I. TODOS

1) Make sure that we loose nothing while preparing the short paper (backup: in branch, in directory(svn-style branch)?)
2) Apply the ACM style and see how much space do we have
3) Decide on the subset of results to publish (I would minimize those: if a figure on LB fits, it would be vital to include it)
4) (and hence, draw a figure)
5) Decide on the order of results
6) Include intro, related work etc. from PhD (Machine Migration, DYnamic Mapping, Related Work, Problem Definition, References)
7) Couple iterations on existing results (these are mostly fine already, though)
8) Make an ACM account for the submission and sum up all the steps needed for the submission

II. INTRODUCTION

Main technical contribution of this paper is $\Omega(k\ell)$ lower bound for any deterministic algorithm for Online Balanced Partition problem. Previous known lower bound were $\Omega(k)$ (cite Marcin-Chen). The gap between upper and lower bound is still substantial, as the best known algorithm is $O(k^2\ell^2)$ (cite Marcin-Chen).

We contribute to the search for the optimal algorithm by presenting $O(k\ell)$-competitive algorithm for the special case where the perfect partition of the input graph is assumed and every algorithm must colocate every communicating pair. Note that our $\Omega(k\ell)$ lower bound uses a perfect partition input graph and colocates communicating pairs, and hence we provide a tight result for such scenario.

Maciek: Future directions of research: How well does the Perfect Partition Learner performs when applied to a single epoch in general online clustering problem (without the perfect partition?)? Epoch = C * l * (k-1) external requests.

III. THE MODEL

Formal introduction of the general model.

We say that an online algorithm DET is component-respecting if it never reaches a configuration with a component split between clusters. If DET splits a component, we issue intra-component requests until DET joins it. If it never joins a component, it is not competitive.

Maciek: + description of perfect partition + ground components model

IV. WARM-UP: LOWER-BOUND $\Omega(\ell)$ FOR $k = 3$

Maciek: The intention of this section is to present a simpler version of a lower bound (that should be skipped if we are short on space) AND to present a lower bound for the case with augmentation and discuss the gap

Theorem 1. *The competitive ratio of any deterministic algorithm for Online Balanced Partition with $k = 3$ is $\Omega(\ell)$.*

Proof.
Maciek: ToDo □

Observation 1. *The lower bound of $\Omega(\ell)$ can be generalized for arbitrary $k = c \cdot 3$. The adversary starts with a serie of "glueing requests", so that each cluster contains 3 components of size $k/3$. We repeat the argument of Theorem 1 using these glued components instead of singletons. OPT pays $O(k)$ for migration of one such component, and ALG pays $\Omega(k \cdot \ell)$.*

Observation 2. *The lower bound of $\Omega(\ell)$ holds even with $1/3 - 1/k$-augmentation, i.e., in scenarios where the capacity of each cluster of the algorithm is $(1 + 1/3) \cdot k - 1$, while the capacity of OPT's clusters is k. To this end we use the construction from Observation 1. We observe that each cluster of the algorithm can fit exactly 3 components of size $k/3$.*

Maciek: Note that this does not contradict the existence of $O((1 + 1/\epsilon) \cdot k \cdot \log(k))$ algorithm of Avin, Bienkowski et al, as the algorithm requires $2 + \epsilon$-augmentation.

Maciek: Note that we have a gap here. What is the maximum augmentation (between 1+1/3 and 2) that still has the $\Omega(\ell)$ lower bound?

V. LOWER BOUND $\Omega(k \cdot \ell)$

Maciek: I assume $\alpha = 1$ to simplify calculations, but everything should hold for general α

In this section we consider the Online Balanced Partition problem. We show a lower bound of $\Omega(k \cdot \ell)$ that uses an input that has a perfect partition. Therefore, this result is a lower bound for the general Online Balanced Partition problem, as well as a lower bound for Online Balanced Partition with perfect partition. The latter complements an upper bound $O(k \cdot \ell)$ for Online Balanced Partition with perfect partition that we present in Section **??**.

Theorem 2. *The competitive ratio of any deterministic algorithm for Online Balanced Partition is $\Omega(k \cdot \ell)$.*

Proof.
Mahmoud: A simpler proof but still not fully rigorous. E.g., what if ALG collocates on S_1? Easy technicalities most likely.
We construct an instance of the problem with ℓ clusters $\{S_1, S_2, \ldots, S_\ell\}, |S_i| = k$. Let $I(C)$ denote the cluster where nodes in a component C are located initially. Consider any online algorithm ALG. We construct the input sequence of requests for ALG as follows. First, we issue $k - 2$ (internal) requests so that ALG form a component of $k - 1$ nodes on

clusters S_1. Note that this does not cost ALG. Let x_0 be the only single nodes left on S_1 and $y_0 \in S_2$ be any single node on S_2. Next, we issue the request between x_0 and y_0. Since this is an external request, ALG joins them into one component and collocates them in some cluster other than S_1. For this, ALG moves to a new configuration that replaces x_0 and y_0 with two other single nodes x_1 and y_1 respectively. Next, we issue a request between x_1 and the largest component C s.t. $I(C) = I(x_1)$. ALG must collocate x_1 and C on some cluster other than S_1 and consequently replaces x_1. We repeat issuing requests between the single node x_i on S_1 and the largest component C' s.t. $I(C') = I(x_i)$, until there are only two single nodes left that originate from the same cluster. I.e., two single nodes $x^*, y^*, I(x^*) = I(y^*)$ that constitute the last pair of such nodes. At this point there are at most $\ell + 1$ single nodes left since there must be at least two nodes with the same initial cluster in any subset of $\ell + 1$ nodes. Given this sequence of requests, the optimal strategy is to migrate $\{x_0, y_0\}$ to the cluster $I(x^*)$ by swapping x_0 and y_0 with x^* and y^* respectively. Hence, OPT pays 4 for node migrations and ALG pays at least one for each node in the sequence $X := x_0, x_1, \ldots$. We exclude at most $(k-1) + (\ell+1)$ nodes out of $k.\ell$ nodes, therefore $|X| \geq k.\ell - k - \ell \in \Omega(k.\ell)$. □

Proof.
Maciek: Warning! This proof requires still major rewrite before submission!
Maciek: The argument of connecting the length of sequence and the number of merged singletons should be stated as an auxiliary lemma.
Maciek: Maybe we do not need to construct I_2 and then I'_2: we can construct I'_2 on-the-fly and bound its length. However, the subtracting argument (as it is written now) is quite simple.
Maciek: ToDo: (one) figure containing: input I_1, cluster D, growing components v_i^C, and the "searched area (by ALG)" (in searching for d_1, d_2)

Maciek: ToDo: check all off-by-one errors, especially this one: check if the existence of b_1 in one of the clusters was considered in all calculations

Fix a component-respecting deterministic algorithm DET. Fix a cluster A containing nodes $a_1, \ldots, a_k$ and a cluster B containing (among others) a node b_1. The adversary begins with issuing $k-2$ internal requests (a_i, a_{i+1}), so that nodes $a_1, \ldots a_{k-1}$ form a single component. These internal requests are free for OPT, and it does not perform additional actions at this point. The next request issued by the adversary is an external request (a_k, b_1). Note that DET must relocate the component $\langle a_k, b_1 \rangle$ to some other cluster, as otherwise it is not component-respecting. OPT also relocates the component $\{a_k, b_1\}$ to some carefuly chosen cluster D, and it evicts nodes $d^{(1)}$ and $d^{(2)}$ (to be fixed later) from D to the previous locations of a_k and b_1 (i.e., to the clusters A and B). For these actions, OPT pays 2, and DET pays at least 2. We call the requests issued so far the requests I_1.

Now, we construct a sequence of requests I_2, but we do not feed them to DET yet. Instead, we simulate DET's actions on $I_1 I_2$ to decide at which point to terminate the input sequence earlier, i.e., we choose a prefix I'_2 of I_2 and we feed DET the input $I_1 I'_2$.

The input I_2 consists of requests $(x^{(1)}, c(x^{(1)})), (x^{(2)}, c(x^{(2)})), \ldots$. The node $x^{(i)}$ is a node in the current configuration of DET that is co-located in one cluster with the component $a_1, \ldots, a_{k-1}$. After a request $(x^{(i)}, c(x^{(i)}))$, the node $x^{(i)}$ is a part of a non-singleton component and must be relocated to some other cluster (as its current cluster hosts a component $a_1, \ldots, a_{k-1}$). Then, DET places the node $x^{(i+1)}$ in place of $x^{(i)}$, and the next request concerns that node. The sequence continues as long as after serving the request, a perfect partition of the input graph exists.

For each $x^{(i)}$, we set $c(x^{(i)})$ to some (to-be-determined) node that in the initial configuration was co-located with $x^{(i)}$. While producing an input sequence, we would like to additionally order the initial nodes V^C of each cluster (i.e., order the nodes that were present in each cluster in its initial configuration). The ordering needs to be produced on-the-fly, as this ordering is used to produce subsequent requests.

Now, we formally define $c(x^{(i)})$, and the ordering of nodes in the initial configuration. Consider a set of nodes from an initial configuration of a cluster C (where $C \neq A$). By v_1^C we label the first node of a cluster C that appears in the sequence $\langle x^{(i)} \rangle$. At this point, we fix an arbitrary other node of C and we label it v_0^C, and we set $c(v_i^C) = v_0^C$ for all i. For the following $k-2$ nodes of the cluster C, we label them $v_2^C, v_3^C, \ldots$ in the order of appearance in the sequence $\langle x^{(i)} \rangle$.

If some nodes of C never appear in the sequence $\langle x^{(i)} \rangle$ (e.g. DET might terminate prematurely or might reside in a configuration with components split) , we number them arbitrarily, with the restriction that the nodes that do appear in the sequence have smaller number than the ones that do not **Maciek: This should not happen at all for component-respecting DET, though. Would be better to provide an additional lemma that shows that every node of each cluster appears in the sequence (outside of v_0's, $a_1, \ldots, a_{k-1}, a_k, b_1$)**

Now, we claim that given the input $I_1 I_2$, every component-respecting DET must produce a sequence $\langle x^{(i)} \rangle$ of length $L = k \cdot \ell - (k-1) - 2 - (\ell - 1) - 1$.
Maciek: ToDo: a sentence that explains each summand of L
Maciek: Name of L should indicate a connection to I_2, e.g., maybe just $|I_2|$, and then $L - \ell - 1$ from last paragraph should be related to I'_2Consider a request $(x^{(i)}, c(x^{(i)}))$. If in this configuration at least one singleton component is present (other than the components of $(x^{(i)}, c(x^{(i)}))$), then serving this request is possible (i.e., the perfect partition exists), and hence a singleton node appears as $x^{(i+1)}$ and the sequence continues. Otherwise, no perfect partition exists, as we have a component $\{a_1, \ldots, a_{k-1}\}$ of size $k-1$ and it can only be complemented by a singleton component.

Maciek: A candidate for auxiliary lemma: intuitions before the claim, then the formal formulation, and a rewrite of the argument from this paragraph Hence, the length of a sequence $\langle x^{(i)} \rangle$ is closely related to the number of singleton components. Before the arrival of the first request from I_2, the number of singleton components is $L + (\ell - 1) + 1$, and after serving the last request, the number of singleton components is 1. After each request, we join this singleton component with another component. This join decreases the number of singleton components by either 1 or 2: if we join two singletons $\{v_0^C\}, \{v_1^C\}$ (there are $\ell - 1$ such joins: for all clusters but A), then the number of singletons decrease by 2, and it decreases by 1 otherwise. We conclude that the number of requests $\langle x^{(i)} \rangle$ to decrease the number of singletons to 1 is L.

Now, we define the input I_2' as a prefix of I_2. Recall that OPT serves the first external request (a_k, b_1) by placing the component $\langle a_k, b_1 \rangle$ in some cluster D, and it evicts some nodes $d^{(1)}$ and $d^{(2)}$ from D to the previous locations of a_k and b_1. It remains to determine the cluster D and nodes $d^{(1)}$ and $d^{(2)}$, and our aim is that OPT would not pay any cost for I_2'. As nodes $d^{(1)}$ and $d^{(2)}$ are not in their initial cluster, a request between a node from (intitial configuration of) D and either $d^{(1)}$ or $d^{(2)}$ would not be free for OPT.

At this point we already know the sequence $\langle x^{(i)} \rangle$ of DET. For the requests of I_2' to be free for OPT, it suffices to choose v_0^D that is neither $d^{(1)}$ nor $d^{(2)}$ (which is always possible for $k > 3$), and to guarantee that $d^{(1)}$, $d^{(2)}$ does not appear in the sequence $\langle x^{(i)} \rangle$, i.e., to finish the input sequence before that. Furthermore, we would like to guarantee the length of $\langle x^{(i)} \rangle$ to remain $\Omega(k \cdot \ell)$.

To this end, we inspect the sequences v_i^C for each cluster, and select such the cluster D whose node v_{k-2}^D, i.e., the penultimate node appears last in the sequence $\langle x^{(i)} \rangle$, and we end the sequence I_2' just before its appearance. Then, we set $d^{(1)} = v_{k-2}^D$ and $d^{(2)} = v_{k-1}^D$. Such choice guarantees that requests from I_2' are free for OPT. Furthermore, the number of requests of I_2' is at least $|I_2| - \ell - 1$: as D is the last cluster to remain with two singleton clusters, all other $\ell - 1$ clusters have at most 1 singleton cluster.

We conclude that for the input $I_1 I_2'$ OPT pays 2, and DET pays at least 2 for each of $L - \ell - 1 = \Omega(\ell \cdot k)$ requests of I_2'. □

VI. An $O(kl)$-Competitive Algorithm For the Scenario With the Perfect Partition

A. *On a Cost of a Single Repartition*

1) A Costly Repartition Example: A repartition cost is bounded by $O(kl)$ (cite marcin-chen). Now we show that the rebalance may be expensive (cubic) in terms of k, even for l linear in k.

This justifies taking a different approach to repartitioning then bounding the repartition cost in $O(k^2 \cdot \ell^2)$ algorithm (described in Section "A simple upper bound" in the DISC paper). As the cost of a specific repartition is $\Omega(k^2)$, then the cost of DET during a single phase is: for each of $O(k \cdot l)$ edges revealed, we already have the cost $O(k^2)$ (precisely, $O(min\{k^2, k\ell\})$, and the competitive ratio would be $O(k^3 \cdot \ell)$.

Maciek: This would be an improvement in terms of ℓ still, but no hopes for $O(k\ell)$ algorithm just by improving the rebalancing analysis

Maciek: Possibly simplify $\Omega(k^3)$? Or just present $\Omega(k^2)$?

2) For Constant k, Repartition Cost is Constant?:
Maciek: Interesting research problem: is a single repartition cost bounded by $O(poly(k))$? This would improve currently best-known $O(k^2\ell^2)$ to $O(k \cdot poly(k) \cdot \ell)$, i.e., would tighten the lower bound in terms of ℓ.
Maciek: We have such examples for $k = 3, 4, 5$. It gives us an $O(\ell)$-competitive algorithm for $k = 3$

Lemma 1. *The rebalancing cost for the greedy algorithm for $k = 3$ is $O(1)$.*
Maciek: Greedy = find the cheapest (closest to the current) feasible partition

Proof. We distinguish among three configuration types of algorithm's clusters: C_1, C_2, C_3. In C_1 we have 3 singleton components (this is also the initial configuration of any cluster). In C_2 we have one component of size 2 and one component of size 1. Finally, in C_3 we have one component of size 3.

Consider a request (u, v) and let U, V be their clusters. If $U = V$, then the request does not require a reconfiguration. Now, we consider cases upon the type of clusters U and V. Note that this is impossible that either U or V is C_3 as otherwise the resulting component would have the size 4, and this contradicts the existence of the perfect partition. If either U or V is C_1, then either cluster can fit the merged component, and the rebalance is local within U and V, for the cost of at most 3 migrations.

Now, we focus on the case where both U and V are C_2. Note that (u, v) cannot both belong to a component of size 2, as the merged component would have size 4. If either u or v belongs to a component of size 2 then it suffices to exchange components of size 1 between U and V. Finally, if u and v belong to components of size 1, then we must place them in a cluster different from U and V. Note that in such case, a C_1-type cluster exists, as otherwise the input graph would not have a perfect partition. The cost of rebalancing is then at most 12
Maciek: can show smaller constant than 12, but it does not matter that much. □

Theorem 3. *There exists a $O(\ell)$-competitive algorithm for $k = 3$.*
Mahmoud: is this needed anymore?

Proof. The adversary issues at most 3ℓ requests, as otherwise the perfect partition would not exist. Each request issues at most one re-balancing for the greedy algorithm, and by Theorem 1, the total cost of any algorithm is $O(k \cdot \ell)$. □

We leverage the simple algorithm $ON_{k=3}$ that seeks a perfect partition for $k = 3$, towards an algorithm for the general online problem. Note that in the general model, an optimal partition is not a necessarily a perfect partition. We divide the sequence of requests into phases. A phase terminates when there is no perfect partition of components. We apply requests to $ON_{k=3}$ as before. Once the current phase terminates, we reset all components by removing (forgetting) all revealed edges, and then we proceed to the next phase.

Corollary 1. *There exists $O(\ell)$-competitive algorithm for the general online problem when $k = 3$.*

Proof. Consider any phase and assume it terminates upon arrival of some request r. By Lemma 1, ON pays $O(\ell)$ during the phase. Regardless of the of OPT's partition at the beginning of the phase, as long as OPT stays in the same configuration, it incurs remote communication cost for at least one request r' (possibly $r' = r$) by the end of the phase. Else, OPT incurs the cost of moving to new partitions. In any case, OPT must pay at least one during the phase. □

B. Perfect Partition Learner Algorithm

Maciek: $. \to \cdot$

Maciek: Font for OPT, ALG, DET, dist etc

Let $P_I = I_1, \ldots, I_\ell$ denote the initial partition with which both ON and OPT begin and $P_F = F_1, \ldots, F_\ell$ the final partition.

Definition 1. *The* distance *of a partition $P = C_1, \ldots, C_\ell$ is the number of nodes in P that do not reside in their initial cluster. That is, $dist(P, P_I) = \sum_{j=1}^{\ell} |C_j \setminus I_j|$.*

In other words, at least $dist(P, P_I)/2$ node swaps are required in order to reach the partition P from P_I. Therefore, $OPT \geq \Delta := dist(P_F, P_I)$. We devise an online algorithm that mimics OPT by minimizing the distance to the initial partition P_I with each re-partitioning. As a result, ON never ends up in a partition that is further away from P_I than Δ. This invariant ensures that ON does not pay too much while reaching P_F.

Maciek: "re-partition" procedure name should indicate the fact that this is a specific repartition that is close to initial configuration

Mahmoud: address the *re-partition()*

Property 1. *Let P be any partition chosen by Algorithm 1 at Line 7. Then, $dist(P, P_I) \leq \Delta$.*

Lemma 2. *The cost of re-partitioning at Line 7 is at most $2.OPT$.*

Algorithm 1 Perfect Partition Learner

```
Input: $k, \ell$, initial partition $P_I$, sequence of edges $\sigma_1, \ldots, \sigma_N$
Output: Final partition $P_F$
1: for each node $v$ create a singleton component $C_v$ and add it to $\mathcal{C}$
2: on communication request $\sigma_t = \{u, v\}$:
3: Let $C_1 \ni u$ and $C_2 \ni v$ be the container components
4: if $C_1 \neq C_2$ then
5:     unite the two components into a single component $C'$ and $\mathcal{C} = (\mathcal{C} \setminus \{C_1, C_2\}) \cup \{C'\}$
6:     if cluster($C_1$) $\neq$ cluster($C_2$) then
7:         re-partition($k, \ell, P_I, \mathcal{C}$)
```

Proof: Consider the re-partitioning that transforms P_{t-1} to P_t upon the request σ_t. Let $M \subset V$ denote the set of nodes that are migrated during this process. We have $M = M^+ \cup M^- \cup M^\circ$, where M^- (M^+) denotes the subset of nodes that enter (leave) their original cluster during the re-partitioning, and M° denotes the set of remaining nodes in M. Since $|M^- \cup M^\circ|$ nodes are not in their original cluster before the re-partitioning, by Definition 1, the distance before the re-partitioning is $dist(P_{t-1}) \geq |M^- \cup M^\circ|$. Analogously, the distance afterwards is $dist(P_t) \geq |M^+ \cup M^\circ|$. Thus, $|M| \leq dist(P_{t-1}) + dist(P_t)$. By Property 1, $dist(P_{t-1}), dist(P_t) \leq \Delta \leq OPT$ and thereby we have $|M| \leq 2.OPT$. ■

Theorem 4. *Algorithm 1 reaches the final partition while being $(2.k.\ell)$-competitive.*

Proof: The algorithm eventually reaches the final partition since it evaluates all possible ℓ-way partitions of components on each external request, including the request that completes revealing of all ground truth components. There are at most $(k-1).\ell < k.\ell$ calls to *re-partition()* and by Lemma 2 each costs at most $2.OPT$. The total cost is therefore at most $2.OPT.k.\ell$ which implies the competitive ratio. ■

VII. $O(k)$-Competitive Algorithm for the Ring

Maciek: todo

We know that the lower bound is k for the ring communication pattern (cite Marcin-Chen). The simple matching (assymptotically

Maciek: maybe it is easy to have k exactly?) upper bound is to fix the reconfiguration to be the cyclic shift. We reconfigure upon encountering inter-cluster request. We use phases: upon discovering the edge on each cut we charge OPT 1 and reset counters.

Appendix