Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Polynomial Time Competitive Repartitioning of Dynamic Graphs
Kompetitive Repartitionierung Dynamischer Graphen in polynomieller Zeit

Tobias Forner

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Polynomial Time Competitive Repartitioning of Dynamic Graphs

Kompetitive Repartitionierung Dynamischer Graphen in polynomieller Zeit

Tobias Forner

Supervisor:	Prof. Dr. Harald Räcke
Advisors:	Prof. Dr. Harald Rcke
	Univ.-Prof. Dr. Stefan Schmid
Submission Date	15.03.2020

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

Date	Tobias Forner

Abstract

...

Contents

1 Introduction

2 Problem Definition

The task is to maintain a partitioning of a dynamic graph consisting of $n = k \cdot l$ nodes that communicate with each other into k parts, each of size l while minimizing both the cost due to communication and due to node migrations defined as follows. The communication cost is zero if both nodes are located on the same server at the time the request needs to be served and it is normalized to one if they are mapped to different servers. An algorithm may perform node migrations in order to change the mapping of nodes to servers prior to serving the communication request at time t. Such a move of one vertex incurs cost $\alpha > 1$.

More formally we are given 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.

We are in the realm of competitive analysis and as a result we compare an online algorithm ON to the optimal offline algorithm Opt. ON 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 while Opt starts with perfect knowledge of the complete sequence σ at time 0.

The goal is to design an online algorithm Alg with a good competitive ratio with regard to Opt defined as follows.

A online algorithm ON is $\rho - competitive$ if there exists a constant β such that

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

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

Often we allow the online algorithm to use larger capacities per server. In this case we speak of an *augmentation* of δ in the case where the online algorithm is allowed to assign $\delta \times n/k$ nodes to each server where $\delta > 1$.

3 Related Work

3.1 Dynamic Balanced RePartitioning

Avin et al.[1] initiated the study of the online variant of the Balanced RePartitioning (BRP) problem that is at the core of this also the topic of this thesis. We discovered flaws in their competitive analysis which makes us question their results. These flaws are discussed in greater detail in Section 4.

3.2 Learning Variant of BRP

[5] studies a special *learning variant* of the Dynamic Balanced Graph Partitioning problem specified above. In this version 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.

[5] 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 $(1+\epsilon)n/2$ the authors construct an exponential-time algorithm that achieves a competitive ratio of $\emptyset((l \log n \log l)/\epsilon)$ 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 $\epsilon \in (0, 1/2)$.

It is important to stress that the assumption that the requests reveal a perfect partitioning of the communication nodes is not applicable for most practical applications and thus it is important to study the general BRP problem without restricting σ.

4 Problems With Analysis in Previous Work

The problem in the analysis of [1] arises due to their usage of the concept of $F(c)$: This contains only edges incident to components from $S(c)$ that arrived after the involved components from $S(c)$ were created by Crep.In the analysis it is assumed that only those edges that are contained in $F(c)$ can contribute to the creation of component c. As we will show that is not the case and as a result it is very challenging to separate the costs of Opt that are due to the requests that actually led to the creation of c.

In order for the approach in [1] to work the different sets $F(c)$ that occur as Crep handles input σ need to form a partition of all requests such that each request can be mapped to one unique set $F(c)$. Only in this case it is possible to lower bound the cost of Crep via the requests in $\bigcup_{c \in \mathrm{DEL}} F(c)$.

Figure 1 shows an example sequence of requests for which this approach does not work. The diagram shows horizontal lines, each representing one of the vertices. A vertical line represents a communication request between its end points. For example the sequence shown contains a communication request between nodes 1 and 2 at time $t = 5$.

In this sequence the first two requests are not contained in $F(c)$ for any component c as nodes 3 and 4 were part of other components that were eventually deleted by Crep before finally being merged at time $t = 15$.

In an older version ([2]) of the paper mentioned above we have also discovered some problems. In this version merges and the component structure is very similar, but upon a deletion of a component the algorithm not only deletes inner edges but also those that leave the component, i.e. all edges $e = u, v$ are deleted where u or v are inside the component. However the additional deletions of edges that are leaving the

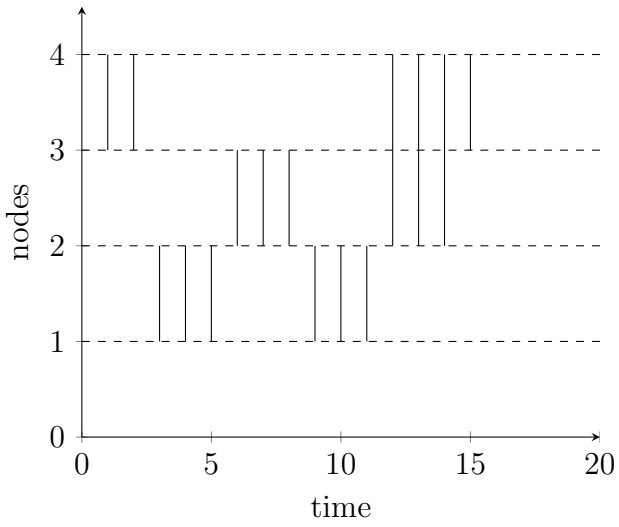

Figure 1: problem in the new version

component are not accounted for in the analysis. We show that this approach may lead to scenarios where the amount of edges that are deleted that leave the component may greatly exceed the amount of inner edges. Just like in the case above this fact makes it very challenging to attribute specific costs to Opt as this makes it very difficult to uniquely identify the actions of Opt that may lead to a reduction of communication costs with a given component that Crep discovers during its execution. This complicates the analysis, but we show in Section 6 how it can be adapted while preserving the competitive ratio.

Figure 2 illustrates an example of the problems mentioned above for this older version of Crep. In the illustration it is assumed that $\alpha = 3$ and $k = 3$. In this case the input sequence leads Crep to first merge nodes 1 and 2 at time $t = 3$ and then to add node 3 to the resulting component at time $t = 6$. The next two requests do not quite lead to the merge of the node 4 with the component. Instead a series of requests follows where node 1 communicates with other nodes that are outside of its component without any merges. Note that this sequence can be extended until that node 1 has communicated with every node except from the nodes $1, 2, 3, 4$. Finally the first 4 nodes are merged at time $t = 15$ at which point the resulting component is deleted as well as *all* edges in the sequence.

5 Algorithm Overview

In this section we give a high-level overview of the algorithm. Implementation details will be discussed in greater detail in Section 8.

5.1 Overview

The general idea of the algorithm is to maintain a second-order partitioning of the nodes into *communication components* which are sets of nodes that communicated frequently. As more requests from the input sequence σ are revealed to the algorithm

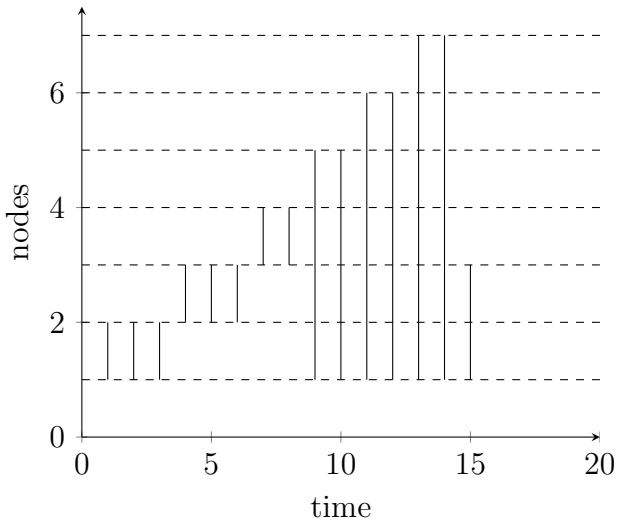

Figure 2: problem in the older version

the components grow in size until the algorithm discovers a component that is too big and hence decides to delete it.

More formally, initially each node is assigned to a singleton component, but as the algorithm is given the input sequence σ new communication patterns unfold which the algorithm keeps track of by maintaining a graph in which the nodes represent the actual communication nodes and the weighted edges represent the number of paid communication requests between the end points of each edge, 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 o different servers at the time of the request.

Our approach defines sets of nodes that communicate frequently as those that represent subgraphs with high connectivity, in this case those with connectivity at least α.

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. If the subset has cardinality greater than k the algorithm instead splits the components whose nodes are contained in S into singleton components and deletes all edges that were contained in S, i.e. every edge where both end points are elements of S.

The collocation of such component sets of at most k individual communication nodes is always possible due to the allowed augmentation of $2 + \epsilon$. This guarantees by an averaging argument that there is almost at least one cluster with capacity at least k. We have also included a pseudocode description of the algorithm in Algorithm 1.

Algorithm 1 DynamicDecomp

```
Initialize an empty graph on n nodes
turn each of the n nodes into a singleton component
for all r = {u, v} ∈ σ do
  if S(v) ≠ S(u) then
    w({u, v}) ← w({u, v}) + 1
  end if
  if ∃ component set X with connectivity at least α and |X| > 1 and nodes(X) ≤ k
  then
    merge(X)
  end if
  if ∃ component set Y with connectivity at least α and |Y| > 1 and nodes(Y) > k
  then
    delete(Y)
  end if
end for
```

6 New Analysis

6.1 Algorithm Definitions

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

Note that such an edge can only have positive weight if its endpoints are in different components.

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

6.2 Structural Properties

Note: These properties are changed to use the connectivity based approach which generally simplifies them, but guarantees slightly less minimum edge weight within mergeable component sets.

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

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.

Proof. We proof 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 true subset S of components with connectivity at least α. We may assume that t is the earliest time for which S has connectivity α.

Then the incrementation of the weight of edge e at time t raised the connectivity of S, but S was not merged into the α-connected component C that was created at time t and must hence 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.

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 true subsets S of components have connectivity at most α and a mergeable component set S has connectivity exactly α.

Proof. This lemma follows directly from lemma 2 as connectivities can only increase by at most 1 at each time t.

Lemma 6. The weight between the components of a component subset S of connectivity α is at least $|S|/2 \cdot alpha$.

Proof. Consider the sum of the weighted degrees of all components:

$$\sum_{c \in S} deg_S(c) = 2 \sum_{e \in S} w(e)$$

The equality follows as the left sum counts each edge twice, once for each endpoint. Now consider the fact that each component must have degree at least α with respect to the edges in S as S has connectivity α and hence the lemma follows.

Lemma 7. The weight between the components of a component subset S of connectivity α is at most $(|S| - 1) \cdot \alpha$.

Proof. We iteratively partition S into subsets via MinCuts with regard to edge weight, i.e. we consider a MinCut of S which partitions S into the subsets S_1 and S_2 and iteratively partition the resulting sets until all sets contain only one component each. As this required at most $|S| - 1$ cuts of value at most α.

6.3 Upper Bound On CREP

We define the following notions for a component $C \in \text{DEL}(\sigma)$, i.e. the subgraph induced by the nodes of C has connectivity at least α and C consists of more than k nodes:

Let $epoch(C)$ denote the (node, time) pairs of nodes in C starting at the time after the time $\tau(node)$ when n was last turned into a singleton component, i.e. $epoch(C) = \bigcup_{n \in nodes(C)} \{n\} \times \{\tau(n) + 1, ..., \text{del-time}(C)\}$. This allows us 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 $epoch(C)$ whose corresponding requests are deleted because of the deletion of component C and call the set of those requests $req(C)$. We split the requests from $req(C)$ into two sets: $core(C)$ contains all requests for which both nodes have already been assigned to C at the time of the request, i.e. $core(C) = \{r = \{u, v\} \in \sigma | (u, time(r)) \in epoch(C)$ and $(v, time(r)) \in epoch(C)\}$. These are the requests that led to the creation of component C. $halo(C)$ contains all requests from $req(C)$ for which exactly one end point was associated with C at the time of the request. Note that this means that $halo(C) = req(C) \backslash core(C)$.

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

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

Proof. The lemma follows directly from Lemma 7 due to the fact that component sets of connectivity α get merged immediately by Crep as shown in Lemma 5. □

Together with the fact that Crep pays for all requests in $halo(C)$ for deleted components C this gives us an upper bound on the costs of Crep together with the cost incurred to Crep due to requests and migrations of the components that do not get deleted during its execution, i.e. those components that are present after all requests from σ have been processed. □

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

Lemma 9. With augmentation $2+\epsilon$, Crep pays at most migration costs of $\sum_{C \in \text{DEL}(\sigma)} |C| \cdot \log k \cdot \alpha + \sum_{C \in FINAL-COMPS(\sigma)} |C| \cdot \log k \cdot \alpha$.

Proof. First note that Crep only performs migrations when it merges components. We fix a component $C \in \text{DEL}(\sigma) \cup FINAL - COMPS(\sigma)$ 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 reserves additional space $\lceil \epsilon \cdot |B|$ for each component B and only moves component B when a merge results in a component of size more than $(1+\epsilon) * |B|$ each node of C was moved at most $\log k$ times. Summing over all nodes in C bounds the number of migrations by $|C| \cdot \log k$ and leads to the desired bound on the migration costs as each node migration incurs cost α to Crep. □

Finally we summarize our results in the following lemma.

Lemma 10. With augmentation $2+\epsilon$, Crep pays at most cost $2 \cdot \sum_{C \in \text{DEL}(\sigma)} |C| \cdot \log k \cdot \alpha + |halo(C)| + 2 \cdot \sum_{C \in FINAL-COMPS(\sigma)} |C| \cdot \log k \cdot \alpha + FINAL - WEIGHTS$.

Proof. Note that the communication costs from Lemma 8 are bounded by $\sum_{C \in \text{DEL}(\sigma)} |C| \cdot k \cdot \alpha$. Similarly we can bound the migration costs due to components from $FINAL - COMPS$ by $\sum_{C \in FINAL-COMPS(\sigma)} |C| \cdot \log k \cdot \alpha$. Accounting for the requests in $halo(C)$ for a deleted component C and weight of the edges that are present at the end of the execution of Crep leads to the lemma. □

6.4 Lower Bound on Opt

We split the analysis of the cost Opt has to pay into two parts. First we examine the cost incurred to Opt due to requests that led Crep to create (and delete) a component $C \in \text{DEL}(\sigma)$. Then we provide a lower bound on the cost incurred to Opt due to requests from $halo(C)$ for $C \in \text{DEL}(\sigma)$.

Lemma 11. Opt pays at least $\lceil \frac{|C|}{k} \rceil \cdot \alpha/2$ due to requests from $core(C)$ for any $C \in \text{DEL}(\sigma)$.

Proof. We fix $C \in \text{DEL}(\sigma)$ arbitrarily and observe that any solution that does not move any vertices of C during the epoch of C incurs cost at least $\lceil \frac{|C|}{k} \rceil \cdot \alpha/2$ as the nodes of C need to be distributed on at least $\lceil \frac{|C|}{k} \rceil$ servers and Lemma 6 gives a suitable lower bound on the weight between these servers and hence also on the number of communication requests an algorithm which chooses this static configuration must pay for.

We show that this static solution can not be improved via migrations. Observe that a migration involving only one node that is mapped to a server from the static configuration may only increase the communication costs for requests in $core(C)$ as it increases the number of partitions in the result of Lemma 6. A migration between two servers used in the static configuration may only decrease the weight between the two servers involved and incurs cost 2α to Opt. Note that the proof of Lemma 6 only requires the weight between one component set S and all the other component sets that together form a mergeable set to be at least α. If Opt now performs a swap of two nodes we can attribute cost 2α to Opt for the connection between these two servers which may only increase the cost we can attribute to Opt. Hence the lemma follows. □

Lemma 12. Opt pays at least $\sum_{C \in \text{DEL}(\sigma)} |halo(C)|/k$ for serving requests from $\bigcup_{C \in \text{DEL}(\sigma)} halo(C) =: H$.

Proof. First note that we may ignore requests from H that both Crep and Opt pay for as including these may only give Crep an advantage in our analysis.

In the following we only consider those requests from H that Crep pays for while Opt does not. We relate the number of those requests that Opt does not pay to the number of migrations Opt performs by assigning each request from H to the next migration performed by Opt that moves a node involved in the request.

We now bound the number of requests from H Opt may be able to serve without cost by assigning nodes to servers optimally. Consider all requests from H that happen between nodes that Opt has placed on the same server between Opt migrations that involve nodes from that server. As the requests from H have not led Crep to perform a migration there can be at most $(k-1) \cdot \alpha$ such requests before Opt changes the nodes mapped to the server by performing a migration. As this migration incurs cost 2α to Opt we conclude that Opt may only improve its cost by a factor of k over the cost paid by Crep for serving requests from H. □

6.5 Competitive Ratio

7 Old Analysis

Note: The following analysis is mostly the same as in [2].

7.1 Lower Bound on OPT

Definition 13. Define the time of creation of a component c managed by Crep as $\tau(c)$. For a non-singleton component c it is the time at which other components were merged

resulting in c, if c is a singleton component $\tau(c)$ is 0 if c existed at time 0 and the last time a component containing c was deleted otherwise.

Definition 14. For a non-singleton component c, define

$$F(c) = \biguplus_{b \in S(c)} \times \{\tau(b)+1, ..., \tau(c)\}.$$

Definition 15. A communication request between nodes x and y at time t is *contained* in $F(C)$ if $(x, y) \in F(c)$ and $(y, t) \in F(c)$.

Definition 16. A migration of node x performed by OPT at time t is *contained* in $F(C)$ if $(x, t) \in F(c)$

Definition 17. For any component c, define $\textsc{Opt}(c)$ as the cost incurred by Opt due to requests contained in $F(c)$ plus the cost due to migrations contained in $F(c)$.

Definition 18. Define Y_c as the set of clusters containing nodes of c in the solution of Optafter Optperforms its migartions (if any) at time $\tau(c)$.

Then the total cost of Opt can be lower-bounded: $\textsc{Opt} \leq \sum_c \textsc{Opt}(c)$.

Note: The following lemma is a changed from lemma 5 in [2] in order to account for the fact that the connectivity-based approach only guarantees at least weight $|B| \cdot \alpha/2$ between different bundles whereas the Crep-density approach from [2] guarantees a higher edge weight of $(|B| - 1) \cdot \alpha$.

Lemma 19. For any non-trivial component c, it holds that $\textsc{Opt}(c) \geq 1/2 \cdot |Y_c| \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha)$.

Proof. Fix a component $b \in S(c)$ and any node $x \in b$. Let opt-mig(x) be the number of Optmigrations of node x at times $t \in \{\tau(b)+1, ..., \tau(c)\}$. Furthermore, let Y'_x be the set of clusters that contained x at some moment in time $t \in \{\tau(b)+1, ..., \tau(c)\}$ (in the solution of Opt). We extend these notions to components: $\textsc{opt-mig}(b) := \sum_{x \in b} \textsc{opt-mig}(x)$ and $Y_{b'} := \bigcup_{x \in b} Y'_x$. Observe that $|Y'_b| \leq |Y_b| + \textsc{opt-mig}(b)$.
Now aggregate components of $S(c)$ into component-sets called *bundles*, so that any two bundles always have their nodes in disjoint clusters. To achieve this, we construct a hypergraph, whose nodes correspond to clusters from $\cup_{b \in S(s)} Y'_b$. Each component $b \in S(c)$ defines a hyperedge that connects all nodes (clusters) that are in Y'_b. Now we examine the connected components of this hypergraph which we call *hypergraph parts* from now on. We bound the number of hypergraph parts by using the fact that the hyperedge related to component b connects $|Y'_b|$ nodes:

$$\begin{aligned} B &\geq |\bigcup_{b \in S(s)} Y'_b| - \sum_{b \in S(c)} (|Y'_b| - 1) \\ &\geq |Y_c| - \sum_{b \in S(c)} (|Y_b| - 1) - \sum_{b \in S(c)} \textsc{opt-mig}(b). \end{aligned}$$

Due to Lemma 6 the number of communication requests in $F(c)$ that are between different bundles is at least $|B| \cdot \alpha/2$. Each of these requests is by definition paid by Optas each request involves a communication between nodes which Optstored in different clusters. Opt(c) also involves $\sum_{b \in S(c)}$ opt-mig(b) node migrations in $F(c)$.
Therefore $2\textsc{Opt}(c) \geq |B| \cdot \alpha + \sum_{b \in S(c)} \textsc{opt-mig}(b) \cdot \alpha \geq |Y_c| \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha$ and hence also $\textsc{Opt}(c) \geq 1/2 \cdot (|Y_c| \cdot \alpha - \sum_{b \in S(c)} (|Y_b| - 1) \cdot \alpha)$.

Note: This lemma does not work like lemma 6 in [2] as there is no obvious telescope sum.

Lemma 20. For any input σ, let $\textsc{del}(\sigma)$ be the set of components that were eventually deleted by Crep. Then $\textsc{Opt}(\sigma) \geq \sum_{c \in \textsc{DEL}(\sigma)} |c|/(2k) \cdot \alpha$.

Proof. Fix any component $c \in \textsc{del}(\sigma)$. Consider a tree $T(c)$ which describes how component c was created: the leaves of $T(c)$ are singleton components containing nodes of c, the root is c itself, and each internal node corresponds to a component created at a single time by merging its children. We now sum Opt(b) over all components b from T(c), including the root c and the leaves L(T(c)). The lower bound given by Lemma 19 does not sum telescopically:

7.2 Upper Bound on Crep

Fix any input σ of Crep and define the following notions.

Definition 21. Let $M(\sigma)$ be the sequence of merge actions (real and artificial ones) performed by Crep. For any real merge action $m \in M(\sigma)$, let size(m) denote the size of the smaller component that was merged. For an artificial merge action set size(m)=0.

Definition 22. Let $\textsc{final}(\sigma)$ be the set of all components that exist when Crep finishes sequence σ. Note that $w(\textsc{final}(\sigma))$ is the total weight of all edges after processing σ.

We split $\textsc{Crep}(\sigma)$ into two parts: the cost of serving requests, $\textsc{Crep}^{req}(\sigma)$, and the cost of node migrations, $\textsc{Crep}^{mig}(\sigma)$.

Note: This lemma was adapted from lemma 7 in [2] and changed to only give an upper bound instead of an equality as alpha-connectivity only gives upper and lower bounds and that is all that is needed for the analysis.

Lemma 23. For any input σ, $\textsc{Crep}^{req}(\sigma) \leq |M(\sigma)| \cdot \alpha + w(\textsc{final}(\sigma))$.

Proof. The lemma follows by an induction on all requests of σ. Whenever Crep pays for the communication request, the corresponding edge weight is incremented and both sides increase by 1. At a time when s components are merged, $s-1$ merge actions are executed, and due to Lemma 7 the total edge weight decreases by at most $(s-1) \cdot \alpha$.
Note: This lemma and its proof do not use connectivity or Crep-density at all as they only rely on the general migration behaviour and hence is exactly the same as lemma 8 in [2].

Lemma 24. For any input σ, with $(2+\epsilon)$-augmentation, $\textsc{Crep}^{mig}(\sigma) \leq (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \textsc{size}(m)$.

Proof. If CREP has more than $2k$ nodes in cluster V_i (for $i \in \{1, ..., l\}$), then we call this excess *overflow* of V_i, otherwise, the overflow of V_i is zero. We denote the overflow of cluster V_i measured right after processing sequence σ by $\text{OVR}^\sigma(V_i)$. It suffices to show the following relation for any sequence σ:

$$\text{CREP}^{mig}(\sigma) + \sum_{j=1}^{l} (4/\epsilon) \cdot \alpha \cdot \text{OVR}^\sigma(V_j) \leq (1 + 4/\epsilon) \cdot \alpha \cdot \sum_{m \in M(\sigma)} \text{SIZE}(m). \tag{1}$$

As the second summand of 1 is always non-negative, 1 will imply the lemma. The proof will follow by an induction on all requests in σ. Clearly, 1 holds trivially at the beginning, as there are no overflows, and thus both sides of 1 are zero. Assume that 1 holds fo a sequence σ and we show it for sequence $\sigma \cup \{r\}$, where r is some request. We may focus on a request r that triggers component(s) migration as otherwise 1 holds trivially.

Such a migration is triggered by a real merge action m of two components c_x and c_y. We assume that $|c_x| \leq |c_y|$, and hence $\text{SIZE}(m) = |c_x|$. Note that $|c_x| + |c_y| \leq k$, as otherwise the resulting component would be deleted and no migration would be performed.

Let V_x and V_y denote the cluster that held components c_x and c_y, respectively, and V_z be the destination cluster for c_x and c_y ($z = y$ is possible). For any cluster V, we denote the change in its overflow by $\Delta\text{OVR}(V) = \text{OVR}^{\sigma \cup \{r\}}(V) - \text{OVR}^\sigma(V)$. It suffices to show that the change of the left hand side of 1 is at most the increase of its right hand side, i.e.:

$$\text{CREP}^{mig}(r) + \sum_{V \in \{V_x, V_y, V_z\}} (4/\epsilon) \cdot \alpha \cdot \Delta\text{OVR}(V) \leq (1 + 4/\epsilon) \cdot |c_x| \cdot \alpha. \tag{2}$$

In order to show that 2 holds, consider three cases:

1. V_y had at least $|c_x|$ empty slots. In this case, Crep simply migrates c_x to V_y and pays $|c_x| \cdot \alpha$. Then, $\Delta\text{ovr}(V_x) \leq 0.\Delta\text{ovr}(V_y) \leq |c_x|, V_z = V_y$, and thus 2 follows.
2. V_y had less than $|c_x|$ empty slots and $|c_y| \leq (2/\epsilon) \cdot |c_x|$. Crep migrates both c_x and c_y to component V_z and the incurred cost is $\text{Crep}^{mig}(r) = (|c_x| + |c_y|) \cdot \alpha \leq (1+2/\epsilon) \cdot |c_x| \cdot \alpha < (1+4/\epsilon) \cdot |c_x| \cdot \alpha$. It remains to show that the second summand of 2 is at most 0. Clearly, $\Delta\text{ovr}(V_x) \leq 0$ and $\Delta\text{ovr}(V_y) \leq 0$. Furthermore, the number of nodes in V_z was at most k before the migration by the definition of Crep, and thus is at most $k + |c_x| + |c_y| \leq 2k$ after the migration. This implies that $\Delta\text{ovr}(V_z) = 0 - 0 = 0$.
3. V_y had less than $|c_x|$ empty slots and $|c_y| > (2/\epsilon) \cdot |c_x|$. As in the previous case, Crep migrates c_x and c_y to component V_z, paying $\text{Crep}^{mig}(r) = (|c_x| + |c_y|) \cdot \alpha < 2 \cdot |c_y| \cdot \alpha$. This time, $\text{Crep}^{mig}(r)$ can be much larger than the right hand side of 2, and thus we will resort to showing that the second summand of 2 is at most $-2 \cdot |c_y| \cdot \alpha$.

   As in the previous case, $\Delta\text{ovr}(V_x) \leq 0$ and $\Delta\text{ovr}(V_z) = 0$. Observe that $|c_x| < (\epsilon/2) \cdot |c_y| \leq (\epsilon/2) \cdot k$. As the migration of $|c_x|$ to V_y was not possible, the initial number of nodes in V_y was greater than $(2 + \epsilon) \cdot k - |c_x| \geq (2 + \epsilon/2) \cdot k$, i.e., $\text{ovr}^\sigma(V_y) \geq (\epsilon/2) \cdot |c_y|$. As component c_y was migrated out of V_y, the number of overflow nodes in V_y changes by

$$\Delta\text{ovr}(V_y) = -\min\{\text{ovr}^\sigma(V_y), |c_y|\} \leq -(\epsilon/2) \cdot |c_y|.$$

   Therefore, the second summand of 2 is at most $(4/\epsilon) \cdot \alpha \cdot \Delta\text{ovr}(V_y) \leq -(4/\epsilon) \cdot \alpha \cdot (\epsilon/2) \cdot |c_y| = -2 \cdot |c_y| \cdot \alpha|$ as desired.

□

7.3 Competitive Ratio

Note: The following lemma is the same as lemma 9 in [2].

Lemma 25. For any input σ, it holds that $\sum_{m \in M(\sigma)} \text{size}(m) \leq \sum_{c \in \text{DEL}(\sigma)} |c| \cdot \log k + \sum_{c \in \text{FINAL}(\sigma)} |c| \cdot \log |c|$, where all logarithms are binary.

Proof. We prove the lemma by an induction on all requests of σ. At the very beginning, both sides of the lemma inequality are zero, and hence the induction basis holds trivially. We assume that the lemma inequality is preserved for a sequence σ and we show it for sequence $\sigma \cup \{r\}$, where r is an arbitrary request. We may assume r triggered some merge actions as otherwise the claim follows trivially.
First, assume r triggered a sequence of real merge actions. We show that the lemma inequality is preserved after processing each merge action. Let c_x and c_y be merged components with sizes $p = |c_x|$ and $q = |c_y|$, where $p \leq q$ without loss of generality. Due to such action, the right hand side of the lemma inequality increases by

$$\begin{aligned}
&(p + q) \cdot \log(p + q) - p \cdot \log p - q \cdot \log q \\
&= p \cdot (\log(p + q) - \log p) + q \cdot (\log(p + q) - \log q) \\
&\geq p \cdot \log(p + q)/p \\
&\geq p \cdot \log 2 = p.
\end{aligned}$$

As the left hand side of the equality changes exactly by p, the inductive hypothesis holds.
Now assume that r triggered a sequence of artificial merge actions (i.e., followed by a delete action) and let $c_1, c_2, ..., c_g$ denote components that were merged to create component c that was immediately deleted. Then, the right hand side of the lemma inequality changes by $\sum_{i=1}^{g} |c_i| \cdot \log |c_i| + |c| \cdot \log k \geq -\sum_{i=1}^{g} |c_i| \cdot \log k + |c| \cdot \log k = 0$. As the left hand side of the lemma inequality is unaffected by artificial merge actions, the inductive hypothesis follows also in this case. □

8 Implementation Details

8.1 Algorithm Pseudocode

Algorithm 2 insertAndUpdate(a,b)

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

8.2 Algorithm Explanations

- Algorithm 2 calls the other routines as needed
- Algorithm 3 starts at the smallest subgraph containing the nodes a and b in the decomposition tree and computes a new decomposition of the subgraph. Specifically it uses the decomposition approach from [4] to decompose one subgraph and then also computes the subgraphs with the next higher connectivity and recurses until the connectivity has reached alpha.
- updateMapping checks whether the alphaConnectedComponents were changed. If yes then it either collocates them if the resulting component is small enough or it adds the component to its return value. Then all the returned components are deleted, i.e. the edges connecting its nodes are deleted and the decomposition is recomputed
- this deletion is performed by Algorithm 4

9 Evaluation

For this evaluation we compare our implementation described in Section 8 to a static algorithm available via Metis (METIS_PartGraphRecursive) and an adaptive/dynamic algorithm (ParMETIS_V3_AdaptiveRepart) implemented in the ParMetis framework.

Algorithm 3 updateDecomposition(a,b)

```
q← findSmallestSubgraph(a, b)
while q not empty do
  current← q.popFront()
  if res.connectivity==alpha then
    continue
  end if
  res← decompose(current, current.connectivity+1)//decomposition based on s-t-cuts
  current.connectivity ← value of smallest encountered cut
  if current.connectivity≥alpha then
    continue
  end if
  childrenQueue ← res
  //make sure that only subgraphs with higher connectivity are added as children
  while childrenQueue not empty do
    c←childrenQueue.pop()
    cRes←decompose(c, current.connectivity+1)
    c.connectivity ← 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
```

Algorithm 4 delComponents(del)

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

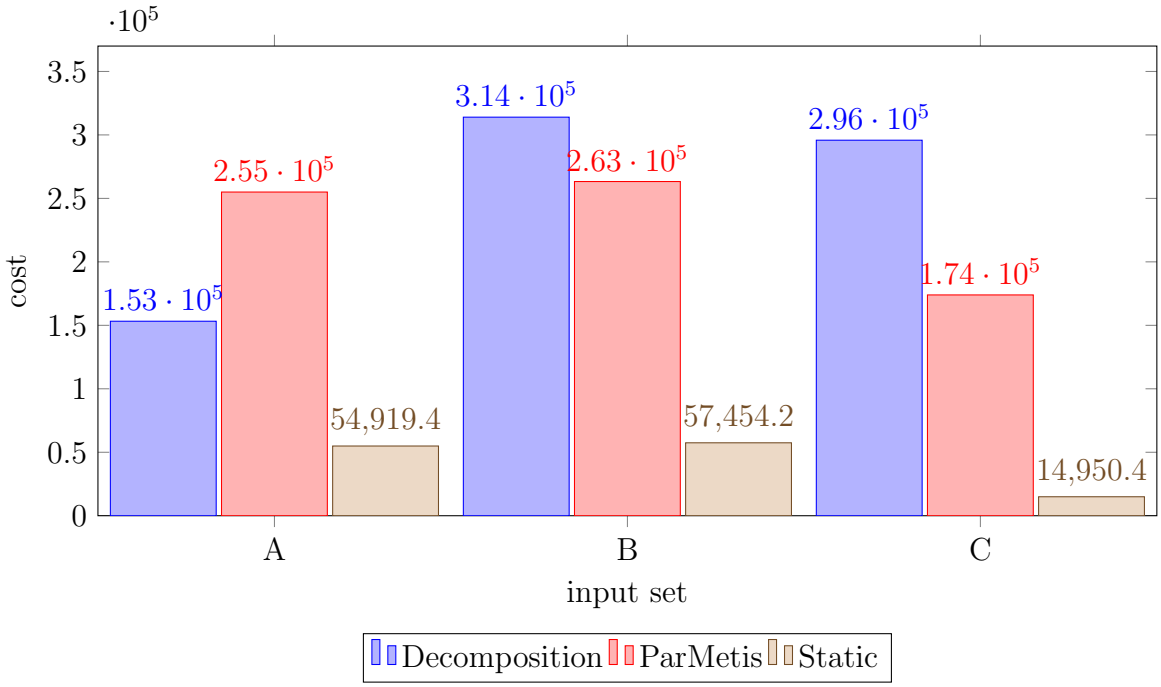

Figure 3: comparison of total cost

Both frameworks are known to produce very good results and to be very fast.
As input data we use several HPC traces, the nature of the data is described in more detail by Avin, Ghobadi, Griner and Schmid in [3].

All data sets contain 1024 different communication nodes and are limited to the first 300 000 requests. The value of α is set to be 6 and the algorithm was tasked to partition the nodes into 32 clusters of size 32 each. The dynamic algorithms were allowed to use augmentation with a factor of 2.1, i.e. for the dynamic algorithms the maximum cluster capacities were $\lfloor 32 \cdot 2.1 \rfloor = 67$.
To our knowledge it is not possible to specify hard limits for the capacities used by the static algorithm implemented in Metis and as a result there are some occurences where the algorithm exceeds the capacities that are allowed by a small amount.

We will first discuss the overall results of our experiments, i.e. we will describe the quality of the results (see Figure 3) as well as the running time needed for each examined algorithm (see Figure 4).
The static algorithm is shown to give the best results in the shortest time, but the low running was also to be expected as it is only called once as opposed to the 300 000 times the other algorithms need to decide whether they want to change their partitioning. The static algorithm also has knowledge of all requests and as a result is able to produce the best results.
For data set A our decomposition algorithm beats the adaptive ParMetis algorithm by a significant amount of about one third of the cost of the latter while ParMetis is significantly faster. For data sets B and C ParMetis is shown to produce slightly better results within drastically less computation time. It is worth mentioning that ParMetis uses the ... graph description format that does not allow for easy adaptation on the fly

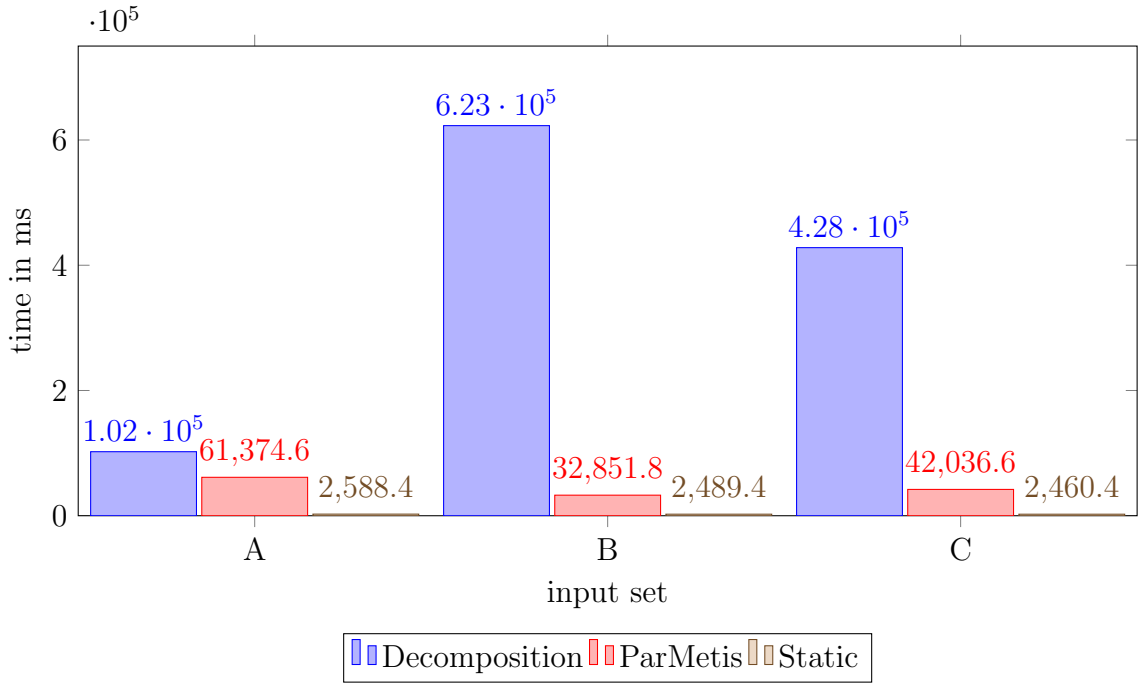

Figure 4: comparison of run time

and needed to be recomputed after every request during our tests. However, we chose not to include this in the running time calculations.

In the next section we discuss the general distribution of the total costs of each algorithm to communication (see Figure 5) and migration costs (Figure 6).
Both ParMetis as well as our decomposition algorithm produce significantly more migration cost than communication cost while the static algorithm predominantly pays for communication. This shows that the dynamic algorithms tend to migrate too much while the static implementation is restricted to only migrate once to a static configuration it finds suitable and as a result has to pay more for communication. This also shows that there is potential to refine the dynamic implementations in such a way that they produce more balanced, and hopefully also less, cost overall.

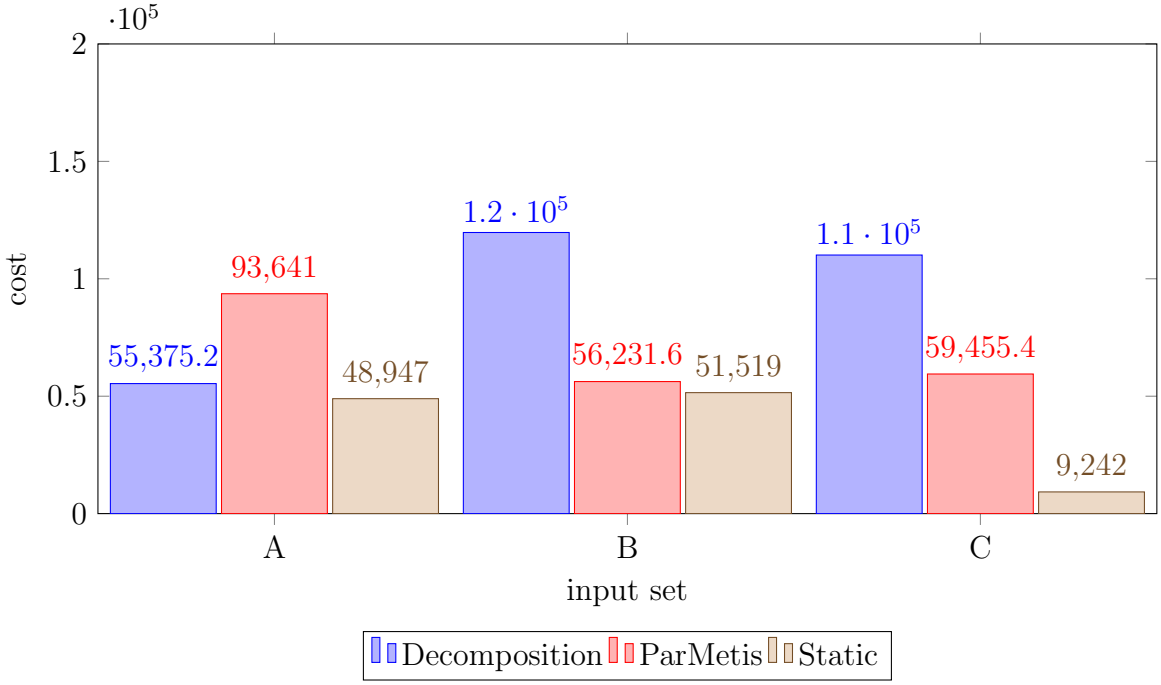

Figure 5: comparison of communication cost

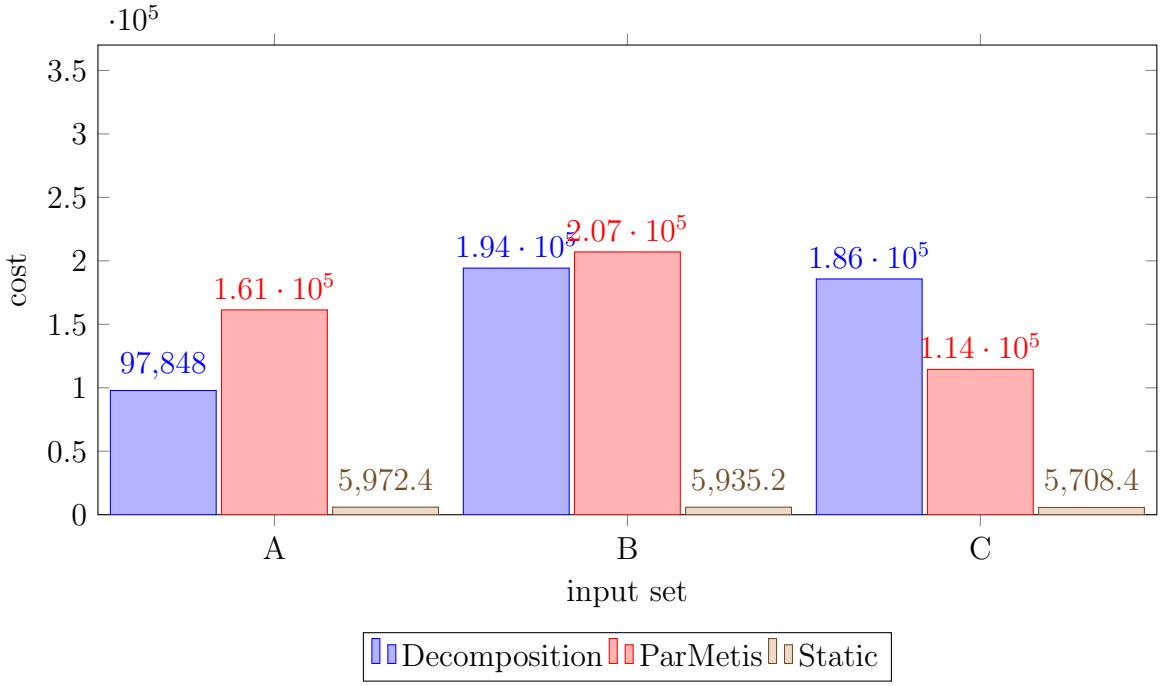

Figure 6: comparison of migration cost

References

[1] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic Balanced Graph Partitioning.

[2] Chen Avin, Marcin Bienkowski, Andreas Loukas, Maciej Pacut, and Stefan Schmid. Dynamic Balanced Graph Partitioning. *no idea*, 2015.

[3] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. Measuring the Complexity of Packet Traces.

[4] 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.

[5] Monika Henzinger, Stefan Neumann, and Stefan Schmid. Efficient Distributed Workload (Re-)Embedding. *no idea*, 2019.