Internship report - Vienna University

Julien Dallot

01/07/20 to 12/09/20

My internship was 2 months long, from 1st July to 12th September. During those two months, I used the vast majority of my time trying to solve a theoretical problem called *online balanced partitioning* (explained in next section). My supervisors were Maciej Pacut and Stefan Schmid, from the University of Vienna. In this report I present the major results of my internship in chronological order.

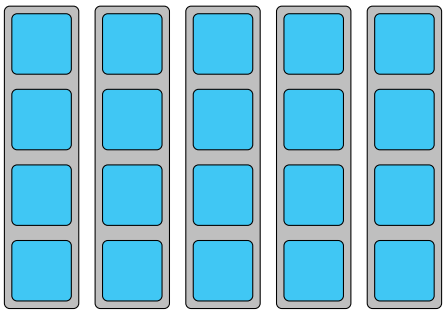
Contents

1 The problem : Online Balanced Partitioning

This problem aims to solve a real world issue encountered in data centers: there are multiple *virtual machines (VMs)*, that can communicate between one another. A *request* forces 2 VMs to communicate (a request can simply be seen as a pair of VMs). Those requests are coming and executed step by step in the order they arrives. Since the problem is an online version, we basically don't know anything about the incoming requests until they finally arrive.

The VMs are all stored in *clusters*. There is a certain number of identical cluster (l of them) and each one of them can store exactly k virtual machines. In this version of the problem, we consider that every cluster is totally fulfilled at any time (so they are exactly $k.l$ virtual machines in all). Here follows a

diagram that describe the situation; in this example, $k = 4$ and $l = 5$, the VMs are in blue, clusters in grey.



But here comes the heart of the problem. Each communication between two VMs needs a certain cost to be paid, which cost is related to the amount of additional traffic the communication involves. We obviously want to reduce that cost to avoid congestion problems. Let A and B be two VMs. Here is the cost of request (A, B):

- 0 if A and B are in the same cluster
- 1 if they are in different clusters

At last, it is possible at any time to move as many VMs as we want from a cluster to an other. This operation is called a *shifting* and it costs $\alpha \geq 1$ for each shifted VM. Since there never is a free space in any cluster, moving a virtual machine means that we must at least move one other VM, which makes the cost of any shifting more than 2α.

The main objective was to find an algorithm that could orchestrate the VMs shiftings, while trying to lower the total cost (communication + shifting cost) and making sure it doesn't exist a request sequence that could lead to arbitrarily bad costs (to avoid potential cyberattacks that might use on purpose traffic congestion). When I started my internship had Maciej Pacut and Stefan Schmid already imagined a quite simple algorithm based on empirical observations about the communications between VMs. More precisely, they noticed that when two VMs communicate in a data center, it is far more likely that those two communicate a lot afterwords. In other words, VMs in data centers tend to only communicate with a restricted number of neighbours; that's what inspired them the following algorithm.

Each time a new request arrives:

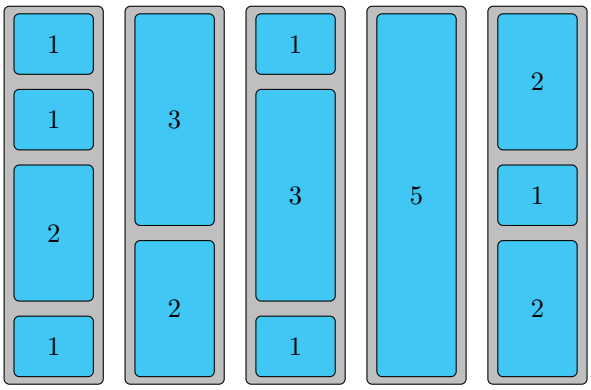
1 if the two VMs of the request are hosted in the same cluster, then nothing needs to be done and the VMs just communicate for free

2 else, if the two VMs aren't hosted in the same cluster, then

2.1 if there exists a shifting resulting with the two VMs of the request being in the same cluster, then compute such a shifting that has minimum cost and apply it. Until the case 2.2 is not reached, those two VMs will be "linked" to each other, meaning that they must always be hosted in a common cluster

2.2 else, in case there's no shifting resulting with the two VMs being in the same cluster without breaking links, then break all links and go back to step 2

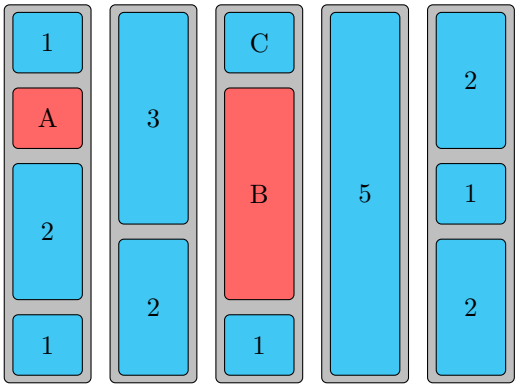
That algorithm trivially forces pairs of VMs which had already communicated with each other, to be hosted in a common cluster. It is to make sure that any further requests between those pairs, supposedly likely to happen, are costless. Besides, it is quite clear, once a communication occurred between 2 VMs, that those VMs now behave like a unique entity, as they will always be shifted together. We call such an entity a *component*, namely a group of VMs that must always be hosted in a common cluster. For the same reasons, we also consider that a request actually gives the order to gather two components, and not two VMs, since the results is the same. Finally, we define a *phase* as being any set of successive steps in the algorithm, where 1^{st} step has no link between VMs and last step reaches point 2.2 (when it's no more possible to gather the two VMs of the incoming request, while respecting component's constraints). From now on, we won't any more talk about VMs, but rather about components; here follows a diagram depicting such an instance, with $k = 5$ and $l = 5$. The figure in each component stands for the number of VMs it is composed of.



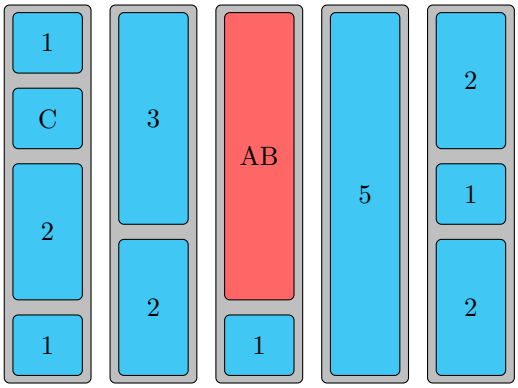
Since every traces of the previous iterations are erased at the beginning of a new phase, we decided only to consider what's happening during a phase. So here is the main question of the whole internship:

let be l clusters and k VMs per cluster, what is the maximum cost of a phase?

At first glance, the shiftings resulting from the arrival of a request seems to be quite simple and cheap, and that is true at the beginning of a phase, where all component's sizes are close to 1 (then it is always sufficient to move only two VM to fulfil the request). The shiftings can also be really cheap at a more advanced moment of the phase. For instance, let's go back to the previous example.

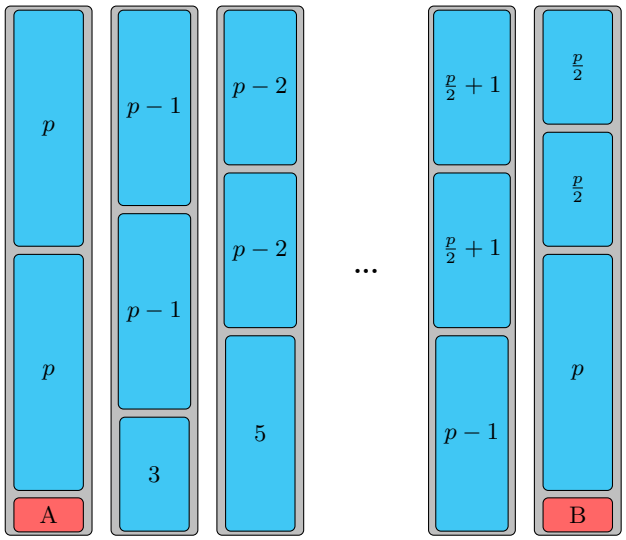


Here the components A and B must be merged, they are colored in red. In that case there is a really cheap shifting: just make an exchange between A and one of the components of size 1 in the same cluster as B, for example C.

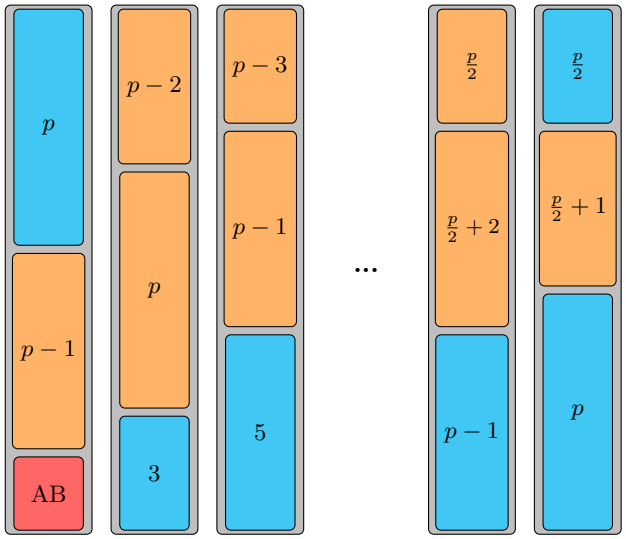


Here is the resulting configuration after the exchange of components A and C. Components A and B have merged into a bigger component of size 4. So the cost of the request (A, B) was 2α.

However the cost of a shifting can be far higher than this. Before I began the internship, Maciej Pacut and Stefan Schmid had found some impressively costly examples where almost every components were forced to move, even if we always chose the least expansive shifting. What's even more surprising is that the corresponding request only asks for two one-size components to be merged. This observation already prevent us from making any link between the size of the components to be merged, and the cost of the involved shifting. Because of that ability to bring about big shiftings while only asking for tiny moves, we gave to those shiftings the name *cascade*. Let p be some positive even integer. Here follows the most costly example we found so far, with $k = 2p + 1$ and $l = p/2 + 1$.



Here components A and B, each of size 1 and colored in red, must be merged. Notice that when B moves to the same cluster as A (ie cluster 1), there is no simple way to retrieve a valid configuration. For B to be able to come in cluster 1 we need to free up exactly one slot in it, that's why we exchange one component of size p in cluster 1 with a component of size $p-1$ in cluster 2. But then the max capacity of cluster 2 is exceeded by 1 slot; we need to exchange one of its components of size $p-1$ with another of size $p-2$ in cluster 3, etc. This chain finally ends when we reach the last cluster, where there precisely lacks one slot since component B moved to cluster 1.



Here is the resulting configuration after components A and B merged in cluster 1. We colored the components that moved during the cascade in orange. To sum things up, we had to move the two upper components for each cluster (except in the first and last clusters where we only moved one). Since the number of clusters l is a $\mathcal{O}(p)$ and so a $\mathcal{O}(k)$, **this cascade has a cost which is in** $\mathcal{O}(k^2)$.

Given those kind of enormous cascades that move almost every components, it first seems not so naive any more to think about the following trivial upper bound. As no component can move two times during one minimum cascade (otherwise it would not be a minimal one), then an upper bound on a cascade's cost trivially is the overall sum of all component's sizes, namely $k \cdot l$ (from now on, we consider that $\alpha = 1$ because it lightens the notations). Then, since every requests involves a strict increase of one component's size by at least 1, then

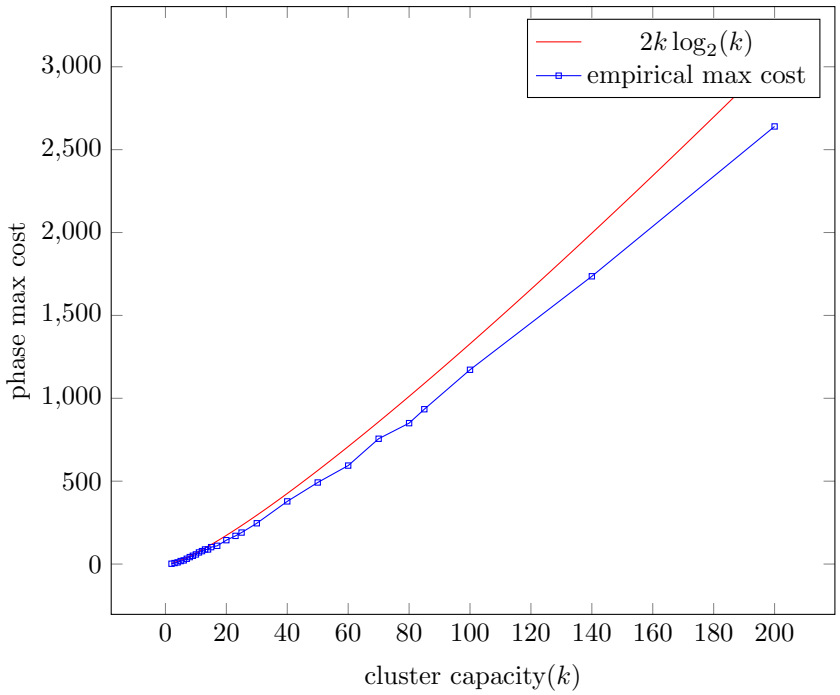
there cannot be more than $k \cdot l$ cascades during a phase. Finally, the cost of any phase cannot exceed $(k \cdot l)^2$. However, the empirical results that we got during this internship clearly suggest that there exists a far better upper bound that we suspect to be $l \cdot k \cdot \log_2(k)$. Before I began the internship had Maciej Pacut and Stefan Schmid already found a lower bound on the maximum cost of a phase of $\mathcal{O}(k \cdot l)$ (Brief Announcement: Deterministic Lower Bound for Online Balanced Repartitioning).

2 Encouraging empirical results (01/07/20 to 06/07/20)

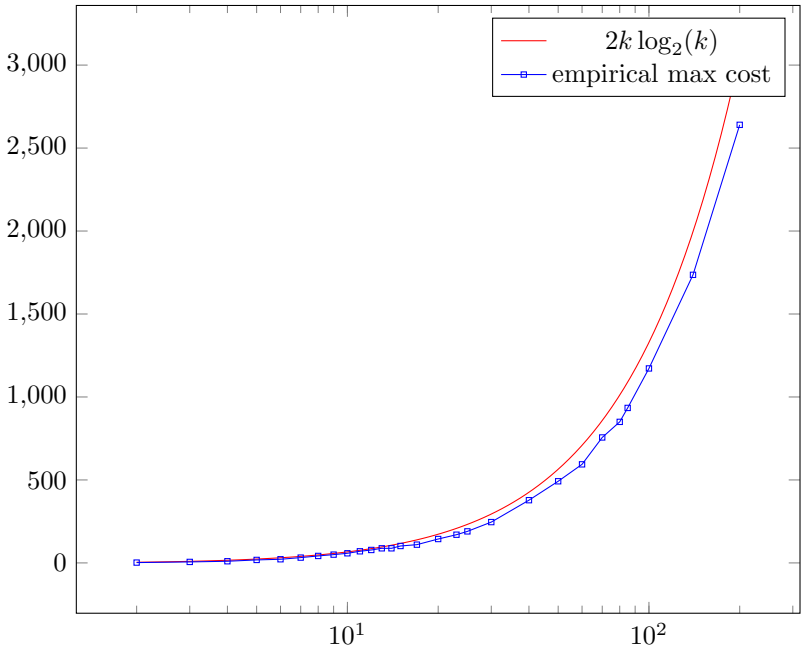
During the first week of the internship, Mr Pacut and I decided to have a clearer and faster way to compute cascades, that's why I coded a Python script which automatically computed a valid minimum cascade given a starting configuration and a request (get the github link here). This problem unfortunately is NP-Complete, thus we were highly restricted in the size of the problem instances, otherwise the computing time literally exploded. To lighten computation time, I used a Linear Programming with Integrity Constraint solver which exploited branch & bound techniques to run it faster. However it wasn't that helpful only to compute valid cascades so we managed to extend the code to whole phases computation.

Hence I coded a program that simulates entire phases with random requests, ie from the beginning with one-size components to the final point when the incoming request cannot be applied. Since this way of proceeding was highly biased, it was crucial to have a very large number of achieved simulations so that we potentially have significant results. Therefore we almost entirely focused on cases where $l = 2$ (ie when there's 2 clusters) during the two months since it spared computation time. To be faster, we had the script running for days and night, in parallel between my personal computer and the Vienna University's data center, whose power made it possible to achieve more than 2 millions trials, with k varying from 2 to 200. The following graph shows our results:

Max cost dependance of cluster capacity, with $l = 2$



Same as above with logarithmic scale on the abscisse axis

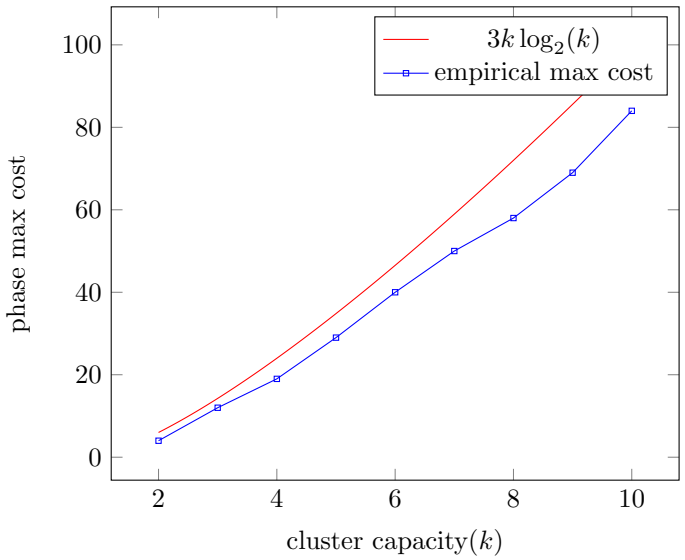


The empirical results the simulation gave us are in blue (I made the constant α equal 1), whereas the red curve is just a plot of the function $f(k) = 2k\log_2(k)$. One can better notice, looking at logarithmic scale plot, how much of a good approximation $2k\log_2(k)$ is. Even better, after millions of trials never had a phase's cost exceeded that function, however tight the approximation is. Those good results gave us new tracks to look into, and we tried for one month and a half to prove that $2k\log_2(k)$ indeed always is an upper bound of a phase's cost.

However, those observations weren't sufficient at all to deeply convince us that this upper bound actually holds and always fit the curve in a tight way. We can approximately explain the relative separation of the two curves as k grows by observing, that the higher k is, the numerous phase possibilities are and thus the worse our empirical max cost estimation is. But they are some huge bias factors. First, we obviously can reach admissible results only for small instances. Second, our MILP solver (Mixed Integer Linear Programming) favors without doubts a certain type of cascades, which narrows our field of investigations (this last bias actually is not that worrying, as we can integrate the specific kind of cascade it does in our algorithm so that it keeps the results we find, if we ever find any).

Even if this upper bound eventually gets exceeded, this at least shows us that it is worth to seek for a better upper bound than $(l.k)^2$, or at least $4k^2$ if we think it's only true for $l = 2$. About the other values of l, we've done more than 100.000 trials for $l = 3$ and k from 2 to 10 and the simulations gave us enthusiastic results in that the function $lk\log_2(k)$ still is an upper bound and a tight approximation. Here follows the results I had with $l = 3$, even if they certainly are not significant.

Max cost dependance of cluster capacity, with $l = 3$



In the remaining parts I will have the pleasure to explain the main tracks we had think about, to finally overcome this problem.

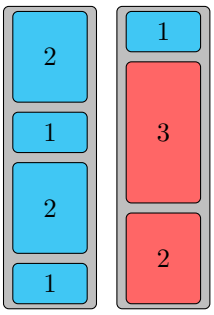
3 The global model (06/07/20 to 30/07/20)

After some brain storming with Mr Pacut, we decided to focus on that strange appearance of the logarithm function that our empirical data seems to highlight. We also decided to completely focus on the $l = 2$ case for it is simpler to compute and to figure out. How could a logarithm intervene in such a problem? The usual way for a logarithm to appear is when we do successive divisions by 2, basically within fusion sort for example (notice that we used a logarithm in base 2 and that it fits particularly well with our empirical results, hence we found it worth our time to especially look for divisions by 2).

Meanwhile I've had the feeling that the problem we study could be completed, in the sense that every time we supposed that two components merge, we had to make sure they are from different clusters; why couldn't we merge components whose clusters are the same? To me this appeared like a hole in our theory, as if you didn't consider 0 to be a part of the integers. By filling this hole, you could end up with a more consistent and understandable theoretical framework. So, what should be the cost of such a merge that take place within components from the same cluster? Let A and B two components, merging them has the following cost:

- Cost of a minimum cascade, if A and B are in two different clusters
- $2 \cdot \min\{|A|, |B|\}$ if A and B are in the same cluster, with $|.|$ the number of VMs it is composed of

Here follows an example with $l = 2$ and $k = 6$:

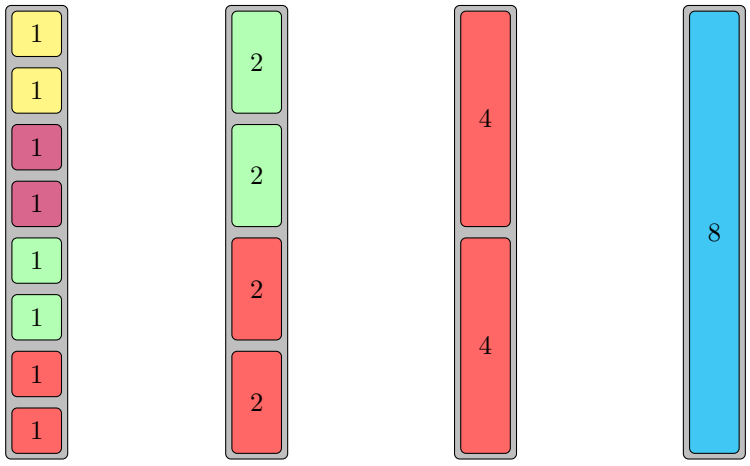


In this example, the request that just came concerns the components colored in red. Since those components are located in the same cluster, then the cost of the merge equals $2 \cdot \min\{2, 3\} = 2$.

We called it the *Global Model* since any component can be merged with any other. To find this cost of $2 \cdot \min\{|A|, |B|\}$, I made a lot of personal trials and I intuitively ended up thinking, this was the most natural way to do it. Indeed, the costs this model involves are quite comparable with those of the normal

model. Furthermore it intuitively makes sense to pay that cost for intra-cluster merges; figure out A and B actually are in different cluster, and it is possible to simply move A in the same cluster as B and vice versa. Then the minimal cascade would be to move the component with lower size, hence the cost would indeed be $2 \cdot \min\{|A|, |B|\}$.

Notice that the global model "contains" the normal model, in a sense that we can compute a normal phase within a global one. This means that if an upper bound holds for the global model, then it will hold for the normal one as well. But what was actually promising with that model, is that it finally gives us a track to explain the appearance of the logarithm function. First, let's wonder what would be the maximum cost of a phase, if we can only merge components that are in the same cluster. We call this framework the intra-cluster model. Then, every clusters become isolated from the others and thus we can focus on only one of them. After some calculation Mr Pacut and I proved that the asymptotic upper bound of a phase's cost with only one cluster is $k \log_2(k)$. This result comes from the fact that we can maximize the cost by always merging components of same size (when its not possible, merge components whose sizes are close). Here is an example for $k = 8$, components that will be merged are in the same color.



We indeed end up with a cost that exactly is $24 = 8 \log_2(8)$. This upper bound is only reached for the k that are powers of 2, but it is also a good upper bound for the others. Since the clusters are totally independent during a phase, then the total maximum cost is obtained by summing the cluster's maximum costs. Hence we have the wanted upper bound $l \cdot k \cdot \log_2(k)$. But how to make a direct link between this result and our problem? Here unfortunately ends this track, because we weren't able to extend it to the global model, where merges can also occur in different clusters ... I also coded a script that simulates

global phases and I found some counter examples where the upper bound gets exceeded. Even worse, I found some phase examples in the normal model, whose costs where greater than his of the intra-cluster model, which definitely prevented us from finding a link between those two models. We had to find some other tracks.

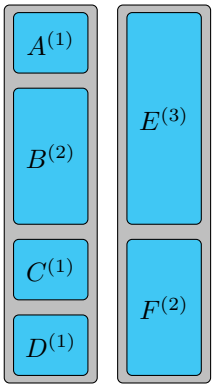
4 Sub-clusters (30/07/20 to 20/08/20)

Mr Pacut had the feeling that we might prove our assumptions by using some induction proof on the number applied requests since the beginning of the phase. We spent many hours trying to make a proof of it in front of a white board, and something was still missing. First, the $k \log_2(k)$ formula didn't quite fit the induction reasoning as the heredity part brought pretty complex terms that we weren't able to link with anything solid. Second, I intuitively found it highly unlikely to find anything relevant by doing so, since we only based our induction on the number of already applied requests.

Indeed, as we have seen it in the first section, they can exist really costly cascades; that observation could lead us to think that phases can be really costly as well. The empirical results highlighted though that we can find a better upper bound. Thus the successive cascades must be amortizing each other's cost during a phase, that is to say, for each costly cascade they are many other cheap ones. That's why I introduced a new notion to work with, that deeply took into account the previous cascades: *sub-clusters*.

Definition (sub-cluster). In a configuration where $l = 2$, a sub-cluster is a subset of components, whose total size on first and second clusters are the same.

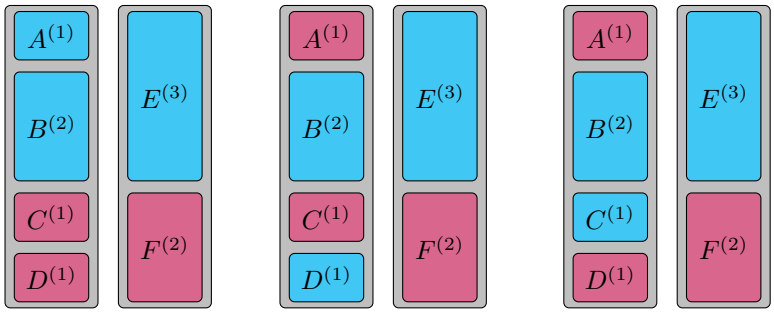
For instance, let's look at the following configuration:



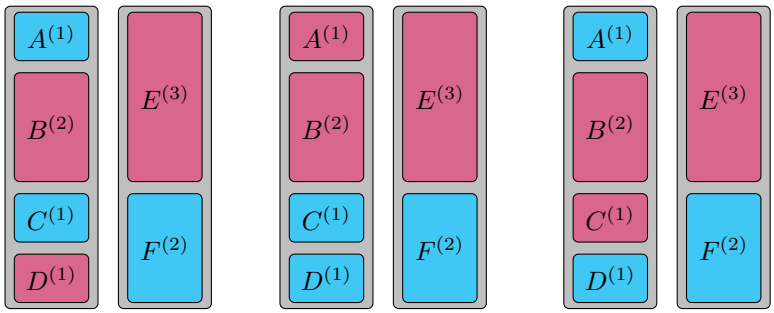
Here, the size of each component is indicated as an exponent of its name. The sub-clusters are sets of components that have the same total size on the left and on the right. For instance, $\{A, B, C\}$ is a sub-cluster as its size on the left and on the right is the same and equals 3.

But how many sub-clusters are they? We can count them using their size. There is only one sub-cluster of size 0, which is the empty set $\emptyset$. There is

obviously no sub-cluster of size 1 since the right cluster has no subset of size one. There are 3 sub-clusters of size 2, here there are colored in purple:



There are as well 3 sub-clusters of size 3:



Finally, there are no sub-clusters of size 4, and only 1 sub-cluster of size 5 which is the set of all components $\{A, B, C, D, E, F\}$. So in total there are $1 + 3 + 3 + 1 = 8$ sub-clusters in this configuration.

How could the sub-clusters be any helpful? First, we can notice that a cascade namely is a sub-cluster that gets exchanged its left and right parts. There is a bijection between the two of them, so introducing sub-clusters gives a simple theoretical framework to work on cascades. But what's definitely interesting about sub-clusters, is that their number gives an other indicator to follow the evolution of the configuration during a phase. At the beginning of any phase, they are $\binom{2k}{k}$ sub-clusters (one can retrieve this result with an explicit summation), their number goes down to 2 at the end of a phase, assuming that the phases we consider still go on until it's no more possible to merge something. But what was really promising to me is the following Lemma:

Lemma. In a configuration where $l = 2$, the number of sub-clusters remains unchanged after any shifting. As a result, that number only changes when components get merged; it also strictly decreases during a phase.

Proof. Let C be a configuration during a phase with $l = 2$ and k an integer. Let $S = (S_L, S_R)$ be any sub-cluster in C with S_L its left part and S_R its right part, let $S' = (S'_L, S'_R)$ be the next sub-cluster to be exchanged because a request just came. We show that after S' gets swapped, then a new sub-cluster appears. The following holds:

$$\begin{aligned}
& w(S_L) = w(S_R) \\
\iff \quad & w(S_L \setminus S'_L) + w(S_L \cap S'_L) = w(S_R \setminus S'_R) + w(S_R \cap S'_R) \\
\iff \quad & w(S_L \setminus S'_L) - w(S_R \cap S'_R) = w(S_R \setminus S'_R) - w(S_L \cap S'_L) \\
\iff \quad & w(S_L \setminus S'_L) + w(S'_R) - w(S_R \cap S'_R) = w(S_R \setminus S'_R) + w(S'_L) - w(S_L \cap S'_L) \\
\iff \quad & w(S_L \setminus S'_L) + w(S'_R \setminus S_R) = w(S_R \setminus S'_R) + w(S'_L \setminus S_L)
\end{aligned}$$

After the swap, $S_L \setminus S'_L$ and $S'_R \setminus S_R$ are in the left cluster, $S_R \setminus S'_R$ and $S'_L \setminus S_L$ are in the right one. The above equations show that their total cost is equal on each cluster, so $\{S_L \setminus S'_L, S'_R \setminus S_R, S_R \setminus S'_R, S'_L \setminus S_L\}$ indeed is a sub-cluster. Since we only used equivalence relations, then the described process is reversible. Finally, at each sub-cluster before the cascade corresponds only one sub-cluster after the cascade and vice versa. Hence the claim holds. □

I spent a lot of time trying to work with those sub-clusters, to use their properties within a proof. Anyway it didn't give any relevant results, and it took me quite a while to figure out, that this property on the number of sub-cluster is no more than a trivial observation that cannot be sufficient to prove anything. Here follows that observation: let C be our current configuration and let $\mathscr{C}$ be the set of the configurations we can reach from C. Notice that the cardinality of $\mathscr{C}$, $|\mathscr{C}|$, equals the number of sub-clusters in C since we can by definition obtain any element of $\mathscr{C}$ by swapping a sub-cluster in C. Let C_1 be an other configuration and S_1 a sub-cluster in C that lead to C_1. Its quite easy to see, that we can still reach any configuration of $\mathscr{C}$ starting from C_1. Indeed let C_2 be an other configuration in $\mathscr{C}$ that we can reach from C with the sub-cluster S_2. Then, starting from C_2, we just need to apply the inverse swap of S_1 to go back to configuration C, and then apply S_2 to reach C_2. Since a composition of cascades still is a cascade, we can reach any element of $\mathscr{C}$ from C_1 and from C, meaning that C and C_1 have the same number of sub-clusters. Even if this attempt hadn't been successful, those cascade compositions and inversion gave me ideas for the last track I looked into.

5 Using Group theory (20/08/20 to 12/09/20)

For a long time I had the feeling that it was somehow possible to study this problem under the framework of group theory. The cascades made think about rubiks cube moves, but without being able to investigate any further. Take the case where $l = 2$ to make things easier, let C be your starting configuration.

Then try to consider that the sub-clusters you can swap (ie the cascades you can apply, they're both the same) describe a group $(G, .)$. The following must hold:

- The composition law is internal (ie s_1, $s_2 \in G \implies s_1 s_2 \in G$)
- There exist a neutral element $e \in G$ st $\forall s \in G$, $es = se = s$
- Any element has an inverse (ie $\forall s \in G, \exists\ s^{-1} \in G : ss^{-1} = e$)

Clearly, G is the set of sub-clusters that we can swap on a given configuration. The composition law is the resulting sub-cluster that we get after swapping two sub-clusters consecutively. Te neutral element e is the empty sub-cluster and the inverse of a sub-cluster must be himself, since swapping it twice in a row bring us back to the same configuration. But there's something wrong about all of this. What really is an element of G? It cannot actually be a sub-cluster as we defined it previously, ie a set of components. If we say an element of G simply is "swap components set A with components set B", then what happens if we want to apply such an element on a configuration where A has its elements shared between the two clusters? Well it has no sense any more and G is not a group.

I tried a lot of things and didn't manage to find a group to describe the whole problem even if I had the strong intuition there was a link. Anytime I thought about new ideas, there was always a single point of the group definition that didn't match. An then, after weeks of trials, I had the idea not express the problem with one group, but rather with many groups at the same time. For that, let's define a *matching*:

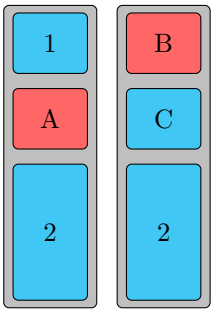
> **Definition (matching).** In the special case of two clusters, a matching is a pairwise matching between VMs. In other words, it is a set of couples of VMs where all the VMs are present.

Let C be the current configuration where all components are of size 1 (at the beginning of a phase) and M be a matching. We say that M is valid on C if every couple of M have one element in the left cluster, the other on the right, which means that swapping those two VMs has a sense. Then we can define a group G_M as follows:

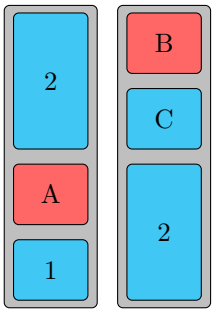
- For any $m \in M$, $m \in G_M$
- For $s_1, s_2 \in G_M$, $s_1 s_2$ is "swap s_1, and then swap s_2"

Now we can see that G_M really is a group. Its neutral element is the empty set (meaning "swap nothing"), each element has an inverse which is himself. The main difference with before is that, within a group G_M, a given VM can only be swapped with its match. Then, on a given configuration C, if it happens that two matching VMs are hosted in the same cluster, we just say that the group G_M is not valid, meaning that is just cannot be used by now.

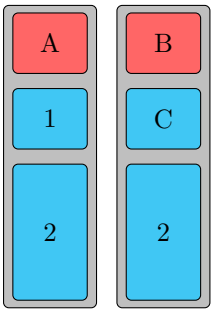
We have created many groups ($\frac{k!}{2^k}$ in total, the number of different matchings); given a configuration C, some of them are valid, the rest are not valid. But what happens when we actually merge two components A and B? There are two case. Let M be a matching and G_M its associated group. If A and B were matched in G_M, then it means that G_M won't ever be useful any more since it won't be possible to swap A and B. We can say that G_M is somehow deleted. Otherwise, if A and B weren't matched in G_M, then we end up with a subgroup of G_M after the merge. Here is an example. In the following diagrams M_1, M_2 and M_3 are three matchings, two VMs are matched if they are opposite each other. The to-be-merged components A and B (of size 1) are in red, and we assume that they will merge after A swapped with with a component C.



Matching M_1

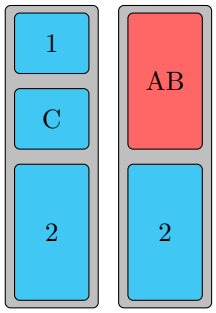


Matching M_2



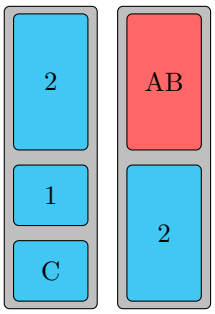
Matching M_3

Those three matchings all are valid in this configuration, however they are going to have different reactions toward the merge of A and B. Let's start with M_1:



After the merge, we see that its no longer possible to swap A and C alone. Hence we lost one degree of freedom in group G_{M_1}

Matching M_1



After the merge, we see that there's no difference for group G_{M_2}. Hence the same swaps are possible, no more no less.

Matching M_2

About matching M_3 finally, we can see that it's no longer possible to swap A and B after the merge. Hence the group G_{M_3} won't be valid any more during the rest of the phase.

Now that we described how do the groups evolve as we merge components, let's have a closer look to the costs of the swaps. Because G_M is an abelian group for any matching M (meaning its elements are commutative ie $s_1 s_2 = s_2 s_1$), then its cardinality $|G_M|$ is a power of 2. At the beginning of a phase, the cardinality of any group obviously is 2^k, but afterwords it ends up being 2^d with $1 \leq d \leq k$ (it can never be 0 since there are always at least 2 elements, the neutral element e and the full element which consists in swapping all the components). We call d the dimension of the group G_M, since we can retrieve every element of G_M by combining only d well chosen elements, kinda like a basis in a vector space.

Now that we have those notions, let C be the current configuration with $l = 2$ (at any time within the phase), and let's say the next request concerns two components A and B. Let M a valid matching on C and G_M its associated group, and let $H_M \subseteq G_M$ defined as follows: $\forall s \in G_M, s \in H_M \iff A \in s$ and $B \in s$, or neither A nor B are in s. In other words, H_M actually contains the elements that don't put A and B in the same cluster. It seems like these precisely are the elements we aren't interested in, but H_M actually plays a vital role. Indeed, the following holds:

Lemma. Let M be any matching, then H_M is a subgroup of G_M and, if $H_M \neq G_M$ then $\dim(H_M) = \dim(G_M) - 1$.

Proof. To prove that H_M is a subgroup of G_M, just see that the neutral element e belongs to H_M (e indeed contains neither A nor B). For $s_1, s_2 \in H_M$, then $s_1 s_2 \in H_M$ and for $s \in H_M$, then $s^{-1} \in H_M$ (I'm not proving it here due to lack of place).

Then let's take a look at the dimension of H_M, and let d be the dimension of G_M. There are 2 cases: if $d = 1$, then it means that the only 2 possible swaps are to swap the entire clusters (which contains A and B) or the neutral element e (which contains neither A nor B). G_M and H_M contain the same

elements, they are equal and H_M's dimension is d. On the other hand, let's assume that $d > 1$, then we're going to prove that H_M's dimension is $d-1$. Let $s_1,\ s_2 \in G_M \setminus H_M$, then s_1 and s_2 contain only A or B. Due to lack of space, I will take for granted that $\exists h \in H_M$ st $s_2 = h.s_1$. With that said, I prove that we can reach any element of G_M by combining elements of H_M with only one element of $G_M \setminus H_M$. Let $s \in G_M \setminus H_M$, let $g \in G_M$; if $g \in G_M \setminus H_M$, then there exists $h \in H_M$ st $sh = g$. Otherwise if $g \in H_M$, then no problem. So, a set containing s and a base of H_M totally describes G_M: $\dim(G_M) = \dim(H_M) + 1 \implies \dim(H_M) = d - 1$. □

Let $(h_1, h_2, \ldots, h_{d-1})$ a base of H_M, and $s \in G_M \setminus H_M$, then $B = (h_1, h_2, \ldots, h_{d-1}, s)$ is a base of G_M. In other words, we have that $G_M/H_M = \{H_M, sH_M\}$; we can fully explore the set of swaps that put A and B in the same cluster by taking a "seed" swap $s \in G_M \setminus H_M$, and then composing it by the elements of H_M. Then let's prove an upper bound on the cost of the minimal swap that put A and B in the same cluster.

Let $(h'_1, h'_2, \ldots, h'_{d-1})$ be an orthogonal basis of H_M, which means that $\forall i, j \in \{1, \ldots, d-1\},\ i \neq j \implies h'_i \cap h'_j = \emptyset$. There exists only one i such that $A, B \in h'_i$, let's call h'_{AB} that swap. Since, the full swap of size k belongs to H_M, we have that

$$
\begin{aligned}
& \sum_{i=1}^{d-1} |h'_i| = k \\
\implies \quad & |h'_{AB}| = k - \sum_{\substack{i=1 \\ i \neq AB}}^{d-1} |h'_i| \\
\implies \quad & |h'_{AB}| \leq k - d + 2 \\
\implies \quad & |s \cap h'_{AB}| \leq k - d + 2 \text{ and} \\
& |h'_{AB} \setminus (s \cap h'_{AB})| \leq k - d + 2
\end{aligned}
$$

Let s_{min} be a minimum swap in the matching M. Then, by choosing the minimum swap between $s \cap h'_{AB}$ and $h'_{AB} \setminus (s \cap h'_{AB})$, we get the following inequality since their intersection equal $\emptyset$.:

$$|s_{min}| \leq \frac{k-d}{2} + 1 \tag{1}$$

To shorten this inequality, we can now have a look at all the other matchings and take the ones whose dimension d_{max} is maximum. Let now s_{min} be an overall minimum swap, we get:

$$|s_{min}| \leq \frac{k-d_{max}}{2} + 1 \tag{2}$$

Despite this approach gave me a better understanding of the problem, I'm afraid I couldn't go further than that. If we sum this inequality over an entire phase, then we hence get an inequality about the total cost of a phase, but I cannot really figure out how evolves that dimension d over a phase. Moreover, I actually don't think the above inequality is tight enough to reach a goof upper bound; maybe it is possible to get a better one using group theory though.

6 Conclusion

This internship was a step forward in the world of research, and I really enjoyed it. I discovered what it does to only think about one thing over and over, in the office, on my way to the office, during the night ... It was passionate to discover this type of life. I want to thank Stefan Schmid and Maciej Pacut a lot for their patience, their every day kindness and benevolence towards me; I especially thank Maciej Pacut for the amazing hours we had in front of that white board. He gave credits to my ideas, even though I only was a 21-years-old student, and that represented a big step in developing my self-confidence. After that internship, I told myself it wasn't that crazy to become a researcher.