon \cdot k \log k)$-competitive. In the next section, in Theorem 17 in Section 5.4, we will show that this algorithm can be implemented in polynomial time.

We will use the following two general definitions throughout the analysis.

Algorithm 4 delete(Y) of CREP-CORE

```
for all $e = \{u, v\} \in E$ do
  if $u \in Y$ and $v \in Y$ then
    $w(e) \leftarrow 0$
  end if
end for
for all $C \in Y$ do
  $cap(serv(C)) \leftarrow cap(serv(C)) + res(C)$
  $res(C) \leftarrow 0$
end for
```

▶ **Definition 1.** *Define for any subset S of components $w(S)$ as the total weight of all edges between nodes of S.*

▶ **Definition 2.** *Let a set of components of size at least 2 and of connectivity α be a mergeable component set.*

▶ **Definition 3.** *An α-connected component is a maximal set of vertices that is α-connected.*

4.1 Preliminaries

The following lemma states that there is always at most one mergeable component set after the insertion of a new edge which CREP then merges. The earliest point in time a new mergeable component set can emerge is after the next edge is inserted. This lemma is very similar to Lemma 4.1 from [2] in essence. The only difference for CREP lies in the definition of a mergeable component which we have defined via a connectivity measure while the authors of [2] used density.

▶ **Lemma 4.** *At any time t after CREP performed its merge and delete actions, all subsets S of components with $|S| > 1$ have connectivity less than α, i.e. there exist no mergeable component sets after CREP performed its merges and deletions.*

Proof. We prove the lemma by an induction on steps. The lemma holds trivially at time 0. Now assume that at some time $t > 0$ the lemma does not hold, i.e., there is a subset S of components with connectivity at least α and $|S| > 1$. We may assume that t is the earliest time for which S has connectivity α.

Then the incrementation of the weight of some edge e at time t raised the connectivity of S, but S was not merged into a new α-connected component C. If no new component was created at time t we arrive at a contradiction as CREP always merges if there exists a mergeable component set.

Now assume that a component C was created at time t. This means that C must also contain the endpoints of e. But then the conjunction of C and S forms an even larger subset of components with connectivity at least α which is a contradiction to the maximality of C and S. ◀

The following lemma is adapted for our connectivity-based approach from Corollary 4.2 in [2].

▶ **Lemma 5.** *Fix any time t and consider weights right after they were updated by CREP but before any merge or delete actions. Then all subsets S of components with $|S| > 1$ have connectivity at most α and a mergeable component set S has connectivity exactly α.*

Proof. This lemma follows directly from Lemma 4 as connectivities can only increase by at most 1 at each time t and Lemma 4 guarantees that mergeable component sets are merged by Crep directly after they emerge before a new request is revealed. ◀

The following two lemmas combined give us a result similar to Lemma 4.3 in [2]: bounds on the edge weight that is cut when partitioning a mergeable component set, i.e. a set of components of connectivity at least α.

We start by establishing a lower bound on this edge weight in the following lemma.

▶ **Lemma 6.** *Given a mergeable set of components S and a partition of S into $g > 1$ parts $S_1, ..., S_g$. Then the weight between the parts of the partition is at least $g/2 \cdot \alpha$.*

Proof. We construct a graph G with the different parts $S_i, i \in \{1, ..., g\}$ of the partition as nodes. Note that this graph is α-connected. We insert an edge for each edge between the parts S_i. Now consider the sum of the weighted degrees of all such nodes S_i in the constructed graph:

$$\sum_{i \in \{1,...,g\}} deg_G(S_i) = 2 \sum_{e \in G} w(e)$$

The equality follows as the left sum counts each edge twice, once for each endpoint. Now consider the fact that each node S_i must have degree at least α with respect to the edges in G because G is α-connected. Hence

$$2 \sum_{e \in G} w(e) = \sum_{i \in \{1,...,g\}} deg_G(S_i) \leq \sum_{i \in \{1,...,g\}} \alpha = g \cdot \alpha$$

which gives us that $\sum_{e \in G} w(e) \geq g/2 \cdot \alpha$. ◀

In the following lemma we establish the upper bound on the cut edge weight when partitioning a mergeable set of components S into $g \geq 2$ parts.

▶ **Lemma 7.** *Given a mergeable set of components S and a partitioning of S into $g \geq 2$ parts $S_1, ..., S_g$. The weight between the parts S_i is at most $(g-1) \cdot \alpha$ during the execution of Crep (in the version pCrep).*

Proof. Similarly to before we construct a graph $G = (V, E)$ with the different parts $S_i, i \in \{1, ..., g\}$ of the partition as nodes and we insert an edge for each edge between the parts S_i. Note again that this graph is α-connected. We iteratively partition G into subsets via minimum cuts with regard to edge weight, i.e. we consider a minimum edge cut of G which partitions the nodes of G into the subsets V_1 and V_2. We continue to iteratively partition the resulting sets until all sets contain only one node of G each. As this required at most $|V| - 1$ cuts of value at most α and $|V| = g$ by definition of G the lemma follows. ◀

4.2 Proof Overview

Now that we have defined some general notions and fundamental properties we give an overview of the approach we use in order to achieve an upper bound on pCrep, a lower bound on Opt and finally the competitive ratio of pCrep. pCrep allows us to argue about each deleted component and the requests deleted during its deletion separately. Furthermore we know that a component deleted by pCrep is both at least α-connected as well as larger than k, i.e. its nodes must be spread across several servers in the solution of Opt. This allows us to assign significant cost to Opt Additionally we also know by the definition of

Crep that the edges that are reset because they are adjacent to C can not be part of any component. We use this fact in Lemma 12 in order to show that the amount of such requests is at most $k \cdot \alpha$ for each migration performed by Opt. These are the main concepts we use in order to bound the cost of pCrep and to assign cost to Opt in the following sections. In Theorem 15 we finally show the competitive ratio of pCrep .

4.3 Upper Bound On pCrep

We start the analysis of the upper bound of Crep by introducing several notions that we will use throughout the analysis. We define the set $\textsc{del}(\sigma)$ as the set of components that were deleted by Crep during its execution given the input sequence σ. We define the following notions for a deleted component $C \in \textsc{del}(\sigma)$. Let $\textsc{epoch}(C)$ denote the (node, time) pairs of nodes in C starting at the time after the time $\tau(\mathit{node})$ when *node* was last turned into a singleton component, i.e.

$$\textsc{epoch}(C) = \bigcup_{n \in nodes(C)} \{n\} \times \{\tau(n)+1, ..., \tau(C)\}.$$

Note that for $C \in \textsc{del}(\sigma)$, $\tau(C)$ denotes both the time of the creation as well as the time of deletion of C. We can use this definition of a component epoch $\textsc{epoch}(C)$ to uniquely assign each node to a deleted component C at each point in time t (except for nodes in components that persist until the end of sequence σ). We assign all requests to $\textsc{epoch}(C)$ whose corresponding requests are deleted because of the deletion of component C and call the set of those requests $\textsc{req}(C)$. We split the requests from $\textsc{req}(C)$ into two sets: $\textsc{core}(C)$ and $\textsc{halo}(C)$. $\textsc{core}(C)$ contains all requests for which both nodes have already been assigned to C at the time of the request, i.e.

$$\textsc{core}(C) = \{r = \{u, v\} \in \sigma | (u, \textsc{time}(r)) \in \textsc{epoch}(C) \text{ and } (v, \textsc{time}(r)) \in \textsc{epoch}(C)\}.$$

These are the requests that led to the creation of component C by increasing the connectivity within the corresponding subgraph.

We define $\textsc{halo}(C)$ as the set of all requests from $\textsc{req}(C)$ for which exactly one end point was associated with C at the time of the request. Note that this means that $\textsc{halo}(C) = \textsc{req}(C) \backslash \textsc{core}(C)$. These definitions allow us to differentiate between the highly-connected sub-graph induced by the nodes of C which are connected by requests from $\textsc{core}(C)$ and the edges leaving C from $\textsc{halo}(C)$ which are relatively less dense as Crep has not merged any outer node with the component.

We start the analysis by bounding the communication cost of Crep that is due to serving requests from $\textsc{core}(C)$ for $C \in \textsc{del}(\sigma)$.

▶ **Lemma 8.** *With augmentation $2 + \epsilon$, Crep pays at most communication cost $|C| \cdot \alpha$ for requests in $\textsc{core}(C)$ where $C \in \textsc{del}(\sigma)$.*

Proof. First note that due to Lemma 4 Crep merges mergeable component sets as soon as they emerge. Whenever Crep performs a merge of a mergeable component set S, Lemma 7 states that there was at most total edge weight $(|S| - 1) \cdot \alpha$ between the merged components, i.e. $w(S) \leq (|S| - 1) \cdot \alpha$. Each such merge decreases the number of components that need to be merged in order to form component C by $|S| - 1$. Hence Crep has payed at most $|C| \cdot \alpha$ communication cost for requests in $\textsc{core}(C)$. ◀

We define $\textsc{fin-weights}(\sigma)$ as the total amount of edge weight between the components $\textsc{fin-comps}(\sigma)$ which are present after the execution of Crep given input sequence σ. Together with the fact that Crep pays for all requests in halo(C) for deleted components C we use these definitions as well as the previous lemma to bound the total communication cost of Crep in the following lemma.

▶ **Lemma 9.** *The cost of serving communication requests that* Crep *has to pay, denoted by* $\textsc{Crep}^{req}(\sigma)$ *given input sequence* σ *is bounded by*

$$\textsc{Crep}^{req}(\sigma) \leq \sum_{C \in \textit{DEL}(\sigma)} (|C| \cdot \alpha + |\textit{HALO}(C)|) + \sum_{C \in \textit{FIN-COMPS}(\sigma)} |C| \cdot \alpha + \textit{FIN-WEIGHTS}(\sigma).$$

Proof. The number of communication requests that led to the creation of a component C is bounded by $|C| \cdot \alpha$ due to Lemma 7. If component C was deleted by Crep then also the edge weights corresponding to requests from halo(C) were reset to zero. All other edge weights were not changed. The remaining communication requests that have not been accounted for so far have either led to the creation of component $C \in$ fin-comps(σ) and are hence also bounded by $|C| \cdot \alpha$ or have not led Crep to any merge and are hence contained in fin-weights(σ). This concludes the proof. ◀

We continue our analysis by bounding the migration cost of Crep in the following lemma.

▶ **Lemma 10.** *With augmentation* $2 + \epsilon$, Crep *pays at most migration costs of*

$$\textsc{Crep}^{mig}(\sigma) \leq \sum_{C \in \textit{DEL}(\sigma) \cup \textit{FIN-COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha.$$

Proof. First note that Crep only performs migrations when it merges components. We fix a component $C \in$ del$(\sigma) \cup$ fin-comps(σ) and bound the number of times each node of C is moved as Crep processes the requests that led to the creation of C.

As Crep only reserves additional space $\lfloor \epsilon \cdot |B| \rfloor$ for each component B and only moves component B when a merge results in a component of size more than $(1+\epsilon) \cdot |B|$ each node of C is moved at most $(2/\epsilon + 1) + \log k$ times. Summing over all nodes in C that were actually moved by Crep bounds the number of migrations by $|C| \cdot ((2/\epsilon + 1) \log k)$ as components get deleted without migrations once they contain more than k nodes. This leads to the desired bound on the migration costs as each node migration incurs cost α to Crep. ◀

Finally we summarize our results from Lemma 9 and Lemma 10 in the following lemma in order to obtain the final upper bound on the cost of Crep.

▶ **Lemma 11.** *With augmentation* $2 + \epsilon$, Crep *pays at most total cost*

$$2 \cdot \sum_{C \in \textit{COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + \sum_{C \in \textit{DEL}(\sigma)} |\textit{HALO}(C)| + \textit{FIN-WEIGHTS}(\sigma).$$

where $\textit{COMPS}(\sigma) = \textit{DEL}(\sigma) \cup \textit{FIN-COMPS}(\sigma)$.

Proof. We sum the results from Lemma 9 and Lemma 10 to obtain the lemma:

$$\begin{aligned}
\textsc{Crep}(\sigma) &\leq \textsc{Crep}^{req} + \textsc{Crep}^{mig} \\
&\leq \sum_{C \in \text{DEL}(\sigma)} (|C| \cdot \alpha + |\textsc{halo}(C)|) + \sum_{C \in \text{FIN-COMPS}(\sigma)} |C| \cdot \alpha + \textsc{fin-weights}(\sigma) \\
&+ \sum_{C \in \text{COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha \\
&\leq 2 \cdot \sum_{C \in \text{COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + \sum_{C \in \text{DEL}(\sigma)} |\textsc{halo}(C)| \\
&+ \textsc{fin-weights}(\sigma).
\end{aligned}$$

◀

4.4 Lower Bound on Opt

We next bound the cost on Opt by assigning cost to Opt based on the size of the components C that Crep deletes and the associated adjacent edges $\textsc{halo}(C)$ which Crep resets to zero during the deletion of C. In order to achieve this we introduce some additional notions. First we define the term *offline interval* of a node v to be the time between two migrations of v in the solution of Opt. More specifically an offline interval of node v either starts at time zero (if it is the first offline interval of v) or after a migration of v and ends with the next migration of node v that Opt performs.

Furthermore we say that an offline interval is contained in the epoch $\textsc{epoch}(C)$ of a component $C \in \textsc{del}(\sigma)$ if it ends before the time $\tau(C)$. Note that $\tau(C)$ is both the time of the creation of C in the solution of Crep and the time of its deletion as $C \in \textsc{del}(\sigma)$. We assign a request r involving the node v to an offline interval of v if it is both the first offline interval of one of the end points of r that ends and if the offline interval ends before the deletion of the edge representing r due to a component deletion. The requests from $\mathcal{H} = \bigcup_{C \in \text{DEL}(\sigma)} \textsc{halo}(C)$ that are not assigned to any offline interval are then those which are deleted due to the deletion of a component that took place before the corresponding offline interval ended. Let P denote the set of edges from $\bigcup_{C \in \text{DEL}(\sigma)} \textsc{halo}(C)$ that both Crep and Opt pay for and let I denote the set of requests we have assigned to offline intervals.

These definitions are illustrated in Figure 2. Note that we only show some requests explicitly for the sake of readability. The grey horizontal lines represent the nodes at each time t. The red outline surrounds the (node,time) pairs of $\textsc{epoch}(C)$. Blue dots mark migrations of the corresponding node performed by Opt while red dots mark deletions of the component the node was assigned to at that time. The dashed vertical lines in black mark requests that are assigned to another component because it is deleted before component C. The dotted green line is a request from $\textsc{halo}(C)$ assigned to the offline interval of node 5 between the two blue dots. The regular green lines are assigned to an offline interval which is not contained in $\textsc{epoch}(C)$. We define this concept more formally at a later point in the analysis. The lines in magenta are sample requests from $\textsc{core}(C)$.

We start by bounding the total edge weight (the total number of requests) we assign to any one offline interval when limiting ourselves to requests from $\mathcal{H}$ which Crep pays for but Opt does not. We denote the set of these requests by N, i.e. $N = \mathcal{H} \backslash P$. Note that $\mathcal{H}$ only contains requests which Crep payed for due to the definition of $\textsc{halo}(C)$.

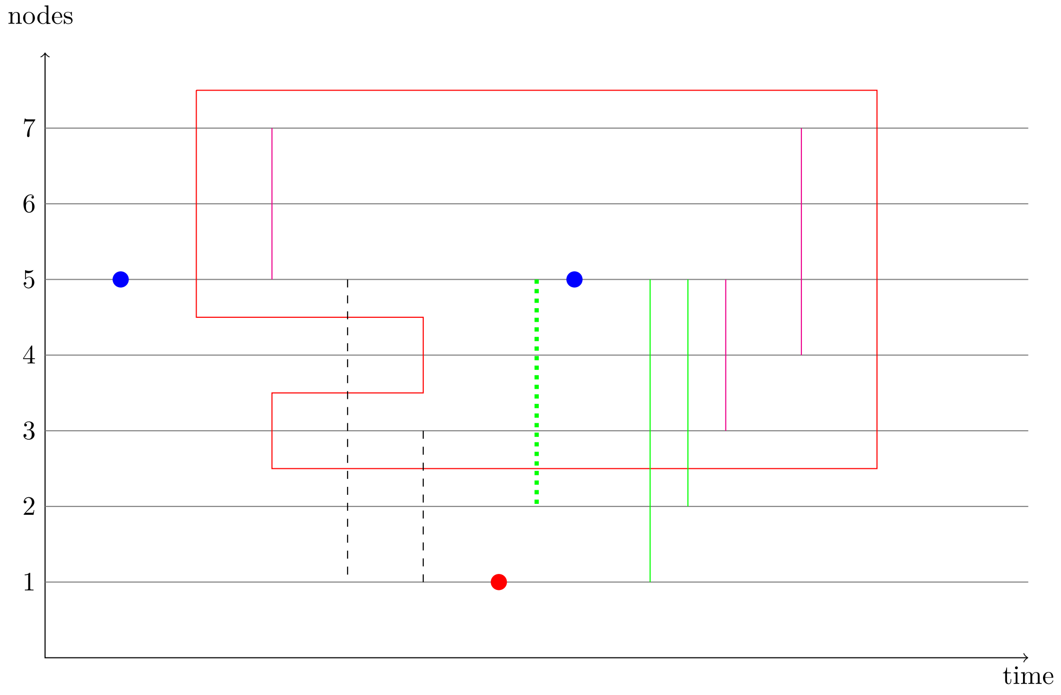

Figure 2 illustration of definitions used in the analysis

▶ **Lemma 12.** *We assign at most $k \cdot \alpha$ requests from N to any one offline interval.*

Proof. We fix an arbitrary offline interval of node v. Observe that none of the nodes involved in the assigned requests are moved by Opt during the offline interval, hence all the requests in question involve only nodes that Opt has placed on the same server as v during the offline interval.

The number of such nodes is hence limited by the server capacity k. As we only examine requests from $\mathcal{H}$ we know that none of these requests have led Crep to perform any merges, hence there were at most α requests between v and any one of the other nodes on its server during the offline interval. This bounds the number of requests assigned to the offline interval by $k \cdot \alpha$. ◀

Let $R(C)$ denote the set of requests from $\textsc{halo}(C)$ that were not assigned to any offline interval for a deleted component $C \in \textsc{del}(\sigma)$.

We say that a migration of node v at time t in the solution of Opt is *contained* in $\textsc{epoch}(C)$ if $(v, t) \in \textsc{epoch}(C)$.

Let $\textsc{opt-mig}(C)$ denote the cost of Opt due to migrations of nodes from component C that are contained in $\textsc{epoch}(C)$ and let $\textsc{opt-req}(C)$ denote the cost of Opt due to serving requests from $\textsc{core}(C)$.

We show the following lower bound on the cost of Opt for migrations from $\textsc{opt-mig}(C)$ and requests from $\textsc{opt-req}(C)$ for all deleted components C.

▶ **Lemma 13.**

$$\sum_{C \in DEL(\sigma)} \textit{OPT-MIG}(C) + \textit{OPT-REQ}(C) \geq |C|/k \cdot \alpha + |R(C)|/k$$

Proof. For the following part of the proof we fix an arbitrary component $C \in \textsc{del}(\sigma)$. Note that the nodes involved in requests from $R(C)$ were not moved by Opt during the processing of requests from $R(C)$ until the time of deletion of C as otherwise they would be assigned to an offline interval.

The number of nodes contained in C or connected to C via edges representing requests from R is at least $|C|+|R(C)|/\alpha$ since requests from $R(C)$ have not led Crep to perform any migrations. Because of this fact Opt must have placed those nodes on at least $\frac{|C|+|R(C)|/\alpha}{k}$ different servers. As Opt does not pay for any requests from R it follows that Opt must have placed the nodes from C in $\frac{|C|+|R(C)|/\alpha}{k}$ different servers.

We first examine the case in which Opt does not move any nodes from C during $\textsc{epoch}(C)$. In this case Opt must partition a graph containing the nodes from C which are connected via edges representing the requests from $\textsc{core}(C)$. As stated earlier Opt placed those nodes in $\frac{|C|+|R(C)|/\alpha}{k}$ different servers at time $\tau(C)$. As Crep merged component C this graph is α-connected and hence Lemma 6 gives that Opt has to cut at least edges of total weight $\frac{|C|+|R(C)|/\alpha}{k} \cdot \alpha = |C|/k \cdot \alpha + |R(C)|/k$.

For the more general case in which Opt may perform node migrations during $\textsc{epoch}(C)$ we adapt the graph construction from above as follows: we add a vertex representing each (node, time) pair from $\textsc{epoch}(C)$. We connect each (node, time) pair p with edges of weight α to the pairs of the same node that represent the time step directly before and directly after p (if they exist in the graph). These edges represent the fact that Opt may choose to migrate a node between any two time steps in $\textsc{epoch}(C)$. Additionally we add an edge of weight one for each request $r = \{u, v\}$ from $\textsc{core}(C)$ by connecting the nodes in the graph that represent the pairs (u, t) and (v, t), respectively. Opt once again has to partition this graph into $\frac{|C|+|R(C)|/\alpha}{k}$ parts.

Note that we only added edges of weight α to the graph and hence this graph is also α-connected. We conclude that once again Opt has to cut edges of weight at least $\frac{|C|+|R(C)|/\alpha}{k} \cdot \alpha = |C|/k \cdot \alpha + |R(C)|/k$.

In both cases only edges representing either requests from $\textsc{opt-req}(C)$ or migrations from $\textsc{opt-mig}(C)$ were cut.

As the sets $core(C)$, $R(C)$, $core(D)$ and $R(D)$ are disjoint for two different components $C, D \in \textsc{del}(\sigma)$ per their definition we conclude that

$$\sum_{C \in \mathrm{DEL}(\sigma)} \textsc{opt-mig}(C) + \textsc{opt-req}(C) \geq |C|/k \cdot \alpha + |R(C)|/k.$$

◀

In the following lemma we combine the results of the previous lemmas in order to bound the cost of Opt given input sequence σ, denoted by Opt(σ).

▶ **Lemma 14.** *The cost of the solution of Opt given input sequence σ is bounded by*

$$\textit{Opt}(\sigma) \geq 1/2 \cdot \sum_{C \in DEL(\sigma)} |C|/k \cdot \alpha + |\textit{halo}(C)|/k.$$

Proof. We combine the results from Lemma 12 and Lemma 13. Note that the cost from Lemma 13 may contain migration costs. In this case the corresponding migrations represent the end of an offline interval. We denote the number of offline intervals by o. This gives us that

$$2\textsc{Opt}(\sigma) \geq \sum_{C \in \mathrm{DEL}(\sigma)} \textsc{opt-mig}(C) + \textsc{opt-req}(C) + o \cdot \alpha + |P|$$

as we account for each migration at most twice.

Consider that due to Lemma 12 we have the inequality $o \geq |N|/k$. We repeat that $\mathcal{H} = \bigcup_{C \in \mathrm{DEL}(\sigma)} \textsc{halo}(C)$. Note that N is the subset of requests of $\mathcal{H}$ for which Opt does not pay while P is the subset of $\mathcal{H}$ Opt pays for. It follows that the disjoint union of N and P is $\mathcal{H}$. Hence we obtain

$$\begin{aligned}
2\textsc{Opt}(\sigma) &\geq \sum_{C \in \mathrm{DEL}(\sigma)} \textsc{opt-mig}(C) + \textsc{opt-req}(C) + o \cdot \alpha + |P| \\
&\geq \sum_{C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + |R(C)|/k + (|N| + |P|)/k \\
&\geq \cdot \sum_{C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + |\textsc{halo}(C)|/k.
\end{aligned}$$

This gives us the lemma. ◀

4.5 Competitive Ratio

We can now combine the results of Lemma 11 and Lemma 14 to obtain the following theorem which gives us the desired competitive ratio.

▶ **Theorem 15.** *With augmentation* $(2+\epsilon)$ *the competitive ratio of* Crep *is in* $O(2/\epsilon \cdot k \log k)$.

Proof. We arbitrarily fix an input sequence σ and use our previous results to bound the competitive ratio of Crep. We define $\textsc{comps}(\sigma) := \textsc{del}(\sigma) \cup \textsc{fin-comps}(\sigma)$ in order to improve readability. Let P denote the set of edges from $\bigcup_{C \in \mathrm{DEL}(\sigma)} \textsc{halo}(C)$ that both Crep and Opt pay for.

$$\begin{aligned}
&\frac{\textsc{Crep}(\sigma) - \textsc{fin-weights}(\sigma)}{\textsc{Opt}(\sigma)} \\
&\leq \frac{2 \cdot \sum_{C \in \mathrm{COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + \sum_{C \in \mathrm{DEL}(\sigma)} |\textsc{halo}(C)|}{1/2 \cdot \sum_{C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + |\textsc{halo}(C)|/k + |P|} \\
&\leq k \log k \frac{2 \cdot \sum_{C \in \mathrm{DEL}(\sigma)} |C| \cdot (2/\epsilon + 1) \cdot \alpha + \sum_{C \in \mathrm{DEL}(\sigma)} |\textsc{halo}(C)|}{1/2 \sum_{(C \in \mathrm{DEL}(\sigma)} |C| \cdot \alpha/2 + |\textsc{halo}(C)|)} + \beta \\
&= O(2/\epsilon \cdot k \log k) + \beta
\end{aligned}$$

where

$$\beta = \sum_{C \in \mathrm{FIN\text{-}COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha$$

Let $\beta' = \beta + \textsc{fin-weights}(\sigma)$. Then it follows that

$$\frac{\textsc{Crep}(\sigma)}{\textsc{Opt}(\sigma)} \leq O(2/\epsilon \cdot k \log k) + \beta'.$$

To obtain the bound on β' we observe that the components in $\textsc{fin-comps}(\sigma)$ each are of size at most k since they were not deleted by Crep. This allows us to derive to bound

$\sum_{C \in \text{FIN-COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \leq l \cdot k \cdot ((2/\epsilon + 1) + \log k)$. Since at the end of the execution of Crep there can be at most $k \cdot l$ components, Lemma 7 allows us to bound FIN-WEIGHTS(σ) by $k \cdot l \cdot \alpha$. Hence we conclude that $\beta' \leq l \cdot k \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + k \cdot l \cdot \alpha \in O(2/\epsilon \cdot k \log k)$. ◀

So far we have shown that Crep is competitive. In Section 5.4 we will show that our version pCrep of Crep can be implemented in polynomial time.

5 From Algorithmic Ideas to Polynomial-Time Implementation

So far we have only shown that the algorithm pCrep described in Section 3 has a competitive ratio $O(2/\epsilon \cdot k \log k)$. In this section we discuss the implementation and running time and conclude in Theorem 17 in Section 5.4 that it is indeed polynomial. We first describe the ideas behind our implementation of Crep and then illustrate them by discussing pseudocode of the implemented algorithm.

5.1 Algorithm Explanations

We now describe our implementation of Crep with augmentation $2 + \epsilon$ in greater detail. In order to limit the section of the graph G maintained by Crep that needs to be updated upon a new request between nodes of different components we maintain a decomposition tree defined as follows: the root represents the whole graph and is assigned the connectivity of the whole graph. Given a node v in the tree that represents a subgraph G' of G, we decompose G' into subgraphs whose connectivity is strictly larger than that of G' and add children to v for each such subgraph. We do not decompose sub-graphs of connectivity at least α any further as we only need to identify whether a new subgraph of connectivity at least α was created by the insertion of the most recent request. Additionally we keep track of the connectivity of each such subgraph. Figure 4 illustrates this decomposition for the graph shown in Figure 3. In the decomposition tree we have labelled each node with the corresponding subset of vertices and the connectivity of the graph induced by these vertices.

If a new request is revealed to Crep then we only need to update the smallest subtree of the decomposition tree which still contains both end points of the request. For example in the case of a new request between 0 and 1 we only need to recompute the decomposition of the sub-graph induced by the vertex set $\{0, 1, 2\}$ in the example. This is correct because we can view each decomposition of a subgraph G' into smaller graphs of a higher connectivity as a set of cuts that separates the nodes of G'. Inserting a new edge within a subgraph G' may only increase the value of the cuts which result in the decomposition of G', but do not affect cuts separating G' itself from other subgraphs. If a new request led to the creation of a new component this means that two old components that were at least α-connected were merged and hence the number of leaves in the decomposition tree decreased. If this is the case then the algorithm checks whether the new component contains more than k nodes. In this case the component is deleted and split into singleton components, each containing one node from the deleted component.

Upon such a component deletion the edges inside of and adjacent to the component are deleted, i.e. their weight is reset to zero. This means that the decomposition tree needs to be recomputed in order to reflect this change. If however the resulting component C contains at most k nodes the algorithm tries to collocate the nodes of the component while minimizing migration costs, i.e. looking for a cluster which contains as many nodes of the newly merged component as possible but which also has enough free capacity for the remaining nodes to be

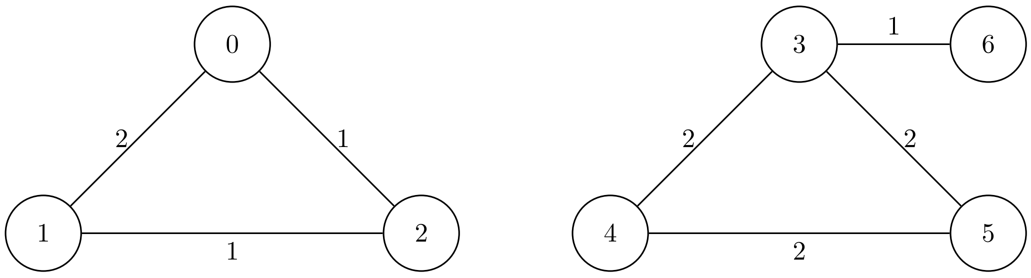

Figure 3 example graph

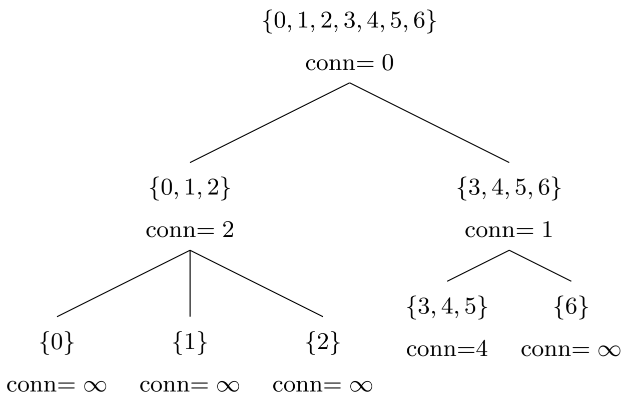

Figure 4 decomposition tree for the graph from Figure 3 for $\alpha = 4$

moved there and for additional reservation $\min\{\lfloor\epsilon \cdot |C|\rfloor, k - |C|\}$. Both the decision whether to delete as well as possible node migrations are handled in the subroutine *updateMapping* which is given the newly computed α-connected components as input. In the next section we provide more detailed pseudocode describing our implementation.

5.2 Algorithm Pseudocode

Algorithm 5 insertAndUpdate(a,b)

```
if comp[$a$] == comp[$b$] then
    return
end if
addEdge($a, b$)
updateDecomposition($a, b$)
$del$ ← updateMapping($alphaConnectedComponents$)
delComponents($del$)
```

Algorithm 5 is the main function that is called upon each new request. It checks whether the new request is between different α-connected components. If this is not the case it determines that this request can not change the decomposition and returns. Otherwise the weight of the corresponding edge is increased and other routines are called that update the decomposition based on this new edge.

Algorithm 6 first determines the smallest sub-graph in the decomposition tree that contains both end points of the request and decomposes this sub-graph. For this decomposition step we use the algorithm proposed by Chang et al. ([8]). We explain this algorithm in greater detail in Section 5.3.

Algorithm 6 updateDecomposition(a,b)

```
$q \leftarrow$ findSmallestSubgraph($a, b$)
while $q$ not empty do
  current$\leftarrow$ $q$.popFront()
  if $res$.connectivity==$\alpha$ then
    continue
  end if
  $res \leftarrow$ decompose(current, current.connectivity+1)//decomposition based on $(s,t)$-cuts
  current.connectivity $\leftarrow$ value of smallest encountered cut
  if current.connectivity$\geq$ $\alpha$ then
    continue
  end if
  childrenQueue $\leftarrow$ $res$
  //make sure that only subgraphs with higher connectivity are added as children
  while childrenQueue not empty do
    $c$ $\leftarrow$childrenQueue.pop()
    $cRes$ $\leftarrow$decompose($c$, current.connectivity+1)
    $c$.connectivity $\leftarrow$ value of smallest encountered cut
    if decompose returned only one graph then
      current.children.add($cRes$)
      if $cRes$ has connectivity smaller than $\alpha$ then
        $q$.push($cRes$)
      end if
    else
      childrenQueue.add($cRes$)
    end if
  end while
end while
```

Afterwards the routine *updateMapping* is called which compares the number of components to the number of components before the arrival of the request. Only if the number of components has decreased it checks for the new component as otherwise there was no component merge. If there was a merge then the routine examines the size of the newly created (merged) component and decides whether to delete or to collocate based on the logic described in the previous section.

If a deletion has to be performed then this step is done in the routine *delComponents* (Algorithm 7) which resets all edge weights both of edges between nodes of the component as well as all adjacent edges and finally starts the decomposition of the whole graph in order to arrive at a new decomposition that follows the definition from the previous section. In the case of a collocation the nodes are moved to a cluster that has enough space while updating the reservations and cluster capacities accordingly.

Algorithm 7 delComponents(del)

```
delAllEdges($del$)
root.connectivity=0
root.children={}
updateDecomposition(0,1)
```

5.3 On the Decomposition of a Subgraph

We next describe our algorithm for the decomposition of a given subgraph represented by a node in the decomposition tree. Specifically we describe the general idea behind our implementation of the subroutine *decompose(treeNode, c)* which we have based on the algorithm described by Chang et al. ([8]). In this algorithm c denotes the connectivity that is the basis of this decomposition step.

Given a node v of the decomposition tree, first a *partition graph* is constructed which is a graph consisting of the nodes in the subgraph represented by v and the edges which are between the nodes of the subgraph. This partition graph also supports merges and cuts of its nodes. More specifically the partition graph is initialized as a graph $P = (V, E)$ with $V = nodes(v)$ and $E = \{e = \{u, w\} \in E' | u \in nodes(v) \text{ and } w \in nodes(v)\}$ where E' represents the set of edges in the graph maintained by CREP. Additionally we maintain a mapping M which assigns each node from V a set of the nodes in the subgraph represented by the decomposition tree node v. Initially M assigns each node in V the subset containing only the node itself.

We now run a *maximum adjacency search* algorithm, sometimes also called *maximum cardinality search* algorithm, ([20]) in order to obtain an arbitrary minimum (s, t)-cut of the graph. The maximum adjacency search algorithm is defined as follows: We start with an empty list L to which we add an arbitrary node of P. We then continually add the most tightly connected node from V to L, i.e. the node which is connected to the nodes in L via edges of the most total weight. Stoer and Wagner ([20]) have shown that the edges between the last two nodes s and t added to L form a minimum (s, t)-cut. We use the value of this cut in order to decide whether to merge the nodes s and t or whether to separate them. If the cut has value less than c we separate the nodes, otherwise we perform a merge. Here the separation of the nodes s and t means that we remove all edges in the cut from the edges E of the partition graph P. In the case of a merge we combine the nodes s and t and merge the outgoing edges, i.e. we replace the set of nodes V of P by the set $V' = V \backslash \{s, t\} \cup \{v'\}$. The edges E of P are modified by removing all edges adjacent to s and t and adding an edge $e' = \{v', u\}$ of weight $w(\{s, u\}) + w(\{t, u\})$ where $w(e)$ denotes the weight of edge e if it exists and is equal to zero otherwise. Furthermore we adjust the mapping M by setting $M(v') = M(s) \cup M(t)$.

We continually run this algorithm until P contains no edges, i.e. until $E = \emptyset$. The sets of nodes mapped to each ot the nodes of P by M now represent candidate subgraphs for the decomposition. Note though that we have only cut and merged according to minimum (s, t)-cuts and not according to minimum cuts. This means that the specific sequence in which we have performed the cuts may influence the result, e.g. if we merged based on a minimum (s, t)-cut which is not a minimum cut. This can be remedied by repeating the procedure on the resulting subgraphs until it returns a subgraph of only one node, i.e. until no separation step is performed during the decomposition. This is due to the fact that this procedure always cuts a subgraph of connectivity less than c at least once, as Chang et al. have shown (see Cutability Property in [8]). In order to speed up this computation we use the heap data structure proposed and analyzed by Chang et al. ([8]).

Thus we conclude that this procedure correctly decomposes a given subgraph as Chang et al. have also stated in Theorem 3.1 in [8]. In the following section we show that this decomposition can be implemented in polynomial time.

5.4 Running Time

The main bottleneck of the algorithm lies in the decomposition updates. The following lemma shows a polynomial running time for these updates. The other parts of the algorithm can be implemented in polynomial time already.

▶ **Lemma 16.** *The subroutine decompose which decomposes a subgraph can be implemented in polynomial time* $O(\alpha|V|^2|E|)$.

Proof. The worst case is given when the whole tree has to be recomputed. We first discuss the time complexity of decomposing a single tree node v. Let the corresponding subgraph be denoted by $G_v = (V_v, E_v)$. Since each iteration of the subroutine *decompose* performs at least one cut as long as the connectivity of the given graph is smaller than the current threshold c we conclude that after at most $|V_v|$ iterations of decompose a correct decomposition is found.

Each step of *decompose* can be performed in $O(|V_v| \cdot |E_v|)$ as the maximum adjacency search algorithm finds an arbitrary minimum (s,t)-cut in time $O(|E_v|)$ as shown in theorem 4.1 in [8] and as there are at most $|V_v|$ minimum (s,t)-cuts computed for each invocation of *decompose*.

Hence the complexity of decomposing the subgraph represented by a tree node v is in $O(|V_v|^2|E_v|)$. Let C_v denote the time needed for the decomposition of the subgraph represented by decomposition tree node v.

We now sum this complexity over the nodes for each connectivity level of the decomposition tree. To this end let *level*(i) denote all nodes in the decomposition tree which are of connectivity exactly i.

$$\sum_{i=0}^{\alpha} \sum_{v \in \text{level}(i)} C_v \leq \sum_{i=0}^{\alpha} O(|V|^2|E|) \in O(\alpha|V|^2|E|).$$

We conclude our analysis of the time complexity by observing the polynomial-time complexity of $O(\alpha|V|^2|E|)$. ◀

We conclude our analysis of the running time by observing that the remaining subroutines can be implemented in polynomial time which results in the following theorem on the running time.

▶ **Theorem 17.** *The algorithm* $_P$CREP *can be implemented in polynomial time.*

Proof. The theorem follows from the Lemma 16 and the fact that the remaining subroutines for edge insertions and component deletions can hence also be implemented in polynomial time. Note in particular that there can be at most one component deletion during the update step after any given request. ◀

6 Evaluation

In order to complement our analytical results and shed light on the performance of our algorithm in practice, we implemented our algorithm and conducted experiments under real-world traffic traces. We also compare our algorithm to reference algorithms.

6.1 Reference Algorithms

We first present the algorithms we compare in order to evaluate our results We consider both algorithmic approaches based on maintaining a second-order partition of the nodes into components we discussed in Section 3, i.e. pCrep where on a component deletion all edges inside of and adjacent to the component are deleted and Crep-Core which only deletes internal edges. Both algorithms start with randomly initialized mappings of nodes to servers We have shown the competitive ratio of pCrep in Theorem 15 and its polynomial running time in the previous section.

We compare these algorithms both with the static graph partitioning algorithm METIS_PartGraphRecursive implemented in the METIS framework ([14, 15]) which we will refer to from now on as Static and the adaptive/dynamic algorithm
ParMETIS_V3_AdaptiveRepart ([16, 19, 18]) available in the ParMetis framework, referred to as Adapt. Both frameworks are known to produce very good results and to be very fast.

We run Adapt in an online manner while maintaining a communication graph by inserting an edge or incrementing a corresponding edge weight for each request, i.e. Adapt is called after each request is inserted into the graph.

For Static we first record the communication graph and give it as input to Static. We then compute the cost of Static due to migrations from the random initial mapping of nodes to servers and due to remote requests, i.e. the total weight of edges between nodes of the communication graph which Static mapped to different servers.

6.2 Input Data

As input data we use several HPC traces (Mocfe, NeckBone, MultiGrid), the nature of the data is described in more detail by Avin, Ghobadi, Griner and Schmid in [4]. For the sake of readability we use the abbreviations NB for NeckBone and MG for MultiGrid.

All data sets contain 1024 different communication nodes and are limited to the first 300 000 requests. The value of α is set to 6 and the algorithm was tasked to partition the nodes into 32 clusters of size 32 each, i.e. $k = l = 32$. The dynamic algorithms are allowed to use augmentation with a factor of 2.1, i.e. for the dynamic algorithms the maximum cluster capacities are $\lfloor 32 \cdot 2.1 \rfloor = 67$.

6.3 Comparison of pCrep and Crep-Core

First we compare the results of pCrep and Crep-Core. Figure 5 shows the resulting total cost of both algorithms on the different input sets as well as how this cost is split between communication and migration costs. One can see that pCrep always produces better results, especially for the second and third input set.

We now discuss the differences of the two algorithms in terms of communication and migration cost. The figures illustrate that pCrep pays only very slightly less for communication requests than Crep-Core whereas the former pays significantly reduced migration cost. This suggests that the deletion of adjacent edges indeed improves the quality of the results by reducing the number of migrations.

Finally Figure 6 compares the running times of both algorithms. One can see that pCrep performs drastically better than Crep-Core in this regard. This may be due to the fact that pCrep deletes edges more frequently and thus needs to take less edges into account when updating the decomposition tree.

Also note that both algorithms generally pay more for migrations than for communication. This suggests that there may be room for fine-tuning these algorithms in order to achieve

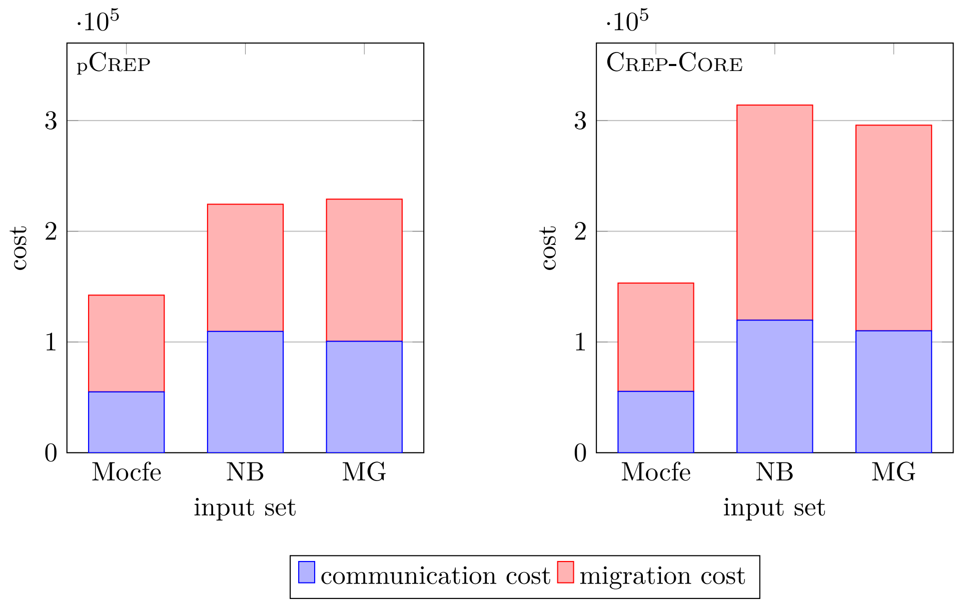

Figure 5 comparison of the total cost of pCrep (left) and Crep-Core (right)

a more balanced distribution of the cost. One such adjustment is investigated in the next section, namely whether we can improve the results by changing the algorithms in such a way that they only merge once a connectivity of $2 \cdot \alpha$ is reached.

6.4 On the Influence of the Connectivity Threshold

In the previous section we observed that both Crep-Core and pCrep have higher migration cost than communication cost. This leads to the question whether we can find adjustments that allow us to achieve a better balance of the different costs. In this section we investigate the influence of the connectivity threshold which determines when components are merged. Up until this point it was set to α. As we show there lies potential in adjusting this threshold in order to improve the quality of the solution.

Namely we discuss the results of both algorithms for the case where they only merge components once a connectivity of at least $2 \cdot \alpha$ is reached. Note that this only affects the analysis by constant factors, namely the statements from Lemma 6 and Lemma 7 are multiplied by a factor of two. This may only impact the other cost bounds by a factor of two which leaves our bound on the competitive ratio of pCrep unaffected. Also note that we do not change the cost of a node migration, this cost is still α.

Figure 7 shows the total cost of pCrep and Crep-Core for this scenario. One can see that the costs of both algorithms are reduced, but Crep-Core has improved significantly more than pCrep.

When looking at the distribution of these costs to communication and migration costs one can see that now the communication costs of both algorithms are higher than the migration cost in all cases. For Crep-Core this has improved the balance of both costs, but pCrep now has significantly higher communication costs than migration costs. This suggests that by adjusting the exact value of connectivity at which these algorithms perform their merges one might be able to further improve the quality of the results. It may even be possible to adjust this parameter dynamically based on some data gathered as requests are processed.

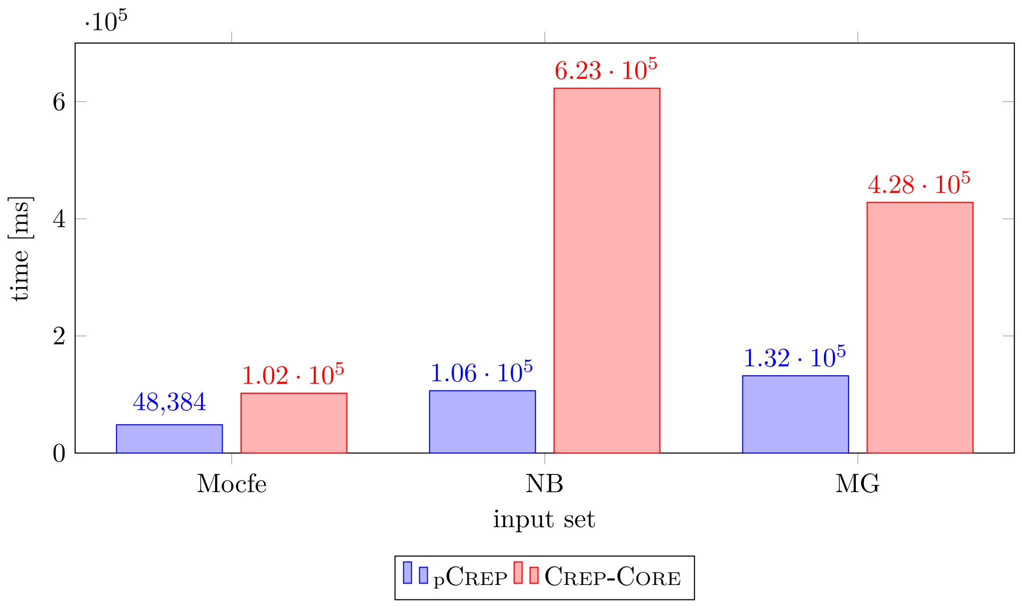

Figure 6 comparison of running time of pCREP and CREP-CORE

These improvements come at a cost of running time as Figure 8 shows, almost doubling or tripling the times of the previous results in some cases. Especially the already worse run times of CREP-CORE are drastically increased.

6.5 Results of ADAPT and STATIC

Figure 9 shows the total cost of the solutions of ADAPT and STATIC. One can see that STATIC is able to produce results that far surpass ADAPT. In fact the results of STATIC are also a significant improvement over the results of pCREP and CREP-CORE presented before. But it is also important to note that STATIC is only useful in scenarios where the communication patterns and frequencies stay mostly the same over time. Otherwise the approach to record the communication graph and then use STATIC in order to compute a partitioning may lead to inconsistent results.

We stress that the implementations of STATIC and ADAPT rely on heuristics only and do not provide any guarantees while we have shown that our algorithm pCREP is a competitive algorithm with competitive ratio $O(2/\epsilon \cdot k \log k)$.

Finally Figure 10 illustrates the run times of ADAPT and STATIC. Note that static is only run once on the communication graph after all requests have been revealed which means that it is expected for STATIC to have the fastest running time. But also ADAPT achieves faster running times than our dynamic implementations.

7 Related Work

The closest work to ours is by Avin et al. [2] who initiated the study of the dynamic balanced graph partitioning problem. The authors present a $O(k \log k)$-competitive algorithm with augmentation $2 + \epsilon$ for any $\epsilon > 1/k$; this algorithm however has a super-polynomial runtime, which we improve upon in this paper. In their paper, Avin et al. also show a lower bound of $k - 1$ for the competitive ratio of any online algorithm on two clusters via a reduction

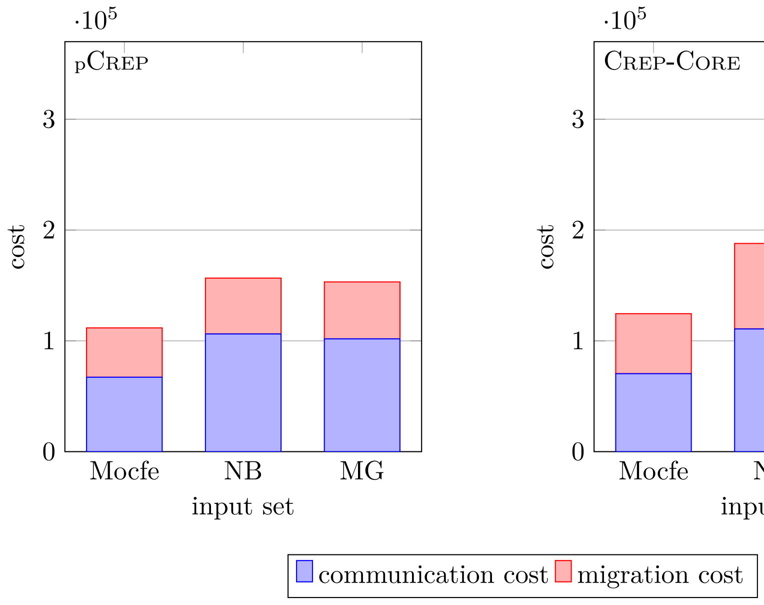

Figure 7 comparison of the total cost of pCrep (left) and Crep-Core (right) in the case where merges are performed at connectivity $2 \cdot \alpha$

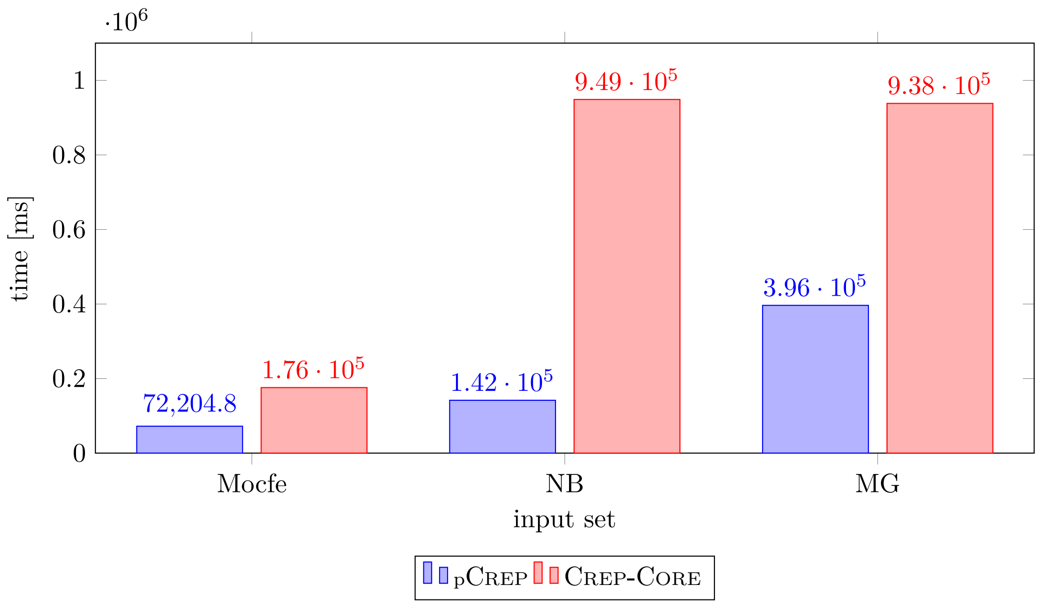

Figure 8 comparison of running time of pCrep and Crep-Core in the case where merges are performed at connectivity $2 \cdot \alpha$

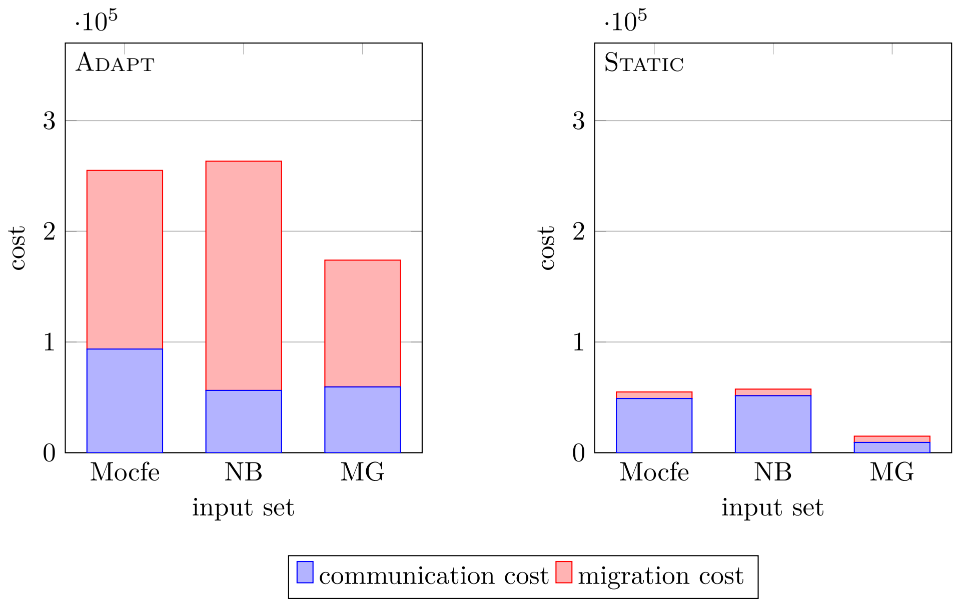

Figure 9 comparison of the total cost of ADAPT (left) and STATIC (right)

to online paging. Restricted variants of the balanced repartitioning problem have also been studied. Here one assumes certain restrictions of the input sequence σ and then studies online algorithms for these cases. Avin et al. [3] assume that an adversary provides requests according to a fixed distribution of which the optimal algorithm OPT has knowledge while an online algorithm that is compared with OPT has not. Further the authors restrict the communication pattern to form a ring-like pattern, i.e. for the case of n nodes $0, ..., n-1$ only requests r of the form $r = \{i \mod n, (i+1) \mod n\}$ are allowed. For this case they present a competitive online algorithm which achieves a competitive ratio of $O(\log n)$ with high probability. Henzinger et al. [13] study a special *learning variant* of the problem where it is assumed that the input sequence σ eventually reveals a perfect balanced partitioning of the n nodes into l parts of size k such that the edge cut is zero. In this case the communication patterns reveal connected components of the communication graph of which each forms one of the partitions. Algorithms are tasked to *learn* this partition and to eventually collocate nodes according to the partition while minimizing communication and migration costs. The authors of [13] present an algorithm for the case where the number of servers is $l = 2$ that achieves a competitive ratio of $O((\log n)/\epsilon)$ with augmentation ϵ, i.e. each server has capacity $(1+\epsilon)n/2$ for $\epsilon \in (0, 1)$. For the general case of l servers of capacity $(1+\epsilon)n/l$ the authors construct an exponential-time algorithm that achieves a competitive ratio of $O((l \log n \log l)/\epsilon)$ for $\epsilon \in (0, 1/2)$ and also provide a distributed version. Additionally the authors describe a polynomial-time $O((l^2 \log n \log l)/\epsilon^2)$-competitive algorithm for the case with general l, servers of capacity $(1+\epsilon)n/l$ and $\epsilon \in (0, 1/2)$.

The dynamic balanced graph partitioning problem can be seen as a generalization (or symmetric version) of online paging. In the online paging problem [11], [9] one is given a scenario with a fast cache of k pages and $n-k$ pages in slow memory. Pages are requested in an online manner, i.e. without prior knowledge of future requests. If a requested page is in the cache at the time of the request it can be served without cost. If it is in slow memory however, then a *page fault* occurs and the requested page needs to be moved into the cache. If the cache is full then a page from the cache needs to be evicted, i.e. moved to the slow

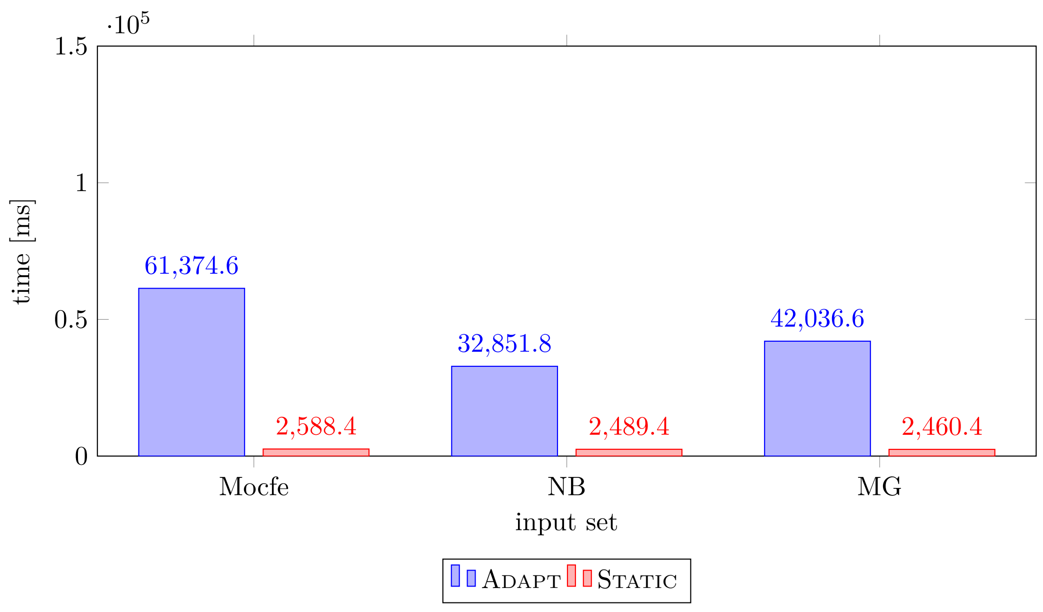

Figure 10 comparison of running time of ADAPT and STATIC

memory in order to make space for the requested one. The goal is to design algorithms which minimize the number of such page faults. However, the standard version of online paging has no equivalent to the option of serving a request remotely as is possible in the Dynamic Balanced Graph Partitioning problem. The variant *with bypassing* allows an algorithm to access pages in slow memory without moving them into the cache, thus providing such an equivalent. It is worth stressing however that in our problem requests involve two nodes while in Online Paging the nodes themselves are requested.

The static balanced graph partitioning problem is the static offline variant of the problem of this paper. In this version an algorithm may not perform any migrations, but has perfect knowledge of the request sequence σ and then needs to provide a perfectly balanced partitioning of the $n = k \cdot l$ nodes into l sets of equal size k that minimizes cost, i.e. the weight of edges between the servers. This scenario can be modelled as a graph partitioning problem where the weight of an edge corresponds to the number of requests between its end points in the input sequence σ. An algorithm then has to provide a partition of the nodes into sets of exactly k nodes each while minimizing the total edge weights between partitions, i.e. an algorithm needs to minimize the edge cut of the graph. This problem is NP-complete ([1]) and for the case where $l \geq 3$, Andreev and Räcke ([1]) have shown that there is no polynomial time approximation algorithm which guarantees a finite approximation factor unless P=NP.

There are several algorithms and frameworks for graph partitioning problems. Usually these frameworks employ heuristics in order to achieve their results. The most successful such heuristic is *Multilevel Graph Partitioning* ([7]). This method consists of three phases. Initially the graph is repeatedly coarsened into a hierarchy of smaller graphs in such a way that cuts in the coarse graphs also correspond to cuts in the finer graphs. On the coarsest level a (potentially expensive) algorithm is used in order to compute an initial partition. This partitioning is then transferred to the finer graphs. In this process one usually uses other local heuristics in order to improve the partition quality even further with every step. METIS ([14, 15]) and Jostle ([21, 22]) are examples of libraries that utilize this multilevel approach. We choose METIS as a reference for our empirical evaluation.

More generally, clustering has been studied within a variety of different contexts from data mining to image segmentation [6, 23, 17], and is the process of generating subsets of elements with high similarity [12]. However, we consider an online problem, i.e. algorithms need to react dynamically to changes in the graph and need to maintain their data structures and adapt accordingly whereas clustering considers complete data sets which are static.

8 Future Work

While our algorithm does not only achieve a polynomial runtime and an almost competitive ratio (up to a logarithmic factor), our work leaves upon several interesting directions for future research. On the theoretical front, it would be interesting to explore how to close the gap between upper and lower bound on the competitive ratio, and to study randomized algorithms. On the practical front, we believe that our algorithm can be further engineered and optimized to achieve a lower runtime in practice, as well as an improved empirical competitive ratio under real (non worst-case) workloads. These kinds of adjustments may be achieved for example by changing certain algorithm parameters such as the connectivity threshold There may also be potential in adapting and improving our decomposition tree data structure in order to improve running times.

References

1 Konstantin Andreev and Harald Räcke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, oct 2006. `doi:10.1007/s00224-006-1350-7`.

2 Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic Balanced Graph Partitioning. *arXiv preprint arXiv:1511.02074v5*, 2015. `arXiv:http://arxiv.org/abs/1511.02074v4`.

3 Chen Avin, Louis Cohen, Mahmoud Parham, and Stefan Schmid. Competitive clustering of stochastic communication patterns on a ring. *Computing*, 101(9):1369–1390, sep 2018. `doi:10.1007/s00607-018-0666-x`.

4 Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. Measuring the Complexity of Packet Traces. *arXiv:1905.08339v1*, 2019. `arXiv:http://arxiv.org/abs/1905.08339v1`.

5 Chen Avin, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Online Balanced Repartitioning. In *Proc. 30th International Symposium on Distributed Computing (DISC)*, pages 243–256. Springer Berlin Heidelberg, 2016. `doi:10.1007/978-3-662-53426-7_18`.

6 Abla Chouni Benabdellah, Asmaa Benghabrit, and Imane Bouhaddou. A survey of clustering algorithms for an industrial context. *Procedia Computer Science*, 148:291–302, 2019. `doi:10.1016/j.procs.2019.01.022`.

7 Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering*, pages 117–158. Springer International Publishing, 2016. `doi:10.1007/978-3-319-49487-6_4`.

8 Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. Efficiently computing k-edge connected components via graph decomposition. In *Proceedings of the 2013 international conference on Management of data - SIGMOD '13*. ACM Press, 2013. `doi:10.1145/2463676.2465323`.

9 Leah Epstein, Csanád Imreh, Asaf Levin, and Judit Nagy-György. On variants of file caching. In *Automata, Languages and Programming*, pages 195–206. Springer Berlin Heidelberg, 2011. `doi:10.1007/978-3-642-22006-7_17`.

10 Leah Epstein and Hanan Zebedat-Haider. Rent or buy problems with a fixed time horizon. *Theory of Computing Systems*, 56(2):309–329, jun 2014. `doi:10.1007/s00224-014-9552-x`.

11 Amos Fiat, Richard Karp, Mike Luby, Lyle McGeoch, Daniel Sleator, and Neal E. Young. Competitive Paging Algorithms. *arXiv preprint cs/0205038*, 2002. `arXiv:cs/0205038v1`, `doi:10.1016/0196-6774(91)90041-V`.

12 Erez Hartuv and Ron Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4-6):175–181, dec 2000. `doi:10.1016/s0020-0190(00)00142-3`.

13 Monika Henzinger, Stefan Neumann, and Stefan Schmid. Efficient Distributed Workload (Re-)Embedding. *no idea*, 2019. `arXiv:http://arxiv.org/abs/1904.05474v1`.

14 George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, jan 1998. `doi:10.1137/s1064827595287997`.

15 George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, jan 1998. `doi:10.1006/jpdc.1997.1404`.

16 George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, jan 1999. `doi:10.1137/s0036144598334138`.

17 M. Pavan and M. Pelillo. A new graph-theoretic approach to clustering and segmentation. In *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* IEEE Comput. Soc, 2003. `doi:10.1109/cvpr.2003.1211348`.

18 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, 2000. `doi:10.1109/sc.2000.10035`.

19 Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, dec 1997. `doi:10.1006/jpdc.1997.1410`.

20 Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, jul 1997. `doi:10.1145/263867.263872`.

21 C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, jan 2000. `doi:10.1137/s1064827598337373`.

22 Chris Walshaw and Mark Cross. JOSTLE: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques*, pages 27–58, 2007.

23 Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993. `doi:10.1109/34.244673`.