Fast Heuristics for Near-Optimal Task Allocation in Data Stream Processing over Clusters

Andreas Chatzistergiou
School of Informatics
University of Edinburgh, UK
a.chatzistergiou@sms.ed.ac.uk

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

We study provisioning and job reconfiguration techniques for adapting to execution environment changes when processing data streams on cluster-based deployments. By monitoring the performance of an executing job, we identify computation and communication bottlenecks. In such cases we reconfigure the job by reallocating its tasks to minimize the communication cost. Our work targets data-intensive applications where the inter-node transfer latency is significant. We aim to minimize the transfer latency while keeping the nodes below some computational load threshold. We propose a scalable centralized scheme that employs fast allocation heuristics. Our techniques are based on a general group-based job representation that is commonly found in many distributed data stream processing frameworks. Using this representation we devise linear-time task allocation algorithms that improve existing quadratic-time solutions in practical cases. We have implemented and evaluated our proposals using both synthetic and real-world scenarios. Our results show that our algorithms: (*a*) exhibit significant allocation throughput while producing near-optimal allocations, and (*b*) significantly improve existing task-level approaches.

1. INTRODUCTION

We address provisioning in contemporary data stream processing systems deployed over clusters. Real-time computing, commonly found in event detection and data stream processing applications, involves the processing of infinite streams of data under tight constraints, *e.g.*, deadlines from event appearance to system detection. In such scenarios, minimizing latency and keeping up with the arrival rate are key priorities. Examples include stock market analysis, click streams, emergency response, to name but a few. A salient problem is provisioning, *i.e.*, ensuring the application has enough resources to perform the necessary processing and meet its deadlines. We adopt a dynamic approach to the problem where we continuously monitor the executing jobs, detect abrupt changes in their profile and then dynamically reallocate their tasks to fit the new profile. This model addresses the volatility of the real-time processing environment. For instance, the arrival rate of the incoming events might fluctuate, or their statistical properties might change and increase processing and communication demands. Failing to deal with such cases may result in performance bottlenecks, and, at worst, data loss and a compromise of quality of service guarantees.

Our work is focused on data-intensive applications with significant inter-node transfer latencies. We aim to reduce the communication latency while keeping the nodes below a computational load threshold. Although previous work [11, 36, 26] suggests decentralized task allocation schemes for avoiding bottlenecks, we argue that a centralized approach can be efficient, as long as we design fast allocation algorithms. We thus devise on fast task allocation heuristics, while keeping the quality of the resulting allocation high. By reviewing state-of-the-art systems (*e.g.*, [16, 23, 25, 33]) we introduce an abstract job model, that allows us to minimize network transfer by minimizing the number of nodes necessary to run a job (Section 2). We introduce a natural grouping of the tasks that naturally arises due to operator parallelism in contemporary systems. By taking advantage of the group-oriented structure of the model we improve current fine-grained task-level approaches (Section 3) and devise a novel efficient top-down algorithm that reduces the complexity of the problem (Sections 4 and 5). Our algorithm not only minimizes the communication cost of a job, but also makes better use of the cluster's processing resources. We have implemented our proposals in Storm (Section 6), a distributed real-time computation system, and evaluated them using both micro-benchmarks and real-world scenarios (Section 7). Our results show that our monitoring system successfully detects changes and our algorithm significantly increases the throughput of the allocator while producing near-optimal allocations. In addition, our fast coarse-grained top-down approach has better optimality guarantees than slow fine-grained task-level approaches. Further, in real-world scenarios, our approach outperforms task-level approaches and the default configuration options of Storm. We finally present related work and conclude (Sections 8 and 9).

CIKM'14, November 3–7, 2014, Shanghai, China.

http://dx.doi.org/10.1145/2661829.2661882.

2. PROBLEM FORMULATION

The dataflow of most cluster-based data stream processing systems dictates a natural grouping of concurrently executing tasks. This constrains both how tasks are allocated to processing nodes and how the tasks communicate. By building on this observation our work departs from previous research in task allocation.

2.1 Stream jobs

A *stream job* is a set of m tasks, $T = \{t_1, t_2, \ldots, t_m\}$ represented as the directed acyclic graph $G(V, E)$. Each vertex $v_i \in V$ is a group of unique tasks such that $v_i \subseteq T$ and $\bigcap_{i=1}^{|V|} v_i = \emptyset$, *i.e.*, all groups are disjoint. Each $e_{i,j} \in E$ connects vertices v_i and v_j and is denoted as $e_{i,j} = v_i \times v_j$. That means that every $t_i \in v_i$ is connected with an edge to every $t_j \in v_j$. Thus, the dataflow of the job is organized

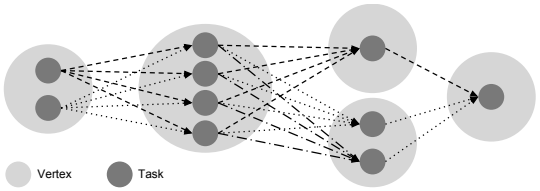


Figure 1: Stream job example

as a graph where every vertex is a group of tasks; between any two communicating groups every task of the first group communicates with every task of the second group. In Figure 1 we depict an example of a stream job. This representation is common in state-of-the-art systems (*e.g.*, [16, 23, 25, 33]). It is a natural abstraction as (*a*) computational units, represented by the vertices, process data and send the output to other computational units; and (*b*) each computational unit is parallelized a number of times to form a group of tasks where each task processes a subset of the input.

Each vertex v_i has a total *processing cost* $c^{pc}_{v_i}$. Similarly, each edge $e_{i,j}$ has a total *intermodule communication cost* $c^{imc}_{v_i,v_j}$. Let $t_i \in v_i$ and $t_j \in v_j$. We assume the following hold:

$$c^{pc}_{t_i} \simeq \frac{c^{pc}_{v_i}}{|v_i|}, \qquad c^{imc}_{t_i,t_j} \simeq \frac{c^{imc}_{v_i,v_j}}{|e_{i,j}|} \tag{1}$$

The above assumes that $c^{pc}_{v_i}$ and $c^{imc}_{v_i,v_j}$ are uniformly distributed between tasks of the same group. This implies (*a*) hardware homogeneity, and (*b*) a lack of significant skew as data is distributed between $t_i \in v_i$. In practical cases the uniform distribution assumption is common, but we discuss and lift both assumptions in Section 5.

Our model is widely applicable in state-of-the-art systems such as: (*a*) S4 [25], where the basic abstraction is the processing element (PE). Each PE processes data and produces output to be consumed by other PEs. Multiple linked PEs make up a job. Several PEs are logically grouped in processing nodes. (*b*) Storm [33], where a job, termed a topology, is a directed acyclic graph of processing units termed spouts and bolts. Spouts model stream sources, while bolts model the computational units. Both spouts and bolts can be parallelized an arbitrary number of times. (*c*) Kafka [23], where producers publish messages (topics) to brokers with messages further pulled by consumers. The system tries to evenly distribute messages to brokers and consumers for load balancing. Consumers are grouped in groups that jointly consume a set of topics.

2.2 Task allocation

Consider a set N of k fully interconnected nodes, $N = \{n_1, \ldots, n_k\}$. Assume that each n_i has a *maximum processing capability*, termed its *capacity* and denoted as $n_i.\mu$: the maximum computational load the node can handle. Our goal is to allocate tasks to nodes so that (*a*) the *total intermodule communication cost*, or IMC, is minimized; and (*b*) the *total processing cost*, or PC, of tasks per node does not exceed the node's capacity. Although our aim is to minimize communication overhead, this model allows us the flexibility to optimize for different objectives at different times. Consider, for instance, an idle cluster where the combined capacity of its nodes (*i.e.*, the sum of their individual capacities) is much higher than the computational requirements of the running jobs. Then, by setting $n_i.\mu$ to each node's maximum processing capability, we allocate the tasks to as few nodes as possible, thus minimizing IMC. Alternatively, if the aim is to load-balance the cluster, we can set each $n_i.\mu$ to the average processing cost per node $\alpha \frac{\sum_{t_m \in T} c^{pc}_{t_m}}{|N|}$, where $\alpha > 1$ is the tolerance level, which increases the average cost by a small percentage to make it easier to produce valid allocations.

Let $X = [x_{t_m,n_i}]$ be an assignment matrix such that:

$$x_{t_m,n_i} = \begin{cases} 1 & \text{if } t_m \text{ is assigned to node } n_i \\ 0 & \text{otherwise.} \end{cases}$$

We define the IMC *gain* as the total IMC that is saved by collocating communicating tasks. This is:

$$\phi(X) = \sum_{n_i \in N} \sum_{t_m \in T} \sum_{\substack{t_n \in T \\ n \neq m}} c^{imc}_{t_m,t_n} x_{t_m,n_i} x_{t_n,n_i}$$

Task allocation is the optimization problem of maximizing $\phi(X)$ under the constraint:

$$\forall n_i \in N, \sum_{t_m \in T} c^{pc}_{t_m} x_{t_m,n_i} \leq n_i.\mu$$

Task allocation is an NP-hard problem (see *e.g.*, [8]), so any practical algorithm must leverage heuristics. Algorithms must guarantee that all pairs will be allocated. But even when the total capacity of the cluster is higher than the total processing cost of the job, we may end up with tasks that cannot fit. This is due to task granularity: the coarser (*i.e.*, the heavier computationally) the tasks are, the higher the chances that we end up with nodes with spare capacity, but that cannot accommodate any of the remaining tasks. Then, we will need to backtrack and try a different allocation. We assume for now that the tasks are fine-grained enough for the algorithm to finish in a single pass; we revisit this in Section 5.

State migration. For simplification we assume task relocation does not involve state migration, *i.e.*, edges represent only data flow. This is a fair assumption in real-time systems. Before reconfiguration we can inject a brief "drain-out" period: the dataflow is blocked until all in-flight data is processed. This ensures that the tasks will not need to migrate any state and can be reinitialized. Live migration is an orthogonal problem left for future research.

Precedence relationships. The efficiency of an allocation algorithm is affected by any precedence relationships between tasks. Consider tasks $a \to b \to c \to d$ and processors p_1, p_2, where $a \to b$ means that the output of a is the input of b; thus a should precede b. If we assign a, b to p_1 and c, d to p_2, processing will be serialized as p_2 must wait for p_1 to finish before it starts. Precedence relationships in stream jobs are not adversely affected, as we deal with pipelined processing. The only latency is at the beginning of processing, until the first processed element reaches the last task of the pipeline. Thus, precedence relationships can be safely ignored.

Notation. We use $\langle \cdot, \cdot \rangle$ to denote a pair of tasks or groups. A pair of groups is an edge of the job graph; we use the pair notation to clarify the correspondence to task pairs. Each task t has two state variables: t.group and t.node. These are references to the task's host group and processing node respectively. The value for t.node is initialized to $\perp$, which denotes the null value. Task allocation assigns a non-null value to t.node for all tasks of the job. Finally, whenever we prioritize a list the result is a heap from which we can either pop elements, or look at the top element without popping.

3. BASIC APPROACHES

We now present the basic task-centric approaches for task allocation that we will subsequently refine.

3.1 A naïve algorithm

The simplest approach is to break the graph of the job into pairs of tasks, sort them by their IMC and process them in descending IMC order, by allocating pairs to nodes in descending processing capacity order of the nodes. We traverse the list of nodes and keep pushing the next most communication-intensive pair onto the cur-

rent node, reducing the node's capacity with each allocation. If the current node cannot host the current pair, we move on to the next node. If a pair cannot fit into any node, we break it into single tasks and allocate each task separately. When a pair is broken, each of its tasks is inserted into a list of unlinked tasks that are allocated last. If we come across a single task that has already been allocated, then we ignore it.[1] We iterate until we exhaust the list of tasks.

The algorithm allocates the most communication-intensive task pairs first while there is more spare capacity and it is more likely to collocate them. As tasks can appear in more than one pairs we can improve the algorithm by checking if one of a pair's tasks is already allocated. If so, we try to collocate the second task; if both tasks are already allocated we move on to the next pair.

Drawbacks. This algorithm is a direct adaptation of the dominant fusion-style heuristic algorithms of [17, 19, 28, 38]. These algorithms ignore the domain-specific group-wise communication pattern between the tasks. A fusion-style algorithm processes each task pair in isolation from its host group pair. This can lead to two inefficiencies. First, sorting by IMC can be misleading. In general, the task pairs of the same group pair will have (approximately) equal IMC. Thus, all pairs will be processed in sequence. A high IMC at the task level, however, does not necessarily imply a high IMC at the group level. A group pair with high IMC that is split in a large number of tasks will be processed after a group pair with a lower IMC and smaller number of tasks. Second, since the algorithm does not try to collocate all task pairs of a group pair, it is likely that tasks of the same group will be allocated to more than one nodes. Consider a group v_n with n tasks, paired with a group v_m with m tasks and a total group IMC of c. There are a total of nm links between v_n and v_m; each of the n tasks retains m links, each with a c/nm IMC. For each task of v_n that is allocated to a different node than the one v_m is allocated to we lose c/m worth of IMC; similarly, we lose c/n worth of IMC in the inverse direction. Allocating a task away from its group can have a significant adverse impact.

3.2 Improving the algorithm

We can improve the algorithm by first prioritizing task pairs not on task IMC but on a combined metric of IMC and PC that we term *relative significance*. Additionally, we leverage the topology of the job to maximize the collocation of task pairs of the same group pair. Given groups v_i and v_j, the relative significance of the pair is:

$$s_{v_i,v_j} = \frac{c^{imc}_{v_i,v_j}}{c^{pc}_{v_i} + c^{pc}_{v_j}} \quad (2)$$

which is essentially the IMC of the pair per PC unit. Group pairs with high combined PC are more likely to be spread across more nodes; thus, the significance of the pair should decrease proportionally to its PC. Since our focus is on devising fast heuristics for a scalable centralized allocator, this simple ratio is a fast and effective way to combine both metrics. A more precise technique would be to iteratively decompose groups at the capacity of the least loaded node and then re-prioritize on the IMC of the resulting groups. This, however, would increase the computational overhead as it would have to be repeated after each allocation. As we will see in Section 7.3, prioritizing task pairs by the relative significance of their host groups paves the way to near-optimality.

The improved algorithm, termed taskLevel(), is shown in Algorithm 1. It accepts as parameters a list of task pairs P and a list of nodes N. We prioritize the list of nodes on their capacity and the list of task pairs on their relative significance. We then iterate over task pairs and allocate task pairs to nodes in decreasing node capacity order. An exception is a pair that has one of its tasks already allocated (recall that this is possible when a task appears in more than one pairs) where we select the node of the allocated task. We use two variables v_i and v_j to track group pairs as we traverse the list of task pairs. These variables are parameters to function pickNode() in line 7, which selects a node from N to allocate the pair of tasks to. Node selection is shown in Algorithm 2. It accepts as parameters the task pair, the last group pair tasks of which were allocated, the last node that we allocated tasks to, and the prioritized list of nodes. The algorithm first checks if any task of the pair has been allocated, in which case it returns the node the task has been allocated to. Otherwise, it checks whether we have switched pairs by comparing the last group pair with the group pair of the current task pair. If we have switched group pairs this is a new allocation so we return the node with the highest available capacity. If we are in the same pair, then we only switch to the node with the next highest available capacity if the task pair cannot fit in the last node used. In any other case, we return the latter. Once a node is selected, we check if it can host the current task pair. If so, we allocate the task pair and update the priority of the node (lines 9 to 13). If we cannot make the allocation, we break the pair into single tasks and push the unallocated tasks into a list R prioritized on c^{pc}_t.

When we come across a pair with both its tasks already allocated, we try to collocate the tasks (lines 17 to 23). We find the node with the highest initial capacity, $n_i.\mu^{init}$, and check if at its current capacity it can accommodate the other node's tasks (variable $n_m.\mu^f$). If so, we move the tasks and update the capacities accordingly.

After exhausting the list of task pairs, we allocate single tasks. In function allocateSingleTasks() of Algorithm 3 we prioritize the list of tasks on to their processing cost, iterate over the list, pick the least loaded node, allocate the task to it, and update the node ca-

Algorithm 1: taskLevel(P,N)

input : list of task pairs P, list of nodes N
output: success if all tasks in P were allocated, failure otherwise

```
1  prioritize N on n_i.μ for all n_i ∈ N;
2  prioritize P on s_{t_m,t_n} for all ⟨t_m,t_n⟩ ∈ P;
3  R ← []; v_i ← null; v_j ← null; n_i ← null;
4  while P not empty do
5      t_m,t_n ← P.pop();
6      if t_m.node = ⊥ or t_n.node = ⊥ then
7          n_i ← pickNode(⟨t_m,t_n⟩,⟨v_i,v_j⟩,N);
8          ⟨v_i,v_j⟩ ← ⟨t_m.group,t_n.group⟩;
9          c_m ← c^pc_{t_m}; c_n ← c^pc_{t_n};
10         if t_m.node ≠ ⊥ then c_m = 0;
11         if t_n.node ≠ ⊥ then c_n = 0;
12         if c_m + c_n ≤ n_i.μ then
13             t_m.node ← n_i; t_n.node ← n_i; n_i.μ ← n_i.μ − c_m − c_n
14         else
15             if t_m.node = ⊥ then R.push(t_m);
16             if t_n.node = ⊥ then R.push(t_n);
17     else
18         n_n ← t_n.node; n_m ← t_m.node;
19         if n_m.μ^init ≥ n_n.μ^init then swap(n_n,n_m);
20         n_m.μ^f ← n_m.μ^init − n_m.μ;
21         if n_m.μ^f ≤ n_n.μ then
22             n_m.μ ← n_m.μ^init; n_n.μ ← n_n.μ − n_m.μ^f;
23             ∀t_i ∈ T, if t_i.node = n_m then t_i.node = n_n;
24 return allocateSingleTasks(R,N);
```

[1]This happens for tasks that are shared between pairs. If we break the first pair the task can still be allocated by a subsequent pair. Thus, the remaining task that was appended to the list will have been allocated when we reach it.

Algorithm 2: pickNode($\langle t_m, t_n \rangle, \langle v_i, v_j \rangle, n_i, N$)

input : task pair to allocate $\langle t_m, t_n \rangle$, last group pair that had tasks allocated $\langle v_i, v_j \rangle$, last node to receive tasks n_i, prioritized list of nodes N
output: node to attempt task allocation to

1 **if** $t_m.node \neq \perp$ **then return** $t_m.node$;
2 **if** $t_n.node \neq \perp$ **then return** $t_n.node$;
3 **if** $\langle v_i, v_j \rangle \neq \langle t_m.group, t_n.group \rangle$ ***or*** $c^{pc}_{t_m} + c^{pc}_{t_n} > n_i.\mu$ **then return** $N.top()$;
4 **return** n_i;

Algorithm 3: allocateSingleTasks(R, N)

input : list of single tasks R, prioritized list of nodes N
output: success if all tasks in R were allocated, failure otherwise

1 prioritize R on $c^{pc}_{t_m}$ for all $t_m \in R$;
2 **while** R *not empty* ***and*** $c^{pc}_{R.top()} \leq n_{N.top()}.\mu$ **do**
3 $\quad t_m \leftarrow R$.pop(); $n_i \leftarrow N$.top();
4 $\quad t_m$.node $\leftarrow n_i$; $n_i.\mu \leftarrow n_i.\mu - c^{pc}_{t_m}$;
5 **if** R *is empty* **then return** *success* **else return** *failure*;

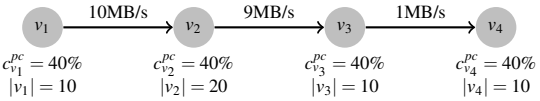


Figure 2: Running example

pacity. This contrasts the naïve approach where we keep allocating tasks to a node until it is full before moving to the next node.

Example 1: Consider the graph of Figure 2 with four task groups of equal processing costs equal to 40% of a node's capacity. The first two pairs have approximately equal communication costs and the third one has a much lower one. The second group has twenty tasks; all other groups have ten tasks. Assume that the spare capacity of each node is also 40%. By prioritizing on relative significance, the taskLevel() algorithm begins with the task pairs of $\langle v_1, v_2 \rangle$ and allocates them one by one. There is no ordering within task pairs, but there is substantial sharing between pairs: even though a task from v_1 appears in twenty pairs it will only be allocated once. Consider the case where the allocation order results in allocating five v_1 tasks and ten v_2 tasks to each of two nodes, filling them to capacity. Next, the tasks of the $\langle v_2, v_3 \rangle$ pair are considered. However, the tasks of group v_2 are allocated to full nodes and thus all task pairs of $\langle v_2, v_3 \rangle$ are split, and the tasks of group v_3 are added to the R list. This leaves the tasks of the $\langle v_3, v_4 \rangle$ pair to be allocated in a similar way to two more nodes, emptying the R list in the process. The resulting allocation will collocate tasks of $\langle v_1, v_2 \rangle$ and $\langle v_3, v_4 \rangle$ without any collocation of tasks from $\langle v_2, v_3 \rangle$. This means a high IMC penalty, even though groups were prioritized on IMC. ■

Complexity. While the running time of the algorithm depends on the properties of the job graph, we can compute the worst case running time. At worst we have $(|T|/2)^2$ task pairs, if there are two groups each containing $|T|/2$ tasks. The task pairs are prioritized on their relative significance in $O((|T|/2)^2)$ time. The nodes are prioritized in $O(|N|)$ time. For each task pair, we must pick the node with the highest available capacity or to probe for a specific node (both are constant-time operations) and delete the pair from the heap in $\log |T|/2^2$ time. After an allocation we update the capacity of the node in $O(\log |N|)$ time. This adds to $O((|T|/2)^2 + |N| + (|T|/2)^2 \log |N| \log |T|/2^2)$ time. In the common case that $|N| \leq (|T|/2)^2$ this is $O(|T|^2 \log |T|^2)$ time. Finally, we allocate the single tasks in $R \subseteq T$. We prioritize R on processing cost in $O(|R|)$ and for each task we pick the node with the currently maximum capacity, allocate the task to it, delete the task in $O(\log |R|)$ time, and update the prioritized list of nodes. Thus, this computation is $O(|R| \log |N| \log |R|)$. In the worst case all task pairs have been split, so $|R| = |T|$; the entire algorithm is still $O(|T|^2 \log |T|^2)$.

4. GROUP-AWARE LINEAR-TIME ALGORITHM

Our novel linear-time algorithm works top-down and balances splitting with fusion by operating at a coarser granularity. Instead of fusing task pairs for collocation, we start from groups, fuse group pairs if possible, and only split a group if absolutely necessary. We split groups in an informed way to maximize collocation.

The algorithm, termed topDown(), is shown in Algorithm 4. It accepts as parameters a list U of group pairs[2] and a list of nodes N. We prioritize N on node capacity and U on relative significance, as given by Eq. 2. We then pop the next element from U and call function resolveSplits() on the resulting pair. This function transforms a group pair by either annotating each group with its sequence of splits and node allocations (if a group has been fully allocated) or filtering out its allocated portion (if it is only partially allocated). We need this functionality as we will be splitting groups as we go along. This may mean that parts of one or both groups will already be allocated when we come across them. We will revisit this function shortly, but for now let us assume the above semantics. The result of resolveSplits() is a pair $\langle u_i, u_j \rangle$ where u_i and u_j are either the subsets of v_i and v_j, respectively, that are not allocated, or the original groups with the sequence of splits attached.

We first consider the cases where not both groups are allocated: either one of them is, or neither is. In the first case, detected in line 9, we identify which group is allocated and which one is not. We set u_i to the allocated group and u_j to the unallocated group. We then collocate as many tasks as possible from u_j to nodes hosting tasks from u_i. We retrieve the list of nodes hosting u_i tasks and prioritize them on expected IMC gain (line 14). We next set a mode variable to second indicating that only the second group will need to be split, as the first is fully allocated. If neither group is fully allocated, we set the prioritized list of nodes to the full list of nodes. We set the mode variable to both so that both groups will be processed (line 17). We allocate the tasks of each group in line 18 (a process detailed below). If the pair is already fully allocated to two different nodes, *i.e.*, all tasks of u_i are allocated to node n_i and all tasks of u_j are allocated to n_j we attempt to fuse the two nodes. Shown in lines 19 to 24, this is performed similarly to the taskLevel() algorithm. If fusion is not possible, we leave the groups as they are. We iterate until we exhaust the list of group pairs, at which point we will be left with a list of single tasks; so we call allocateSingleTasks() of Algorithm 3 and return.

Group pair allocation. The algorithm, allocateGroupPair in Algorithm 5, first tries to fuse a pair and, if that is not possible, it splits it into multiple pairs and recursively processes them. The input to the algorithm is the group pair to allocate, the prioritized list of candidate nodes, the mode of operation (whether we should split only the second group, or both) and a list of single tasks that may be extended by the algorithm. We pop the first element from the prioritized list of nodes, n_ℓ, and check if it can fit the entire pair. If so, we collocate all tasks of the pair and decrease the capacity of the node accordingly. If collocation is not possible, then we must split the pair. When splitting $\langle v_i, v_j \rangle$, we aim to create two new pairs $\langle u_i, u_j \rangle$ and $\langle w_i, w_j \rangle$ so that the IMC is minimized. We split pairs in constant time through Eq. 1. There are two cases to consider: either one of the groups has been allocated, or neither group has.

[2]This is the set of edges represented as a list of group pairs as the pairs may be modified and this should not affect the job graph.

Algorithm 4: topDown(U,N)

input : list of group pairs U, list of nodes N
output: success if all tasks of all group pairs in U were allocated, failure otherwise

1 prioritize N on $n_i.\mu$ for all $n_i \in N$;
2 prioritize U on s_{v_i,v_j} for all $\langle v_i, v_j \rangle$;
3 $R = []$;
4 **while** *U not empty* **do**
5 $\quad \langle v_i, v_j \rangle \leftarrow V$.pop();
6 $\quad \langle u_i, u_j \rangle \leftarrow$ resolveSplits$(\langle v_i, v_j \rangle)$;
7 $\quad$ **if** $\forall t_i \in u_i, t_i.node = \bot$ ***or*** $\forall t_j \in u_j, t_j.node = \bot$ **then**
8 $\quad\quad N_u \leftarrow \bot$; mode $\leftarrow \bot$;
9 $\quad\quad$ **if** $(\forall t_i \in u_i, t_i.node = n_i$ ***and*** $\forall t_j \in u_j, t_j.node = \bot)$ ***or***
10 $\quad\quad\quad (\forall t_i \in u_i, t_i.node = \bot$ ***and*** $\forall t_j \in u_j, t_j.node = n_j)$ **then**
11 $\quad\quad\quad$ **if** $(\forall t_i \in u_i, t_i.node = \bot$ ***and*** $\forall t_j \in u_j, t_j.node = n_j)$ **then**
12 $\quad\quad\quad\quad$ swap(u_i, u_j);
13 $\quad\quad\quad N_u \leftarrow \{t_i$.node $: t_i \in u_i\}$;
14 $\quad\quad\quad$ prioritize N_u on $\left(c^{imc}_{t_i,t_j} |u_i||u_j|\right)_{t_i \in u_i, t_j \in u_j}$ for all $n \in N_u$;
15 $\quad\quad\quad$ mode $\leftarrow$ second;
16 $\quad\quad$ **else**
17 $\quad\quad\quad N_u = N$; mode $\leftarrow$ both;
18 $\quad\quad$ allocateGroupPair$(\langle u_i, u_j \rangle, N_u$, mode$, R)$;
19 $\quad$ **else if** $\forall t_i \in u_i, t_i.node = n_i$ ***and*** $\forall t_j \in u_j, t_j.node = n_j$ **then**
20 $\quad\quad$ **if** $n_j.\mu^{init} \geq n_i.\mu^{init}$ **then** swap(n_i, n_j);
21 $\quad\quad n_j.\mu^f \leftarrow n_j.\mu^{init} - n_j.\mu$;
22 $\quad\quad$ **if** $n_j.\mu^f \leq n_i.\mu$ **then**
23 $\quad\quad\quad n_j.\mu \leftarrow n_j.\mu^{init}$; $n_i.\mu \leftarrow n_i.\mu - n_i.\mu^f$;
24 $\quad\quad\quad \forall t_j \in u_j, t_j.$node $= n_i$
25 $\quad$ **return** *allocateSingleTasks*(R, N);

(a) Splitting one group: Given $\langle v_i, v_j \rangle$ we split v_j to u_j, w_j (lines 8 to 11). We aim to allocate as many tasks of v_j to n_i as we can: for each collocated task we gain $c^{imc}_{v_i,v_j}/|v_i|$ in IMC. The number of tasks we can fit in n_i is $k = \lfloor \mu_i / c^{pc}_{t_j} \rfloor$ where $c^{pc}_{t_j}$ is the average processing cost of Eq. 1. We can slice v_j at index k (a constant time operation) to obtain u_j and w_j. The tasks of u_j will be collocated with the tasks of v_i; the tasks of w_j will be processed in the final step.

(b) Splitting both groups: Given $\langle v_i, v_j \rangle$, no task of either group has been allocated; let us assume that $|v_i| < |v_j|$. We split $\langle v_i, v_j \rangle$ into $\langle u_i, u_j \rangle$ and $\langle w_i, w_j \rangle$, collocate the tasks of $\langle u_i, u_j \rangle$ in n_ℓ, and recursively process $\langle w_i, w_j \rangle$. The issue is devising a good splitting strategy. As the communicating tasks are in $v_i \times v_j$, there is no point in collocating tasks of only one group: the IMC is only reduced when we collocate at least one $t_i \in v_i$ with at least one $t_j \in v_j$. Thus, the maximum number of splits per group pair is equal to $|v_i|$. The best we can do for each split is to maximize $|u_i| \times |u_j|$. This can be achieved by collocating as many pairs of tasks as possible, and then filling the spare space with tasks from either v_i or v_j. However, if $|v_i| \ll |v_j|$ this may hinder the allocation of the next pair to be considered. Thus, we split proportionally based on $\lfloor |v_j| / |v_i| \rfloor$. Let the *minimum balanced split processing cost* σ be:

$$\sigma = c^{pc}_{v_i} + c^{pc}_{v_j} \lfloor |v_j| / |v_i| \rfloor$$

The number of minimum balanced splits that can fit in n_ℓ is $k = \lfloor n_\ell.\mu / \sigma \rfloor$. We slice v_i and v_j at index k to create u_i, u_j, w_i, and w_j. After $\langle v_i, v_j \rangle$ has been split into $\langle u_i, u_j \rangle$ and $\langle w_i, w_j \rangle$ we collocate the tasks of u_i and u_j (lines 12 to 17).

Finally, we recursively process w_i and w_j (line 20). This process will terminate if: (*a*) the last split pair fits in the least loaded node; or (*b*) the smaller group is exhausted first and we are left with a number of tasks that will be processed in isolation; or (*c*) we are done processing at the same time (*i.e.*, $|w_i| = |w_j| = 0$), thus no recursive call is made. When only one of the two groups is empty the other group is pushed to R for later processing.

Algorithm 5: allocateGroupPair$(\langle v_i, v_j \rangle, N$, mode$, R)$

input : group pair to allocate $\langle v_i, v_j \rangle$, prioritized list of nodes N, mode of possible splitting (second or both), list of single tasks R
output: allocations for tasks in v_i, v_j, insertion into R of remaining single tasks that could not be collocated

1 $n_\ell = N.top()$;
2 $c_i \leftarrow c^{pc}_{v_i}$; $c_j \leftarrow c^{pc}_{v_j}$;
3 **if** *mode* $=$ *second* **then** $c_i = 0$;
4 **if** $c_i + c_j \leq n_\ell.\mu$ **then**
5 $\quad$ **if** *mode* $=$ *both* **then** $\forall t_i \in v_i, t_i$.node $\leftarrow n_\ell$;
6 $\quad \forall t_j \in v_j, t_j$.node $\leftarrow n_\ell$; $n_\ell.\mu \leftarrow n_\ell.\mu - c_i - c_j$;
7 **else**
8 $\quad$ **if** *mode* $=$ *second* **then**
9 $\quad\quad c^{pc}_{t_j} \leftarrow c^{pc}_{v_j}/|v_j|$; $k \leftarrow \lfloor n_\ell.\mu / c^{pc}_{t_j} \rfloor$;
10 $\quad\quad u_j, w_j \leftarrow$ slice(v_j, k); $\langle u_i, w_i \rangle \leftarrow \langle v_i, \emptyset \rangle$;
11 $\quad\quad \forall t_j \in u_j, t_j$.node $\leftarrow n_\ell$; $n_\ell.\mu \leftarrow n_\ell.\mu - c^{pc}_{u_j}$;
12 $\quad$ **else if** *mode* $=$ *both* **then**
13 $\quad\quad$ **if** $|v_i| > |v_j|$ **then** swap(v_i, v_j);
14 $\quad\quad \sigma = c^{pc}_{v_i} + c^{pc}_{v_j} \lfloor |v_j| / |v_i| \rfloor$; $k = \lfloor n_\ell.\mu / \sigma \rfloor$;
15 $\quad\quad u_i, w_i \leftarrow$ slice(v_i, k); $u_j, w_j \leftarrow$ slice(v_j, k);
16 $\quad\quad \forall t_i \in u_i, t_i$.node $\leftarrow n_\ell$; $\forall t_j \in u_j, t_j$.node $\leftarrow n_\ell$;
17 $\quad\quad n_\ell.\mu \leftarrow n_\ell.\mu - c^{pc}_{u_i} - c^{pc}_{u_j}$;
18 $\quad$ **if** $|w_i| > 0$ ***and*** $|w_j| = 0$ **then** $\forall t_i \in w_i, R$.push(t_i);
19 $\quad$ **else if** $|w_i| = 0$ ***and*** $|w_j| > 0$ **then** $\forall t_j \in w_j, R$.push(t_j);
20 $\quad$ **else if** $|w_i| + |w_j| > 0$ **then** allocateGroupPair$(\langle w_i, w_j \rangle, N$, mode$, R)$;

Resolving splits. We now revisit function resolveSplits() at line 6 of Algorithm 4. Splitting a group pair can complicate the processing of subsequent pairs as it implies updating the relevant edges to refer to the resulting groups. This may be expensive: for each split we must scan the list of groups and update the relevant pairs. However, we can apply a lazy approach: leave the group pairs intact in the prioritized list and only perform the necessary processing when the next pair to consider contains split groups. Thus, we only mark a split group as such whenever we split it, along with the sequence of splits and node allocations. These are appends to a group-specific list and thereby constant-time operations. We rely on resolveSplits() to annotate each group of a pair with its sequence of splits. There are two cases to consider. The first is when the group is partially allocated. Then, we take the non-allocated tasks and process these; the rest of the tasks have already been processed. The second case is when a group is fully allocated to more than one nodes. Then, we return the sequence of splits, which will be prioritized on expected IMC gain (line 14 of Algorithm 4).

Complexity and remarks. Operating on groups and not on tasks significantly reduces the complexity. At the task level the number of edges is quadratic to the number of tasks; the common case at the group level is $k|V|$ edges for some constant k. The worst case is still quadratic, since there are at most $n(n-1)/2$ edges in a DAG of n nodes, though this is a degenerate case and is rarely found in practice. Note that taskLevel() has quadratic complexity even when $|V| \ll |T|$, if each group has $|T|/|V|$ tasks. Effectively, topDown() factors out the quadratic complexity due to the communication pattern between tasks. In the worst case $|V| = |T|$ and $|E| \simeq |V|^2$, *i.e.*, each group has a single task and the number of edges is quadratic. Apart from that pathological case, we need $O(|V|)$ time to prioritize group pairs and $O(|N|)$ time to prioritize nodes. Then, $O(|V| \log |V|)$ time is needed to perform the allocation for a total of $O(|V| + |V| \log |V| + |N|)$. In the typical case where $|N| \leq |T|$ and $|V| \leq |T|$ the total cost is $O(|V| \log |V|)$. The last step, as in Section 3.2, is the allocation of single tasks. This is $O(|R| \log |R|)$; at worst, $|R| = |T|$ for a complexity of $O(|T| \log |T|)$.

Example 2: In the example of Figure 2, topDown() allocates symmetrically the task pairs of each group pair. The algorithm allo-

cates five tasks from v_1 along with ten tasks from v_2 to each of the two nodes. This leads to 50 collocated links per node for a total of 100 (*i.e.*, $2\left(|v_1|/2 \cdot |v_2|/2\right)$). The taskLevel() algorithm allocates seven tasks from v_1 and six tasks from v_2 to the first node and the rest (three tasks from v_1 and fourteen from v_2) to the second node. This results in 84 collocated links, 16 fewer than topDown(). ■

The main advantage of the top-down approach is its linear-time complexity compared to the quadratic approach of Section 3.2. Operating top-down allows us to consider more factors when allocating tasks. After having considered a few pairs, the algorithm forms as many fused groups as the number of nodes. The result is that the group pairs with the highest IMC form a "center of gravity," so to speak, for the remaining groups. The latter are usually fused with the groups of an already occupied node. As allocation proceeds, the splitting algorithm is called more often: the capacity of the nodes shrinks, so chances are that they will not fit entire group pairs, thereby triggering group splitting. Note that the algorithm is essentially greedy so it may fall into the trap of local minima.

5. RELAXING ASSUMPTIONS

We have so far argued that task granularity affects the quality of an allocation. Even if the processing needs of a job are less than the processing capacity of the cluster, a valid allocation may not be possible for coarse-grained tasks. In the systems we model it is possible to tune task granularity. A way to ensure allocation, if failure to allocate is initially detected, is the following. We find the greatest common divisor d between the processing costs of the groups and the capacity of the nodes. Then, we set the number of tasks of each group to $c_{v_i}^{pc}/d$. This will result in equally-sized tasks in all groups that will exactly fit. Thus, it will result in successful allocations so long as the capacity of the cluster can accommodate the job. In the worst case $d = 1$, which is the finest-grained worst-case: each group is broken into tasks with 1% processing cost each.

Another assumption is hardware homogeneity. We can relax this assumption by scaling the capacity of a node to reflect its hardware capability. We add a preprocessing step that adjusts the capacity of the nodes by normalizing with respect to the least capable node of the system. For instance, a node with a quad-core CPU can have four times the capacity of a normal CPU. Though these are approximations, they simplify allocation and introduce flexibility.

Finally, we tackle the uniform distribution assumption of Eq. 1. This can be resolved with a preprocessing stage that in a linear scan divides groups of tasks into subgroups of tasks with approximately the same costs (within a margin). Then, we can run our allocation algorithms on the new groups. This increases the number of groups but guarantees the uniform distribution assumption within these groups. This is effective against the case of only a skewed distribution around a few key values as it will result in a small number of new groups. In the degenerate case this may lead to the number of groups being equal to the number of tasks.

6. IMPLEMENTATION

We have implemented our algorithms on Storm [33]: a data stream processing framework for clusters. Like MapReduce [16], the Storm runtime automatically parallelizes user programs and executes them on the cluster in a fault tolerant way.

Storm. A Storm job, termed a *topology*, is a *directed acyclic graph* (DAG). Vertices represent processing units and edges the data that flows between them. There are two types of processing units: *spouts* and *bolts*. Spouts model stream sources, while bolts are the computational units. Each processing unit is parallelized a number of times that is provided by the developer through an abstraction

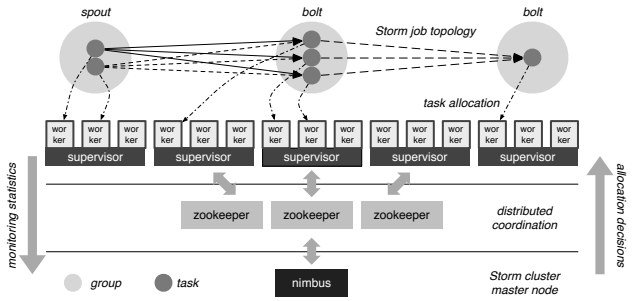


Figure 3: Storm architecture and example topology

called an executor. Thus, every vertex is executed as a group of tasks communicating with *all* tasks of a connected vertex. An executor may contain more than one threads that can be aggregated at the executor level to map to our model's tasks. For the rest of the paper the term tasks refers to Storm's executors. There is also a separation between logical nodes and physical nodes: the tasks of a group may be executed on any number of nodes of the cluster. The task group of a source vertex forwards data to the task group of a target vertex through partitioning strategies called *groupings*. The dominant groupings are shuffling, replication, key partitioning or direct grouping (the producer decides which task to forward data to). The former two result in uniform distributions while the latter two are subject to our relaxation techniques of Section 5.

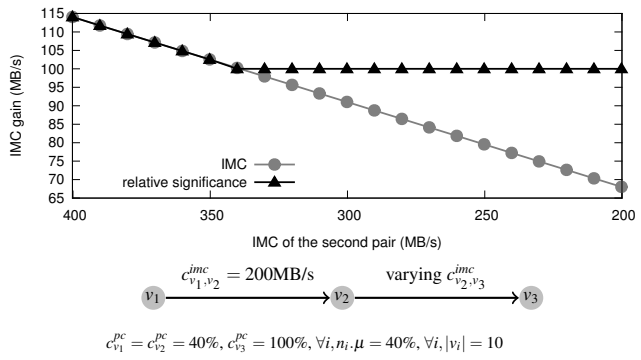
A Storm cluster has one master node, called the *nimbus* node, a number of *zookeeper* nodes, and a large number of worker nodes. Job topologies are submitted to the nimbus, which allocates tasks to the workers and monitors the cluster for failures. Distributed coordination, *i.e.*, maintenance of cluster state, is provided by the zookeeper nodes [39] and does not affect task-specific data exchange. Worker nodes execute the tasks of the job. A supervisor daemon at each execution node checks for tasks assigned to that node by probing the zookeeper cluster. If there are such tasks it launches them as worker processes. The same physical node may be running multiple tasks; all workers of the same physical node report to the same supervisor at the physical level. The architecture of Storm, and an example topology are shown in Figure 3.

Monitoring. We have enhanced Storm daemons with a monitoring subsystem. We extended the worker daemon with a monitoring thread that measures the processing costs of all worker threads along with the number of bytes sent to other workers. Measurements are taken periodically, timestamped, and written to persistent storage on the node, to be picked up by the supervisor. Supervisors aggregate statistics, tag them with the supervisor identifier, and send them to the zookeeper cluster. The nimbus node checks for supervisor updates and, for each update, it globally aggregates statistics and, if there is a large difference between successive measurements, runs the task allocation algorithm to reconfigure the job.

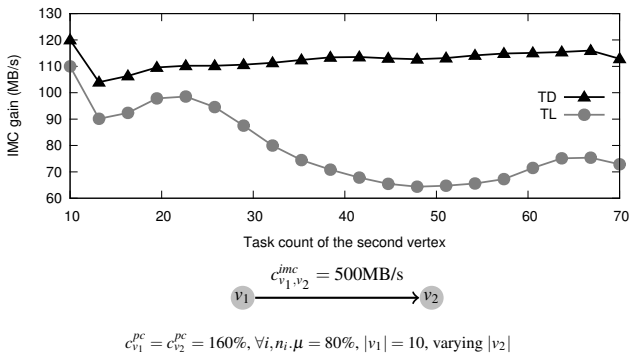
Reconfiguration. To reconfigure the cluster we leverage the Storm infrastructure, which supports suspending and resuming jobs. This is done by blocking the spouts of the job, thus preventing new stream elements from being propagated to the bolts and forwarded through the topology. This is followed by a "drain out" phase where in-flight data is propagated through the bolts until all communication queues between bolts are empty. Our task allocation algorithms then reconfigure the cluster by *explicitly* assigning tasks to worker nodes. Processing is resumed by unblocking the spouts.

7. EXPERIMENTAL EVALUATION

We implemented our algorithms on Storm and experimented using single-node micro-benchmarks and real workloads on a cluster



(a) Impact of prioritization



(b) Impact of task count

Figure 4: Results in micro-benchmarks

deployment. The micro-benchmarks were evaluated on an Intel Core i3-2100 CPU at 3.10GHz, 4GB of physical memory, running 32-bit Ubuntu 12.04. For the real workload experiments we used an Amazon EC2 homogeneous cluster of 102 Amazon M1 Small Instance nodes. We used one node for the nimbus, one zookeeper node, and 100 workers. At the time of writing, each Amazon instance consisted of one EC2 compute unit with 1.7GB of physical memory running 32-bit Ubuntu 12.04. We obtained cluster measurements after waiting 4 minutes for the system to initialize and reach a stable data flow. We report the average measurements of three consecutive runs. The variance was significantly low and it is thus omitted. We use TL to refer to the task-level algorithm of Section 3.2; TD is the the top-down algorithm of Section 4; TL was prioritized with IMC while TD was prioritized with *relative significance*; Storm refers to the default allocation algorithm.

7.1 Micro-benchmarks

We report the results of the sensitivity analysis of our techniques in Figure 4; we annotate plots with their originating job graph.

Task prioritization. We tested how our prioritization policy based on relative significance improves the existing policy based on IMC (Section 3.2). We used a pipeline of three groups, with the first two having equal processing costs and the third group a significantly higher one. To isolate any other factors that may affect the allocation we used the TL algorithm. We shrunk the IMC of the second pair from two times the IMC of the first pair to the two being equal. We report the IMC gain of each approach (y-axis) with every IMC decrement (x-axis). Both policies start with the same allocation. As the IMC of the second pair is reduced, its relative significance drops and it is given lower priority. Conversely, while the IMC of the second pair is higher, it is given higher priority by the IMC prioritization policy. The IMC gain of the IMC policy decreases linearly with the reduction in IMC; prioritization based on relative significance, however, is balanced and gives a constant improvement.

Impact of task count. Our algorithms address the case when one group has disproportionally many tasks in comparison to other groups. This arises when the task count is correlated with processing cost. We started with a group pair of ten tasks per group and increased the task count in the second group to a factor of seven. We employ only six worker nodes. The processing cost of each group was twice the processing capacity of each worker node; the spare capacity of each worker node was at 80%. We compare the TL and TD algorithms in Figure 4(b). The performance, measured in IMC gain (y-axis), of the TL algorithm substantially degrades with increasing task count (x-axis). The TD algorithm remains unaffected as it splits groups in a balanced way. This even results in a marginal improvement as the task count grows.

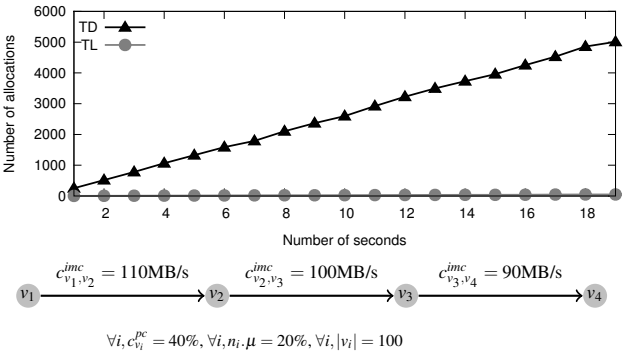


Figure 5: Invocation throughput

7.2 Throughput

A central allocator scheme requires fast allocation algorithms; otherwise, the task allocator becomes a bottleneck. We tested the throughput of the TL and TD algorithms by counting the number of completed invocations of each algorithm in a fixed time interval for a pipeline of four groups with 100 tasks each. All groups had equal processing costs and approximately equal IMC. In Figure 5 we plot the duration of the throughput run (x-axis) and the number of successful allocations of each algorithm (y-axis). The TD algorithm dominates TL by two orders of magnitude. At the peak of Figure 5, TD achieves a throughput of $5012/19 \simeq 264$ allocations per second which allows a centralized allocator scheme to be viable in large clusters. The results indicate that our top-down algorithm delivers its promise for fast and lightweight allocations. Considering its high-quality allocations it is the best choice of algorithm overall.

7.3 Comparison to the optimal solution

The aim of this benchmark is to show that, for the studied class of problems and volatile environments, we can quickly produce allocations without a significant loss in optimality. For this purpose we compare our algorithms with the optimal solution for 2000 random generated graphs. The allocations were computed for a cluster of 8 single-core nodes with a total capacity of 800%. The latter is defined as the percentage of a single processing core's available computational resources. A node with no tasks will have a capacity equal to its number of cores times 100%. The optimal solution was computed with a brute force algorithm *i.e.*, by computing the IMC gain for all possible allocations of a given graph. To allow the exhaustive enumerator to complete within a reasonable amount of time we set the number of tasks equal to 17.

We used a custom graph generator to produce random graphs of a specific class as follows. The generator starts by picking the

number of groups, ranging from 3 to 7, and then distributes a total number of tasks to groups. Every job has a total processing cost of 500% which is also randomly distributed across the available groups. We do this by correlating the amount of PC assigned to the number of tasks in each group. This captures the fact that groups with higher PC are parallelized into more tasks. Then, we connect the groups acyclically with each group having a fanout ranging from 1 to 3, with probabilities 70%, 25% and 5% respectively. Finally, we assign IMC to each edge ranging from 5MB/s to 20MB/s.

We compare the allocations of TD and TL to the optimal, by accumulating for each approach the IMC gain achieved for all graphs. The total IMC gains of TD, TL and the optimal are 24.5GB/s, 22.6GB/s and 26.3GB/s respectively, which means that TD and TL achieve 93.1% and 85.9% optimality, respectively. This suggests that, in group-oriented graphs, designing complicated and expensive algorithms does not pay off as simpler solutions can achieve optimality. By leveraging the graph-based structure we can design simple heuristics that quickly produce results of high quality. Note that this benchmark favors TL as the task count is low and asymmetric task counts are harder to occur (see also Figure 4(b)).

7.4 Real workloads

We tested our algorithms on real-time feeds of Twitter and stock market data. We measured the processing and communication costs of the jobs at the EC2 cluster of 102 nodes and then evaluated the allocation algorithms for different node capacities. The latter are defined as in Section 7.3. A capacity greater than 100% denotes that the processing node has multiple cores available; a processing cost greater than 100% means that the processing needs of a task exceed the capability of a single core. Recall from Section 6 that a single worker is capable of accommodating more than one tasks and our algorithms explicitly allocate tasks to workers. As node capacity increases the tasks are assigned to fewer nodes. For each allocation we plot the IMC gain. We present three algorithms: TL with IMC prioritization, TD and the default Storm allocator.

We saw in micro-benchmarks that small changes in the environment can affect the decisions of the algorithms. We revisit this in realistic settings where what will likely change will be (*a*) the available node capacity, as jobs come and go, and (*b*) the processing and intermodule communication costs as inputs exhibit burstiness. These are exactly the use-cases we target: abrupt environmental changes that make the existing configuration suboptimal and require adaptation. By varying node capacity we see how the algorithms behave as spare space becomes available. We start with the lowest node capacity that ensures a successful allocation and scale it until all algorithms converge. Convergence occurs naturally: as capacity increases, tasks are assigned to fewer nodes. Thus, achieving a good allocation becomes easier. For a fixed capacity, a burst in incoming traffic means increased processing and communication costs as more data needs to be processed in a shorter period of time. Our goal is to seamlessly deal with this increased demand.

Real-world jobs can result in significant network load. The Twitter job pushes a total of 143MB/s of data to the network while the stock market job pushes a total of 790MB/s; the amount of data transferred is continuous. Real-world production clusters may have to run multiple jobs like these at the same time. Thus, efficient allocation algorithms are vital in mitigating the network load.

Twitter trending topics. We built a topology to compute the top k trending twitter topics. We gathered data by sampling Twitter[3] to a total of ten million tweets. We saved the sample so we could replay it on demand and feed it to the cluster through Storm spouts.

[3]Using the Twitter4j API, see `http://twitter4j.org`.

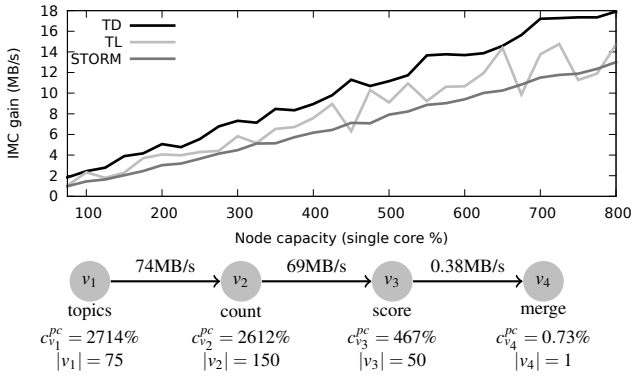


Figure 6: Twitter trending topics

We streamed the set of tweets continuously at a constant rate. The results for $k = 10$ and the topology of the job in general are shown in Figure 6. The topology is a pipeline of (*a*) a single spout that pushes topics to the cluster; (*b*) a counting bolt that updates the count of each topic, partitions topics alphabetically, and propagates the topic/count pair; (*c*) a ranking bolt that receives per-partition topic/counts and maintains a list of the top k topics per partition; and (*d*) a merging bolt that merges all lists to produce a single list of the current top k topics. Each vertex of the topology is annotated with the total PC of the corresponding group as a percentage of node capacity as measured by our framework. Each edge is annotated with the IMC between its groups, again as measured by our framework. TD outperforms Storm's default task allocation algorithm by up to 91% and is 79% better than TL. The savings of TD compared to the total network load of the job reach up to 13%.

Bargain detection. Based on stock market analysis, the goal is to identify bargain stock quotes and raise alerts. One way to do this is to compare the stream of quotes to the *volume weighted average price* (VWAP) of each symbol. Bargain quotes are a function of the current price, volume, and VWAP of the symbol. We used the per-minute ticks of quotes and trades of the companies of the NASDAQ 100. To increase the load we replayed the stream at a faster rate keeping the same relative temporal distance between quotes and trades. The allocation results and Storm topology are shown in Figure 7. The topology consists of (*a*) two spouts, one for trades and one for quotes; (*b*) an aggregation bolt on trades computing the VWAP of each symbol; we set the window size to ten minutes and the slide size to one element, *i.e.*, we compute the VWAP with each arrival; (*c*) a joining bolt matching symbols to averages; and (*d*) a filtering bolt detecting bargains by comparing the current price to the VWAP. (*e*) a result bolt that writes the bargain quotes to a file. The join bolt keeps an in-memory hash table on the VWAP of each symbol. For each new VWAP aggregate it updates the hash table; for each new quote it probes the hash table and returns the symbol and its VWAP to the bargain filter. This job is indicative of a real-world scenario: continuous, trigger-based processing with relatively few operations per job. The TD algorithm outperforms the Storm task allocator by up to 89% and the TL algorithm by up to 31%. The network load savings of the TD algorithm vary up to 22%.

Feature performance breakdown. In Figure 8 we give a breakdown of the different algorithm features based on the bargain detection example to showcase how they contribute to the final result. We compare the Storm algorithm with the TL and TD (TD-IMC) algorithms both prioritized with IMC. Then, we add TD prioritized with relative significance (TD-RS). The TD-IMC algorithm yields efficiency comparable to TL but at a lower complexity. Relative significance (TD-RS) further improves the results.

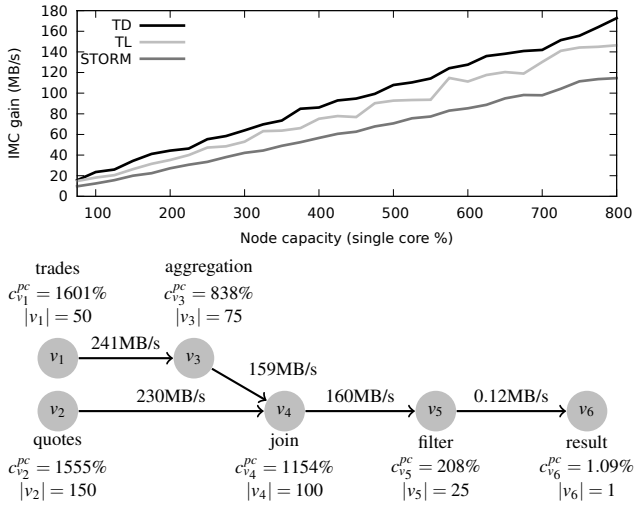


Figure 7: Stock market bargain detection

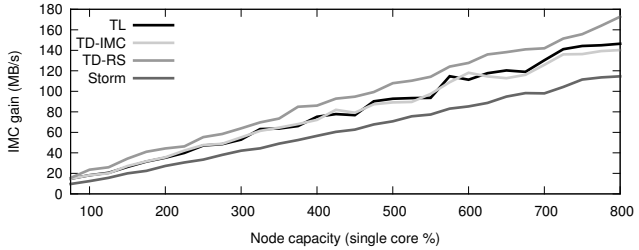


Figure 8: Feature breakdown in bargain detection

Adapting to varying input rates. We now illustrate the adaptability of our approach. We continuously streamed the Twitter data to the EC2 cluster as we varied the input rate; each node was limited to a capacity of 400%. Each time the input rate was detected to change, the task allocator would be triggered. We would then measure the total IMC gain. Note that processing and communication costs scale proportionally to the input rate. We plot the results in Figure 9 in two parts. The bottom part is the incoming input rate in tweets per second (y-axis) as time progresses (x-axis). The top part compares the IMC gain of TD with reconfiguration, termed as TD-RC, with TD and the default allocator in Storm. The allocation of both TD and Storm was based on over-provisioning by running the algorithms once at the maximum input rate (minutes 2 to 5). The resulting allocation was used throughout the experiment independently of the input rate. The fluctuation of the input rate affects the IMC gain of both TD and Storm. TD performs a factor of two better than Storm but when the input rate is reduced the performance of both algorithms drops significantly. TD-RC reconfigures the cluster to reduce IMC, so the hit on the network load is not as severe. When the input rate drops the framework detects it. Our allocator leverages the spare processing capacity to fuse more tasks to nodes, resulting in an almost constant IMC gain throughout the experiment independently of the input rate, which eases network load (from minute 5 onwards). These results show that our monitoring framework is capable of detecting changes in incoming traffic and also reacting quickly to those changes by reconfiguring the cluster.

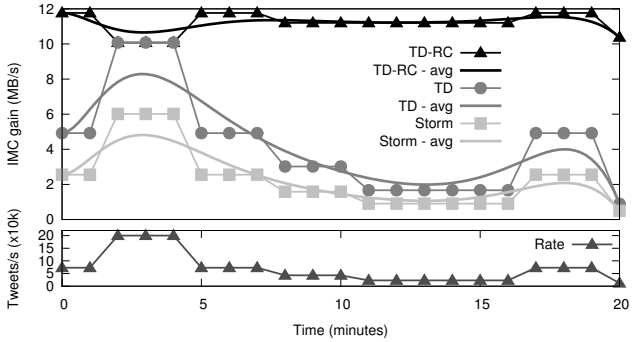


Figure 9: Adapting to varying input rate

8. RELATED WORK

Task allocation in distributed systems has been extensively studied. The main body of research stems from three sources: 0-1 integer linear programming, graph theory, and heuristics. Approaches based on 0-1 integer linear programming (*e.g.*, [13, 14, 18]) produce optimal solutions by formulating task allocation as an optimization problem and solving it by using mathematical programming techniques. Variants include branch-and-bound *e.g.*, [10, 24], which may reduce the computation required. At worst, the time needed grows exponentially with the problem size. Graph-theoretic approaches [6, 7, 20, 21, 27, 32] apply graph algorithms to the task graph (*e.g.*, *min-cut* or shortest path) to find a solution. Similarly, [12] uses Markov decision theory: the system is modeled as a semi-Markov process with rewards. Mathematical optimization and graph-theoretical approaches produce optimal solutions at the expense of high computational costs. In a real-time system this is not an option. Thus, we have employed heuristics and exploited the properties of the problem to improve the quality of the solutions.

Existing heuristic approaches prioritize tasks either in pairs or through centroids [17, 19, 28, 38]. In the first case the job graph is decomposed into connected task pairs, which are then sorted by their IMC. We have adapted these algorithms for our setting and presented them as the basic algorithms of Section 3. In the second case, the first step is to identify the maximally linked tasks, *i.e.*, the centroids. Then, tasks are prioritized on centrality. In both cases, tasks are fused to form clusters while the cluster can be collocated in a single node. Once the capacity of a node is exceeded a new cluster is formed, and so on. When it is not possible to finish the allocation, these algorithms revisit previous decisions until they reach a valid allocation. We follow similar principles in our task-level algorithms and the techniques of Section 5. In [15], the authors account for precedence relationships between the tasks. This is not applicable in a streaming setting. Overall, existing work, though applicable, is not the best option in our setting as it does not take into account task grouping. Our evaluation has shown that group-aware policies outperform task-level ones. Other approaches include graph matching between tasks and processors [31], by formulating the problem as a state-space search problem and applying general optimization algorithms like A* [22] or simulated annealing [30]. In line with our decision not to use mathematical optimization algorithms we have chosen to do away with general optimization algorithms as well and provide a specialized solution.

In the database community and in push-based systems there is a solid body of work [1, 2, 34]. In [36] a decentralized scheme based on load coalescing is proposed. This work has a similar aim with our work for minimizing communication delays, however, it takes a different direction as we focus on a centralized approach with a fast allocation algorithm. In [35] low latency is achieved by minimizing load variance and maximizing load correlation while, in later work [37], resilient allocations are proposed to handle short-term bursts. Both works in contrast to ours assume an unlimited network bandwidth. Other work in decentralized allocators includes [11, 26]. The former focuses on load balancing while the latter consid-

ers the network latency but is designed for pools of wide-area overlay networks. SODA [34] is the scheduler of System S [3] which produces allocations based on several metrics with a combination of heuristics and exact algorithms. It targets scenarios where the load significantly exceeds systems capacity. Such scenarios favor more expensive algorithms as the margin for error is low. [29] introduces a dataflow operator which adaptively adjusts data partitions. This is orthogonal to our work since it focuses on the intra-operator load distribution. Finally, [4] presents an improved scheduler for Storm for adapting to workload changes. It differs from our work since it is designed and modeled specifically for Storm and, more importantly, it does not take into consideration the group-oriented structure of Storm jobs, which is the main contribution of our work. Their allocation approach is at a task-level granularity ordered by the task IMC which is quite similar to our baseline.

Other efforts [9] propose scheduling techniques, for deciding which operators, their order, and the amount of tuples to process. This differs from our work since it focuses on scheduling at a single machine level rather than on operator allocation to a cluster of nodes. Likewise [5] proposes a scheduling algorithm for minimizing the run-time system memory for a single processor.

9. CONCLUSIONS AND OUTLOOK

We have presented a framework for dynamic reconfiguration in data stream processing systems running on clusters. We profile currently executing jobs to detect changes in their behavior. We then dynamically reconfigure jobs by reallocating their tasks across the nodes of the cluster. Our techniques leverage the group-oriented way state-of-the-art systems represent jobs. Based on this we devise fast and efficient task allocation algorithms. We have implemented and evaluated our proposals. Our results show that our top-down algorithms can provide fast allocations that are also near-optimal rendering a central allocator scheme as a scalable alternative to various decentralized schemes. Given the groundwork we have presented, future directions include: (*a*) extending the monitoring framework and the allocation algorithms to cater for more objectives; (*b*) workflow reorganization (*e.g.*, changing the order of operations) in addition to task allocation as a means for improving performance; and (*c*) accounting for task sharing across jobs.

10. REFERENCES

[1] D. J. Abadi *et al.* Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 2003.

[2] D. J. Abadi *et al.* The design of the Borealis stream processing engine. In *CIDR*, 2005.

[3] L. Amini *et al.* SPC: A distributed, scalable platform for data mining. In *DMSSP*, 2006.

[4] L. Aniello *et al.* Adaptive online scheduling in Storm. In *DEBS*, 2013.

[5] B. Babcock *et al.* Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, 2004.

[6] S. Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *Software Engineering, IEEE Transactions on*, SE-7(6), 1981.

[7] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Trans. Softw. Eng.*, 5(4), 1979.

[8] A. Burns. Scheduling hard real-time systems: a review. *Softw. Eng. J.*, 6(3), 1991.

[9] D. Carney *et al.* Operator scheduling in a data stream manager. In *VLDB*, 2003.

[10] M.-S. Chern *et al.* An lc branch-and-bound algorithm for the module assignment problem. *Inform. Process. Lett.*, 1989.

[11] M. Cherniack *et al.* Scalable distributed stream processing. In *CIDR*, 2003.

[12] T. C. K. Chou and J. A. Abraham. Load balancing in distributed systems. *IEEE Trans. Softw. Eng.*, 1982.

[13] W. Chu. Optimal file allocation in a multiple computer system. *Computers, IEEE Transactions on*, C-18(10), 1969.

[14] W. Chu *et al.* Task allocation in distributed data processing. *Computer*, 13(11), 1980.

[15] W. W. Chu *et al.* Task allocation and precedence relations for distributed real-time systems. *IEEE Trans. Comput.*, 1987.

[16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[17] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, 15(6), 1982.

[18] O. I. El-Dessouki *et al.* Distributed enumeration on between computers. *Computers, IEEE Transactions on*, 1980.

[19] V. B. Gylys and J. A. Edwards. Optimal partitioning of workload for distributed systems. In *COMPCON*, 1976.

[20] K. Haessig and C. J. Jenny. Partitioning and allocation computational objects in distributed computing systems. In *Proc. IFIP Congr.*, 1980.

[21] C. J. Jenny. Process partitioning in distributed systems. In *Proc. NTC*, 1977.

[22] M. Kafil *et al.* Optimal task assignment in heterogeneous distributed computing systems. *Concurrency, IEEE*, 1998.

[23] J. Kreps *et al.* Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[24] P.-Y. R. Ma *et al.* A task allocation model for distributed computing systems. *Comp., IEEE Trans. on*, 1982.

[25] L. Neumeyer *et al.* S4: Distributed stream computing platform. In *ICDM Workshops*, 2010.

[26] P. Pietzuch *et al.* Network-aware operator placement for stream-processing systems. In *ICDE '06*, 2006.

[27] G. Rao *et al.* Assignment of tasks in a distributed processor system with limited memory. *Comp., IEEE Trans. on*, 1979.

[28] A. Sarje and G. Sagar. Heuristic model for task allocation in distributed computer systems. *Computers and Digital Techniques, IEEE Proceedings*, 138(5), 1991.

[29] M. A. Shah *et al.* Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2002.

[30] J. Sheild. Partitioning concurrent vlsi simulation programs onto a multiprocessor by simulated annealing. *Computers and Digital Techniques, IEEE Proceedings*, 134(1), 1987.

[31] C.-C. Shen *et al.* A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *Computers, IEEE Transactions on*, 1985.

[32] H. S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Softw. Eng.*, 1977.

[33] Storm. `http://github.com/nathanmarz/storm`.

[34] J. Wolf *et al.* SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. Lecture Notes in Computer Science, 2008.

[35] Y. Xing. Dynamic load distribution in the Borealis stream processor. In *ICDE*, 2005.

[36] Y. Xing. Load distribution for distributed stream processing. In *EDBT Workshops*, 2005.

[37] Y. Xing *et al.* Providing resiliency to load variations in distributed stream processing. In *VLDB*, 2006.

[38] X. Yang and X. Zhang. A general heuristic algorithm of task allocation in distributed systems. In *Comp. and Appl.*, 1987.

[39] Apache Zookeeper. `http://zookeeper.apache.org/`.