T3-Scheduler: A topology and Traffic aware Two-level Scheduler for stream processing systems in a heterogeneous cluster

Leila Eskandari, Jason Mair, Zhiyi Huang, David Eyers

PII: S0167-739X(17)32643-2
DOI: https://doi.org/10.1016/j.future.2018.07.011
Reference: FUTURE 4329

To appear in: *Future Generation Computer Systems*

Received date : 17 November 2017
Revised date : 5 July 2018
Accepted date : 10 July 2018

T3-Scheduler: A Topology and Traffic Aware Two-Level Scheduler for Stream Processing Systems in a Heterogeneous Cluster

Leila Eskandari*, Jason Mair, Zhiyi Huang, David Eyers

Department of Computer Science
University of Otago
Dunedin
New Zealand
Email: {leila,jkmair,hzy,dme}@cs.otago.ac.nz

Abstract

To efficiently handle a large volume of data, scheduling algorithms in stream processing systems need to minimise the data movement between communicating tasks to improve system throughput. However, finding an optimal scheduling algorithm for these systems is NP-hard. In this paper, we propose a heuristic scheduling algorithm—T3-Scheduler—for a heterogeneous fog or cloud cluster that can efficiently identify the tasks that communicate with each other and assign them to the same node, up to a specified level of utilisation for that node. Using three common micro-benchmarks and an evaluation using two real-world applications, we demonstrate that T3-Scheduler outperforms current state-of-the-art scheduling algorithms, such as Aniello et al.'s popular 'Online scheduler' and R-Storm, improving throughput by up to 32% for the two real-world applications.[1]

Keywords:
Stream processing, Scheduling, Big Data, Heterogeneous Cluster

*Primary contact: Phone: +64 3 479 8498 Fax: +64 3 479 8529
[1]This work is an extension of an Auto-DaSP workshop paper in [1]

1. Introduction

The increasing amount of data generated by new applications such as social networks, low latency stock trading and real-time search, have necessitated the development of new data processing frameworks. Data Stream Processing Systems (DSPSs) process unbounded streams of data in real-time, as the data arrives without the need to store it first. Over the last few years, a broad range of research has advanced stream processing systems [2, 3, 4, 5]. An extensive survey on different generations of DSPSs can be found in [6].

As DSPSs need to quickly process large amounts of data, they are often located in cloud data centers which can be distant from the the sources of data streams, resulting in higher network latency and lower system throughput. A potential solution to this problem is edge node/fog computing [7, 8], where streaming applications are run on compute nodes on the edge of the network [9]. This improves data locality by placing the computations and data sources closer to each other, providing lower network latency. However, fog nodes can consist of multiple types of hardware, resulting in heterogeneous clusters of nodes. To fully utilise the edge node resources, task allocation policies in DSPSs need to be made more heterogeneous aware, such that they can effectively schedule streaming applications across multiple nodes of different types. These policies can further be applied to data centers, where multiple generations of hardware can exist as new nodes are added to existing systems over time [10].

To further improve data locality, an efficient task allocation policy can place the tasks which communicate with each other on the same compute nodes to reduce data movements between compute nodes, improving network latency and system throughput [11]. A number of scheduling algorithms have been proposed in the literature to improve the performance of DSPSs. The common practice in these heuristic algorithms is to find the tasks that communicate with each other and put them in the same node. However, these methods inspect each communicating pair of tasks or groups of tasks in isolation, which can result in task assignments, where tasks are spread across nodes, increasing communication cost. This can also leave some nodes underutilised in a heterogeneous cluster, further increasing communication cost. To address the above issues, we propose T3-Scheduler, a Topology and Traffic aware Two-level Scheduler for DAG-based DSPSs which can find highly communicating tasks and assign them to the same compute node in a heterogeneous cluster such that each node remains fully utilised. T3-

Scheduler is *topology* aware, as it considers the shape and connectivity of the topology graph when finding highly communicating tasks. It is also *traffic* aware as it monitors the streaming application execution to find the data transfer rates between tasks communicating with each other. Finally, it is *two-level* as follows:

- First level: T3-Scheduler divides the application graph into multiple parts, where each part includes highly communicating tasks, with a size relative to the capacity of a compute node in the heterogeneous cluster. This allows T3-Scheduler to reduce inter-node communication by placing groups of highly communicating tasks together and to fully utilise each compute node, by filling it to capacity. The novel technique used by T3-Scheduler to find groups of neighbouring tasks in the application graph provides a broad view of the communicating tasks, rather than the localised view of some previous approaches.
- Second level: Once tasks are assigned to a compute node, T3-Scheduler finds the best assignment of those tasks within the node by placing highly communicating tasks in the same worker process. This helps to minimise the communication between the worker processes within a compute node. To the best of our knowledge, this is the first work to consider the scheduling of tasks within the compute node.

This paper makes the following contributions:

- We propose a scheduling algorithm–T3-Scheduler–that searches the application graph to find groups of highly communicating tasks, which are grouped and assigned to the same compute node, based upon the available capacity of computing nodes within the cluster. This novel approach of searching the application graph to find neighbouring groups of communicating tasks and assigning them to the same node reduces the inter-node communication, while fully utilising each node.
- We evaluate the communication cost of T3-Scheduler by comparing it to a theoretically optimal scheduler, implemented in CPLEX [12], when run on three micro-benchmarks, each representing a different communication pattern. The evaluation shows that T3-Scheduler can achieve results that are close to optimal in a number of different cluster configurations.

- We implement T3-Scheduler in Apache Storm 1.1.1 [4] and through experimental results show that our proposed scheduler outperforms the adaptive 'Online scheduler' [13] (for brevity we refer to this scheme as OLS in this paper) and R-Storm [14] when run on three micro-benchmarks and two real world applications. The results show that T3-Scheduler increases throughput by up to 32% for the two real-world applications.

The rest of the paper is organised as follows. In Section 2, some existing schedulers for DSPs are discussed. Section 3, presents a system model and formulates the scheduling problem. Section 4 presents our proposed method. In Section 5, we compare T3-Scheduler with a theoretically optimal scheduler, followed by an experimental evaluation of T3-Scheduler in Section 6. Finally, Section 7 concludes the paper and discusses future work.

2. Related Work

There is an extensive amount of research on scheduling in data stream processing systems. Generally, there are three main approaches to tackle the scheduling problem [15]: Mathematical programming, graph-based approximation and heuristics.

Cardellini et al. [16] formulate the optimal schedule using integer linear programming. It considers the heterogeneity of computing and networking resources, finding the optimal solution for a small number of tasks. The resolution time of this approach grows exponentially with the problem size, making it impractical for larger problems. The same authors propose a model in [17] to find the optimal number of replicas for an operator in a DSPS, which has the same scalability problem. SQPR [18] presents an optimisation model for query admission, allocation and reuse. If an optimal solution is not able to be found within a given time, the best solution up to that point will be used. While it does not guarantee an optimal solution, the scheduling algorithm can find a schedule in a fixed amount of time. SODA [19] is an optimised scheduler designed for System S [20]. It models the placement of Processing Elements (PEs) on the processing nodes for admitted jobs as a mixed integer programming problem. A heuristic approach is used as backup, in case the optimal solution fails. DRS [21] finds the optimised number of tasks for each operator based on the theory of Jackson open queuing networks, minimising the processing time for the input data.

In graph-based approximation approaches, the application graph is considered as a DAG where vertices and edges in the graph represent tasks and data flow between the tasks respectively. Then, the graph-based approximation techniques are used to tackle the problem. For example, to reduce the inter-node communication, the vertices are assigned to compute nodes using graph algorithms such that the communication cost is minimised. COLA [22] in System S uses a graph partitioner to fuse multiple PEs of a stream processing graph into bigger PEs in order to decrease inter-process communication. Quincy [23] for Dryad [24] maps the problem of task to worker assignment into a graph over which a min-cost flow algorithm is used. It then minimises the cost of a model which includes data locality, fairness, and starvation freedom. Fischer et al. [25] propose a mapping of workload scheduling to graph partitioning problem. Ghaderi et al. [26] develop a stochastic model of resource allocation for graph-based streaming applications. P-Scheduler [27] first calculates the number of nodes required to run the application. It then exploits graph partitioning algorithms in a hierarchical manner to schedule the DAG on a homogeneous cluster. Re-Stream [28] identifies the critical path of a data stream graph and reallocates the critical vertices on the critical path to minimise response time. It also consolidates non-critical vertices on non-critical path to maximise the energy efficiency. Locality-Aware Routing in [29] is reduced to a graph clustering problem. The authors' aim is to partition the bipartite graph such that pairs of keys that appear together frequently are in the same group. They analyse the correlation between keys used in successive fields grouping, and explicitly map correlated keys to operator instances executed on the same server. Hence, data tuples containing these keys can be passed from one operator to the next through an address in memory, instead of copying the data over the network. This has the advantage of being faster, and avoids the saturation of the network infrastructure.

To work effectively, heuristic approaches make a number of simplifying assumptions that can result in sub-optimal solutions. Despite this, they are useful when an optimal solution is not essential, or when determining such an optimal solution would be too time consuming [34]. A number of heuristic approaches have been proposed in the literature that are able to find efficient solutions in a timely manner [35]. The optimisation goal of [36] and [37] is to minimise the network usage. They both use a latency space as the search space to find the optimal placements in a distributed way. The former is based on the physical model of springs where a spring relaxation technique is used to find operator placement in the virtual nodes in the cost space

Table 1: Related work comparison

Scheduling algorithm	**Solution Approach**	**Optimisation Goal**	**Dynamic**	**Heterogeneity**
Aniello et el. [13]	Best fit	Inter-node communication	✓	✓
R-Storm [14]	3D knapsack	Resource utilisation, Inter-node communication	×	✓
T-Storm [30]	Bin packing	Resource utilisation, Inter-node communication	✓	✓
Cardellini et al. [16]	Integer programming	Resource utilisation, Inter-node communication, QoS Metrics	✓	✓
Chatzistergiou et al. [31]	Task grouping	Inter-node communication	✓	✓
Fischer et al. [25]	Graph partitioning	Inter-node communication	✓	×
Ghaderi et al. [26]	Dynamic graph partitioning	Resource utilisation	✓	✓
P-Scheduler [27]	Graph partitioning	Resource utilisation, Inter-node communication	✓	×
Rychly et al. [32]	Benchmarking applications	Workload balancing	✓	×
DRS [21]	Jackson open queuing networks	Resource utilisation	✓	✓
Cardellini et al. [33]	Searching a cost space	QoS Metrics	✓	✓
Re-Stream [28]	Critical path in graph	Energy consumption Response time	✓	✓
Caneill et al. [29]	Finding correlations between the keys	Inter-node communication	✓	✓

and then the closest physical nodes to these solutions are found. The latter iteratively finds an optimal virtual node for each operator in the latency space, based on the Weber problem, which is then mapped to a physical node. Similar to [36], Cardellini et al. [33] optimise scheduling by using a spring based formulation to search a 4-dimensional cost-space including latency, throughput, utilisation and availability.

Aniello et al. [13] use a greedy best fit approach in order to co-locate communicating task pairs in the same compute node. Their scheme monitors the execution and sorts the communicating task pairs in descending order based on their data transfer rate. It then iterates over the sorted task pair list and assigns each pair to the least loaded node. A different heuristic approach is proposed in [31], which inspects communicating groups of tasks instead of task pairs and has provided some inspiration for this paper. Using a greedy approach, a group pair is selected to be assigned to the node with the highest remaining capacity. R-Storm [14] is an offline, resource-aware scheduler that considers the CPU, memory and bandwidth requirements of each task, as specified by the user. It then deals with the scheduling as a

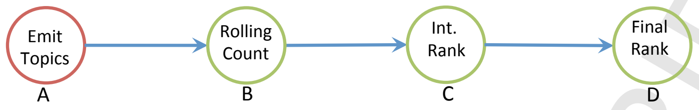


Figure 1: Operator view of a streaming application

3D knapsack problem. T-Storm [30] fully utilises each node by filling it to capacity. The algorithm sorts all of the tasks in descending order of their incoming and outgoing load. Then, it assigns tasks to available nodes using a bin packing approach, where nodes are sorted by capacity. Rychly et al. [32] profile each application and assign tasks based upon the dominant resource i.e. tasks that heavily use the CPU are assigned to nodes with the largest CPU capacity, while graphic-intensive tasks are placed on nodes with powerful GPUs.

Table 1 provides a comparison of the schedulers presented above, each of which was implemented in Apache Storm. We have done such a selection for a fair comparison. From this table, it can be seen that each of these algorithms is based upon varying assumptions and optimisation goals. In this paper, our optimisation goal is to reduce inter-node communication and to fully utilise compute resources. The heuristics above with the same optimisation goal either under utilise the resources or inspect each task or task pair in isolation. In the following sections, we describe our algorithm which can have a broader view of communicating tasks and assign them to the same compute node. We then compare it with the algorithms in [14] and [13], which have the same optimisation goal.

3. System Model and Problem Definition

An application in data stream processing systems such as Storm [4], S4 [2], Flink [3], and Heron [5] can be represented as a Directed Acyclic Graph (DAG), shown as $G = (V, E)$. These systems are designed to process large volumes of data, at high velocity, as it flows through the system. Each vertex v_i in V represents an operator, and each edge e_j in E represents the data flow between two operators communicating with each other. An operator can either be a source or a computation unit. A source reads data from an external source and emits it as a stream into the streaming application. Computational units process the stream, received from either a source or another computational unit, before emitting the results to another computational unit.

A common streaming application used on these systems is the real-time analysis of data streams, such as the top trending topics on twitter, which provides insights into what is being tweeted about at a given time. Figure 1 shows the operator-view graph of this streaming application. From this figure, it can be seen that it has a linear layout with one source and three computational operators, where an edge connects two operators communicating with each other.

Each operator in a streaming application has one or more tasks, where each task is an executing instance of the operators code. This allows each task within an operator to perform the same computation on different data streams, providing parallelism. The number of tasks can be determined by the developer of the streaming application and changed during run-time if needed. When more than one task is used in an operator, *stream grouping* indicates how the tasks of the emitting operator are connected to the tasks of the receiving operator, and how the stream is partitioned among the receiving tasks. In a communicating operator pair (A, B), the communication between a task in Operator A and a task in operator B can be shown as a connected edge, resembling a bipartite graph.

For example, Figure 2 shows the task-view graph of the top trending topics streaming application, previously shown in Figure 1. From the figure, it can be seen that operators A, B, C and D have 5, 9, 4 and 1 tasks respectively. This streaming application will be described in detail in Section 6.3. As it can be seen from Figure 2, the tasks of every two communicating operators are fully connected due to the parallelism and stream groupings.

Each vertex and edge in the task view graph is weighted by the task load and data transfer rate between the two communicating tasks respectively. We define the scheduling problem as mapping the vertices in the task view graph to a set of compute nodes with different capacities, such that the sum of the inter-node edge weights is minimised and the sum of the vertex weights does not exceed the capacity of a given node. In other words, the DAG should be divided into a number of parts, where each part is sized relative to the capacity of the respective node, while minimising the sum of edge weights between compute nodes.

Similar to [31, 16], the scheduling problem in a DAG-based DSPS is formulated as follows. We assume that $(t_i$,$t_j)$ represents two tasks t_i and t_j in V which communicate with each other in a streaming application, which are connected by an edge in E. $R(t_i, t_j)$ is the data transfer rate between these

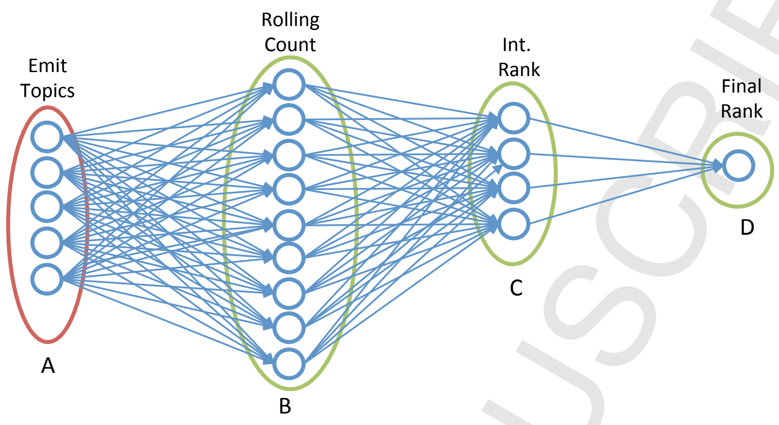


Figure 2: Task view of a streaming application

two tasks, represented as the edge weight in G. $L(t_i)$ indicates the load of task t_i, represented as vertex weight for this task, n_u represents node u in the node set N, and $C(n_u)$ is the available capacity of n_u. The problem of minimising data movement between the communicating tasks in an application can be expressed as minimising the following sum:

$$\sum R(t_i, t_j) \times X_{ij} \quad \forall (t_i, t_j) \tag{1}$$

Such that:

$$\sum L(t_i) < C(n_u) \quad \forall n_u \in N, \; \forall t_i \text{ assigned to } n_u \tag{2}$$

Where X_{ij} is 0 if t_i and t_j are on the same node; otherwise, it is 1.

While it is possible to formulate the problem of optimally scheduling tasks onto a set of nodes, it has been proven to be NP-hard [38, 39, 40] due to the large search space and computational complexity. This means that small problems sizes might be solved in a feasible time. However, as the problem size grows, it becomes impractical to find the optimal solution because of the increased computational complexity. To overcome this practical limitation of an optimal scheduler, there is a need for a heuristic approach to find a near optimal schedule in real-time. Our heuristic approach reliably and efficiently finds the highly communicating tasks, discussed in detail in Section 4.

4. T3-Scheduler Algorithm

To efficiently assign tasks to compute nodes in a DAG-based DSPS, a scheduler needs to find groups of highly communicating tasks, which can then be assigned to the same node, reducing the inter-node communication cost. This can be seen as a graph partitioning problem where the application graph is partitioned into multiple parts such that the number of edge cuts between the partitions is minimised. Each part can then be assigned to a compute node, such that the nodes capacity is not exceeded. Usually, graph partitioning algorithms such as [41] are used to produce partitions of roughly equal parts, suitable for a homogeneous configuration as proposed in [25, 27]. However, to produce partitions of different sizes, graph partitioning algorithms are dependent upon *a priori* information. That is, they rely on knowing the number of tasks to be assigned to each node before the graph can be partitioned. This is a barrier to practical deployments as it is difficult to reliably know this information at scheduling time.

To address this limitation, T3-Scheduler uses a heuristic method to produce partitions of different sizes such that the inter-part communication is reduced. Each part can then be assigned to a node with a relative capacity. This is achieved without requiring any a priori information, such as the number of tasks to be assigned to each compute node.

T3-Scheduler consists of five main steps as follows.

1. **Monitoring:** To facilitate the task scheduling, T3-Scheduler needs to initially profile the task graph execution, measuring the tasks' load and data transfer rate between the communicating tasks.
2. **Constructing a simplified graph:** The profile from the previous step is then used to build a weighted simplified graph. This graph is initially similar to the operator-view graph, and is updated with new vertices and edges if necessary.
3. **Node selection:** T3-Scheduler selects a node by taking the capacity into account.
4. **First level of scheduling:** At this level, T3-Scheduler determines which groups of tasks should be co-located within each node by finding a sub-graph of highly communicating tasks from the simplified graph.
5. **Second level of scheduling:** At this level, T3-Scheduler finds the number of workers required for each node and then divides the sub-graph found by first-level of scheduling into the number of workers such

Table 2: Notation for T3-Scheduler Pseudo-code

Symbol	Meaning
op	Online profile
sg	Simplified graph
g	A group of tasks, represented as a vertex in sg
sbg	A sub-graph, consisting of highly communicating tasks
t	Number of tasks within sbg
T	Max number of tasks per worker
w	Required number of workers
n	Current node

that highly communicating tasks are grouped together into a worker in order to reduce inter-worker communication.

Algorithm 1 presents these steps and the following subsections provide a detailed discussion of each step. Table 2 shows the notation used in this paper for the pseudo-codes.

Algorithm 1 Pseudo-code for T3-Scheduler Algorithm

```
op = MONITORING(Stream_Application)
sg = CONSTRUCT_SIMPLIFIED_GRAPH(op)
SCHEDULE(nodes, sg)

function SCHEDULE(nodes, sg)
    while all $g$ in sg are not assigned do
        $n$ = NODE_SELECTION(nodes)
        sbg = FIRST_LEVEL($n$, sg)
        SECOND_LEVEL($n$, sbg, $T$)
    end while
end function
function FIRST_LEVEL($n$, sg)
    sbg ← ∅
    while $n$ is not full do
        g = GROUP_SELECTION($n$, sg)
        Add $g$ to sbg
        Mark $g$ in sg as assigned
    end while
    return sbg
end function
function SECOND_LEVEL($n$, sbg, $T$)
    $w = \lceil \frac{t}{T} \rceil$
    parts ← Partition sbg into $w$ parts using METIS
    Assign each part in parts to one worker on $n$
end function
```

4.1. Monitoring

As the first step, T3-Scheduler monitors the execution of the streaming application. This involves measuring the data transfer rate between each of the task pairs, providing a profile of all communications and also the task loads. The collected values are stored regularly in a monitoring log which T3-Scheduler can read periodically when rescheduling.

4.2. Constructing a simplified graph

T3-Scheduler constructs a weighted simplified graph, initially similar to the operator-view graph, using the online profile, collected from the monitoring step. T3-Scheduler initially aggregates all of the tasks within each operator into a single group, representing a vertex in the graph. The weight of the new group/vertex in the simplified graph is found by summing the load of each task within the group. Each edge in the graph, connecting two groups, is the aggregation of all the communications between two groups, weighted with the sum of the data transfer rates of their communicating tasks. This is made possible by the fact that the tasks within two communicating operators are fully connected in typical DSPSs because of stream grouping.

The simplified graph has the advantage of having a view of the connectivity between all the tasks. Therefore, a sub-graph, consisting of highly communicating tasks, can be found and assigned to the same compute node. Additionally, by considering communicating groups of tasks instead of communicating tasks, a slight change between some data transfer rates of communicating tasks within the communicating groups will not result in rescheduling, making the scheduling more stable. Vertices and edges are updated regularly if any vertex has to be partitioned in order to fit in a compute node.

4.3. Node selection

T3-Scheduler considers the capacity and resource availability of each node, selecting the highest capacity node. This allows T3-Scheduler to take steps towards minimising the inter-node communication as a result of placing more communicating tasks in the higher capacity nodes. If multiple nodes have the same capacity, ties are broken by selecting a node randomly among the potential nodes. T3-Scheduler fills a node with as many communicating tasks as possible, up to its capacity, and then moves to the next highest capacity node.

4.4. First level of scheduling

Algorithm 2 Pseudo-code for Group Selection Algorithm

```
function GROUP_SELECTION($n$, sg)
    if n is empty then
        $p$ = FIND_STARTING_POINT(sg)
        if $p$ can fit in $n$ then
            return $p$
        else
            $(p1, p2)$ = GROUP_PAIR_PARTITIONING($p$)
            UPDATE_SIMPLIFIED_GRAPH(sg)
            return $p1$
        end if
    else
        $g$ = FIND_NEIGHBOUR(sg)
        if $g$ can fit in $n$ then
            return $g$
        else
            $(g1, g2)$ = GROUP_PARTITIONING($g$)
            UPDATE_SIMPLIFIED_GRAPH(sg)
            return $g1$
        end if
    end if
end function
```

The goal of the first level of scheduling in T3-Scheduler is to divide the simplified graph into multiple parts where each part consists of highly communicating groups of tasks. T3-Scheduler takes a heuristic approach to divide the graph into multiple parts which are sized according to the capacity of the heterogeneous node to be scheduled on. For every empty compute node, T3-Scheduler begins by finding the highest communicating group pair as a starting point in the simplified graph and expanding it by repeatedly selecting the most highly communicating neighbouring groups, forming a sub-graph, until the node is full. Once the node becomes full, a new compute node is selected for which a new starting point is found and the same procedure is applied. By following this approach, T3-Scheduler can find highly communicating tasks and place them in the same compute node. In the following, we provide further details of the first level of scheduling.

4.4.1. Forming a sub-graph

After locating the group pair with the highest weight as the starting point, we evaluate if the pair of groups is able to fit within the current compute node. This condition is checked by comparing the sum of the tasks' load of each group, and the capacity of the selected compute node. This will

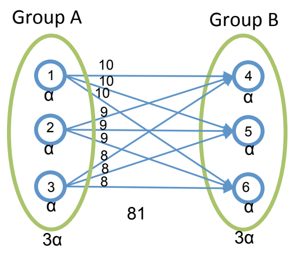


Figure 3: Group pair (A, B), with load 6α, to be assigned to a node, with capacity of 4α

have two possible outcomes. If the compute node has sufficient capacity to accommodate the group pair, it is assigned to the compute node. However, in the event of the compute node having insufficient capacity, a fine-grained partitioning will be performed on the group pair, where the number of tasks within one or both groups is reduced. This will enable a new, smaller group pair to be assigned to the current compute node which would not otherwise be possible. Partitioning a group pair will be explained in more detail shortly.

If the node still has some remaining capacity, we expand the starting point by finding the most highly communicating neighbours. Having located the immediate neighbour with the highest weight, an evaluation is performed to check if this group can fit within the compute node. If the node has sufficient capacity for the neighbouring group, it is added to the sub-graph and the next highest weighted neighbour will be evaluated. But, in the event that the highest weight neighbour is not able to fit within the compute node, an additional fine-grained partitioning will be performed on this group. Partitioning a single group will be explained in more detail shortly. After performing the single group partitioning, we expand the sub-graph to include the new group, which now has a size equivalent to the node's capacity, meaning it is fully utilised. Algorithm 2 presents the pseudo-code for the process of group selection in forming a sub-graph step.

4.4.2. Fine-grained group pair partitioning

To resolve the issue of insufficient capacity to accommodate the group pair selected as the starting point, we partition this group pair into two smaller group pairs, thereby allowing a subset of the initial group pair to be assigned to the compute node. For instance, assume that the group pair denoted as (A, B), shown in Figure 3, has a total size of 6α, which is unable to fit within the compute node with a total capacity of 4α, where α is the average load for each task. Therefore, we have to partition the group pair, (A, B), into two smaller group pairs (A_1, B_1) and (A_2, B_2) such that (A_1, B_1) can fit. The aim of partitioning (A, B) is to minimise the edge cuts between (A_1, B_1) and (A_2, B_2) while maximising the number of tasks in (A_1, B_1). To achieve this, the task pairs with the highest rate from (A, B), are repeatedly selected and assigned to (A_1, B_1) until the node's capacity is reached.

However, when selecting a task pair (t_i, t_j) with the highest data transfer rate from (A, B), three scenarios are possible. To keep track of which tasks have been selected from (A, B), selected tasks are marked as 'selected' in (A, B) when they are assigned to (A_1, B_1).

1. Both tasks, t_i and t_j, are new and have not been selected before. In this case, both tasks will be assigned to (A_1, B_1) if the sum of the load of t_i and t_j is less than or equal to compute node's remaining capacity.
2. One of the tasks is not new and is already marked as selected in A or B. To simplify the explanation, we assume that t_i in task pair (t_i, t_j) is already marked as 'selected' in A and is not new, but t_j is new. We first find all the task pairs connected to t_j from (A, B), denoted as (t_k, t_j), where t_k in A is not marked as 'selected'. Then, among these task pairs, we pick the task pair with the highest rate that can fit in the compute node and assign it to (A_1, B_1). By doing this, (t_i, t_j) is also included in (A_1, B_1) due to full connectivity of two communicating groups. This has the benefit of assigning two new tasks to (A_1, B_1) at each step with the highest data transfer rate and therefore minimising the edge cuts between (A_1, B_1) and (A_2, B_2).
3. Both tasks have already been assigned to (A_1, B_1) and are marked as 'selected' in (A, B) . In this case, no further processing is done and we move to the task pair which has the next highest data transfer rate.

If one group is smaller than the other, for example A is smaller than B, we cannot always find a task pair with two new tasks. In this case, we just assign the task pair with the highest data transfer rate, that can fit in the compute node, to (A, B_1). After repeatedly selecting the task pairs and reaching the compute node's capacity, all the unmarked tasks in B are assigned to B_2 which will be inspected for assignment to another compute node later by T3-Scheduler. The simplified graph is updated with the new vertices, edges and their weights every time a partitioning is performed.

A detailed example of partitioning a pair is presented in Appendix A.

4.4.3. Fine-grained single group partitioning

When expanding the sub-graph, the situation can arise where an entire group, denoted as A, is unable to fit within the current compute node's remaining capacity, requiring A to be partitioned. The aim is to utilise the node by filling it with the most highly communicating tasks from A. At each step, the task with the highest data transfer rate from A, which is connected to the sub-graph within the compute node, is found and assigned to a smaller group, denoted as A_1, until the compute node is full. The selected task from A is marked as 'selected' in A after assignment to A_1. In the case that the selected task's load is higher than available capacity, the task with the next highest data transfer rate is inspected for assignment. This process is repeated until we find a task that can fit in the node. Otherwise, the group partitioning process is complete. The remaining tasks from A, which are not marked as selected, form a new group, denoted as A_2. Then, the simplified graph is updated with the new vertex, edges and their weights. The new group, A_2, has this chance to be placed in another compute node with the unassigned task groups in the simplified graph that it communicates with. A detailed example of partitioning a pair is presented in Appendix B.

4.5. Second level of scheduling

In stream processing systems, such as Apache Storm, each of the compute nodes contains one or more worker processes, where each of these workers run one or more tasks. The scheduler needs to determine how tasks should be assigned to each worker, such that inter-worker communication is minimised. Therefore, T3-Scheduler needs to schedule the tasks to workers within each node based upon the task communication for the sub-graph. Therefore, at the second level of scheduling, it is determined which tasks should be assigned to the same worker process.

Finding the number of workers per node is hard to know *a priori*. In determining the number of workers per compute node, a balance needs to be struck between performance and reliability. That is, assigning all the tasks within a compute node to a single worker will be less reliable, as a single task failure will cause all tasks within the worker to be interrupted and restarted. Whereas, assigning fewer tasks per worker will require more workers, increasing the inter-worker communication.

Therefore, in our simple heuristic we introduce a threshold T, for the maximum number of tasks per worker, which sets a balances between better tolerating failures, and reducing inter-worker communication. The value of

T is empirically determined, by observing which value gives reliable performance results.

To find the number of workers, denoted as w, for each compute node, we divide the number of tasks to be assigned to the compute node, denoted as t, by T, for the given application:

$$w = \left\lceil \frac{t}{T} \right\rceil \tag{3}$$

After finding w for each compute node, T3-Scheduler needs to find highly communicating tasks to be assigned to each worker. This problem can be viewed as partitioning the sub-graph assigned to each node into w parts. To achieve this, T3-Scheduler uses well-studied and performant K-way partitioning proposed in [41, 42], which has commonly been used for graph partitioning [43, 44, 45, 46, 47]. K-way partitioning divides the vertices of each sub-graph g into w roughly equal parts, such that the number of edge cuts between workers is minimised. It is worth noting that it is not possible to use K-way partitioning in the first level of scheduling as the heterogeneous cluster requires groups of varying sizes.

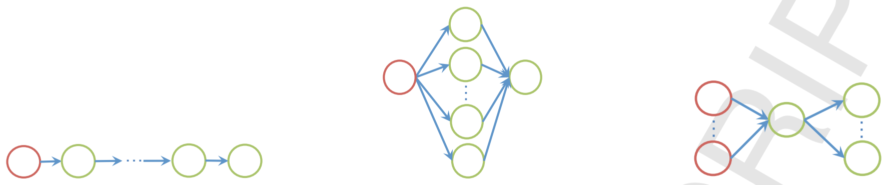
The partitions, found by K-way partitioning, are of roughly equal size, which means each part may contain fewer than T tasks. The number of tasks will remain close to T, but will not exceed this threshold value. Each part is then assigned to a worker, where the most highly communicating tasks are grouped together, minimising inter-worker communication.

Appendix C presents a detailed example of applying T3-Scheduler to the top trending topics example, previously shown in Section 3, describing each step in detail.

5. T3-Scheduler vs Optimal Scheduler

In this section, we compare the communication cost of T3-Scheduler with an optimal scheduler using three micro-benchmarks that represent common shapes of a streaming application [14], as used in previous work [14]. Each of these micro-benchmarks evaluates a different congestion pattern: linear, diamond and star, described as follows:

- **Linear micro-benchmark:** This is one of the most common types of applications, shown in Figure 4a, which consists of a number of operators, where tuples are passed from one operator to the next.



(a) Layout of linear application (b) Layout of diamond application (c) Layout of star application

Figure 4: Layout of micro-benchmark applications [14]

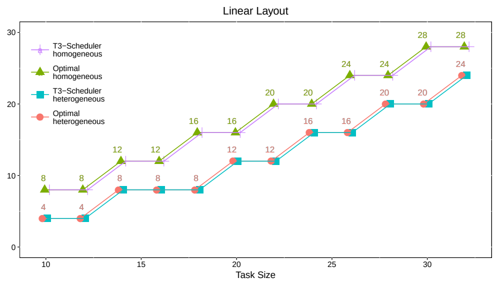
- **Diamond micro-benchmark:** The layout of this micro-benchmark has a diamond shape, shown in Figure 4b, where one source emits tuples to multiple operators. Each operator then passes these tuples to a single sink operator.

- **Star micro-benchmark:** The more complicated layout has a star shape, shown in Figure 4c, where multiple sources emit tuples to one operator. This operator then emits the received tuples to multiple sinks, passing them along.

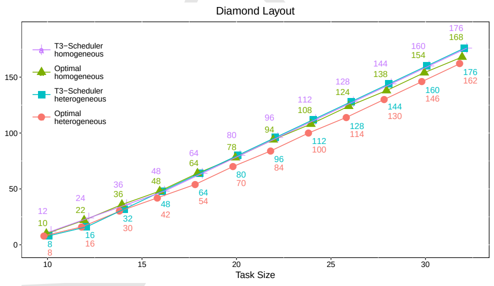
The linear layout is configured with two tasks per operator, while in the diamond layout, the source and sink operators have four tasks, with all other operators having two tasks. In the star layout, each of the source and sink operators has two tasks and the middle operator has four tasks. In the following experiments, we evaluate different problem sizes by increasing the number of operators in each layout, with each new operator containing two tasks. For the linear layout, each new operator is added to the end of the layout, whereas for the diamond, new operators are added to the middle of the layout which communicate with the source and sink operators. Finally, for the star layout, operators are added as either a source or sink, with each new operator alternating between the two, ensuring the layout remains balanced. We configure each of the micro-benchmarks with a problem size of 10 tasks to begin with, increasing the problem size to 32. To simplify the experiments, for illustrative purposes, we assume that the data transfer rate between every two communicating tasks is 1 normalised unit per second, and the load is uniform across all tasks. The simple configuration of these examples is for illustrative purposes, and to ensure the corresponding discussion is clear and easy to follow. It is also worth noting that the schedulers are capable of working with more complex configurations, which do not have a uniform task load or communication.

Table 3: Compute nodes configuration

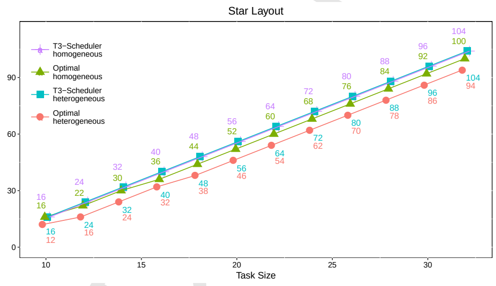
Setting	Number of Nodes	Nodes Capacity
Homogeneous	10 nodes	Each with a capacity of 4
Heterogeneous	10 nodes	3 nodes with capacity of 6 3 nodes with capacity of 4 4 nodes with capacity of 2



(a) Linear layout



(b) Diamond layout

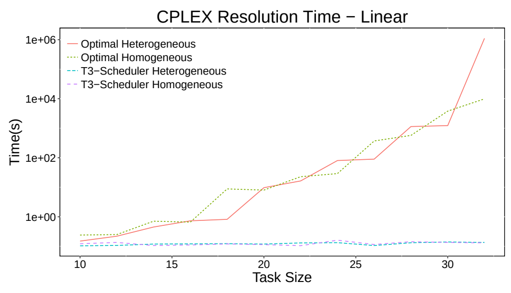


(c) Star layout

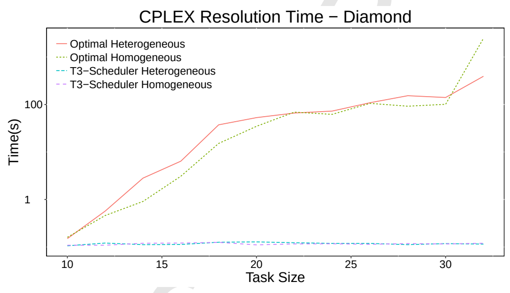
Figure 5: T3-Scheduler vs optimal scheduler in different configurations for linear, diamond and star layouts

We run our experiments on both a homogeneous and heterogeneous cluster configuration, described in Table 3. The homogeneous cluster consists of 10 nodes, each with a capacity of 4, where the capacity of a node is defined as the average number of tasks that can be assigned to the node. The heterogeneous cluster contains 10 nodes, made up of three different node types, with capacities of 6, 4 and 2; there are 3, 3 and 4 nodes of each capacity respectively.

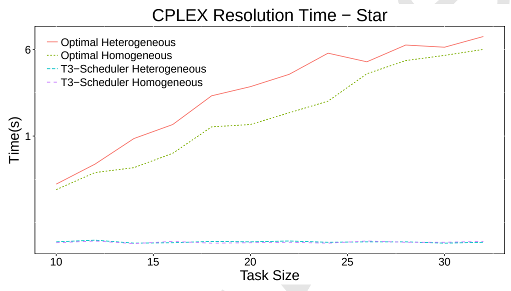
We use IBM CPLEX Studio 12.7 [12] as our optimisation software as it provides state-of-the-art implementations of linear, integer and mixed-integer linear optimisations [18]. We run our experiments to find an optimal solution on a 64-core server with four 16-core AMD Opteron 6276 processors, running at 2.3GHz, with 512GiB of RAM. We require such a large server as it



(a) Linear layout



(b) Diamond layout



(c) Star layout

Figure 6: Resolution time for optimal scheduler and T3-Scheduler in different configurations for linear, diamond and star layouts

becomes increasingly computationally expensive to find the optimal solution for larger problem sizes. In comparison, to find the communication cost for T3-Scheduler, we can use any commodity hardware as T3-Scheduler does not need any specific hardware requirements. We run our experiments for T3-Scheduler on a system with 8GiB of RAM which has a 3.2GHz Intel Core i5-4570 processor with four cores. In each of the experiments, we present the cost, which is a measure of the communication cost between nodes, previously seen in Equation 1. We refer to the solutions found by CPLEX as optimal throughout this paper as done in other work [19, 16]. Further, based on a limited brute force evaluation of small task sizes, we found that CPLEX results were optimal.[2]

The communication costs found by the optimal scheduler and T3-Scheduler for the micro-benchmarks are shown in Figure 5. It is worth noting that while the optimal scheduler can be used to solve small problem sizes in a feasible

[2]A more extensive brute force evaluation is infeasible due to the NP-hard time complexity of the problem.

amount of time, this does not remain the case as the problem size increases due to the increased computational complexity. For the problem sizes greater than 20, the solutions were found in minutes or hours which is not practical. Here, our intended purpose for determining the optimal communication cost is to evaluate how close to optimal the near real-time results of T3-Scheduler are.

From Figure 5a, it can be seen that for both the homogeneous and heterogeneous configurations, T3-Scheduler is able to match the results of the optimal scheduler for the linear layout. There is a notable gap in the communication cost between the homogeneous and heterogeneous configurations. This is a result of the node capacities, where the heterogeneous cluster configuration has some larger nodes, 3 with capacity 6, which allows more of the tasks to be placed in fewer nodes, thereby reducing the communication cost when compared to the homogeneous configuration which uses smaller nodes with a capacity of 4.

Figures 5b and 5c show the results for diamond and star layouts. As can be seen from the figure, there is a difference between the communication costs found by T3-Scheduler and optimal scheduler for some problem sizes. This can be attributed to the different approaches, where T3-Scheduler tries to place all of the tasks within an operator along with its neighbouring operators which results in a sub-optimal solution. In comparison, the optimal scheduler selects tasks from multiple operators to achieve the optimal result.

It can also be seen from Figure 5 that the communication costs of both schedulers for diamond and star layouts are bigger than linear. The reason is that in the linear layout, more communicating task can be grouped and placed in the same node because of the shape of the layout when compared to the other two layouts. In the diamond layout, the source and sink operators only have 4 tasks each, which means that the number of communicating tasks that are able to be placed in the same node is limited. As communicating tasks from the source or sink are selected and assigned, along with the tasks from a few of the middle operators, this leaves a number of the middle operators unassigned. In turn, when these operators are to be assigned, there are no neighbouring operators left for them to be grouped with, as the source and sink were previously assigned. The same happens in the star layout where the middle operator only has 4 tasks and can be placed with a few tasks from the sink or source operators in the same node.

While this evaluation has shown that it is possible to use optimisation software to determine an optimal schedule for some small problem sizes, it is worth noting that it quickly becomes impractical as the problem size grows. To illustrate this, Figure 6 shows the resolution times for the micro-benchmark evaluations. From the figure, it can be seen that the resolution time begins to increase as the task size goes above about 20. For example, for the linear micro-benchmark on the heterogeneous cluster, shown in Figure 6a, the resolution time for a task size of 32 is 1114257 seconds, where 50GiB of RAM is used. This required the optimisation software to be run on the large, 64-core server, meaning it would take substantially longer on a more modestly powered system. While the resolution times for both diamond and star are not quite as high as for linear, they still quickly reach times that are impractical for scheduling, even for these small problem sizes. For 32 tasks, diamond and star have resolution times of 2471 and 6 seconds for the homogeneous cluster and 393 and 7 for the heterogeneous cluster, respectively.

Overall, the T3-Scheduler is able to efficiently allocate tasks to the nodes within the cluster with near optimal results for the diamond and star layouts, and equalling the optimal scheduler for the linear layout. This is an important result for practical application deployments, which commonly implement the linear layout. The heuristic approach adopted by T3-Scheduler is able to quickly find a near optimal solution, making it a good solution for scheduling in practice. We evaluate the practical performance of our implementation in the next section.

6. Experimental Results

We first provide a brief overview of Apache Storm for the sake of completeness, then we provide the evaluation of T3-Scheduler which is implemented within the Storm framework and discuss our experimental results. Storm is an open source, distributed, fault-tolerant and reliable system for processing streams of data. A Storm cluster has three sets of nodes:

- **Master node** runs a daemon called "Nimbus", which is responsible for distributing the tasks across the cluster, assigning tasks to the machines and monitoring for failures.
- **Worker nodes** run a daemon, called the "Supervisor", that receives the tasks assigned by the master node, and starts and stops the worker processes as dictated by Nimbus on the master node. Each worker process within a worker node runs a subset of the Storm topology.

- **ZooKeeper nodes** coordinate the communication between master and worker nodes. The overall size of the ZooKeeper cluster will depend on the scale of the Storm cluster, and the level of resilience desired.

A streaming application in Storm is called a *topology*. There are two types of processing elements/components in Storm: *spouts* and *bolts*. A spout is the source of a data stream and emits data, while a bolt is the computational unit used to process the data before emitting new data to the next bolt in the DAG. A stream is defined as an unbounded sequence of tuples where a tuple is a named list of values. Each component in Storm consists of a number of *executors* that can be run in parallel. In other words, each executor is an executing instance of the component's code that can be run in parallel with other executors of the same component. Each executor normally consists of one task. When a topology, consisting of spouts and bolts, is submitted to a Storm cluster, the tasks are grouped into a number of workers/JVMs. Each compute/worker node is configured with a number of slots/ports where each worker is assigned to one slot.

For every two communicating components, each having multiple tasks, Storm needs to determine which tasks of the receiving component should receive which stream coming from the emitting component. This is called *stream grouping*, and controls the flow of the stream between tasks. The most common stream groupings are discussed in the following list. More stream groupings can be found in the Storm documentation [4].

- **Shuffle grouping:** The target task in the component receiving a tuple is selected randomly such that each task of this component receives on average the same number of tuples emitted by the source component.

- **Fields grouping:** The target task in the component receiving a tuple is selected based on a specific field in the tuple such that the tuples with the same value for that specific field always go to the same task.

- **Global grouping:** The entire stream goes to the task with the lowest ID in the component receiving a tuple.

We implement T3-Scheduler[3] on Apache Storm 1.1.1, running on a heterogeneous Storm cluster, configured with one master node, one ZooKeeper

[3]`https://github.com/leilaeskandari/T3-Scheduler`

node and eight worker nodes. Ubuntu 16.04 LTS is installed on each node inside a VirtualBox VM. We use VMs as this allow us to specify the hardware configuration, with high and low capacity worker nodes. The high capacity nodes are configured with 4 cores and 4GiB of RAM with four slots per node, whereas the low capacity nodes have 2 cores and 2GiB of RAM with two slots per node. This node configurability allows us to use both homogeneous and heterogeneous clusters. Although a heterogeneous environment typically refers to different models of hardware with different configurations, we argue that the same hardware, where not all resources are available, can also be considered a heterogeneous environment. Each node has a 2.7GHz Intel Core i5-3330S processor and is connected to a 1Gbps network. We limit average CPU usage of each node to 80%, which prevents any node from becoming overloaded. T in Equation 3 provides a balance between the desire for more tasks per worker, and the reliability of execution. The failure of any single task within a worker will cause all of the other tasks within that worker to be restarted. As a result, any failure will cause tuples to be lost, where having fewer tasks per worker will reduce these losses. Therefore, we set the user-defined parameter of T to 5, as this had less significant losses. It is also worth noting that the results presented in this paper do not include any runs which had task failures.

We compare our scheduler with Aniello et al.'s 'Online scheduler' [13] (referred to as 'OLS') and R-Storm [14]. Unlike many other Storm schedulers, the OLS and R-Storm implementations are publicly available, allowing for a fair comparison. Furthermore, these two schedulers have the same optimisation goal as T3-Scheduler, which is to reduce the inter-node communication. We use the average throughput, defined as the average number of tuples executed in each bolt's task per 10 second period, and average execute latency for each task as our performance metrics.

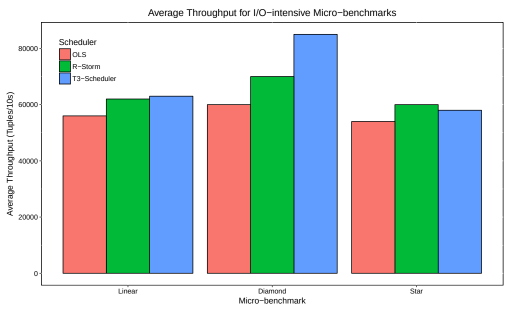
We use three micro-benchmarks and two real-world topologies with real data for our evaluation. Each experimental topology is run ten times for 650 seconds. Each topology is initially run with round-robin scheduling for 50-60s while T3-Scheduler monitors the execution. Rescheduling is then performed once for each experimental run, where Storm migrates the tasks to new cluster nodes. Currently, Storm uses a simple method of task migration, where the execution is stopped, allowing the tasks to be moved, before restarting execution with the new configuration. As a result of this delay, we present all experimental results starting at 150s, after rescheduling has completed. In our future work, we will investigate methods for smooth task migration, which does not stop the entire execution, which will reduce the overhead of rescheduling. Further, we will also continue work on run-time performance monitoring, investigating how workload characteristics change during execution and when rescheduling should be performed. In the following, we describe each experiment in detail and present a typical execution of each topology for each scheduler.

6.1. Micro-benchmarks

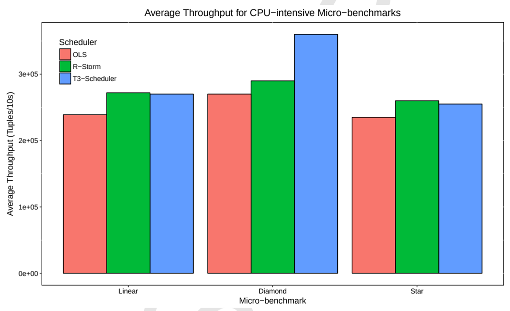
To evaluate T3-Scheduler, we first perform our experiments on the three micro-benchmarks previously presented in Section 5. We configure each of the spouts and bolts in the linear and diamond micro-benchmarks to have eight tasks. The star micro-benchmark is configured with four and eight tasks for the spout and bolts respectively. Each micro-benchmark is run in two different configurations: I/O-intensive and CPU-intensive, described as follows.

I/O-intensive: In this configuration, the throughput of the system is limited by the amount of communication between the nodes. We reduce the workload of each bolt by slowing the rate of the spout, so that each bolt has little processing to do, causing processing time to be limited by the network latency. We run each of the I/O-intensive micro-benchmarks on a heterogeneous cluster consisting of one high capacity node and two low capacity nodes. The small cluster size ensures each scheduler assigns tasks to both high and low capacity nodes, allowing us to evaluate how well highly communicating tasks are co-located together. This configuration also removes the impact of node consolidation, performed by T3-Scheduler and R-Storm. The results for the micro-benchmark execution for T3-Scheduler, R-Storm and OLS are presented in Figure 7a. As can be seen in the figure, T3-Scheduler is able to outperform OLS by 7-41% for all of the micro-benchmarks, while achieving a similar average throughput to R-Storm for the linear and star micro-benchmarks. For the diamond micro-benchmark, T3-Scheduler outperforms R-Storm by 20%, which is the result of a more efficient task placement of all of the bolts. R-Storm was unable to assign any of the tasks in one of the middle bolts to the same node as either the source or sink, increasing inter-node communication. It is also worth noting that these results for R-Storm were only achieved after tuning user-configured parameters, whereas T3-Scheduler does not have such a requirement.

CPU-intensive: In this configuration, the throughput of the system is limited by the CPU utilisation of each node. We increase the workload of



(a) I/O-intensive micro-benchmarks

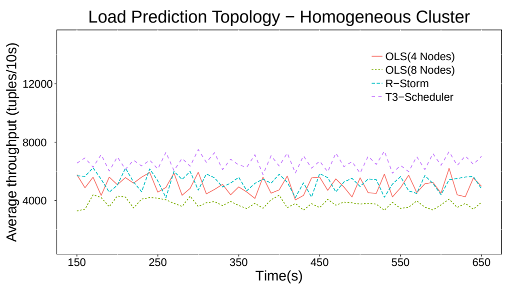


(b) CPU-intensive micro-benchmarks

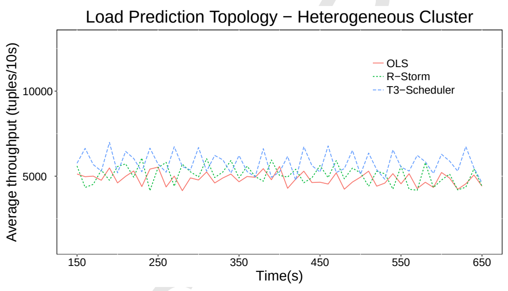
Figure 7: Throughput results of I/O-intensive and CPU-intensive linear, diamond and star micro-benchmarks

each bolt by supplying tuples at a faster rate, ensuring the bolts are fully loaded, resulting in a high CPU load. Each micro-benchmark is run on a heterogeneous cluster consisting of two high capacity nodes and four low capacity nodes. The results for the CPU-intensive micro-benchmarks are shown in Figure 7b. As can be seen from the figure, T3-Scheduler achieves similar results as the previous I/O-intensive configuration, outperforming OLS, with similar average throughput to R-Storm for linear and star, and higher for diamond. The throughput for each of the micro-benchmarks is much higher than was previously seen for the I/O-intensive configuration, placing a greater load on the CPUs, which is a result of a higher rate for the spouts. By placing more communicating tasks closer together, T3-Scheduler has an average throughput 24-33% higher than R-Storm and OLS for diamond micro-benchmark, showing the benefit of our group based assignment approach. T3-Scheduler can also outperform OLS by 8-12% for linear and star micro-benchmarks.

Overall, these results demonstrate the ability of T3-Scheduler to efficiently place more communicating tasks closer together by monitoring the execution, improving overall throughput. In comparison, R-Storm is an offline scheduler which is unable to respond to dynamic changes in the data transfer rates between tasks. While this is not a problem for the micro-benchmarks, due to their consistent communication pattern, this is a limitation for real-world applications. Further, R-Storm needs the user to configure the application requirements before execution which might not be practical or convenient. The best fit approach of OLS, assigns each task pair to the least loaded node, which is likely to spread the communicating tasks over the

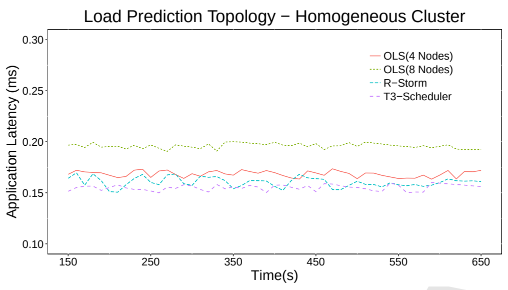


(a) Homogeneous cluster

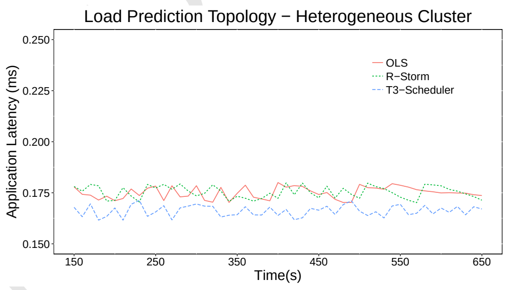


(b) Heterogeneous cluster

Figure 8: Throughput results of smart home load prediction topology



(a) Homogeneous cluster



(b) Heterogeneous cluster

Figure 9: Latency results of smart home load prediction topology

compute nodes.

6.2. Load Prediction Application for Smart Homes

This topology is based on the first query of the DEBS 2014 Grand Challenge [48]. The query predicts the future load for each house and each individual smart plug based on current load measurements and a slice-based prediction model built on historical data. The streaming application implementing this query has a diamond shape, with one spout, two intermediate bolts and one sink bolt as follows:

- Source spout: The spout splits the comma separated records and sends the data to each of the load prediction bolts with shuffle grouping.
- Plug Prediction bolt: At the start of each slice, the sum of the load for all the events which their timestamps are within the slice is calculated. The average load for the current time slice is calculated at the end of

slice and kept in a data structure to calculate the median for future predictions. Then, predicted load for the two time slices from the current time slice is calculated, using the current average load and median of the average load for the same time slices over the past days. The output is sent to the sink bolt with fields grouping.

- House Prediction bolt: For each house, the predicted load is calculated as the sum of all the predicted loads for each smart plug within the house. The output is sent to the sink bolt with fields grouping.
- Sink bolt: A sink bolt formats the output such that depending on the tuples received by the bolt, it either emits the house prediction value along with house ID and predicted time or the plug prediction value with hierarchical identifier for the plug and predicted time.

Our application is configured with a time slice of 15 minutes, with the spout and each of the bolts having 10 tasks. We use a Redis server to store the smart plug records. We first run the topology on a homogeneous cluster with 8 high capacity nodes using T3-Scheduler, R-Storm and OLS. Figure 8a shows the experimental results.

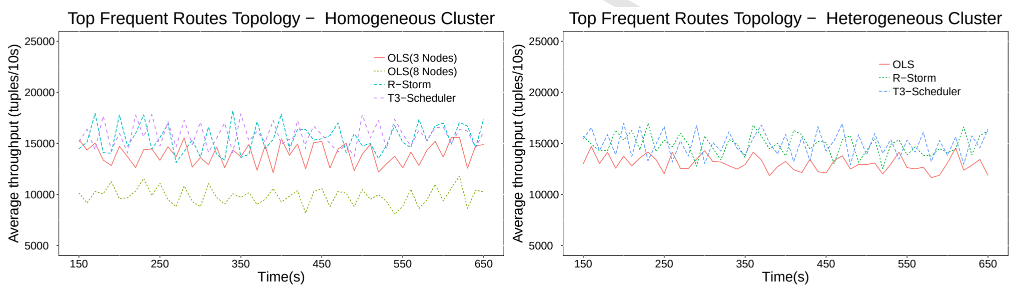
As can be seen from the figure, OLS and R-Storm have an average throughput of 3700 and 5200 respectively, while T3-Scheduler has an average throughput of 6600. This represents an improvement of 26% and 78% over R-Storm and OLS respectively. The lower throughput of OLS is because of the best fit approach that is used, which results in tasks being spread across all the nodes as it assigns each task pair to the least loaded node. While R-Storm co-locates tasks from multiple components reasonably well, the Plug Load component is the exception, where its tasks are assigned separately from all other tasks, resulting in a lower throughput. It also fails to consider the data transfer rates between tasks due to offline scheduling.

On average, R-Storm and T3-Scheduler use 4 out of 8 nodes while OLS uses all 8 nodes because of its best fit approach. Therefore, for a fairer comparison we rerun OLS on 4 nodes, removing the impact of node consolidation. As can be seen in Figure 8a, the throughput of OLS increases when run on fewer nodes, however it is still lower than the other two schedulers.

We also run the topology on a heterogeneous cluster, consisting of 2 high capacity nodes and 4 low capacity nodes, using all three schedulers. This is intended to demonstrate which of the schedulers is better able to place the highly communicating tasks within the limited number of high capacity nodes, reducing inter-node communication. The results are shown in Figure 8b. As can be seen, T3-Scheduler can outperform R-Storm and OLS because of its task selection. It is worth noting that the throughput is lower for all schedulers than homogeneous setting, as there is more inter-node communication.

Figures 9a and 9b show the execute latency for the topology run by the three schedulers. From the figures, it can be seen that as the execute latency is reduced, we can achieve higher throughput. Overall, the experimental results show that T3-Scheduler can achieve a better performance as a result of a more efficient task placement.

6.3. Top Frequent Routes in NYC Taxi Data



(a) Homogeneous cluster

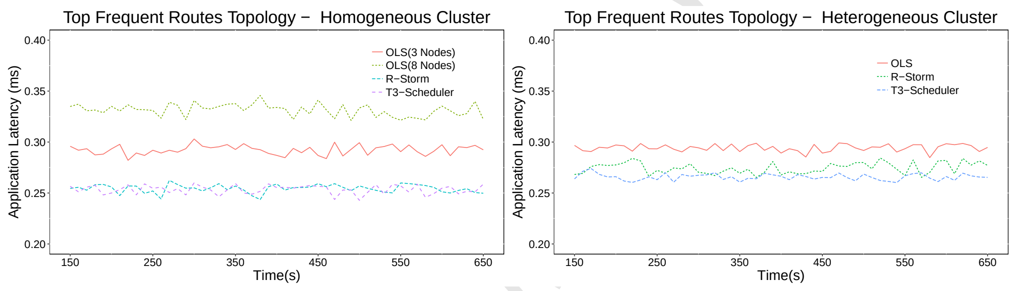
(b) Heterogeneous cluster

Figure 10: Throughput results of top frequent routes topology

This topology is based on the first query of the DEBS 2015 Grand Challenge [49]. The query is to find the top 10 most frequent routes of New York taxis for the last 30 minutes using the 2013 dataset. The layout of this topology has a linear shape with one spout and four bolts as follows:

- Source spout: The spout reads the records of each taxi trip from the dataset and sends the data to the PreProcess bolt with shuffle grouping.
- PreProcess bolt: The PreProcess bolt processes each trip record in order to find the start and end cell numbers based on longitude and latitude coordinates of the pickup and drop off locations. This bolt then emits the routes to Rolling Count bolt with shuffle grouping.
- Rolling Count bolt: This bolt counts the occurrence of each route using a rolling counter, implemented with a sliding window. The data is then passed to the Intermediate Rank bolt with fields grouping.

- Intermediate Rank bolt: This bolt has multiple tasks and is used to distribute the load coming from the Rolling Count bolt. It emits the intermediate ranks to the Final Rank bolt with global grouping. Each task of the Intermediate Rank bolt finds 10 top frequent routes for a specified window.
- Final Rank bolt: This bolt has only one task, aggregating the incoming intermediate rankings from the Intermediate Rank bolt into a final rank.



(a) Homogeneous cluster

(b) Heterogeneous cluster

Figure 11: Latency results of top frequent routes topology

The numbers of tasks for each of the Source, PreProcess, Rolling Count, Intermediate Rank and Final Rank bolts are 16, 16, 8, 4 and 1 respectively. We use a Redis server to store the trip records. We simulate a replay of the taxi data, by setting a simulation time that is a constant ratio with real time. This ratio is set so that 1 minute is equal to 0.1 second of time in our experiment—thus the sliding window in Rolling Count bolt is 3 seconds.

Figure 10a shows the throughput for this topology using the three schedulers, run on an 8 node homogeneous cluster. As can be seen from the figure, the average throughput for T3-Scheduler, R-Storm and OLS is 15700, 15500 and 9800 respectively. Again, OLS suffers from the problem of spreading the tasks across all 8 nodes, while both T3-Scheduler and R-Storm only use 3 nodes. When running the application with OLS on a 3-node cluster, the average throughput is 13900, which is still lower than the other two schedulers.

We further evaluate each of the schedulers on a heterogeneous cluster with 2 high capacity nodes and 2 low capacity nodes. From the results, shown in Figure 10b, it can be seen that T3-Scheduler has an average throughput of

14900 while R-Storm and OLS have an average of 14800 and 13000 respectively. Finally, Figures 11 show the latency for each scheduler running on the homogeneous and heterogeneous cluster configurations.

Although, T3-Scheduler does not provide a significant improvement over R-Storm for Top Frequent Routes topology, it is worth noting that R-Storm requires a considerable amount of tuning by the user. In comparison, T3-Scheduler is able to achieve the same results without requiring such extensive tuning. Both R-Storm and T3-Scheduler outperform OLS as it takes a greedy best fit approach which fails to see the whole communication pattern and co-locate the highly communicating tasks efficiently. As seen in the Smart Home topology, which has a diamond shape, R-Storm is unable to reliably find the communicating tasks because of its task selection method. While R-Storm operates offline and does not have any rescheduling overhead, it is unable to dynamically react to changes in the communication pattern. In comparison, T3-Scheduler incurs an overhead for rescheduling, but is better able to find highly communicating tasks by monitoring task execution and can react accordingly. The oscillating fluctuations in throughput that can be seen in the experimental results are similar to the fluctuations seen in previous work [14]. From the detailed log files, there are no unexpected characteristics, so we are not concerned by this behaviour.

7. Conclusions and Future Work

T3-Scheduler efficiently places tasks within the compute nodes, which leads to lower amounts of inter-node and intra-node communication. We evaluated T3-Scheduler using three micro-benchmarks and two real-world streaming applications. The experimental results showed that T3-Scheduler outperformed Aniello et al.'s state-of-the-art online scheduler, improving throughput by 12-32% for the Smart Home and Top Frequent Routes real-world applications. Further, T3-Scheduler outperformed R-Storm in Smart Home topology by 26% and achieved a similar average throughput for Top Trending Routes without the need for tuning.

As future work, we will evaluate T3-Scheduler on a larger cluster with a larger set of real-world applications. We will also continue work on run-time performance monitoring, investigating how workload characteristics change during execution and when rescheduling should be performed. We will further investigate methods to minimise the overhead for rescheduling such as efficient and reliable state migration of tasks when rescheduling. We will also consider other metrics in our heuristic, including QoS, network bandwidth and memory.

References

[1] L. Eskandari, J. Mair, Z. Huang, D. Eyers, A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster, in: Auto-DaSP Workshop in European Conference on Parallel Processing, Springer, 2017, pp. 68–79.

[2] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: Distributed stream computing platform, in: Proceedings of 2010 International Conference on Data Mining Workshops (ICDMW), IEEE, 2010, pp. 170–177.

[3] Apache Flink, `https://flink.apache.org/`.

[4] Apache Storm, `https://storm.apache.org/`.

[5] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja, Twitter Heron: Stream processing at scale, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 239–250.

[6] T. Heinze, L. Aniello, L. Querzoni, Z. Jerzak, Tutorial: Cloud-based data stream processing, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, ACM, 2014, pp. 238–245.

[7] F. Bonomi, R. Milito, P. Natarajan, J. Zhu, Fog computing: A platform for internet of things and analytics, in: Big data and Internet of Things: A Roadmap for Smart Environments, Springer, 2014, pp. 169–186.

[8] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the first edition of the MCC Workshop on Mobile Cloud Computing, ACM, 2012, pp. 13–16.

[9] S. Yang, IoT stream processing and analytics in the fog, IEEE Communications Magazine 55 (8) (2017) 21–27.

[10] A. Shan, Heterogeneous processing: a strategy for augmenting Moore's law, Linux Journal 2006 (142) (2006) 7.

[11] S. Chakravarthy, Q. Jiang, Stream data processing: a quality of service perspective: modeling, scheduling, load shedding, and complex event processing, Vol. 36, Springer Science & Business Media, 2009.

[12] IBM CPLEX, `https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/index.html` (1988).

[13] L. Aniello, R. Baldoni, L. Querzoni, Adaptive online scheduling in Storm, in: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems, 2013, pp. 207–218.

[14] B. Peng, M. Hosseini, Z. Hong, R. Farivar, R. Campbell, R-Storm: Resource-aware scheduling in Storm, in: Proceedings of the 16th Annual Middleware Conference, ACM, 2015, pp. 149–161.

[15] W. W. Chu, L. J. Holloway, M.-T. Lan, K. Efe, Task allocation in distributed data processing, IEEE Computer 13 (11) (1980) 57–69.

[16] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator placement for distributed stream processing applications, in: Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems, ACM, 2016, pp. 69–80.

[17] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator replication and placement for distributed stream processing systems, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 44, ACM, 2017, pp. 11–22.

[18] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, P. Pietzuch, SQPR: Stream query planning with reuse, in: Proceedings of 27th IEEE International Conference on Data Engineering (ICDE), IEEE, 2011, pp. 840–851.

[19] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, L. Fleischer, SODA: an optimizing scheduler for large-scale stream-based distributed computer systems, in: Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2008, pp. 306–325.

[20] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, C. Venkatramani, SPC: A distributed, scalable platform for data mining, in: Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms, ACM, 2006, pp. 27–37.

[21] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, Z. Zhang, DRS: Dynamic Resource Scheduling for real-time analytics over fast streams, in: Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2015, pp. 411–420.

[22] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, B. Gedik, Cola: Optimizing stream processing applications via graph partitioning, in: ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2009, pp. 308–327.

[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg, Quincy: fair scheduling for distributed computing clusters, in: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, ACM, 2009, pp. 261–276.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: distributed data-parallel programs from sequential building blocks, in: ACM SIGOPS Operating Systems Review, Vol. 41, ACM, 2007, pp. 59–72.

[25] L. Fischer, A. Bernstein, Workload scheduling in distributed stream processors using graph partitioning, in: Proceedings of IEEE International Conference on Big Data (Big Data), 2015, IEEE, 2015, pp. 124–133.

[26] J. Ghaderi, S. Shakkottai, R. Srikant, Scheduling storms and streams in the cloud, ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) 1 (4) (2016) 14.

[27] L. Eskandari, Z. Huang, D. Eyers, P-Scheduler: adaptive hierarchical scheduling in Apache Storm, in: Proceedings of the Australasian Computer Science Week Multiconference, ACM, 2016, p. 26.

[28] D. Sun, G. Zhang, S. Yang, W. Zheng, S. U. Khan, K. Li, Re-Stream: Real-time and energy-efficient resource scheduling in big data stream computing environments, Information Sciences 319 (2015) 92–112.

[29] M. Caneill, A. El Rheddane, V. Leroy, N. De Palma, Locality-aware routing in stateful streaming applications, in: Proceedings of the 17th International Middleware Conference, ACM, 2016, p. 4.

[30] J. Xu, Z. Chen, J. Tang, S. Su, T-Storm: Traffic-aware online scheduling in Storm, in: Proceedings of the 34th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2014, pp. 535–544.

[31] A. Chatzistergiou, S. D. Viglas, Fast heuristics for near-optimal task allocation in data stream processing over clusters, in: Proceedings of the 23rd ACM International Conference on Information and Knowledge Management, ACM, 2014, pp. 1579–1588.

[32] M. Rychlỳ, P. Škoda, P. Smrž, Heterogeneity–aware scheduler for stream processing frameworks, International Journal of Big Data Intelligence 2 (2) (2015) 70–80.

[33] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Distributed qos-aware scheduling in storm, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM, 2015, pp. 344–347.

[34] Z. Abrams, J. Liu, Greedy is good: On service tree placement for in-network stream processing, in: 26th IEEE International Conference on Distributed Computing Systems, 2006. ICDCS 2006., IEEE, 2006, pp. 72–72.

[35] G. T. Lakshmanan, Y. Li, R. Strom, Placement strategies for internet-scale data stream systems, IEEE Internet Computing 12 (6) (2008) 50–60.

[36] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the 22nd International Conference on Data Engineering, ICDE., IEEE, 2006, pp. 49–49.

[37] S. Rizou, F. Dürr, K. Rothermel, Solving the multi-operator placement problem in large-scale operator networks., in: Proceedings of 19th International Conference on Computer Communication Networks (ICCCN), IEEE, 2010, pp. 1–6.

[38] M. R. Gary, D. S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, Journal of Symbolic Logic 48 (2) (1983) 498–500.

[39] U. Srivastava, K. Munagala, J. Widom, Operator placement for in-network stream query processing, in: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems, ACM, 2005, pp. 250–258.

[40] R. Eidenbenz, T. Locher, Task allocation for distributed stream processing, in: Proceedings of 35th IEEE International Conference on Computer Communications, IEEE INFOCOM, IEEE, 2016, pp. 1–9.

[41] G. Karypis, V. Kumar, Multilevel K-way partitioning scheme for irregular graphs, Journal of Parallel and Distributed Computing 48 (1) (1998) 96–129.

[42] METIS, http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[43] M. Tanaka, O. Tatebe, Workflow scheduling to minimize data movement using multi-constraint graph partitioning, in: Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE Computer Society, 2012, pp. 65–72.

[44] L. Wang, Y. Xiao, B. Shao, H. Wang, How to partition a billion-node graph, in: Proceedings of IEEE 30th International Conference on Data Engineering (ICDE), IEEE, 2014, pp. 568–579.

[45] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, J. McPherson, From think like a vertex to think like a graph, Proceedings of the VLDB Endowment 7 (3) (2013) 193–204.

[46] J. Huang, D. J. Abadi, K. Ren, Scalable SPARQL querying of large RDF graphs, Proceedings of the VLDB Endowment 4 (11) (2011) 1123–1134.

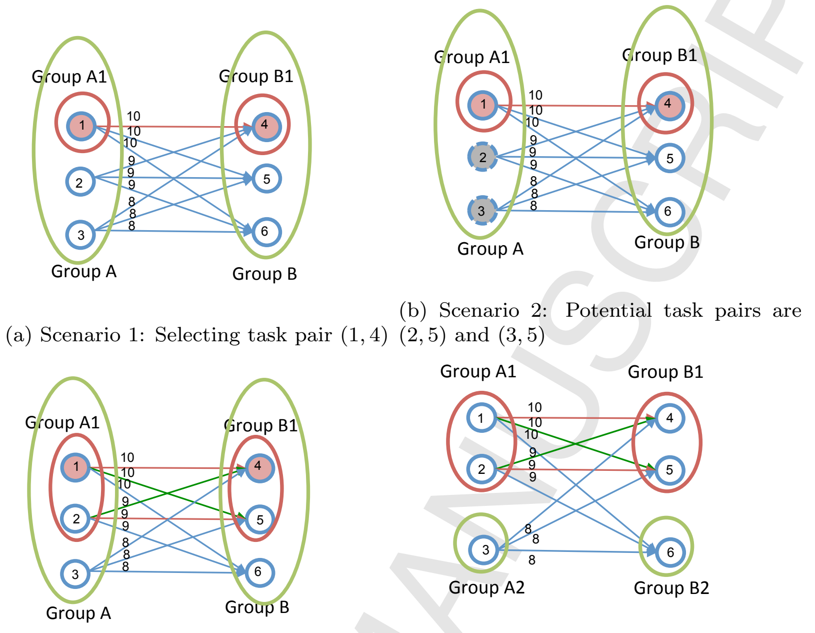
[47] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, Proceedings of the VLDB Endowment 5 (8) (2012) 716–727.

[48] Z. Jerzak, H. Ziekow, The DEBS 2014 Grand Challenge, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, ACM, 2014, pp. 266–269.

[49] Z. Jerzak, H. Ziekow, The DEBS 2015 Grand Challenge, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, ACM, 2015, pp. 266–268.

Appendix A. An example of group pair partitioning

We present the step by step process of partitioning the group pair (A, B) in the example presented In Section 4.4.2. We assume that (A, B) in Figure 3 has an aggregated data transfer rate of 81 (sum of data transfer rate for task pairs) and 3α load for each group (α load for each task) has been selected to be assigned to a node with available capacity of 4α. These numbers have been selected to simplify the explanation of the example. Since the sum of tasks' load for group A and group B is greater than the node's capacity, the group pair (A, B) needs to be partitioned. First, the task pair with the highest data transfer rate should be selected. In this example, there are three task pairs $(1, 4)$, $(1, 5)$ and $(1, 6)$ with a data transfer rate 10, which is the highest. None of the tasks in these task pairs have previously been selected (scenario 1), therefore we select task pair $(1, 4)$ at random and assign it to (A_1, B_1), as shown in Figure A.12a. The next highest data transfer rates for the remaining task pairs belong to $(1, 5)$ and $(1, 6)$. However, one task of both task pairs, *task* 1, has been selected before and is already in the (A_1, B_1) (scenario 2). Having the same condition for both task pairs, task pair $(1, 5)$ is selected at random for further processing. As we want to assign two new tasks to (A_1, B_1) which means the tasks that have not been selected previously, we find all the task pairs in (A, B) that include *task* 5 and a task not marked as selected. There are two possible task pairs $(2, 5)$ and $(3, 5)$, which meet this criteria, shown in Figure A.12b. Among all the potential task pairs, we pick the task pair with the highest rate, which can fit in the compute node. In this example, task pair $(2, 5)$ is selected, as shown in Figure A.12c. After each selection the load and capacity is checked. The node is full at this point, so the rest of the tasks which are not marked as selected, form (A_2, B_2) group pair, shown in Figure A.12d, which will be assigned to another node by following the T3-Scheduler algorithm.



(a) Scenario 1: Selecting task pair $(1, 4)$

(b) Scenario 2: Potential task pairs are $(2, 5)$ and $(3, 5)$

(c) Selecting task pair $(2, 5)$

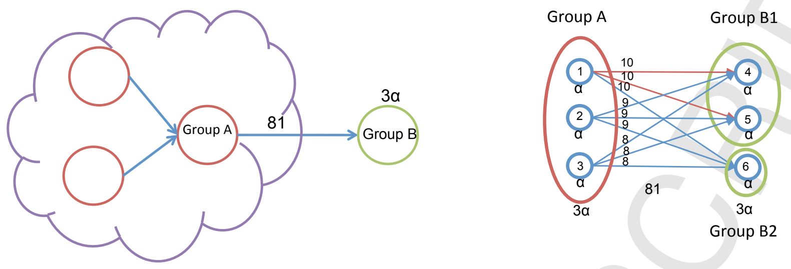
(d) Forming group pairs (A_1, B_1) and (A_2, B_2)

Figure A.12: An example of Group Pair Partitioning by T3-Scheduler

Appendix B. An example of single group partitioning

We present applying the single group partitioning algorithm to the example shown in Section 4.4.3. We assume that the sub-graph shown in red in Figure B.13a has been selected for assignment to the current node. Since the node still has some remaining capacity, Group B is selected as the neighbour with the highest weight for assignment for expansion of the sub-graph. We assume that (A, B) has an aggregated data transfer rate of 81. The remaining capacity of the node is 2α, but the size of the selected group is 3α. These numbers have been selected to simplify the explanation of the example. Therefore, Group B is too large for the remaining capacity and requires single group partitioning such that one group can fit in the remaining capacity.

To do this, we pick the tasks with the highest transfer rate which are connected to the sub-graph. In our example, the task pairs $(1, 4)$, $(1, 5)$ and $(1, 6)$ have the highest rate, so we pick $(1, 4)$ at random. The remaining capacity is α now, so we pick $(1, 5)$ next. The node is full now and by picking



(a) Remaning capacity is 2α (b) Selecting tasks 4 and 5

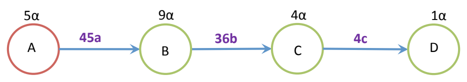
Figure B.13: An example of single group partitioning by T3-Scheduler

the task pairs with the highest transfer rates we have reduced the edge cuts between the sub-graphs which are to be assigned to the different nodes.

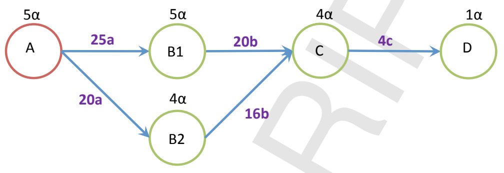
Appendix C. Applying T3-Scheduler to an example

In this section, we apply our algorithm to the top trending topics presented in Section 3. In this example, we have a heterogeneous cluster, consisting of five compute nodes of varying sizes. It is assumed that the task load for all the tasks in the application is the same and equal to α. This is a simplifying assumption we make in this current example, where each task in the application is the same size, however, T3-Scheduler is not dependent upon this assumption and is capable of scheduling tasks of varying sizes, as previously discussed. Based on this assumption, the capacity of each node is defined as the number of tasks which are able to be allocated to each node multiplied by α. *Node* 1, *node* 2, *node* 3, *node* 4 and *node* 5 have the capacities of 6α, 4α, 10α, 6α and 5α respectively. It is assumed that *node* 3 is configured to have up to four workers, whereas the rest of the nodes can have up to two workers.

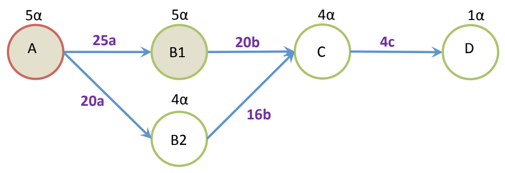
The first step performed by the T3-Scheduler is to monitor the execution and collect the tasks' load and data transfer rate between the tasks, previously described in Section 4.1. We further assume that for the top trending topics application in this example, the communication rates between communicating operators obtained from monitoring step reduce at each subsequent step. This is a valid assumption as less data go through at each operator. In this example, we assume weights a, b and c are obtained for each task pair in (A, B), (B, C) and (C, D) group pairs respectively, assuming that $a > b > c$. Therefore, while the exact communication rates are not known,



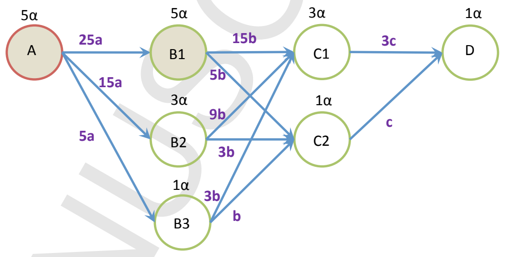
(a) Simplified Graph built by T3-Scheduler for top trending topics application



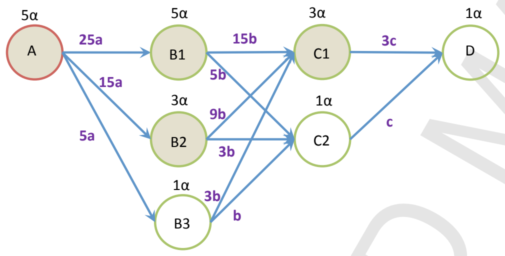
(b) Updated simplified graph after first group pair partitioning



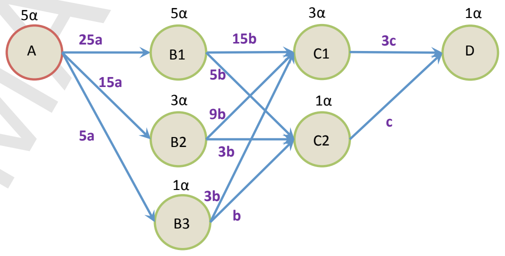
(c) Assigning (A, B_1) to *node* 3



(d) Updated simplified graph after second group pair partitioning



(e) Assigning (B_2, C_1) to *node* 1



(f) Assigning (B_3, C_2) and (C_2, D) to *node* 4

Figure C.14: Applying T3-Scheduler to top trending topics application

these weights allow us to present the way our algorithm works. To further simplify this current example, we assume that the data transfer rate between the task pairs of each group pair is the same. This assumption is based upon the lack of skew in the topics. However, our algorithm can handle different data transfer rates, as previously discussed, the current example making these assumptions are strictly for illustrative purposes.

As the second step, a simplified graph is constructed based on these statistics such that each vertex weight is calculated by the summation of the task loads in each operator. Also, the weight for an edge connecting two groups is calculated by the aggregation of the data transfer rates of all the task pairs. The simplified graph can be seen in the Figure C.14a.

Node selection in T3-Scheduler, begins by selecting the largest capacity node from the heterogeneous cluster. In this case, the selected node is *node* 3, with a capacity of 10α. Having selected a node, in the first level of scheduling, we then need to locate a starting point within the simplified graph. Group pair (A, B), consisting of the Emit Topics and Rolling Count groups, has the highest weight of $45a$, as shown in Figure C.14a. After evaluation of this group pair, it can be seen that the group pair cannot fit in the selected node because the whole group pair has 14 tasks, while *node* 3 has a capacity of only 10 tasks, requiring a group pair partitioning. The results of performing fine-grained group pair partitioning algorithm, as explained in Section 4.4.3, are shown in Figure C.14b. This then allows the smaller group pair, (A, B_1), to be assigned to *node* 3. Further, B_2 can be assigned to another node along with its communicating neighbours. Following this, the graph weights and edges are updated, as they are continually updated at each step after each partitioning. *node* 3 is full at this point with (A, B_1), shown in Figure C.14c, so we move to the next node.

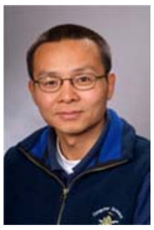
The next node selected by node selection algorithm, is *node* 1, which has a capacity of 6α. The starting point for this node is (B_2, C) with the highest weight of $16b$ and including 8 tasks, however this group pair is unable to fit within *node* 1 with the capacity of 6 tasks. Therefore, the group pair is partitioned and the results are shown in Figure C.14d. After group pair partitioning, (B_2, C_1) with the weight of $9b$ is selected for the starting point of *node* 1. The graph weights are then updated. The immediate neighbours for this pair includes B_3, C_2 and D. However, *node* 1 is full now, Figure C.14e, and there is no further processing on this node.

The next available node selected by node selection algorithm is *node* 4 with capacity of 6α. The starting point of (B_3, C_2) with the weight of b is found for *node* 4. As this node still has some remaining capacity, we find and process the immediate neighbours to find the group with the highest weight which is connected to the groups already assigned to this node. The next group to be picked is D, which is the only immediate neighbouring group. After assigning D to the node, the first-level scheduling finishes as all the groups are assigned, as shown in Figure C.14f.

For the second level of scheduling, T3-Scheduler partitions the sub-graph assigned to each node into the number of required workers. We assume that T is set to 6. The number of required workers for *node* 3, *node* 1 and *node* 4 is 2, 1 and 1 respectively. Therefore, there is a need for further partitioning of the sub-graph assigned to *node* 3. After performing the partitioning by METIS, each part is assigned to one worker and T3-Scheduler finishes here.

Leila Eskandari received a BSc degree from Ferdowsi University of Mashhad in Computer Engineering and a MSc degree from Sharif University of Technology in Network Engineering, Iran. She is currently completing a PhD in Computer Science at the University of Otago, New Zealand. Her research interests include big data processing, distributed computing and computer networks.

Jason Mair received his PhD in Computer Science in 2015 and a BSc Honors degree in 2010 from the University of Otago, New Zealand. His research interests include multicore architectures, green computing, big data processing and distributed computing.



Zhiyi Huang received the BSc degree in 1986 and the PhD degree in 1992 in computer science from the National University of Defense Technology (NUDT) in China. He is an Associate Professor at the Department of Computer Science, University of Otago, New Zealand. He was a visiting professor at EPFL (Swiss Federal Institute of Technology Lausanne) and Tsinghua University in 2005, and a visiting scientist at MIT CSAIL in 2009. His research fields include parallel/distributed computing, multicore architectures, operating systems, green computing, cluster/grid/cloud computing, high-performance computing, and computer networks

David Eyers received his PhD in Computer Science in 2006 from the University of Cambridge, UK, having previously attained his BE in Computer Engineering from the University of New South Wales in Sydney, Australia. He is a Senior Lecturer at the Department of Computer Science at the University of Otago in New Zealand, and a visiting research fellow at the University of Cambridge Computer Laboratory. His has broad research interests including green computing, information flow control, network security, and distributed and cloud computing.

- Proposes a topology and traffic aware two-level scheduling for a heterogeneous cluster
- Forms sub-graphs of highly communicating tasks, relative to the size of the nodes
- Demonstrates a near-optimal placement
- Outperforms a state of the art task placement strategy