Merge, Split, and Cluster: Dynamic Deployment of Stream Processing Applications

Aymen Jlassi, Cédric Tedeschi

Merge, Split, and Cluster: Dynamic Deployment of Stream Processing Applications

Aymen Jlassi
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
aymen.jlassi@irisa.fr

Cédric Tedeschi
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
cedric.tedeschi@inria.fr

***Abstract*—Stream Processing (SP) is now a major paradigm to timely handle large volumes of data generated at the edge of the Internet. Stream processing engines (SPE) are tools easing the specification, deployment and monitoring of SP applications. Such applications are typically programmed as a directed acyclic graph (DAG) of operators to be applied on each data item. Yet, SPEs are mostly equipped to deploy one application at a time without seeking synergies between those applications. Yet, in many domains, the set of operators composing applications overlap for a non-negligible amount. We envision a platform on which applications are submitted dynamically, each new graph of operators potentially sharing some of them with the currently running operators. We assume a homogeneous platform, a graph being deployed over multiple nodes. We need to minimize the inter-node traffic while guaranteeing that the capacity of a node is not exceeded. This paper presents the *Merge, Split and Cluster* approach: each time a new DAG of operators is submitted, i) its operators are first merged with the already running operators, ii) if an operator's load thus created exceeds the nodes' capacity, the operators gets split into several instances, and iii) the operators of the resulting graph are clustered, each cluster being hosted by a single node so as to maximize intra-node traffic. Two heuristics are proposed for this last phase. Simulation results show that i) merging allows to drastically reduce the needs in computing resources, and ii) that the heuristic provides an efficient clustering minimizing the intra-node traffic.**

***Keywords*-Stream Processing, Deployment, Clustering**

I. Introduction

Stream Processing (SP) is becoming the dominant paradigm when it comes to process continuous flows of data. To be able to produce timely knowledge, Stream Processing follows the principle of processing each new data record independently as soon as it is produced, so the information it carries is immediately added to the current knowledge. A Stream Processing application is typically a set of operators structured as a pipeline, or more generally as a directed acyclic graph (DAG), that each data item traverses.

Stream processing has shown adequacy in the support of many application domains ranging from social networks to surveillance. Different applications in a given domain are likely to share part of their operators. This is for instance the case in maritime surveillance, our motivating use case. Platforms such as [1] are dedicated to the aggregation and processing of AIS signals[1]. Such platforms include software tools able to extract relevant information about a particular maritime area or a particular subset of ships of interest. Yet, such processing still heavily rely on a human operator to navigate through this too big amount of data and extract the relevant information out of it.

The Sesame project[2] aims at developing innovating tools to help human operators take critical decisions when dealing with an abnormal situation involving ships. Such tools must quickly raise alerts when such a situation arises. The typical kinds of requests to be answered in such a monitoring system are:

- *How many tankers recently entered a given country's exclusive economic zone (EEZ) at a high speed?*
- *Is there currently any abnormal behaviors in this EEZ involving two fishing vessels?*

Such queries, implemented following the Stream Processing paradigm, will combine filters so that only relevant boats are kept. Also, part of the operators will be present in both pipelines. In many application domains, the set of building blocks out of which most complex applications of this area can be built is actually relatively small.

As such a platform should be able to deal with multiple queries received at different times, detecting overlaps between the graphs can enhance significant reuse and reduce the deployment costs. More precisely, a first step to the deployment of a pipeline is to identify how many of its operators are already running previously submitted query. Then the two graphs can be at least partially merged, providing this did not result in a too high increase in the load of common operators regarding the capacity of the compute node hosting it. In such a case, the operator has to be replicated. Finally, since the new set of operators to host was modified, the grouping of operators over compute nodes has to be revised, with the goal of ensuring that no compute node is overloaded while the traffic between nodes is minimized.

In this paper, we propose an algorithm to handle the dynamic deployment of stream processing applications which exhibit a non negligible overlaps. We assume graphs are

[1]The signal sent by ships to warn about their presence
[2]http://recherche.imt-atlantique.fr/sesame/

submitted to the platform at arbitrary times and put into a queue. Each new graph needs to get merged with the currently running set of operators, the operators which, following this step, are too heavily loaded needs to get split. Finally, the grouping of operators needs to get revised. The approach proposed is composed of three steps: i) **Merging** the currently running operators with the newly submitted DAG, by identifying the common prefixes in these DAG. In the worst case, there is no overlap between the two DAGs, and the graphs are left unchanged. ii) **Splitting** the operators whose load exceeds the compute nodes' capacity. Such operators are duplicated, and the load of the initial node is shared between the replicas. The problem of finding new compute nodes is here out of scope. The following simply assume that new nodes can be allocated dynamically. iii) **clustering** the operators, each cluster being deployed over a different compute node. The algorithm relies on a clustering heuristic minimizing the amount of inter-operator traffic, which translate into actual inter-node traffic: The traffic between two operators being grouped together into a single node does not generate any real traffic.

These algorithms have been implemented in a simulation tool so as to capture their ability to optimize the needed number of nodes to support the applications as well as to minimize the actual network traffic induced by the applications once deployed by the algorithms.

Section II more formally exposes the problem solved. Section III describes globally the proposed resolution scheme and its different phases. Section IV details the two heuristics used for the clustering phases and discusses their complexity. Section V presents the simulation experiments conducted and their results. Finally, before a conclusion, Section VI presents the related work.

II. Problem Description

A. Platform model

We assume a platform composed of a homogeneous set of nodes that can be scaled up or down. This reflects a typical cloud platform, providing a limited number of virtual machine sizes. We assume for simplicity, that there is a single VM flavour which can be allocated, of capacity C, and that the number of actual nodes to be allocated is to be decided based on the workload. C is similar to a processor frequency: it abstracts out the amount of processing it can do during one time unit. VMs are referred to as *compute nodes* or simply *nodes* in the following.

B. Application model

We target platforms dedicated to a given application domain where applications are submitted online by some users so they can be deployed. These applications are submitted as DAGs of stream processing operators. Because these applications answer similar queries, we assume that the chance of having common operators between applications is high. More precisely, an application can be represented by a DAG $G = (V, E)$ where V is the set of operators of the represented application. Each operator is associated with a load $l(v)$ which represents the processing capacity it requires per time unit to process its incoming load without incurring delays. Each edge $e \in E$ is associated with a weight $w(e)$ which expresses the amount of data which traverses this edge per time unit. We do not constraint the possible graphs with a relation between the weights of edges and the loads of operators: high input velocities do not necessarily lead to high processing load: each record may be very simple to process. An operator v can exist in one or more replicas: if some operator v exhibits a load $l(v) = 300$ and that compute nodes have a capacity $C = 100$, it is necessary to have three replicas for it, each running on a distinct node.

C. The clustering problem

At its core, our problem is to be able to cluster the operators into compute nodes while minimizing the inter-node traffic. We denote P_G a **partition** of G. A partition is a set of **clusters** composed of operators in V, where each cluster contains a set of connected operators and each operator belongs to one and only one cluster. In other words, $P_G = \{cl_1, cl_2, \ldots, cl_k\}$. In a partition, we can distinguish the *intra-cluster* edges from the *inter-cluster* edges. Intra-cluster edges are the set of edges whose both endpoints belong to the same cluster:

$$Intra(E) = \{(u, v) \in E : u \in cl_i, v \in cl_j, i = j\}$$

Inter-cluster edges are the set of edges whose endpoints belong to different clusters:

$$Inter(E) = \{(u, v) \in E : u \in cl_i, v \in cl_j, i \neq j\}$$

A partition P_G has a *cost* and a *value*: The value P_G is the sum of the weights of *intra-cluster* edges. The cost of P_G is the sum of the weights of *inter-cluster* edges. Each edge contributes either to the cost or to the value of P_G:

$$cost(P_G) = \sum_{e \in Inter(E)} w(e)$$

$$value(P_G) = \sum_{e \in Intra(E)} w(e)$$

$$value(P_G) + cost(P_G) = \sum_{e \in E} w(e)$$

The objective is to cluster the operators, each cluster being deployed on a distinct node. Such an objective has the following constraints: i) The set of operators deployed on each node must not exceed the node's capacity, and ii) The traffic between nodes should be minimized. More formally, given a platform whose computes nodes' capacity is C, the goal is to find a partition P'_G such that Equations 1 and 2

are verified:

$$\forall cl \in P'_G, \sum_{v \in cl} l(v) < C \quad (1)$$

$$cost(P'_G) \text{ is minimized} \quad (2)$$

Graph partitioning is most commonly addressed in a slightly different version in the literature, where the constraint is not to ensure the cluster's size does not exceed a certain threshold, but where the imbalance between clusters is being minimized [2]. Yet, given our strong constraint on the nodes' capacity, the problem described is closer to a knapsack problem with additional constraints related to minimizing the cost coming from the edges. In both cases, the problem appears to be NP-complete [12], [13]. Yet, when the graph is a tree, optimal pseudo-polynomial algorithms exist [17], [16], [13] for the problem.

D. The problem in context

The partitioning is only one of the needed steps towards the deployment of a given application. We assume a platform upon which multiple applications are submitted online for their immediate deployment.

Each time a new application is submitted, different things need to be ensured, namely, that i) any reuse of the currently deployed operators running due to previous application is possible, ii) that no new operator exhibits a load higher than a node's capacity, in which case, replication of the operator is needed, iii) that the new graph is efficiently clustered as per the objectives described in Section II-C. This led to the development of the Merge, Split and Cluster (MSC) approach, described in Section III.

III. The global MSC algorithm

We now describe the general algorithm applied each time a new application is submitted. It is composed of three phases: i) the **merging** of the submitted application with the current graph of operators running, ii) the **splitting** of any new operator exceeding the capacity of one compute node, and iii) the **clustering** of the resulting graph so it can be deployed efficiently over the compute nodes. We need here to refine a bit the representation of an operator. An operator is represented by the following tuple $(name, replica, level)$ where:

- $name$ denotes the actual function provided by the operator;
- $replica$ is the actual replica number, each replica of an operator having a unique ID (allowing to order them);
- $level$ denotes the level of the operator in the graph, *i.e* its distance to the root of the DAG;

A. The Merging Phase

The merging phase consists in merging the running operators represented by $G_{current}$ with the operators of the newly submitted operators, represented by DAG G_{new}. It is similar to identifying common prefixes between two graphs, as such common prefixes can get merged: if two applications start with a similar sequence of operators, the goal is to avoid a useless duplicated deployment.

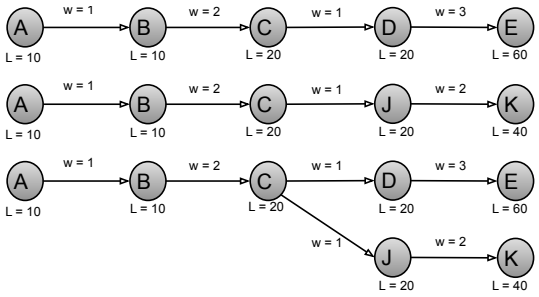


Figure 1: Benefit obtained by the Merging phase.

The merging phase input is composed of the two graphs and the maximum capacity of a compute node. Prefixes are subgraphs composed of *redundant* operators, *i.e.*, the operators in G_{new} that can be found in $G_{current}$ such that their name and in-degree are the same and their predecessors are also redundant. The merging phase output is a graph of operators where the redundant operators appear only once, while the non-redundant operators are unchanged. Figure 1 illustrates this process: the top graph is the graph of current running operators. The middle graph is the newly submitted graph. The third line illustrates the result of the Merging: operators A, B and C form a common prefix of redundant operators. Only J and K are kept and linked to the C operator present in $G_{current}$.

B. The Splitting Phase

After the new graph is merged into the currently running operators, each operator's load is checked against the capacity of one node C. Any operator v whose load exceeds C is split into several replicas v_i, which will receive its own share of the load, yet trying to maximize the utilization of nodes. The number of replicas for v will be $l(v)/C$ where the new load of each node v_i is $l(v_i) = C$ except for one replica which takes the remaining load. New links are created in the process between the parent of v and the set of v_is, the weight of the initial link being splitted similarly. This process illustrated in Figure 2 for a node v whose initial load is 950, the weight of its incoming link is 50 and $C = 300$.

C. The Clustering Phase

The third step of the submission is the clustering phase. It groups together the operators in compute nodes so as to optimize the resource usage of the deployment, as described in Section II-C, specifically maximizing the utilization of compute nodes while minimizing the inter-node traffic generated.

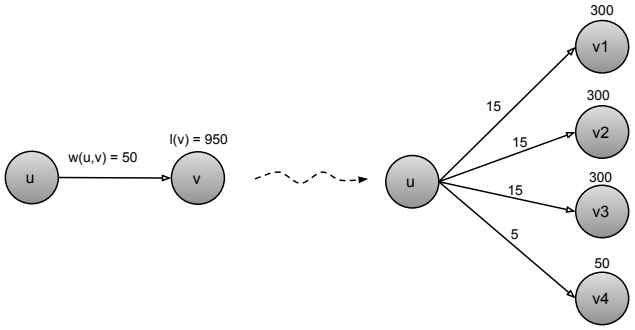


Figure 2: Splitting an operator.

Two heuristics were devised to achieve this phase. The first one, called *Tree-Optimal Clustering (TOC)* is the adaptation on a pseudo-polynomial optimal algorithm [17] having the same objective function, but for the trees. The second one, referred to as *Greedy Clustering (GC)* and is a linear-time greedy heuristic.

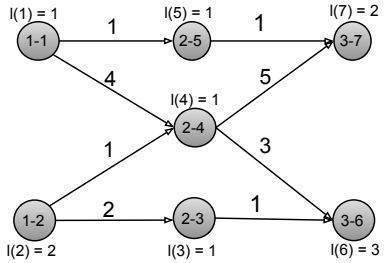
IV. Clustering Heuristics

We here describe the two heuristics mentioned previously in more details. Both rely on dynamic programming to cluster the graph so cluster's cumulated load never exceeds a given maximal capacity and the cumulated inter-cluster weight is minimized. They offer a different tradeoff between closeness to optimality and time complexity. Let us introduce few notations common to both algorithms:

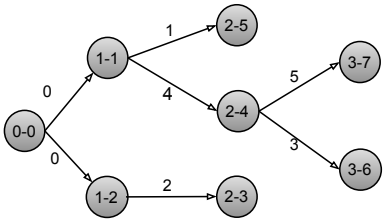
- $(v_1, \ldots, v_k)$ denotes the cluster composed of operators v_1 to v_k
- $(u_1, \ldots, u_k)(v_1, \ldots, v_l)$ is a partition composed of two clusters
- u_l denotes the optimal partition of the graph *rooted* at u assuming a capacity l (u_0 denotes the optimal partition of u subtree)
- $NP[u_l, v_k]$ denotes the partitions created by merging the two partitions u_l and v_k
- $value(u_l)$ indicates the value of the partition u_l

A. Tree-Optimal Clustering (TOC)

The first heuristic is inspired by the pseudopolynomial yet an optimal algorithm for clustering trees [17] with the objectives defined in Section II-C. Thus, before its actual clustering, the input DAG is transformed into a tree by removing some of its edges. The tree creation is made of 4 steps: i) In case the graph has multiple source operators, an extra dummy operator whose load is zero is created as the parent of the original sources. Edges created to connect the dummy operator to its children is also zero-weighted. ii) Each operator is associated with a level, which represents its distance to the root. More precisely, the original source nodes have a level of 1, the potential dummy operator has a level of 0. the children of sources have a level of 2, etc. iii)



(a) Initial merged DAG.



(b) Tree construction.

Figure 3: Transformation of a DAG into a tree.

If a node at level i has multiple parent nodes (*i.e.*, multiple neighbours of level $i-1$), only the edge to its parents with the highest load is kept in the graph, the others are removed. iv) Children of nodes are ordered. Figure 3 shows an example of a DAG (a) being transformed into a tree (b).

The clustering algorithm, described by Algorithm 1 is then applied. It is composed of four steps:

a - The partitions of every leaf operator i with load l_i (Lines 5 to 15) are generated: Trivially, $i_{l_i} = i_0 = (i)$. The optimal partition i_0 of i is composed of a single cluster composed of operator i.

b - A non leaf operator i whose children have been processed is chosen. The optimal partitioning of its subtree is created on its turn, following steps in Lines 16 to 30. In particular, Function $FindSubTreePartitions$ (described on Lines 31 to 45) generates all partitions of the sub-tree rooted at i. It creates all possible partitions $i'_j, \forall j \in [l_i, \ldots, C]$ by combining its current partitions with those of its children. For a given capacity j, to concatenate i_k with y_{j-k}, two operations can be used: either the respective clusters containing i and y are merged, or they are not, and the partitions are simply concatenated.

c - i_0, the optimal partition of the sub-tree rooted at i is chosen amongst $\{i_{l_i}, i_{l_{i+1}}, \ldots i_C\}$ (Line 22). If i is not the root of the tree, Step b is repeated.

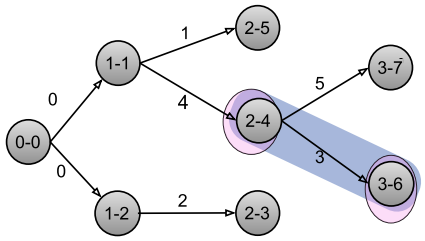
d - After having calculated the solutions of all operators, the result will be the union of the solutions of the operators in the first level (lines 24 to 26). Finally, once all partitions of subtrees rooted at operators of level 1 are found, if the DAG has multiple source operators, the dummy operator and its outgoing links are removed.

Algorithme 1 : Alg. Tree-Optimal Clustering

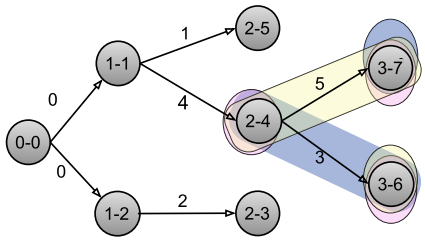
```
1  Main TOC(G, ML):
       Result : Set of clusters having the largest sum of values
       Input  : G // Graph
       Input  : ML // Maximum Level in the graph
       Input  : C // Maximum capacity per cluster
2      Initialization(G, ML);
3      FindFinalPartition(G, ML, C);
4  End TOC
5  Procedure Initialization(G, ML):
6      for (level = ML; level >= 1; level − −) do
7          foreach (i ∈ Graph.getOperatorInLevel(level)) do
8              if (level == ML) then
9                  i_{l_i} = i_0 = (i);
10             else
11                 i_{l_i} = (i);
12             end if
13         end foreach
14     end for
15 End Procedure
16 Function FindFinalPartition (G, ML, C):
       Output : Final clusters result
17     Result ← φ ;
18     for (level = ML − 1; level > 0; level − −) do
19         foreach (i ∈ G.getOperatorInLevel(level)) do
20             if i is not a leaf operator then
21                 ListP_{i,l_i,C} ←
                     FindSubTreePartitions(i, ML) ;
22                 i_0 ← HighestLoadInList(ListP_{i,l_i,C}) ;
23             end if
24             if level = 1 then
25                 Result ← Result ∪ i_0;
26             end if
27         end foreach
28     end for
29     return Result;
30 End Function
31 Function FindSubTreePartitions (i, C):
       Output : List of clusters of i depending on the load
       Input  : i // Operator
32     ListResult ← φ ;
33     foreach (y ∈ GetChildOf(i)) do
34         for (j = l_i; j ≤ C; j + +) do
35             for (a = l_i; a ≤ C; a + +) do
36                 Create the partition i'_j = NP(i_a, y_{j−a});
37             end for
38             if (value(i'_j) > value(i_j)) then
39                 i_j = i'_j;
40                 UpdateList(ListResult, i'_j);
41             end if
42         end for
43     end foreach
44     return ListResult ;
45 End Function
```

Note that the operators at a given level can be processed in parallel.

Let us illustrate the TOC algorithm by its application on the DAG in Figure 3 where the load of each operator and the weight of each edge are shown. We will assume a capacity $C = 4$. Figure 3-b shows the tree obtained from the graph. Note that each operator is labeled by its level and identifier. After the initialization step, Operator $2-4$, whose all children have been processed at initialization time can be processed. It first merges its own partitioning with its Child 3-6, as shown in Figure 4(a). Two partitions (in blue and pink, respectively) are created (see Step b of the algorithm). Then, Child 3-7 is also included, leading to a



(a) Operator 2-4: merging partitions of Child 3-6.



(b) Operator 2-4: merging partitions of Child 3-7.

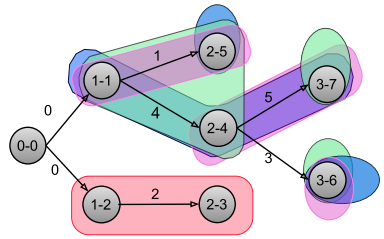
Figure 4: Partitioning for subtree rooted at Op. 2-4

set of 3 possible partitions: the pink and the blue one are extended with Child 3-7, and a new one, in yellow, is created by the merging of Op. $2-4$ with Op. 3-7 in a single cluster.

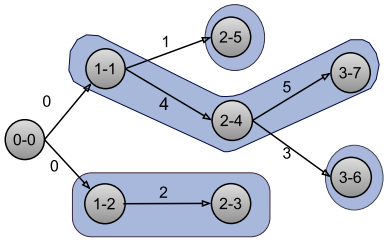
Once Op. 2-4 has been processed, Level 1 can be processed in its turn. This is illustrated in Figure 5. Let us focus on Op. 1-1. Each possible partition whose cumulated load does not exceed the capacity C are represented, each one with a different color in Figure 5(a): The best partition with a maximum load of 3 per cluster is shown in green, the best one with a maximum load of 4 per cluster is shown in blue. The final step of selecting the partition globally minimizing the cost is then performed over these partitions. It is given in blue on Figure 5(b). The final partition is composed of 4 clusters, and its value is 11.

Time Complexity of TOC: Let us consider the processing time to create the possible partitions of a subtree rooted at u with one of its children v. Without loss of generality, let us consider that u has already partitioned part of its subtree: Adding one child in the partitioning consists in trying all possible combinations of the current partitions of u with the set of possible partitions of the subtree of v. The number of the current partitions for u depends on $l(u)$: it already exists at most an optimal partition for each c such that $l(u) \leq c \leq C$. For each partition whose cluster size is limited by c, we can combine it with a partition of v of cost c' such that $c + c' \leq C$. So for $c = 1$, we can combine it with $c - 1$ partitions of v, for $c = 2$, we can combine it with $c - 2$ partitions of v, and so on. In the end the number of combinations to build is bounded by

$$\sum_{i=1}^{C} C - i = O(C^2).$$



(a) Possible partitioning at Level 1.



(b) Final partitioning.

Figure 5: Partitioning at Level 1 and final step.

As, globally for the tree, each child is processed once (when merging with its parent), the time complexity is in $O(n \times C^2)$.

B. Greedy Clustering (GC)

Similarly, as what is done with the TOC algorithm, a 0-loaded dummy operator connecting all the source nodes if there are multiple ones is first added to the DAG. Again, edges going out of the dummy operator have a weight of 0. Operators are again leveled from 0 to N and uniquely identified so they can be sorted.

Then, the GC clustering is composed of three steps, as detailed in Algorithm 2. Firstly, a partitioning is initialized for each operator, consisting of a single cluster composed of the operator itself (Lines 5 to 9). Secondly. the solutions are computed in Lines 10 to 20): The function $FindFinalSolution$ processes levels one by one and, for each non leaf operator i, calls the function $FindPartitions$ (in Line 14) which merges i_p with one of its chlidren's partition y_p to create the new current partition of i including the subtree rooted at y. Two cases are possible here:

- If the sum of the load of the cluster containing i and the load of the cluster containing y is less than C, $FindPartitions$ merges the clusters containing them into a single cluster and adds it to the new partition, the remaining clusters of i_p and y_p being also added in it.
- Otherwise, $FindPartitions$ simply concatenates the original clusters in i_p and y_p into the new one.

Doing so, because the graph is not a tree, some operators can have multiple parents, and so, appears multiple times in the partitioning of the subtree rooted at the common ancestor of these parents. The GC algorithm checks for each level, if such a redundancy exists. Function $Deduplicate$ (Line 17) removes such ocurrences: If an operator u which belongs to multiple clusters in a single partition, u is kept in the cluster which provides the highest value (Lines 33 to 43). The partition thus found at the dummy operator constitutes the final result.

Let us illustrate this simpler procedure on the DAG in Figure 3-a which was already used to exemplify the TOC algorithm. We again assume $C = 4$. Figure 6 shows the partitions created for operators at Level 2: Note that in contrast with the TOC algorithm keeps a single partition for each subtree, which simplifies a lot the partition constructions

Algorithme 2 : Algorithm of Greedy Clustering (GC)

```
1  Main Main(G, ML):
       Result : Set of clusters having the largest sum of values
       Input  : G // Graph
       Input  : ML // Maximum Level in the graph
       Input  : MC // Maximum capacity per cluster
2      Initialization(G, ML);
3      Result ← FindFinalSolution(G, ML, MC);
4  End Main
5  Procedure Initialization(G):
6      foreach (i ∈ G.getAllOperators()) do
7          i_p = (i);
8      end foreach
9  End Procedure
10 Function FindFinalSolution(G, ML, MC):
       Output : Final clusters result
11     for (level = ML − 1; level >= 0; level − −) do
12         foreach (i ∈ G.getOperatorInLevel(level)) do
13             if y ∈ GetSonOf(i) then
14                 P(i) ← FindPartitions(i, y, MC);
15             end if
16         end foreach
17         Deduplicate(level, G.getOperatorInLevel(level)));
18     end for
19     return DUM_p;
20 End Function
21 Function FindPartitions(i, y, MC):
       Output : List of clusters of i depending on the load
       Input  : i // Operator
       Input  : y // Child of the operator i
22     NewPartition ← φ ;
23     o_1 ← CheckClusterOf(i);
24     o_2 ← CheckClusterOf(y);
25     if InterCluster(o_1, o_2) > 0 and load(o_1 ∪ o_2) < MC then
26         NewPartition ← Merge o_1 and o_2;
27     else
28         NewPartition ← o_1 ∪ o_2
29     end if
30     NewPartition ← i_p\{o_1} ∪ y_p\{o2} ;
31     return NewPartition ;
32 End Function
33 Procedure Deduplicate (ListOpe):
       Output : Duplicated operators between clusters are deleted
       Input  : ListOpe // Partitions of operators
34     ListOfDupOp ← CheckDupOp(ListOpe) ;
35     foreach (k ∈ ListOfDupOp) do
36         higherValue ← GetHigClusterValue(k, ListOpe) ;
37         foreach j ∈ ListOfClusterWhereExist(k, ListOpe) do
38             if (higherValue > j.GetIntraCluster()) then
39                 j.delete(k);
40             end if
41         end foreach
42     end foreach
43 End Procedure
```

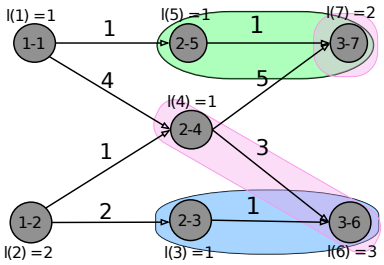


Figure 6: Level 2: Partition creations.

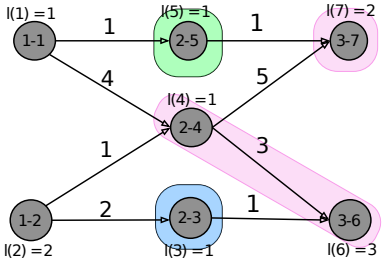


Figure 7: Level 2: Deduplication.

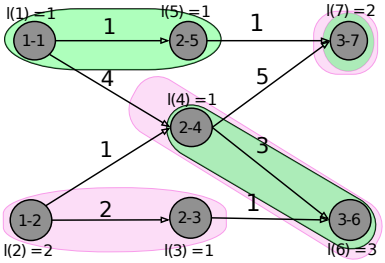


Figure 8: Level 1: Partition creations.

at each step. The partitions kept for the subtree rooted at $2-3$, $2-4$ and $2-5$ are shown in blue, pink and green, respectively. After deduplication of Level 2, the partition is shown on Figure 7.

The result, after processing Level 1 is shown in Figure 8. As specified above, for each subtree, the partition created is the one which is obtained by concatenating the clusters containing a node and the child currently processed if such a merge does not violate the capacity constraint. Deduplicating the graph in Figure 8 is shown in Figure 9. The concatenating of green and pink clusters constitutes the final partition, composed of 4 clusters and whose value is 6 (Remind that the TOC algorithm found a partition with a value of 11).

Time Complexity of heuristic 2: In contrast with TOC, GC process each child y in constant time when merging its partition with its parent i partition. It is simply a matter of merging (or not) the clusters containing y and the cluster containing i in their respective partitions, depending on the result of a test regarding their cumulated load. As each child is processed once by the set of its direct ancestors in the graph (their neighbours of the previous level), the

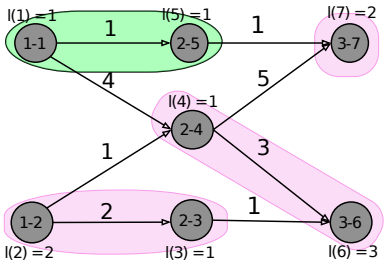


Figure 9: Level 1: Deduplication.

complexity is linear in the number of edges in the graph.

V. Evaluation

A. Simulation set-up

Simulation experiments were conducted to evaluate the general scheme and compare the heuristics devised. The three steps of the method, namely *merging*, *splitting* and *clustering* were implemented in a homebuilt simulator including approximately 10k lines of multi-threaded Java. Both clustering heuristics were implemented leveraging multi-threading: In both algorithms, the thread can launch an operator of the next level if: *(1)* there is no operator to be processed in the current level and *(2)* all children of the target operator have been processed. The merging step is multi-threaded similarly. The number of threads was set to 4.

The input of the simulator is composed of i) two graphs representing the currently running operators and an application newly submitted, respectively, and ii) the maximum capacity C. Both graphs are generated using two parameters, namely S, an approximate size for the graph, and the load sent to the source operators. This load is generated randomly between 1 and the maximum load.

The first graph is generated as follows: i) the number of levels is chosen between 0 and $\sqrt{S}$. The number of operators at each level is also chosen between 0 and $\sqrt{S}$. Edges are added backward: For each level l, each node is linked to a certain number of operators in levels 0 to $l-1$. It is ensured that every operator at level 0 has at least one outgoing link, and that every operator at the last level has at least one incoming link. The weight of edges is again chosen uniformly at random between 0 and a maximum weight. The load of non-source operators are generated so as not to exceed the sum of the loads of previous operators, each one being multiplied by the weight of its associated edges. For instance, the load of Op. $2-4$ in Figure 3 cannot exceed $6 (= 4 * 1 + 2 * 1)$.

The second graph is generated based on the first one so as to ensure a deterministic degree of similarity. For instance, if the similarity is required to be 50%, the second graph will include half of the operators of the first graph. A similarity of 100 % means that all operators from the

first graph are copied to the second graphs, yet the second graph can be larger than the first one. The common operators are systematically chosen at the beginning of the graph, so there is a common prefix between the two graphs. Once the selection of the common operators has been done, the second graph is completed using a similar approach as for the first graph.

We conducted experiments using 9 scenarios numbered from 1 to 9. The targeted graph size given as input for Experiment i is $(i+1)\times 10$. For instance, Scenario 4 created graphs with a target size of 50. The other parameters are as follows: the load for an operator is a positive integer $l \leq 20$. The weight for an edge is a positive float $w \leq 2$. The similarity between the first and the second graph is between 45 and 55%, which is arbitrary but is sufficiently high to be able to study the benefits of the merging. The nodes' capacity is set to 200, which allows to have a significant clustering in regard to the operators' load while having a significant number of clusters. These synthetic graphs were not necessarily created to reflect real maritime traffic surveillance workflows but target the validating the algorithms.

Each scenario is repeated 20 times, using the same initial graph, but 20 different secondary graphs to merge with the first. The numbers provided in the next section are averages computed based on the results of these 20 runs. These experiments were run over an Intel Core i7-8650U CPU having 8 cores and 32 Gb of RAM.

The following results give hints on i) the benefit brought about by the merging phase in terms of number of operators it removes, ii) the relative overhead induced by both heuristics, and iii) the actual minimization of the inter-cluster traffic.

B. Results

Figure 10 shows the benefits brought about by the merge and split steps. This benefit is expressed in terms of reduction of operators' load and reduction of data traffic. The merge and split steps reduce the total operators loads of the two submitted graphs from 24 to 30 % and the total of the data traffic from 18 to 27%, thus confirming the possibility to run more applications even if their cumulative needs in resources seem to exceed the available computing power and network bandwidth at first.

Figure 11 gives the execution time of the merge step, which consists on two substeps: the detecting of overlapping operators between the two graphs and the merging of these operators. In Scenarios 8 and 9, the number of operators increases significantly (more than 200 operators) and the duration of the merging increases similarly.

Figure 12 shows the costs of the clustering obtained during the last phase, *i.e.*, the amount of data exchanges actually leading to traffic between compute nodes. Remind that we want to minimize this cost.

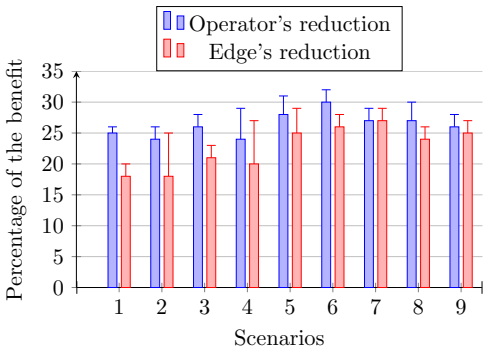


Figure 10: Benefit of the merging.

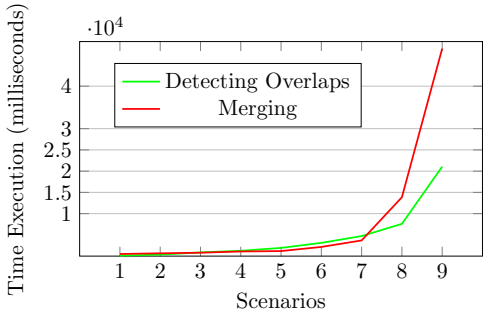


Figure 11: Duration of merging.

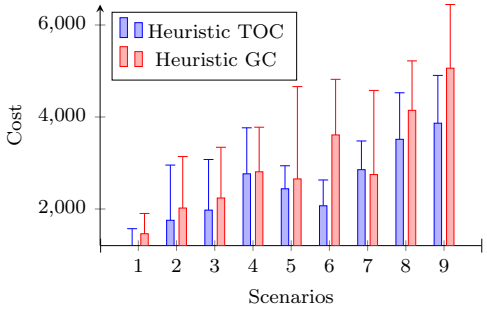


Figure 12: Inter-cluster throughput per Algorithm

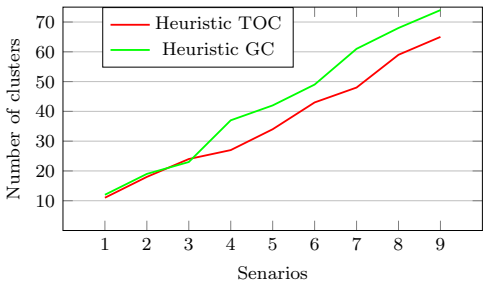


Figure 13: Result : Number Of Clusters per Algorithm

Figure 13 shows the average numbers of clusters calculated by the two algorithms. The TOS algorithm reduces considerably the number of clusters, which is inline with

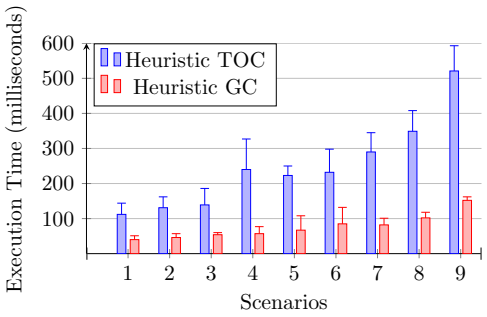


Figure 14: Duration of clustering.

the results on the cost and values. At this level, the TOS algorithm responds our requests concerning the saving of resources. Compared to GC, The TOC algorithm globally gives better results, it reduces the clustering cost by 5 to 60 % (in Scenario 6) and increases the intra-cluster throughput. TOC concomitantly saves computing resources: it reduces the number of clusters by 8 to 13% when the input graphs comprise each more than 50 operators.

Figure 14 shows the execution time of each heuristic in the 9 scenarios. Recall that the number of operators grows linearly with the id of the scenario. We can see that our implementation of the algorithms seem to reflect the complexities of the heuristics as given in Section III. Clearly, GC increases linearly and slowly while TOC shows a non-linear behavior. Thus, they offer a different compromise between the quality of the result and the completion time. TOS offers better results in most scenarios, despite its execution time. GC has a significantly better execution time yet an acceptable quality in its results in term of the resource usage. The choice to use one heuristic or the other depends on the use case: GC can clearly be could be useful for smaller volatile applications, and TOC can be used when the extra-benefit it brings can be useful on a long term basis.

VI. Related Work

When deploying SP applications over these platforms, a prominent problem is the placement problem, namely, how to map the graph of operators over the compute resources. This has been a recently addressed topic, over Clouds [8], hybrid Edge/Cloud platforms [6], [19], or without a particular platform in mind [3], [18]. The placement problem is related but orthogonal to the ability to preprocess the graph before its actual deployment in regard to what is already running on the platform.

A number of works dealt with preprocessing the SP graph. These static analyses aim to detect the potential parallelism of the graph [21] or the ordering semantics hidden in the user-defined operations so as to reorder the operators optimally [11].

The idea of sharing parts of the processing between graphs for reuse is not new [10]. Yet, to our knowledge, only few works put it into practice. Yet, Repantis *et al.* propose a framework where it is possible to find already running components that can be reused to build a new pipeline, based on a peer-to-peer network [20]. More recently, Chaturvedi et al formalized the problem of merging a set of graphs sharing a prefix of their operations [4]. The merging part of our algorithm is similar to their approach. Yet, they ignore the splitting and clustering phases.

Graph clustering (or partitioning) has been the subject of an important series of works over the years [2]. Let us first differentiate several flavors of the problem. Partitioning a graph into k balanced clusters has first been shown to be NP-complete [12]. Also, a problem closer to ours, namely partitioning a graph into balanced clusters (without knowing k beforehand) while trying to minimize the the inter-cluster cost (or *min-cut*) was also shown to be NP-complete [7].

The seminal work by Kernighan and Lin (KL) described the partitioning problem and showed its similarity to the min-cut problem [15]. They propose a heuristic, based on the incremental improvement of an initial arbitrary solution, for the bi-partitioning problem, running in $O(n^2 \log n)$. A number of improvements over the KL algorithm has been proposed, either in terms of complexity or in terms of precision regarding the balance between clusters. Fiduccia and Mattheyses proposed an algorithm running in $O(e)$ for the bi-paritioning problem [5]. Early extensions to the k-partitioning problem were based on recursively applying the KL/FM methods, adding a k factor to the complexity, and increasing the risk for imbalance between clusters.

Multilevel graph partitioning was then introduced as an attempt to reduce the potential imbalance between clusters [9]. In this approach, the graph is first iteratively coarsened up to a *small* graph on which the partitioning is applied. This partitioning is then *propagated* back to the original graph. This technique has been improved by Karypis and Kumar [14] who provided linear-time algorithm.

It is worth noting that most of GP problems are formulated so as to balance the load in clusters. In other words, that the cumulated weight of nodes in each cluster is approximately k. Our goal is slightly different: no cluster can have a weight higher than a given threshold. Following this formulation, the problem is similar to the knapsack problem with additional constraints related to the edges. The problem again appears to be NP-complete [13]. Yet, when the graph is a tree, optimal pseudo-polynomial algorithms exist, as the one which inspired our work, and proposed by Lukes [17].

VII. Conclusion

This paper targets stream processing platforms over which multiple applications are deployed dynamically. It details a general scheme to be applied each time an application is submitted, which optimizes the merging of the new application into the currently deployment. It consists in three steps: i) the application is merged with the currently running

applications, ii) the operator exceeding the capacity of one compute node are split into as many instances as needed, and iii) the resulting set of operators are clustered so as to ensure no node sees its capacity exceeded and the network traffic between nodes is globally minimized. For the last phase, two heuristics providing a different cost/efficiency trade-off were proposed. The first one relies on a pseudo-polynomial algorithm which is optimal in the case of the tree. The second one is a simpler heuristic whose time complexity grows linearly with the number of edges in the graph. The benefit and costs brought about by the different phases of the scheme has been exhibited through simulation.

This work opens different perspectives: i) considering heterogeneous, geographically distributed platforms, and ii) decentralizing the scheme for an improved scalability.

ACKNOWLEDGMENT

This project was partially funded by ANR grant ASTRID SESAME ANR-16-ASTR-0026-02.

REFERENCES

[1] European maritime safety agency - vessel tracking globally (lrit). http://www.emsa.europa.eu/lrit-home.html. Accessed December 2019.

[2] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016.

[3] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Perform. Eval. Rev.*, 44(4).

[4] S. Chaturvedi, S. Tyagi, and Y. Simmhan. Collaborative reuse of streaming dataflows in iot applications. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 403–412, Oct 2017.

[5] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Design Automation Conference*, pages 175–181, 1982.

[6] Rajrup Ghosh and Yogesh Simmhan. Distributed scheduling of event analytics across edge and cloud. *ACM Trans. Cyber-Phys. Syst.*, 2(4).

[7] Olivier Goldschmidt and Dorit S. Hochbaum. A Polynomial Algorithm for the k-Cut Problem for Fixed k. *Mathematics of Operations Research*, 19(1):24–37, 1994.

[8] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[9] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 28–28, Dec 1995.

[10] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[11] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. *Proc. VLDB Endow.*, 5(11):1256–1267, July 2012.

[12] Laurent Hyafil and Ronald L. Rivest. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical Report 33, IRIA.

[13] David S Johnson and KA Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8(1):1–14, 1983.

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

[15] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[16] Sukhamay Kundu and Jayadev Misra. A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1):151–154, 1977.

[17] Joseph A. Lukes. Efficient Algorithm for the Partitioning of Trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[18] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, April 2006.

[19] Laurent Prosperi, Alexandru Costan, Pedro Silva, and Gabriel Antoniu. Planner: Cost-efficient Execution Plans Placement for Uniform Stream Analytics on Edge and Cloud. In *WORKS 2018: 13th Workflows in Support of Large-Scale Science Workshop*, pages 1–10, Dallas, United States, November 2018.

[20] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-aware component composition for distributed stream processing systems. In Maarten van Steen and Michi Henning, editors, *Middleware 2006*, pages 322–341. Springer, 2006.

[21] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2012.