Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Polynomial Time Competitive Repartitioning of Dynamic Graphs

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 Räcke
	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 Dynamic Balanced Graph Partitioning problem was first introduced by Avin, Bienkowski, Loukas, Pacut and Schmid ([2]).

The task is to maintain a partitioning of a dynamic graph consisting of $n = k \cdot l$ nodes that communicate with each other into l parts, each of size k 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 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 $\rho-$*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.

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 $\delta > 1$. This augmented online algorithm is than compared with the optimal offline algorithm Opt which is not allowed to use any augmentation.

3 Related Work

3.1 Dynamic Balanced RePartitioning

Avin et al.[2] initiated the study of the online variant of the Balanced RePartitioning (BRP) problem that is the topic of this thesis. They propose a deterministic algorithm for the Dynamic Balanced Graph Partitioning problem with augmentation $2 + \epsilon$ for any $\epsilon > 1/k$. They also show a lower bound of $k - 1$ for the competitive ratio of any online algorithm for the Dynamic Balanced Graph Partitioning problem on two clusters via a reduction to online paging. Furthermore they show that no δ-augmented deterministic online algorithm can achieve a competitive ratio smaller than k for any augmentation $\delta < l$.

3.2 Restricted Variant of Balanced RePartitioning

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, Cohen, Parham and Schmid ([4]) study one such case: the authors 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 they 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, Neumann and Schmid ([7]) 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.

The authors of [7] 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 $\epsilon \in (0, 1/2)$.

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.

3.3 Clustering

3.4 Online Paging

3.5 Static Balanced Graph Partitioning

The Static Balanced Graph Partitioning problem is the static offline variant of the topic of this thesis. 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. 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.

For the case where $l \geq 3$, Andreev and Räcke have shown that there is no polynomial time approximation algorithm which guarantees a finite approximation factor unless P=NP ([1]).

4 Algorithmic Ideas

In this section we describe two different solution approaches to the Dynamic Balanced Graph Partitioning problem. We call these approaches Core-Del and Adj-Del respectively. We first describe this general approach that is common to both algorithms and then address the specific differences and analysis ideas.

Both methods share a similar concept at their core: 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. As more requests from σ are revealed to the algorithms they merge the corresponding components once they are suitably connected and relocate the nodes of the new component in such a way that all the nodes of a component are always located on the same server.

More formally, 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 o 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. If the resulting component has size at least $2/\epsilon$ the algorithm reserves additional space $\min\{\epsilon \cdot |C|, k\}$, otherwise the reservation is zero. This reservation guarantees that nodes are not migrated too often for the analysis to work.

The collocation of such component sets of at most k individual communication nodes is always possible without moving a node not in C 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 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 differentiating factor 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. Please note that the subroutine *delete(Y)* of a component set Y is different for each of the algorithms. We also describe the particular challenges each approach entails when it comes to the competitive analysis.

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 comp(v) ≠ comp(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) and update reservations
    end if
    if ∃ component set Y with connectivity at least α and nodes(Y) > k then
        delete(Y)
    end if
end for
```

These approaches are fairly similar to the algorithms defined in previous work ([3], [2]). The main differentiating factor is that we merge once a component set reaches connectivity α while the approaches by the other authors do so once the component set reaches a certain density threshold. More specifically they 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 [3] and Lemma 6 in this thesis.

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

4.1 Core-Del

Algorithm 2 delete(Y) of Core-Del

```
for all e = {u, v} ∈ E do
    if u ∈ Y and v ∈ Y then
        w(e) ← 0
    end if
end for
```

We address Core-Del first. In this version edges are reset which are *contained* in the deleted component, i.e. 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 resembles the one suggested by Avin et al. ([2]) and is also written in pseudocode in Algorithm 2.

The idea of the analysis is then to relate the cost of Core-Del with the cost of Opt by considering the respective costs due to requests from a deleted component C in the solution of Core-Del as these are of high connectivity and are impossible for Opt to collocate on one server as each deleted component contains more than k nodes. Then one could sum these costs over all such deleted components in order to

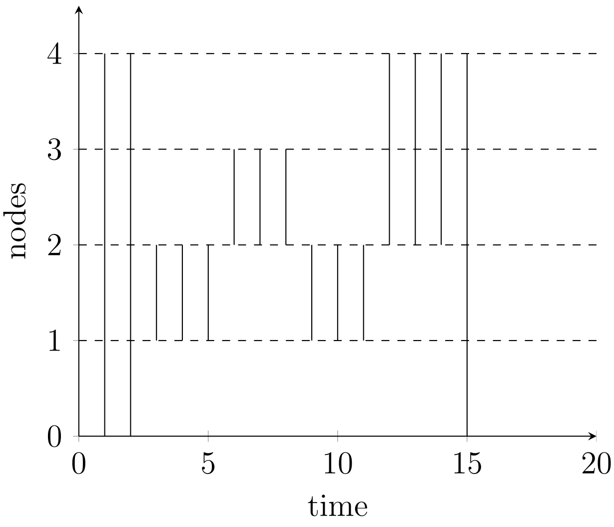

Figure 1: illustration of the analysis problem with the approach CORE-DEL

bound the costs. For this approach to work however one would have to find a way to cleanly separate requests belonging to one deleted component from those belonging to another in order to establish a lower bound on the cost of OPT.

In this case the edges which are adjacent to a deleted component C remain even after the deletion of C and may then contribute the the creation of a new component D. It is now very challenging to attribute any significant cost to OPT for these requests.

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 $t = 5$. Now consider the case where $\alpha = 3$ and $k = 3$.

In this sequence the first two requests between nodes 0 and 4 happen without leading to a merge. The following 12 requests lead to a merge of a new component C consisting of the nodes 1,2,3 and 4 which gets deleted by CORE-DEL at time $t = 14$. Note now that the edges corresponding to the requests between nodes 0 and 4 are still present even after this deletion. Finally the request at time $t = 15$ leads to a merge of nodes 0 and 4.

One can see that such cases may also happen at a much larger scale, where almost all requests happen much earlier than the time at which an actual merge of the nodes involved in the requests happens.

4.2 ADJ-DEL

Algorithm 3 delete(Y) of ADJ-DEL

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

The second algorithm, ADJ-DEL, resets all the edges contained in the deleted component C but also resets the weights of edges *adjacent* to C, i.e. all edges

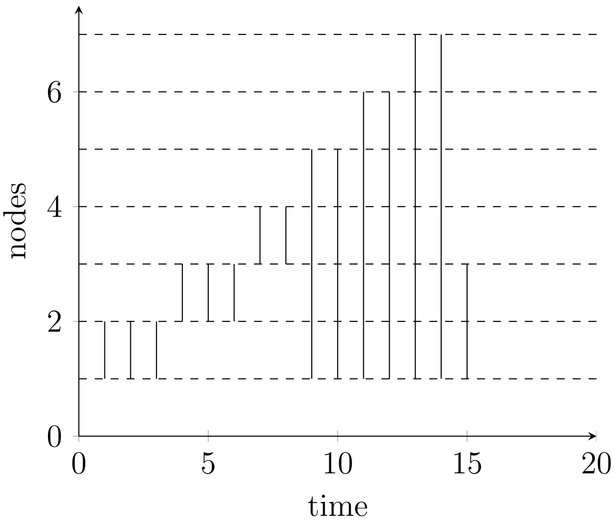

Figure 2: illustration for the Adj-Del approach

$e = \{u, v\}$ are reset to zero if u or v were contained in component C at the time of its deletion. This second version is very similar to the algorithm proposed by Avin et al. in [3]. The deletion method is also described in pseudocode in Algorithm 3.

The idea for the analysis is once again to relate the cost of both Opt and Adj-Del to the deleted components in the solution of Adj-Del.

The fact that Adj-Del 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 section on Core-Del. However 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.

Figure 2 illustrates an example of this challenge mentioned above for the Adj-Del approach.. In the illustration it is assumed that $\alpha = 3$ and $k = 3$. In this case the input sequence leads Adj-Del 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 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. This means that the amount of adjacent edges that are reset may greatly exceed the amount of edges inside a deleted component. We show in our analysis in Section 5 that this approach is valid and allows for a competitive ratio of $O(k \log k)$.

5 Analysis

For the sake of this analysis we examine the Adj-Del algorithm from Section 4 and name the algorithm Crep from now on.

We analyse the competitive ratio of Crep with augmentation $(2 + \epsilon)$ and show in Theorem 14 that Crep is $O(k \log k)$-competitive.

5.1 Algorithm Definitions

We begin our analysis by introducing two general definitions that we will use throughout the analysis.

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.

5.2 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 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 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 then 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. ☐

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 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 $(|S|-1)\cdot\alpha$ during the execution of Crep.

Proof. We iteratively partition S into subsets via minimum cuts with regard to edge weight, i.e. we consider a minimum edge cut 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 the lemma follows. □

5.3 Upper Bound On Crep

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 component $C \in \textsc{del}(\sigma)$, i.e. the subgraph induced by the nodes of C has connectivity at least α and C consists of more than k nodes:

Let $\textsc{epoch}(C)$ denote the (node, time) pairs of nodes in C starting at the time after the time $\tau(node)$ 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: $\textsc{core}(C)$ 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. $\textsc{halo}(C)$ contains 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 $\textsc{halo}(C) = \textsc{req}(C)\setminus\textsc{core}(C)$.

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. 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. □

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 $\textsc{halo}(C)$ 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

$$\textsc{Crep}^{req}(\sigma) \leq \sum_{C\in\mathrm{DEL}(\sigma)}(|C|\cdot\alpha + |\textsc{halo}(C)|) + \sum_{C\in\mathrm{FIN\text{-}COMPS}(\sigma)}|C|\cdot\alpha + \textsc{fin-weights}(\sigma).$$

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 requests from HALO(C) were deleted. 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 let CREP to any merge and are hence contained in FIN-COMPS(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

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

Proof. First note that CREP only performs migrations when it merges components. We fix a component $C \in$ DEL(σ) $\cup$ 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 components of size at least $2/\epsilon$ 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 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 in the following lemma.

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

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

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

$$\begin{aligned}
\text{CREP}(\sigma) &\leq \text{CREP}^{req} + \text{CREP}^{mig} \\
&\leq \sum_{C \in \text{DEL}(\sigma)} (|C| \cdot \alpha + |\text{HALO}(C)|) + \sum_{C \in \text{FIN-COMPS}(\sigma)} |C| \cdot \alpha + \text{FIN-WEIGHTS}(\sigma) \\
&+ \sum_{C \in \text{DEL}(\sigma) \cup \text{FIN-COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha \\
&\leq 2 \cdot \sum_{C \in \text{DEL}(\sigma) \cup \text{FIN-COMPS}(\sigma)} |C| \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + \sum_{C \in \text{DEL}(\sigma)} |\text{HALO}(C)| \\
&+ \text{FIN-WEIGHTS}(\sigma).
\end{aligned}$$

□

5.4 Lower Bound on OPT

In this section we bound the cost on OPT by assigning cost to OPT based on the size of the components C CREP deletes and the associated adjacent edges HALO(C) CREP resets to zero during the deletion of C. In order to achieve this we first 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 $\mathcal{H} = \bigcup_{C \in \mathrm{DEL}(\sigma)} \textsc{halo}(C)$ 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.

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 requests in question by N, i.e. those from $N = \mathcal{H} \backslash (P \cup OP)$ where OP denotes the requests from $\mathcal{H}$ only OPT pays for.

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

Proof. We fix an arbitrary offline interval. 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. This bounds the number of requests assigned to the offline interval by $k \cdot \alpha$. □

In the following lemma we combine this result with a bound on the cost of OPT due to requests we have not accounted for, namely those from $\textsc{core}(C)$ and those that are not contained in an offline interval because they get deleted by CREP due to a component deletion that takes place before their offline interval ends.

Lemma 13. The cost of the solution of OPT given input sequence σ is bounded by

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

Proof. Let P denote the set of edges from $\bigcup_{C \in \mathrm{DEL}(\sigma)} \textsc{halo}(C)$ that both CREP and OPT pay for and let I denote the set of requests we have assigned to offline intervals.

For the following part of the proof we fix an arbitrary component $C \in \textsc{del}(\sigma)$. Let R denote the set of requests from $\textsc{halo}(C)$ that were not assigned to any offline interval. This means that the nodes involved in requests from R were not moved during the processing of requests from R until the time of deletion of C.

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

We first examine the case in which OPT does not move any nodes from C during EPOCH(C). In this case OPT must partition a graph containing the nodes from C which are connected via edges representing the requests from CORE(C)into migrations. Because of this fact OPT must have placed those nodes on at least $\frac{|C|+R/\alpha}{k}$ parts. 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 migrations. Because of this fact OPT must have placed those nodes on at least $\frac{|C|+R/\alpha}{k} \cdot \alpha = |C|/k \cdot \alpha + R/k$.

For the more general case in which OPT may perform node migrations during EPOCH(C) we adapt the graph construction from above as follows: we add a vertex representing each (node, time) pair from 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 EPOCH(C). Additionally we add an edge of weight one for each request $r = \{u, v\}$ from 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 $\frac{|C|+R/\alpha}{k}$ 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 $\frac{|C|+R/\alpha}{k} \cdot \alpha = |C|/k \cdot \alpha + R/k$.

Finally we need to account for the fact that the migrations of nodes from C that OPT performs also end offline intervals and might hence be accounted for twice in our analysis up to this point:

$$\begin{aligned} 2 \cdot \textsc{Opt}(\sigma) &\geq \sum_{(C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + R(C)/k + I/(k \cdot \alpha) \cdot \alpha + |P| + |OP| \\ &\geq \sum_{(C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + \textsc{halo}(C)/k \end{aligned}$$

where the last equality follows from the fact that $\bigcup_{C \in \mathrm{DEL}(\sigma)} R(C) \cup I \cup P \cup OP = \bigcup_{C \in \mathrm{DEL}(\sigma)} \textsc{halo}(C)$ as the different $R(C)$ as well as I, P and OP are disjoint. Hence the cost of OPT is at least $1/2 \cdot \sum_{(C \in \mathrm{DEL}(\sigma)} |C|/k \cdot \alpha + \textsc{halo}(C)/k$ as OPT pays for requests from P by the definition of P. □

5.5 Competitive Ratio

In this section we combine the results of Lemma 11 and Lemma 13 to obtain the following theorem.

Theorem 14. With augmentation $(2 + \epsilon)$ the competitive ratio of CREP is in $O(k \log k)$.

Proof. We arbitrarily fix an input sequence σ and use our previous results to bound the competitive ratio of CREP. We define COMPS(σ) := DEL(σ)∪FIN-COMPS(σ) 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.

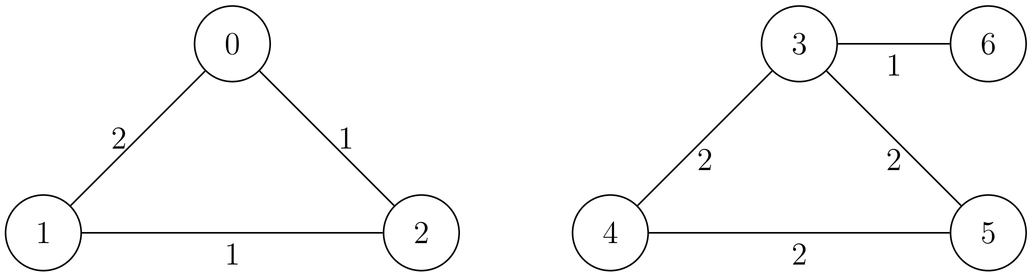

Figure 3: example graph

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

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

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 $\textsc{fin-weights}(\sigma)$ by $k \cdot l \cdot \alpha$. Hence we conclude that $\beta \leq l \cdot k \cdot ((2/\epsilon + 1) + \log k) \cdot \alpha + k \cdot l \cdot \alpha \in O(k \log k)$. □

6 Implementation Details

In this section we 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 (sub-)graph G' of G, we decompose G' into sub-graphs whose connectivity is strictly larger than that of G' and add children to v for each such sub-graph. We do not decompose sub-graphs of connectivity at least α as we only need to identify whether a new sub-graph of connectivity at least α was created by the insertion of the most recent request.

Section 6 illustrates this decomposition for the graph shown in Section 6.

6.1 Algorithm Pseudocode

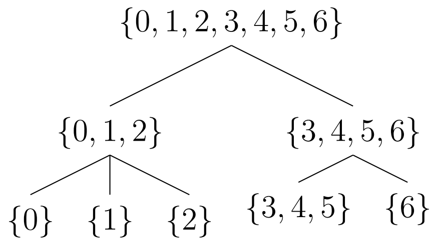

Figure 4: decomposition tree example for the graph from Section 6 for $\alpha = 4$

Algorithm 4 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 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 6 delComponents(del)

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

6.2 Algorithm Explanations

- Algorithm 4 calls the other routines as needed
- Algorithm 5 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 [6] 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 6

7 Evaluation

In this section we evaluate the quality and performance of our algorithm implementation.

For this evaluation we compare our implementation described in Section 6 to a static algorithm available via Metis (METIS_PartGraphRecursive) ([8, 9]) and an adaptive/dynamic algorithm (ParMETIS_V3_AdaptiveRepart)([10, 12, 11]) implemented in the ParMetis framework. 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 [5].

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 occurrences 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 5) as well as the running time needed for each examined algorithm (see Figure 6).
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

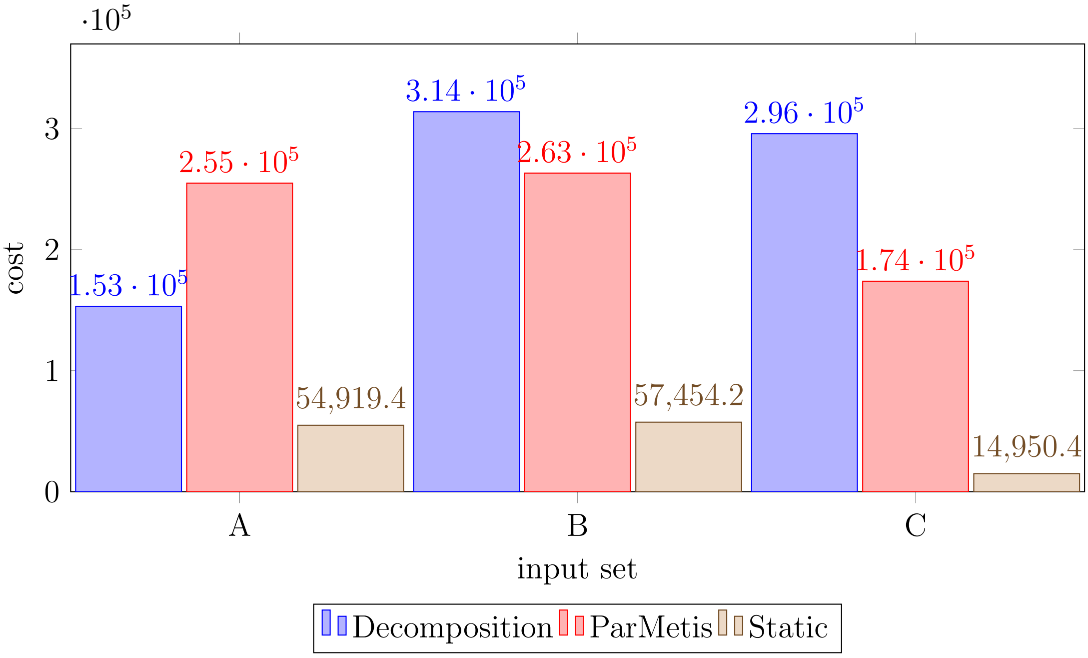

Figure 5: comparison of total cost

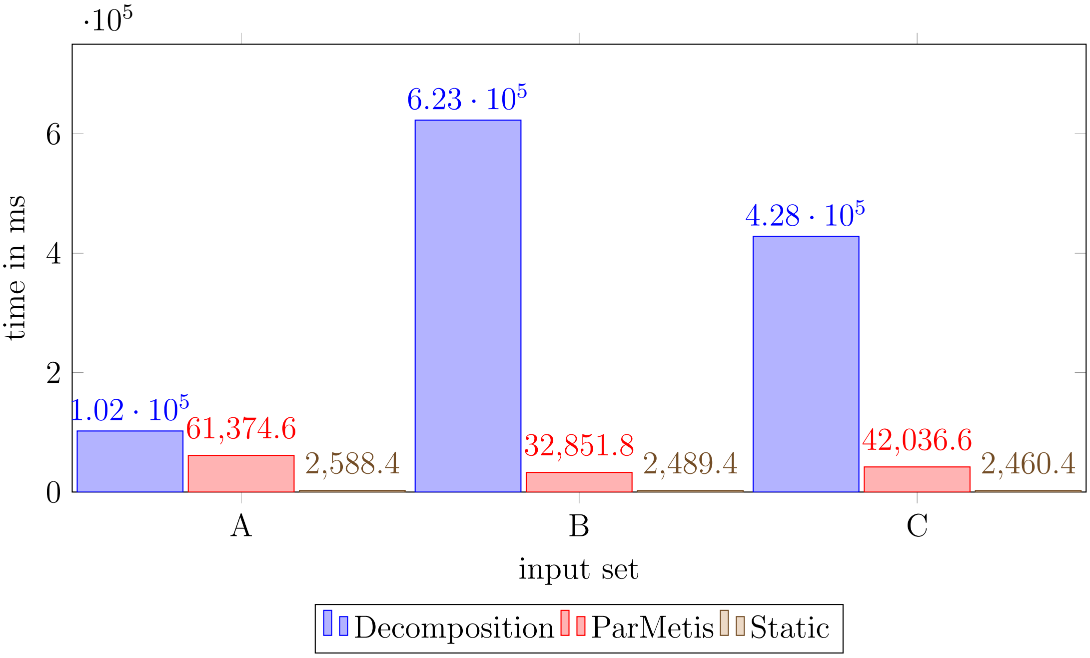

Figure 6: comparison of run time

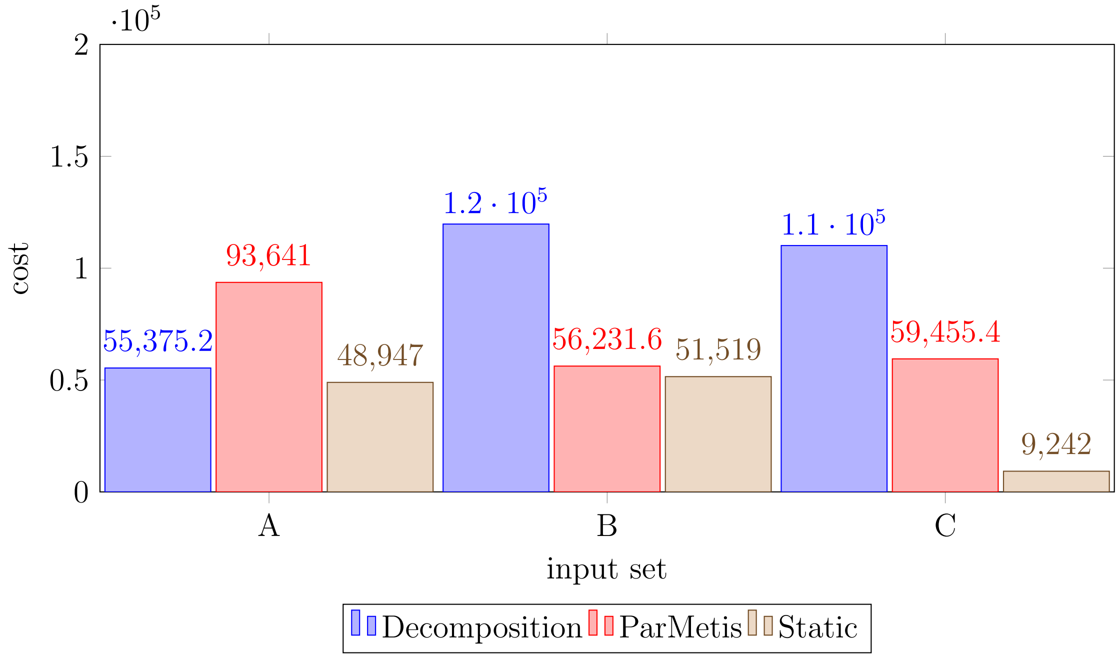

Figure 7: comparison of communication cost

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 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 7) and migration costs (Figure 8).
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.

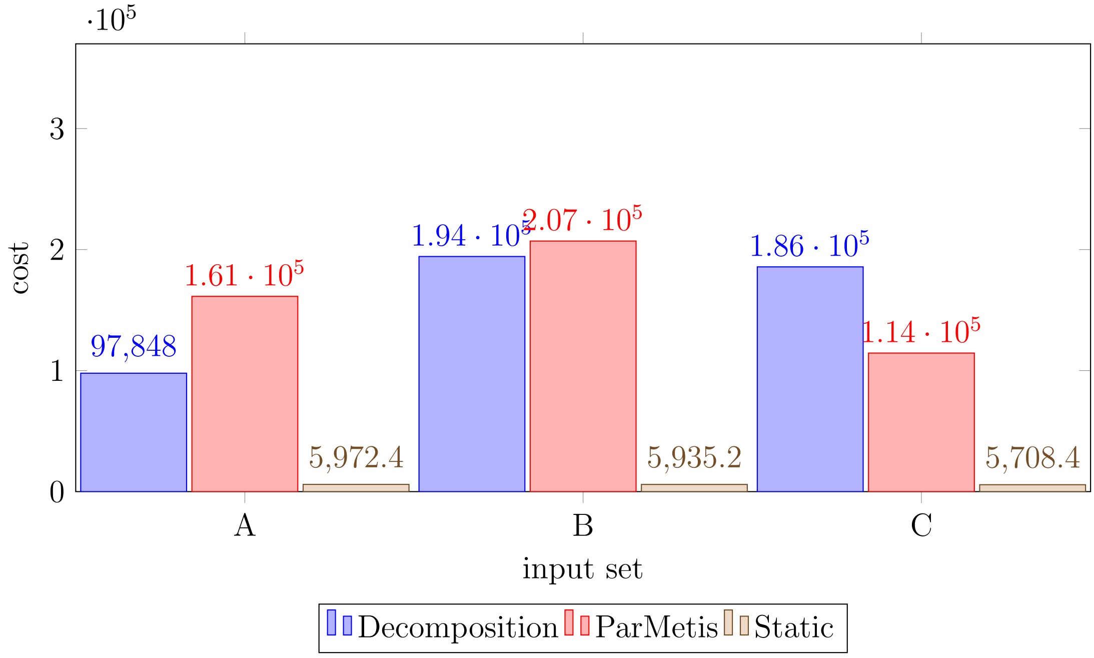

Figure 8: comparison of migration cost

References

[1] Konstantin Andreev and Harald Racke. Balanced Graph Partitioning. *Theory of Computing Systems*, 39(6):929–939, oct 2006.

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

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

[4] 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.

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

[6] 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.

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

[8] 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.

[9] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, jan 1998.

[10] George Karypis and Vipin Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, jan 1999.

[11] 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.

[12] 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.