I-Scheduler: Iterative scheduling for distributed stream processing systems

Leila Eskandari, Jason Mair, Zhiyi Huang, David Eyers

PII: S0167-739X(20)33035-1
DOI: https://doi.org/10.1016/j.future.2020.11.011
Reference: FUTURE 5919

To appear in: *Future Generation Computer Systems*

Received date : 16 May 2020
Revised date : 11 October 2020
Accepted date : 15 November 2020

Highlights

- I-Scheduler is an iterative K-way graph partitioning-based algorithm.
- It reduces the task graph size so that it can be solved by optimisation software.
- An iterative heuristic fall back algorithm is used to solve large problem sizes.

I-Scheduler: Iterative Scheduling for Distributed Stream Processing Systems

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

Task allocation in Data Stream Processing Systems (DSPSs) has a significant impact on performance metrics such as data processing latency and system throughput. An application processed by DSPSs can be represented as a Directed Acyclic Graph (DAG), where each vertex represents a task and the edges show the dataflow between the tasks. Task allocation can be defined as the assignment of the vertices in the DAG to the physical compute nodes such that the data movement between the nodes is minimised. Finding an optimal task placement for DSPSs is NP-hard. Thus, approximate scheduling approaches are required to improve the performance of DSPSs. In this paper, we propose a heuristic scheduling algorithm which reliably and efficiently finds highly communicating tasks by exploiting graph partitioning algorithms and a mathematical optimisation software package. We evaluate the communication cost of our method using three micro-benchmarks, showing that we can achieve results that are close to optimal. We further compare our scheduler with two popular existing schedulers, R-Storm and Aniello et al.'s 'Online scheduler' using two real-world applications. Our experimental results show that our proposed scheduler outperforms R-Storm, increasing throughput by up to 30%, and improves on the Online scheduler by 20–86% as a result of finding a more efficient schedule.[1]

*Primary contact: Phone: +64 3 479 8498 Fax: +64 3 479 8529
[1]This work is an extension of I-Scheduler [1]

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

1. Introduction

In the era of big data, with streaming applications such as social media, surveillance monitoring and real-time search generating large volumes of data, efficient Data Stream Processing Systems (DSPSs) have become essential. More than 30,000 gigabytes of data is generated every second and the rate is accelerating [2]. According to IBM,[2] 90% of the data that existed in 2012 was created in the two years prior. These data sources need to be analysed to gain insights, and find trends such as determining the most frequent events for a continuous dataflow occurring over a certain period of time. DSPSs are designed to process such dataflows, by operating on data streams, which are a continuous, unbounded sequence of data items, with a number of data attributes, processed in the order in which they arrive. In comparison, batch processing systems store the data before performing *ad hoc* queries, which is not suited for real-time analysis. A general purpose DSPS faces a number of competing challenges, such as task allocation, scalability, fault tolerance, QoS, parallelism degree, and state management, among others. It is not possible to optimise for all of the challenges at the same time as there will be tradeoffs. The specific streaming application requirements determine which challenges need to be addressed. While each of these challenges are current areas of research, we focus on scheduling in this paper as low latency response times are a consistent priority across many streaming applications. The scheduling policy determines how tasks are distributed in the DSPS, which can have a significant impact on the performance metrics of the system such as tuple latency (the time taken to process a tuple) and system throughput (the number of tuples processed in a given time) [3]. A scheduling policy needs to strike a balance between system performance, the use of system resources and run-time overhead.

Finding an optimal placement for DSPSs is NP-hard [4, 5, 6]. Thus, approximate approaches are required to improve the performance of DSPSs [7].

[2]`https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/`

An efficient task scheduler will adapt to changes in the communication pattern of a streaming application, ensuring that the communication between compute nodes, referred to as inter-node communication, is minimised. Specifically, by placing highly communicating tasks on the same node, communication between compute nodes can be reduced [8, 9, 10, 11]. The term "highly communicating tasks", which is used throughout this paper, refers to a pair or group of tasks which exchange a larger amount of data than other neighbouring tasks. Additionally, prioritising the use of higher capacity compute nodes allows more highly communicating tasks to be co-located within a compute node, requiring fewer nodes to be used, which helps to further reduce inter-node communication. To achieve this, the scheduler monitors the run-time communication of a streaming application, logging the communication rates between tasks and tasks' load, which is then used when rescheduling.

Existing work on task schedulers that aim to minimise the inter-node communication have a number of limitations. Firstly, the compute nodes might be underutilised, which results in using more compute nodes than required. Secondly, many of the schedulers are not designed for heterogeneous clusters, which is an important requirement for many deployments, as they evolve over time, as new hardware is added. Further, a multi-user homogeneous cluster where not all of the system resources are available can be viewed as a heterogeneous system. Finally, offline schedulers are incapable of adapting to the run-time changes in the traffic patterns of streaming applications. In this paper, we aim to address these limitations and propose I-Scheduler, for DAG-based DSPSs, which partitions the DAG in order to minimise the communication between each part, such that inter-node communication is minimised when each part is assigned to a compute node. The contributions of this paper are summarised as follows:

- We propose I-Scheduler, for homogeneous and heterogeneous DSPSs, which reduces the size of the task graph by fusing highly communicating tasks, allowing mathematical optimisation software to be used to find an efficient task assignment. A fallback heuristic is also proposed for cases where the optimisation software cannot be used, which iteratively partitions the application graph based on the capacity of the heterogeneous nodes and assigns each partition to a node with a relative capacity.

- We evaluate the communication cost of I-Scheduler by comparing it to a theoretically optimal scheduler, implemented in CPLEX, when run on three micro-benchmarks, each representing a different communication pattern. The results show that I-Scheduler can achieve results that are close to optimal in a number of different cluster configurations.

- We implement the proposed scheduler in Apache Storm 1.1.1 and through experimental results, show that our proposed scheduler can outperform state of the art R-Storm [9] and Aniello et al.'s 'Online scheduler' [8] (for brevity we refer to this scheduler as OLS in this paper). The results show that I-Scheduler outperforms OLS, increasing throughput by 20–86% and R-Storm by 3–30% for the real-world applications.

The rest of this paper is organised as follows. In Section 2, we discuss key related work. We then present the problem definition in Section 3. Section 4 describes our proposed scheduler, which is then followed by a comparison with an optimal scheduler in Section 5. Section 6 evaluates our proposed scheduler with two real world applications. Finally, Section 7 concludes the paper.

2. Related Work

There is a broad range of research on task placement in DSPSs with different optimisation goals. Generally, there are three main approaches to tackle each optimisation problem [12]: mathematical programming, graph-based approximation and heuristics. In the following, we discuss a number of scheduling algorithms that use different approaches to meet specific optimisation goals.

In the mathematical programming approach, the scheduling problem is formulated as an optimisation problem and solved using mathematical programming techniques. Cardellini et al. [13] formulate the optimal schedule using integer linear programming. Their formulation considers the heterogeneity of the computing and networking resources when finding the optimal solution for a small number of tasks. The same authors propose a model in [14] to find the optimal number of replicas for an operator in a DSPS. Both optimal solutions become impractical for larger problems as the resolution time grows exponentially with the problem size. Eidenbenz et al. [6] provide a theoretical analysis of task allocation and prove its NP-hardness. They propose an approach to compute optimal resource assignments for each task, where the application graph is from the class of series-parallel decomposable graphs. Their algorithm assumes that the computational cost of the tasks dominates the communication cost, and achieves an approximate solution for resources with uniform capacities and bandwidths. SODA [15], designed for System S [16], uses mixed integer programming to model the placement of Processing Elements (PEs) on the processing nodes for admitted jobs. A heuristic approach is used as backup, in case the optimal solution fails. Jiang et al. [17] propose a max-min fairness approach for scheduling multiple streaming applications on a heterogeneous cluster, by formulating the problem as a mixed 0-1 integer program. As this problem is NP-hard, they propose an approximate approach by converting the non-convex constraint into linear constraints using linearisation and reformulation techniques. Wang et al. [18] model the system as G/G/M priority queues using the Allen-Cunneen formula to estimate the average end-to-end response times. They formulate the resource allocation as an optimisation problem, with three cost functions: reducing QoS violation, maximising CPU utilisation and minimising overhead. The state of the underlying platform is continuously monitored and the allocation of resources is periodically adjusted to achieve the optimisation goals. Each of these mathematical techniques attempt to solve a different optimisation goal. However, as the problem is NP-hard, many of these techniques rely on heuristic methods as a fallback, in the event of being unable to find a solution within a feasible amount of time. This is a key limitation of such approaches, where the intended method of finding an optimal solution is not practical.

In graph-based approximation approaches, the application graph is considered as a DAG where vertices and edges represent tasks and the dataflow between tasks respectively. This allows existing graph analysis techniques to be applied to the problem of scheduling, which includes graph partitioning and critical path analysis. 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. Quincy [19] for Dryad [20] 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. A commonly used method for finding groups of highly communicating tasks is to partition the DAG, where each partition can then be assigned to a compute node. COLA [21] 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. However it is an offline scheduler and cannot react to the changes in the data stream rates. Fischer et al. [22] propose a mapping of workload scheduling to a graph partitioning problem. P-Scheduler [23] 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. Both of the schedulers work on homogeneous clusters and cannot be used for heterogeneous nodes.

Ghaderi et al. [24] develop a dynamic stochastic graph partitioning and packing method for allocating resources of graph-based streaming applications. Their algorithm also provides a tradeoff among average partitioning cost and average queue lengths. The scheduler handles multiple streaming applications, which continually arrive and depart, unlike other systems which assume an application will persist on the system until explicitly terminated. An alternative method of improving the performance of a streaming application is to reduce the critical path when assigning tasks to nodes, thereby reducing the response time. Such an approach is used in Re-Stream [25] where the critical path of a data stream graph is identified, allowing the critical vertices on the critical path to be reallocated, minimising the response time. It also consolidates the non-critical vertices on the non-critical path to maximise energy efficiency. Another approach to reduce the critical path is proposed in [26] where an online scheduler achieves system stability with a makespan guarantee. First, some relevant dynamic information, such as the input rate of the data stream, and the current available capacity of each compute node, needs to be collected. The new task placement scheme is obtained through the dynamic critical path based on an earliest finish time priority strategy. However, neither of these algorithms consider task parallelism, and only assume a single task per operator, limiting the performance, and responsiveness of the application. Further, they do not react to changes in the application during execution.

The use of graph analysis has the advantage of a global view of the application graph which can help to better locate partitions of highly communicating tasks. However, many of the graph-based algorithms are essentially based on heuristic methods and can result in sub-optimal solutions.

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 [27]. A large number of heuristic approaches have been proposed in the literature that are able to find efficient solutions in a timely manner [7]. One of the most commonly adopted optimisation goals is the minimisation of inter-node communication. That is, by assigning highly communicating tasks to the same node, the communication between nodes will be reduced, improving performance. This is especially important on current, cloud-based DSPSs which are highly scalable, where a large number of nodes may be used. Aniello et al. [8] propose a static, offline graph analysis method that derives a partial order of the components of a streaming application based on their connectivity. It then places the tasks of each operator within the same node as those of the adjacent operator. However, such a linear set is incapable of representing the entire streaming application communication pattern. Additionally, offline analysis cannot identify the amount of communication between tasks. To address this inability to use run-time values, an online scheduler is also proppopsed in [8] (that we refer to as OLS in this paper). OLS monitors the data transfer rate between communicating tasks, CPU load and memory usage at the beginning of execution. These values are used to minimise the inter-worker and inter-node traffic, through a greedy, best fit approach. The limitations of a task pair view is improved in [11] which uses a different heuristic approach to inspect communicating groups of tasks instead of task pairs. It defines a metric for two communicating tasks, called relative significance, which is calculated as the data transfer rate between the tasks over the sum of tasks' load. The nodes are sorted based on capacity and the communicating groups of tasks are sorted based on relative significance. Using a greedy approach, a group pair is selected to be assigned to the node with the highest remaining capacity. However, the nodes are not fully utilised, because this approach has a tendency to spread the groups of tasks across the cluster as a result of inspecting each group pair in isolation. To better consolidate the cluster nodes and fully utilise the resources, Peng et al. [9] propose R-Storm,[3] an offline resource-aware scheduler, which considers the CPU, memory and bandwidth requirements of each task, as specified by the user. It then deals with the scheduling as a 3D knapsack problem. However, the need for users to provide such detailed task requirements beforehand might not always be a practical or convenient solution.

Similarly, T-Storm [10] 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 pack-

[3]Available in the implementation of Apache Storm

ing approach, where nodes are sorted by capacity. Each node in T-Storm can only have one worker in order to remove inter-process traffic between workers. However, when assigning tasks to the node, each task's total load is considered separately and the traffic rate between tasks is ignored. Similar to T-Storm, D-Storm [28] schedules tasks using bin packing, where tasks are prioritised based on a formula which weights tasks by their newly introduced and potential intra-node communication. Tasks are then assigned, in descending order of priority, using a First Fit Decreasing (FFD) heuristic, which places them in the first node with sufficient capacity. Basanta-Val et al. [29] characterise streams as real-time entities, allowing existing real-time scheduling theory to be applied. This results in more predictable performance, which is used to tune the number of required compute nodes, before using bin packing algorithms to assign streams to nodes. Similar to T-Storm, these approaches also check each task in isolation when performing bin packing. For a more dynamic scheduler, which adapts to each workload type, Rychly et al. [30] 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. This approach does not consider the communication pattern between tasks. TCEP [31] proposes a scheduler which adapts to changes in the environment and desired QoS by choosing between multiple placement mechanisms. As a single placement mechanism may not handle these changes, a lightweight genetic algorithm model is used to determine the best operator placement, improving QoS. The algorithm allows for seamless transitions between the operator placement mechanism at run-time.

Each of the previous works discussed above have used a variety of approaches to meet their optimisation goal of QoS, resource utilisation, decreasing latency or reducing inter-node communication, among others. However, they each had a particular limitation, such as being offline, taking a localised pair-based view of the task graph, not fully utilising compute nodes or only working on homogeneous clusters. In this paper we address each of these limitations, while focusing on the optimisation goal of reducing inter-node communication. This can be achieved by fully utilising compute nodes and placing highly communicating tasks on the same node, which will reduce the amount of data needing to be moved between the nodes. We design an online scheduler to react to the recent changes in the communication between tasks, that works on heterogeneous clusters. We leverage the strengths of each general approach by using optimisation software, graph analysis and a

Table 1: Notation for problem formulation

Symbol	**Description**
t_i	Task i
T	Task set
$R(t_i, t_j)$	Data transfer rate between t_i and t_j
$L(t_i)$	Load of t_i
n_i	Node i
N	Node set
$C(n_i)$	Available capacity of n_i
$P_{i,u}$	Placement of t_i on n_u
T_i	All of the tasks assigned to n_i
$X_{i,j}$	Inter-node communication between t_i and t_j
$W(i, j)$	Communication cost between t_i and t_j

heuristic method, for task scheduling. Optimisation software is used to guide the task placement on each of the compute nodes, which is made possible by reducing the problem size by fusing groups of highly communicating tasks, determined through global view graph analysis. To handle situations where the task graph cannot be sufficiently reduced in size, a fallback heuristic is used to provide a near optimal schedule. In the following sections, we present our proposed scheduler in detail.

3. System Model and Problem Definition

Data stream processing systems, such as Apache S4 [32], Flink[4], Storm[5] and Twitter Heron [33], process large volumes of data, on the fly, as it flows through the system, without first being stored. This makes them ideal for providing real-time analysis of fast flowing information, with a common example being Twitter's top trending topics, where real-time streams of tweets are analysed to determine the most popular topics at a given time.

Streaming applications consist of a number of interconnected operators, which are either source or compute operators. A source operator receives streams of data, which it then emits into the streaming application, by passing it to a compute operator. These compute operators then perform opera-

[4]`https://flink.apache.org`
[5]`https://storm.apache.org/`

tions on the data stream, before emitting the results to the next compute operator in the application. Each operator contains multiple tasks, where each task is an executing instance of the operator's code. This allows each task to process a different data stream, increasing the throughput and parallelism of the operator. The number of tasks per operator is initially determined by the application developer, but is able to be adjusted at run-time in response to changes in the amount of data flowing through the application.

The dataflow between tasks is determined by *stream grouping*,[6] which specifies how the tasks of the emitting operator are connected to the tasks of the receiving operator. These policies control how the stream is split across multiple tasks, with varied objectives such as always sending similar streams to the same task, or ensuring that all tasks receive a balanced number of streams.

Each of the streaming systems represents applications as a DAG, $G = (V, E)$, where each task is a vertex v_i in V and the dataflow between two communicating tasks is shown by the edge e_j in E. The vertex is weighted by the task load, while the edge weights represent the communication rate between two given tasks. With this representation of a streaming application, scheduling can be seen as a problem of assigning the vertices to compute nodes, such that the inter-node communication is minimised. That is, the graph is divided into parts, which are sized relative to the capacity of the nodes in the system, such that the nodes are fully utilised without exceeding the capacity of a given node. The division also seeks to minimise the sum of edge weights between compute nodes, which reduces the communication cost. This problem has been proven to be NP-hard [4, 5, 6, 34] due to the large search space and computational complexity.

Similar to [11, 13, 35], the optimal scheduling problem is formulated as follows. It is assumed that t_i and t_j are two communicating tasks in the task set T and $R(t_i, t_j)$ is the data transfer rate between these two tasks. $L(t_i)$ indicates the load of task t_i, n_i represents node i in the node set N, and $C(n_i)$ is the available capacity of n_i. Table 1 shows the notation used in the problem formulation.

In order to represent the allocation of t_i on n_u, a placement matrix P is

[6] http://storm.apache.org/releases/current/Concepts.html

defined such that:

$$P_{i,u} = \begin{cases} 1 & \text{if task } t_i \text{ is assigned to } n_u \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In order to formulate whether two tasks, t_i and t_j are on the same node or different nodes, an inter-node communication matrix X is defined such that:

$$X_{i,j} = \sum_{u=1}^{|N|} \sum_{v=1}^{|N|} P_{i,u} \times P_{j,v} \qquad \text{for } u \neq v \tag{2}$$

The problem of minimising data movement between the communicating tasks in an application can be expressed as minimising the following sum:

$$W(i,j) = \sum_{i=1}^{|T|} \sum_{j=1}^{|T|} R(t_i, t_j) \times X_{i,j} \qquad \forall t_i, t_j \in T \text{ and } i < j \tag{3}$$

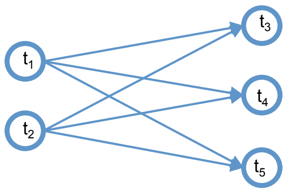
subject to:

$$\sum_{j=1}^{|T_i|} L(t_j) < C(n_i) \qquad \forall n_i \in N \tag{4}$$

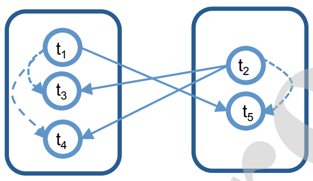
$$\sum_{u=1}^{|N|} P_{i,u} = 1 \qquad \forall i \in T \tag{5}$$

The limitations in 4 and 5 ensure that the capacity of a node is not exceeded and each task is assigned to only one node. It is worth noting that matrix X is symmetric and only $i < j$ is considered in Formula 3.

In the following, we provide an example to demonstrate the optimisation problem formulated above. Figure 1a shows an example DAG application, with 5 tasks that communicate with each other. We assume a task load of 1 for each task and a communication rate of 1 to simplify this example. These tasks are to be assigned to 2 nodes, each with a capacity of 3 tasks. The placement matrix P, given by Formula 1, is:



(a) A simple DAG application



(b) One possible optimal task placement

Figure 1: A simple DAG application and one possible optimal task placement

$$P = \begin{bmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \\ P_{31} & P_{32} \\ P_{41} & P_{42} \\ P_{51} & P_{52} \end{bmatrix}$$

where each row represents a task and each column shows a node. An entry P_{iu} shows the placement of task i on node u.

The inter-node communication matrix X for all 5 tasks, found by Formula 2, shows all of the possible task placements as follows:

$$X = \begin{bmatrix} 0 & P_{11}P_{22}+P_{21}P_{12} & P_{11}P_{32}+P_{31}P_{12} & P_{11}P_{42}+P_{41}P_{12} & P_{11}P_{52}+P_{51}P_{12} \\ P_{11}P_{22}+P_{21}P_{12} & 0 & P_{21}P_{32}+P_{31}P_{22} & P_{21}P_{42}+P_{41}P_{22} & P_{21}P_{52}+P_{51}P_{22} \\ P_{11}P_{32}+P_{31}P_{12} & P_{21}P_{32}+P_{31}P_{22} & 0 & P_{31}P_{42}+P_{41}P_{32} & P_{31}P_{52}+P_{51}P_{32} \\ P_{11}P_{42}+P_{41}P_{12} & P_{21}P_{42}+P_{41}P_{22} & P_{31}P_{42}+P_{41}P_{32} & 0 & P_{41}P_{52}+P_{51}P_{42} \\ P_{11}P_{52}+P_{51}P_{12} & P_{21}P_{52}+P_{51}P_{22} & P_{31}P_{52}+P_{51}P_{32} & P_{41}P_{52}+P_{51}P_{42} & 0 \end{bmatrix}$$

For example, entry X_{12} shows the inter-node communication between task 1 and task 2, which is a result of either, task 1 being placed on node 1 and task 2 on node 2 or task 1 on node 2 and task 2 on node 1. Note that if both tasks are placed on the same node, there is no inter-node communication between them. As the data transfer rate is 1, the communication cost matrix, previously formulated in Formula 3 is the same as X. This problem can be given to the optimisation software with the limitations, given in Formulas 4 and 5, that the capacity of 3 tasks per node is not exceeded and that a task is only assigned to one node. Figure 1b shows one possible optimal solution, with an inter-node communication cost of 3, illustrated by the arrows.

While it is possible to formulate the scheduling problem as an optimisation problem, in practice, it is infeasible to determine an optimal schedule for larger problem sizes due to the computational complexity. As the problem of scheduling tasks to compute nodes is NP-hard [6], a heuristic approach is required to find near optimal schedules in a reasonable time. Our heuristic approach, presented in Section 4, leverages graph partitioning algorithms and optimisation software to reliably and efficiently schedule tasks.

4. I-Scheduler Algorithm

As discussed in Section 3, using optimisation software to find an optimal solution to the scheduling problem is often not practical. One approach is to reduce the size of the task graph by fusing each pair of tasks into a single task [12]. However, this has a number of drawbacks. Firstly, fusing task pairs takes a localised view of the task communications and may be unreliable at placing communicating tasks within the same node. Secondly, this approach may not be able to scale to larger problem sizes as it is only able to halve the task graph size.

To address this, I-Scheduler exploits the K-way partitioning algorithm to reliably find groups of highly communicating tasks. As this algorithm only finds partitions of equal size, we iteratively apply K-way partitioning to the application graph, where partitions are sized according to the current heterogeneous node capacity. I-Scheduler can find a solution in two ways. Firstly, a threshold task graph size is reached by fusing the partitions such that the problem size is sufficiently reduced to be solved by optimisation software. This allows use-cases which can tolerate taking longer to find a schedule to benefit from the optimisation software. Secondly, I-Schedule can continue to partition the graph iteratively until all tasks are assigned to a node with a relative capacity. This provides an efficient fallback heuristic method for use-cases that cannot tolerate the delay in using the optimisation software. It is worth noting that our fallback heuristic is triggered either from the beginning, when there is no tolerance for use of optimisation software, or when it becomes clear that reducing the task sizes will not be sufficient to meet the threshold. This is different with other works which use their fallback heuristic algorithm only if an attempt to find an optimal solution fails.

I-Scheduler consists of four main steps as follows.

1. **Monitoring:** To operate online, I-Scheduler monitors the communication between tasks, measuring the tasks' load and data transfer rates.

Algorithm 1 Pseudo-code for the I-Scheduler algorithm

```
 1: $\theta \leftarrow$ time allowed for scheduling
 2: if $\theta > 0$ then
 3:     $use\_optimisation\_software \leftarrow$ True
 4:     $\phi \leftarrow$ FIND_THRESHOLD($\theta$)  ▷ Find the threshold task size based on $\theta$ from previous experiments
 5: else
 6:     $use\_optimisation\_software \leftarrow$ False
 7:     $\phi \leftarrow 0$
 8: end if
 9: $Online\_Profile \leftarrow$ MONITOR($Streaming\_Application$)
10: $g \leftarrow$ CONSTRUCT_GRAPH($Online\_Profile$)
11: $T \leftarrow$ Maximum number of tasks per worker  ▷ An empirically determined threshold
12: $N \leftarrow$ SORT($[(c_1, q_1), (c_2, q_2), \ldots, (c_j, q_j)]$)
13: SCHEDULE($N$, $g$, $\phi$, $T$)
14: function SCHEDULE($N$, $g$, $\phi$, $T$)
15:     $task\_map \leftarrow$ FIRST_LEVEL($N$, $g$, $\phi$)
16:     for each $t\_m$ in $task\_map$ do
17:         $sbg \leftarrow t\_m.tasks$
18:         $n \leftarrow t\_m.node$
19:         SECOND_LEVEL($n$, $sbg$, $T$)
20:     end for
21: end function
22: function FIRST_LEVEL($N$, $g$, $\phi$)
23:     $i \leftarrow 1$                          ▷ Iteration number
24:     $t \leftarrow$ FIND_TASK_SIZE($g$)          ▷ Number of tasks in $g$
25:     $f \leftarrow 0$                          ▷ Number of fused tasks
26:     while $t + f > \phi$ do
27:         $k_i \leftarrow \left\lceil \frac{t}{N[i].c} \right\rceil$
28:         if $k_i = 1$ then
29:             FUSE_TASKS($g$, 1)  ▷ Fuse $t$ remaining tasks in $g$ into a single task
30:             break
31:         end if
32:         if $k_i > \phi$ and $use\_optimisation\_software$ then
33:             $use\_optimisation\_software \leftarrow$ False
34:             $\phi \leftarrow 0$
35:         end if
```

```
36:         GRAPH_PARTITION($g$, $k_i$)   ▷ Partition $g$ into $k_i$ parts on unfused vertices
37:         if use_optimisation_software then
38:             $\alpha \leftarrow \min(\left\lceil \frac{t-\phi}{N[i].c-1} \right\rceil, N[i].q)$
39:         else
40:             $\alpha \leftarrow \min(k_i, N[i].q)$
41:         end if
42:         FUSE_TASKS($g$, $\alpha$)                ▷ Fuse the tasks in $\alpha$ partitions
43:         $t \leftarrow t - (\alpha \times N[i].c)$                ▷ Update remaining task size
44:         $f \leftarrow f + \alpha$                    ▷ Update the number of fused tasks
45:         UPDATE_GRAPH($g$)       ▷ Update $g$ with new vertices and edges
46:         $i \leftarrow i + 1$
47:     end while
48:     if use_optimisation_software then
49:         task_map ← SOLVE_MODEL($N$, $g$)            ▷ Use an optimisation software to solve the optimisation problem
50:     else
51:         task_map ← DIRECT_MAPPING($N$, $g$)   ▷ Map each fused task in $g$ to a node in $N$ with relative capacity
52:     end if
53:     return task_map
54: end function
55: function SECOND_LEVEL($n$, $sbg$, $T$)
56:     $t \leftarrow$ FIND_TASK_SIZE($sbg$)
57:     $w = \lceil \frac{t}{T} \rceil$
58:     parts ← GRAPH_PARTITION($sbg$, $w$)       ▷ Partition $sbg$ into $w$ parts
59:     for each part in parts do
60:         Assign part to one worker on $n$
61:     end for
62: end function
```

2. **Constructing a weighted graph:** The online profile from the previous step is then used to build a weighted graph, providing I-Scheduler with a global view of the task loads and communication, which is updated with new weights for vertices and edges when rescheduling.
3. **First level of scheduling:** At this level, I-Scheduler determines which groups of tasks should be co-located within each node by finding a sub-

graph of highly communicating tasks from the weighted graph.

4. **Second level of scheduling:** To reduce inter-worker communication within each node, I-Scheduler partitions the sub-graph assigned to each node into the required number of workers using K-way partitioning. This ensures highly communicating tasks are placed in the same worker.

Algorithm 1 presents the pseudo-code for I-Scheduler. Table 2 shows the notation used in this section for the I-Scheduler algorithm. In the following subsections, each step is discussed in detail.

4.1. Monitoring

I-Scheduler monitors the execution of the streaming application to construct a profile of all task communications and task loads. This is an important step for any online task scheduler, which requires up-to-date information about the streaming application traffic patterns. For this step, I-Scheduler collects the data transfer rate, $R(t_i, t_j)$ for every task pair t_i and t_j, and the load, $L(t_i)$ for every task t_i, previously presented in Section 3 (Algorithm 1 Line 9). These values are regularly collected and stored, ensuring they are available for effective rescheduling.

4.2. Constructing a weighted graph

Each of the individual task pair values, collected from the monitoring step, are used to construct the weighted application graph, G, where the vertices represent tasks and edges show the communication between tasks (Algorithm 1 Line 10). Vertices are weighted by the task load and the edges are weighted by the total data transfer rate between the corresponding tasks. This creates a global view of the application graph which helps I-Scheduler find and co-locate highly communicating tasks on the same node when scheduling. The weights in the graph are updated when rescheduling occurs, keeping the graph up-to-date.

4.3. First level of scheduling

By exploiting graph partitioning to find partitions of highly communicating tasks, sized according to node capacities, we can fuse each partition into a single task. This process is performed iteratively for each cluster node capacity until we reach a task graph size that is solvable by the optimisation software. However, in the event that the task graph is too large and complex to be sufficiently reduced in size, we can continue using the iterative task

Table 2: Notation for I-Scheduler algorithm

Symbol	Description
g	Weighted application graph
t	Number of unfused tasks within g
f	Number of fused tasks within g
θ	A feasible amount of time to solve the optimisation problem
ϕ	Threshold task size, solvable in θ
T	Threshold task size per worker
N	Sorted array of available nodes (capacity, quantity)

fusions until all tasks are fused and then assign each of the fused tasks to a node with a relative capacity, providing a heuristic fallback.

Since the time permitted for scheduling may vary between use-cases, a user parameter, θ, is defined for this time interval (Algorithm 1 Line 1). For use-cases that tolerate longer scheduling times, a larger θ can be chosen, which will result in an improved scheduling solution as the optimisation software does more of the work. This parameter provides the ability to tune the scheduling efficiency according to the time allowed for scheduling by the use-case. Based on the value set for θ, we can then determine a task threshold ϕ based on previous experiments, such that a solution for ϕ tasks can be found in a time interval of θ (Algorithm 1 Line 4). t is defined as the total number of tasks within an application graph, g. Therefore, by reducing t to ϕ, we can then use the optimisation software to find a near optimal solution. While this approach can help improve the efficiency of the schedule, it is not necessarily optimal as the steps used to reduce the problem size are heuristic. Despite this lack of a guarantee, our experimental results, presented later in Section 6, show that this approach helps to improve the performance of task scheduling.

To determine highly communicating tasks, we use K-way partitioning [36] to reliably find partitions of highly communicating tasks, which allows us to fuse some of these partitions into single larger tasks. This partitioning is performed iteratively, where each iteration, and corresponding graph partition size, is based upon the largest capacity node in the heterogeneous cluster. Each iteration reduces the size of t, which continues until t reaches the threshold ϕ. Note that this scheduling approach would only require one iteration on a homogeneous cluster, as all the node capacities are the same.

The graph partitioning is performed on a weighted graph such that each vertex weight is the task load and each edge weight is the data transfer rate between two communicating tasks. To simplify our formulation, we assume that a node's capacity is defined as the number of tasks it can host, however it can be defined as the sum of the CPU speed for all cores within a node. We assume that all available nodes are stored in an array, $N = \{(c_1, q_1), (c_2, q_2), \ldots, (c_j, q_j)\}$, such that each element of the array has two properties: capacity, c, and quantity, q. We use '.' notation to denote accessing each property of an element of the array, such that $N[i].c$ is the capacity of node N_i, and $N[i].q$ is the number of nodes with the given capacity in the heterogeneous cluster.

We begin by sorting the available nodes in descending order of capacity. By starting with the highest capacity nodes, we are able to assign the largest groups of tasks to these nodes, reducing the overall communication cost. In comparison, using smaller nodes results in smaller task groups which are spread across more nodes, increasing the inter-node communication cost, resulting in a less efficient schedule.

In order to partition the task graph g of size t into k_i parts of roughly size $N[i].c$, we need to calculate, k_i, for iteration i as follows (Algorithm 1 Line 27):

$$k_i = \left\lceil \frac{t}{N[i].c} \right\rceil \tag{6}$$

After finding k_i at the start of each iteration, we compare it with ϕ as follows:

- $k_i \leq \phi$: In this case, we are able to partition (Algorithm 1 Line 36) and select k_i partitions for fusion in order to reach ϕ. However, to maximise the work performed by the optimisation software, we may not require all k_i partitions to be fused to reach ϕ. Instead we find the minimum number of fusions required, β, to reach ϕ. By selecting β partitions, the task size t is reduced by $\beta \times N[i].c$ tasks, and increased by β as we place β fused tasks back in the graph. This relationship between β, t and ϕ is shown as:

$$\phi = t - \beta \times N[i].c + \beta \tag{7}$$

Therefore, the minimum number of fusions required, β out of k_i, is calculated as follows:

$$\beta = \left\lceil \frac{t - \phi}{N[i].c - 1} \right\rceil \tag{8}$$

While β partitions will reach ϕ, as desired, there may not be enough nodes, $N[i].q$, of the current capacity, $N[i].c$, to host the fused partitions. Therefore, we compare β with $N[i].q$ as follows:

- $\beta \leq N[i].q$: we fuse the tasks in β partitions out of k_i and we have successfully reduced t to ϕ.
- $\beta > N[i].q$: This means that the number of available nodes for the current capacity is not enough to host β fused tasks. Therefore, only $N[i].q$ partitions are fused and a further iteration of partitioning is performed on the remaining unfused tasks, where the size of each partition is the next available largest capacity node.

This decision is formulated as (Algorithm 1 Line 38):

$$\alpha = \min(\beta, N[i].q) \tag{9}$$

After fusing α partitions (Algorithm 1 Line 42), the value of t is updated to the new remaining task size, calculated as: $t \leftarrow t - \alpha \times N[i].c$ (Algorithm 1 Line 43). A new application graph is built at each step based on the connectivity and weights from the original application graph (Algorithm 1 Line 45). The task size of the new graph (which includes remaining and fused tasks) is compared to ϕ and the process of K-way graph partitioning continues until we reach ϕ. Once reached, we use the optimisation software to perform the final task assignment with the fused, and remaining unfused tasks (Algorithm 1 Line 49).

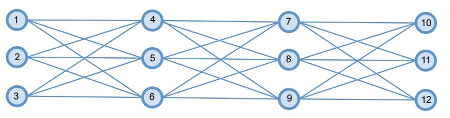
- $k_i > \phi$: This means that even if we fuse all of the k_i partitions into k_i tasks, we will not be able to reduce the task size t to ϕ. Therefore, the optimisation software cannot solve the problem within the time interval θ. This can occur at any stage during iterative partitioning, which is resolved by setting the threshold ϕ to zero, and falling back to the heuristic iterative partitioning algorithm (Algorithm 1 Line 32–35) . We then continue with the I-Scheduler algorithm. This will result in every task in g being part of a fused group, which can fit within a given node, as there is no longer a stopping threshold. Therefore, for each iteration i, α task groups (Algorithm 1 Line 40) are fused and assigned to the $N[i].q$ nodes with a corresponding capacity $N[i].c$ (Algorithm 1 Line 51).

In summary, I-Scheduler uses iterative graph partitioning to reduce the size of the task graph by fusing tasks of some partitions to reach a threshold. The smaller graph size enables the optimisation software to be used for task assignment. The configurable threshold determines the number of task fusions that are required, setting the amount of work to be performed by the optimisation software. The fallback heuristic method is used when the task graph is too large to be reduced and can be viewed as iterative graph partitioning. In this case, I-Scheduler essentially treats each step of the problem as a homogeneous cluster, using K-way partitioning to determine the groups of tasks for each node capacity. It then assigns different sized partitions of the task graph to their relative capacity nodes.

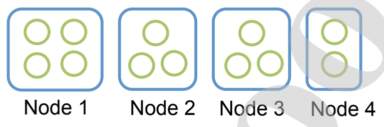
4.4. Second level of scheduling

In data stream processing systems such as Apache Storm, each compute node can have a number of workers where each worker runs one or more tasks. Once the first level is complete, each compute node has been assigned a subgraph by I-Scheduler. In the second level of scheduling, I-Scheduler decides how these tasks are distributed across the workers within each compute node so that the inter-worker communication is minimised. Similar to our previous work [35], I-Scheduler first calculates the number of workers per node, where a balance needs to be struck between performance and reliability. That is if all the tasks are assigned to a single worker, there is no inter-worker communication. However, in the event of a failure of a task within the worker, the worker will reschedule the failed task, along with all other tasks within that same worker. Therefore, the failure of any one task will impact the performance of all other tasks, which encourages fewer tasks per worker for reliability. Nevertheless, using more workers comes at a performance cost, where the inter-worker communication is increased, resulting in higher latency.

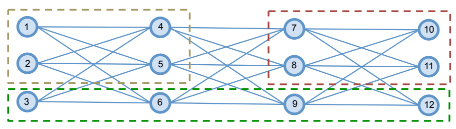
In our simple heuristic, an empirically determined threshold T is introduced, which represents the maximum number of tasks per worker (Algorithm 1 Line 11). This value sets a balance between better tolerating failures,



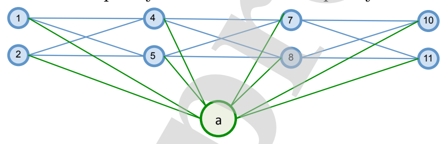
(a) The initial task graph g has 12 tasks



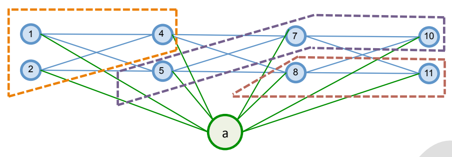
(b) The cluster consists of 1 node of capacity 4, 2 nodes of capacity 3 and 1 node of capacity 2



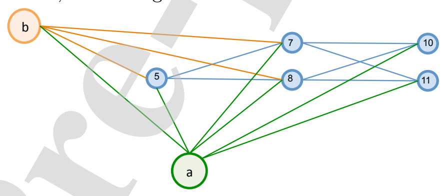
(c) Partitioning g into 3 partitions of size 4, calculated by Equation 6



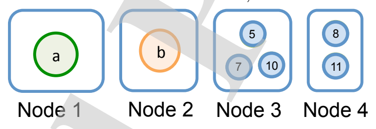
(d) Fusing 1 partition out of 3, calculated by Equation 9, shown in green



(e) Partitioning the remaining tasks into 3 partitions of size 3, calculated by Equation 6



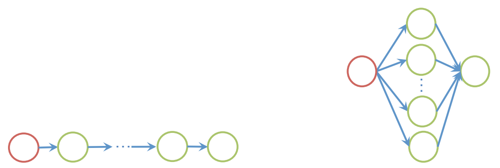
(f) Fusing 1 partition out of 3, calculated by Equation 9, shown in orange



(g) Final assignment

Figure 2: An example of task assignment by I-Scheduler

and reducing inter-worker communication. I-Scheduler calculates the number of workers per node by dividing the total number of tasks assigned to a compute node by T, the maximum number of tasks per worker (Algorithm 1 Line 57). The sub-graph assigned to each compute node is then partitioned into the number of worker nodes by using K-way partitioning (Algorithm 1 Line 58), similar to the first level of scheduling, such that each worker is assigned a maximum of T tasks, and the inter-worker communication is minimised by assigning the most highly communicating tasks together to the same worker.



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

Figure 3: Layout of micro-benchmark applications [9]

4.5. An example of task assignment by I-Scheduler

Consider the scheduling problem, with a task graph of size 12, shown in Figure 2a and a heterogeneous cluster, made up of 3 types of nodes, 1 node with capacity of 4, 2 nodes with capacity of 3, and 1 node with capacity of 2, shown in Figure 2b. The threshold ϕ is set to 8. The first iteration of I-Scheduler partitions the 12 task graph into 3 partitions of size 4, calculated by Equation 6 as shown in Figure 2c. In order to reach the threshold of 8, we need to fuse 2 out of the 3 partitions calculated by Equation 8. However, only 1 of these partitions can be fused, based on Equation 9, as we only have 1 node of capacity 4. So we fuse 1 partition and place it back into the graph as the new fused task, shown in Figure 2d. This reduces the unfused task size to 8 tasks, shown in the figure. The number of tasks in the new graph is 9 (including 8 unfused tasks plus 1 fused task) which is greater than the threshold of 8. Therefore, a further iteration of graph partitioning is performed on the remaining tasks, where the unfused tasks in the graph are partitioned into 3 parts of roughly size 3, shown in Figure 2e. We fuse 1 of the 3 partitions, shown in orange in the figure, as this is all that is required to reach the threshold of 8, shown in Figure 2f. Finally, the optimisation software finds a solution and I-Scheduler performs the final task assignment, shown in Figure 2g.

5. I-Scheduler versus Optimal Scheduler

In this section, we compare the communication cost and resolution time of I-Scheduler with a theoretical optimal scheduler when run on three micro-benchmarks. Each of these micro-benchmarks, shown in Figure 3, is named after its shape of the topology and communication pattern: linear, diamond and star, and are based on the implementation originally presented in [9]. These are common communication patterns that can be found in real world

Table 3: Compute nodes' configuration

Setting	**Number of Nodes**	**Capacity of Nodes**
Homogeneous	10 nodes	All nodes, capacity of 4
Heterogeneous	10 nodes	3 nodes, capacity of 6 3 nodes, capacity of 4 4 nodes, capacity of 2

applications, which consists of either just one of these communicating patterns, or a combination of them [37]. Given the absence of a standard benchmark suite [37], micro-benchmarks have been adopted in other works for evaluation [38, 39]. In the following, we describe each layout.

- **Linear micro-benchmark:** This is one of the most common topology layouts, shown in Figure 3a, where operators are connected in sequence passing tuples along. Each of the sources, sinks and operators are configured with two tasks each.
- **Diamond micro-benchmark:** This layout has a single source and sink operator, with a variable number of operators in between, which gives it the diamond shape, shown in Figure 3b. The source and sink are configured with four tasks each, while the operators in the middle have two tasks each. New operators are added to the middle.
- **Star micro-benchmark:** This more complicated topology has multiple sources and sinks, connected to a single operator in the middle. This makes this one operator a point of congestion where multiple sources emit tuples to it, before it sends the tuples out to multiple sinks. Each source and sink is configured with two tasks, while the middle operator has four tasks. New operators are alternately added as either a source or sink, keeping the topology balanced.

We evaluate different problem sizes for each micro-benchmark by increasing the number of operators, increasing the task count by two each time, starting with a problem size of 10 tasks, and continuing until 32 tasks are reached. Each micro-benchmark is evaluated on a homogeneous and heterogeneous cluster, 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

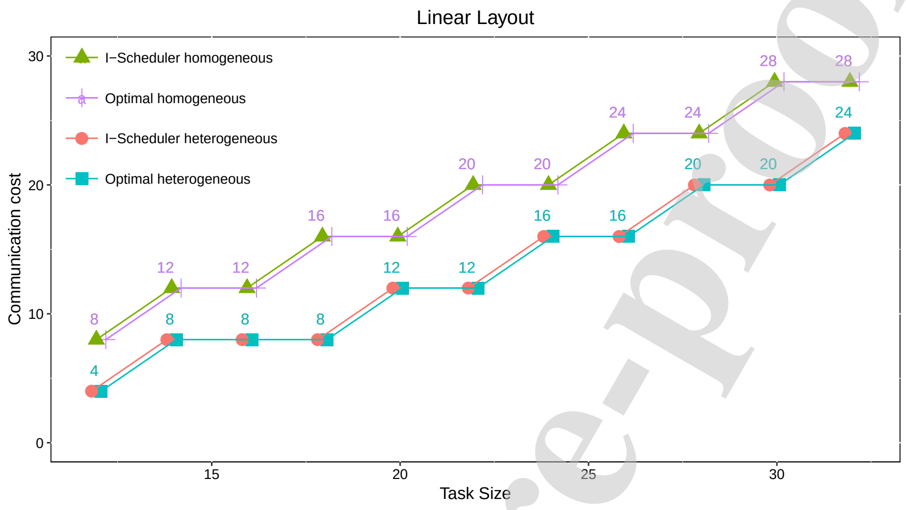


Figure 4: I-Scheduler vs. optimal scheduler for linear layout

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 [40] as our optimisation software as it provides state-of-the-art implementations of linear, integer and mixed-integer linear optimisation [41]. As it becomes computationally expensive to find the solution to problems as they get bigger, a 64-core server with four 16-core AMD Opteron 6276 processors, running at 2.3GHz, with 512GiB of RAM was used to find the optimal solution. In comparison, I-Scheduler does not have such onerous resource requirements, and was run on a system with a 3.2GHz Intel Core i5-4570 quad core processor with 8GiB of RAM. We refer to the solutions found by CPLEX as "optimal" throughout this paper as done in other work [15, 13]. This is supported by our brute force evaluation of a limited number of small task sizes, where we found that CPLEX results were indeed optimal.[7] To simplify the experiments, we assume that the data

[7]A more extensive brute force evaluation is infeasible due to the time complexity of the problem.

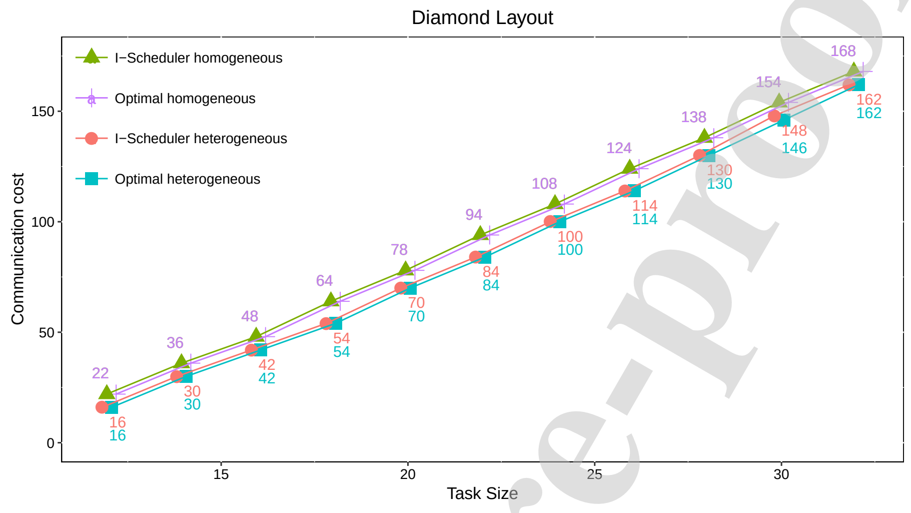


Figure 5: I-Scheduler vs. optimal scheduler for diamond layout

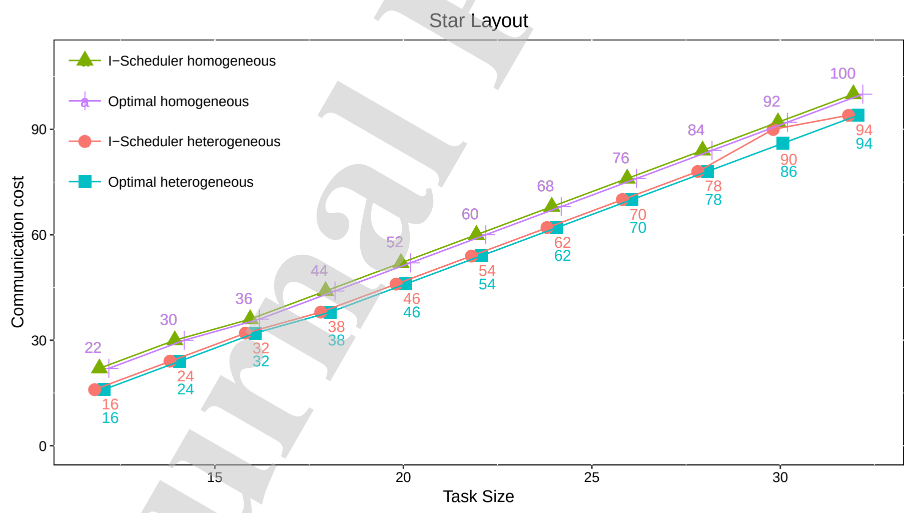


Figure 6: I-Scheduler vs. optimal scheduler for star layout

transfer rate between every two communicating tasks is 1 normalised unit per second, and the load is uniform across all tasks. We configure I-Scheduler's θ parameter to 1 second. In each of the experiments, we present the cost, which is a measure of the communication cost between nodes, previously seen in Equation 3. It is worth noting that the communication cost does not change between runs with the same configuration. This is due to I-Scheduler's use of the K-way partitioning algorithm and the optimisation software, which produce the same result when given the same weighted graph as input.

Figures 4, 5 and 6 show the results for the linear, diamond and star micro-benchmarks respectively. For the linear layout, shown in Figure 4, we can see that I-Scheduler is able to match the results of the optimal scheduler for both the homogeneous and heterogeneous configurations. There is a notable gap in the communication cost between the homogeneous and heterogeneous configurations. This is due to the differences in cluster capacities, where the larger nodes in the heterogeneous cluster allow more tasks to be placed in a single node, reducing the inter-node communication.

Figure 5 shows that I-Scheduler can achieve the same results as optimal for the diamond micro-benchmark, except for 30 tasks run on the heterogeneous cluster, where I-Scheduler has a communication cost of 148, which is two units higher than the optimal scheduler's 146. Looking at the task assignment of both schedulers, we found that the tasks in partitions fused by I-Scheduler while reducing the problem size were slightly different from those found by the optimal scheduler. This is an example where the heuristic K-way partitioning algorithm does not result in the optimal partitions used for task fusion, but I-Scheduler is still able to find a near optimal result.

Figure 6 shows the results for star layout for I-Scheduler and optimal scheduler. As can be seen from the figure, I-Scheduler can find the optimal solution in all cases but for the heterogeneous configuration with 30 tasks where the optimal solution is 86 and I-Scheduler finds 90. This is similar to the previous case for diamond where the tasks within each fused partition differ slightly from optimal. Despite this, I-Scheduler can still find an efficient, near optimal task assignment.

The communication cost for diamond and star layouts, shown in Figures 5 and 6, is much higher than that of the linear layout, Figure 4. This is a result of the shape of each layout, where the linear shape allows more communicating tasks to be placed together within each node, while diamond and star provide fewer opportunities to place communicating tasks together.

Figures 7, 8 and 9 show the resolution times for I-Scheduler and the optimal scheduler for each of the micro-benchmarks. From the figures, it can be seen that the resolution time of the optimal scheduler increases significantly with the problem size. This is a clear demonstration showing that while the optimal scheduler can be used for small problem sizes, it quickly becomes impractical as the problem size increases. In comparison, I-Scheduler has a resolution time of less than θ for all problem sizes. These results demonstrate that I-Scheduler is able to schedule the tasks within a practical time, which is not always possible with other previous approaches that made use of optimisation software. While there is some minor fluctuation in the resolution times, these are not the result of variations in the execution time, as the same trend was seen across multiple runs. Instead, these fluctuations are a result of differences in the size and complexity of the problem solved by the optimisation software. Differences in the problem size occur because entire groups of tasks are fused during each iteration, which can result in a task size that is slightly smaller than the target size ϕ. This results in a shorter resolution time than those with task sizes closer to ϕ.

The complexity of the problem is affected by the size of the partitions created and the cluster node configuration, among other factors. During each iteration, the K-way algorithm partitions the task graph into roughly equal parts where a number of them are chosen for fusing. If any of these fused partitions are not equal to the compute node size, the optimisation software will have to determine which of the remaining tasks should be assigned to the same node to minimise inter-node communication, consequently increasing the complexity of the assignment problem. Also, the cluster configuration can affect the complexity of the task assignment problem handled by the optimisation software. For example, a heterogeneous cluster might present fewer possibilities for assigning larger fused tasks, as seen for the star layout in Figure 9, with a problem size of 28 tasks, which has the lowest resolution time. In this case, the partitions found by the K-way algorithm are all of size 6, which match the size of the 3 large capacity nodes once fused. This reduces the complexity of the final task assignment to assigning 3 tasks of size 6, to 3 nodes of size 6, and assigning the 10 remaining tasks of size 1 to the 3 nodes of size 4 and 4 nodes of size 2. This demonstrates how the size of the fused tasks, and the corresponding cluster configuration, can affect the resolution time.

Overall, I-Scheduler is able to efficiently assign tasks to the homogeneous and heterogeneous cluster nodes and quickly find near optimal results for each of the three micro-benchmarks. This approach can be used to reduce

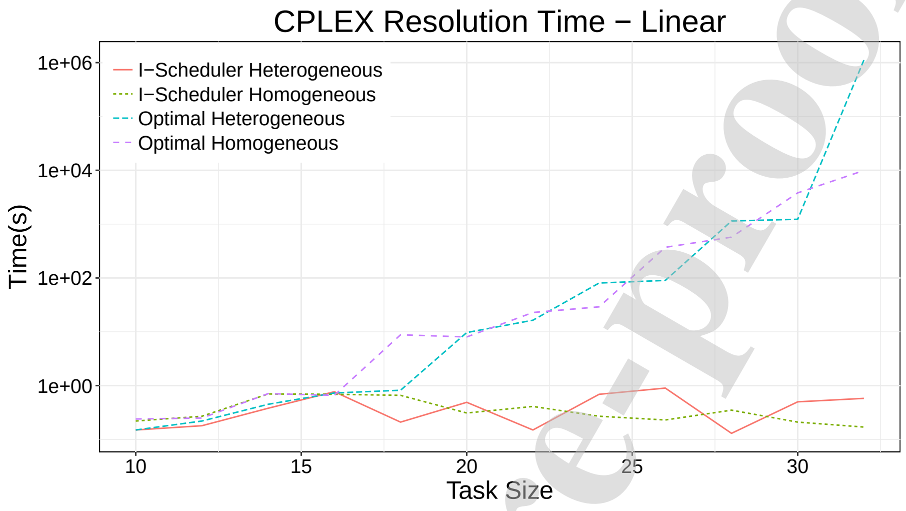


Figure 7: Resolution time for optimal scheduler and I-Scheduler for linear layout

even larger scheduling problems, making them optimisation problems that can be solved in a practical time. We evaluate the practical performance of I-Scheduler in the next section.

6. Experimental Evaluation

In this section, we evaluate I-Scheduler on a homogeneous and heterogeneous cluster, using three micro-benchmarks and two real-world applications. These results are then compared with Aniello et al.'s 'Online scheduler' (referred to as 'OLS') [8] and R-Storm [9], previously discussed in Section 2, which were chosen for two main reasons. Firstly, OLS and R-Storm have the same optimisation criteria as I-Scheduler, which is to reduce the inter-node communication. Secondly, the source code for both schedulers is publicly available, allowing for fair comparisons to be made. Each scheduler is evaluated on the average latency for each task and system throughput, defined as the average number of tuples executed in each operator's task per 10 second period. I-Scheduler is implemented in Apache Storm 1.1.1, and is run on a Storm cluster, configured with one master node, one ZooKeeper node and eight worker nodes. Each node runs Ubuntu 16.04 LTS, has a 2.7GHz Intel Core i5-3330S processor and is connected to a 1Gbps network. Ora-

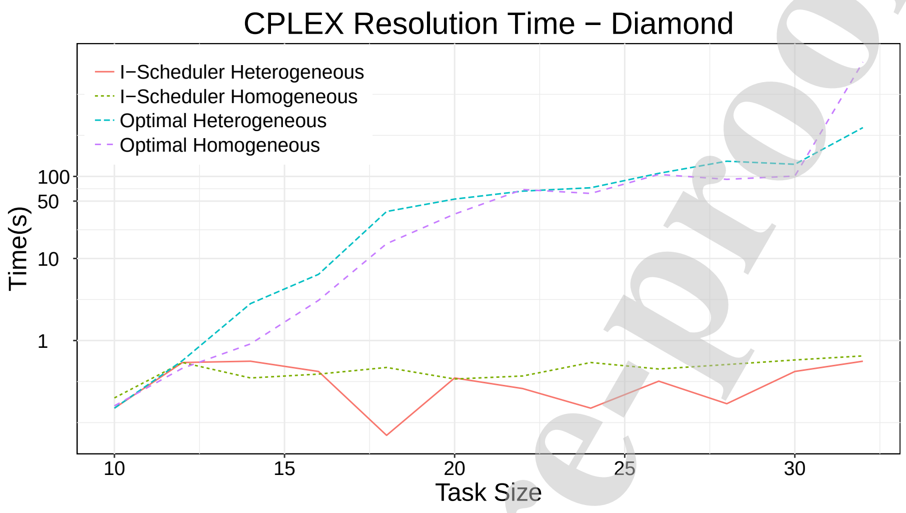


Figure 8: Resolution time for optimal scheduler and I-Scheduler for diamond layout

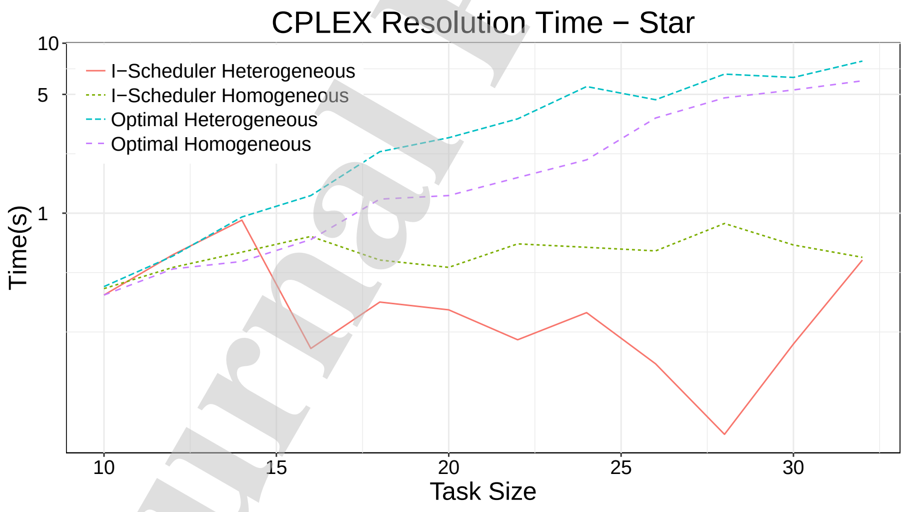


Figure 9: Resolution time for optimal scheduler and I-Scheduler for star layout

cle VM VirtualBox is used to provide a virtualised environment, similar to how large-scale DSPS services may be run in the cloud. This also allows each of the compute nodes to be configured as either high or low capacity, where the high capacity nodes have 4 cores, 4GiB of RAM and four slots per node, while the low capacity nodes have 2 cores, 2GiB of RAM and two slots per node. This configurability allows us to evaluate each scheduler on both homogeneous and heterogeneous clusters. While many clusters start out as homogeneous, they grow over time, as new hardware is introduced, resulting in heterogeneous clusters. Further, a homogeneous cluster where not all compute resources are available can be treated as being a heterogeneous environment, making it important to evaluate both cluster configurations. We initially validated the use of VMs by comparing the performance with a bare metal configuration, finding the difference in performance to be insignificant within the scale of our performance results.

To ensure stable execution performance during evaluation, the number of tasks per worker in the second level of scheduling is set to 5. This provides a balance between the desire for more tasks per worker, and the reliability of execution, where the failure of any single task within a worker will cause all of the other tasks within that worker to be restarted, resulting in data loss. It is also worth noting that the results presented in this paper do not include any runs which had task failures.

Each experimental application is run ten times for 650 seconds. Applications are initially run with round-robin scheduling for 50-60s while I-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, in order to 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 application for each scheduler, similar to how results were presented in OLS [8] and R-Storm [9].

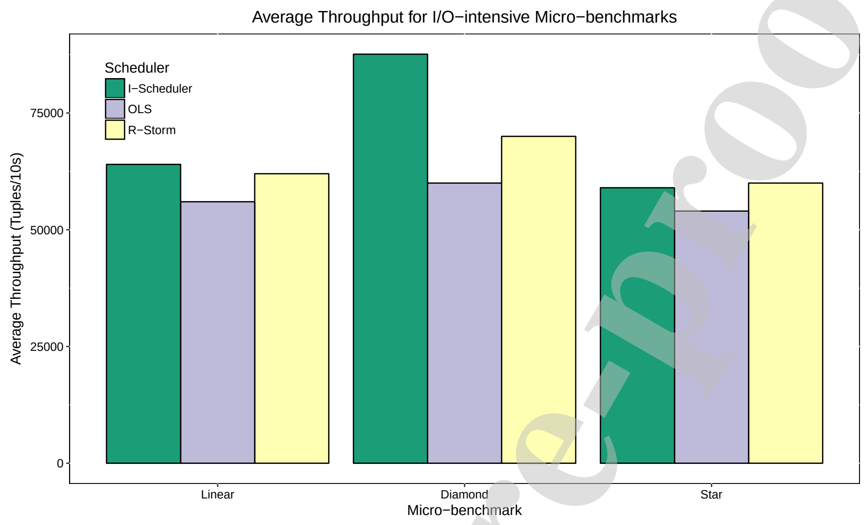


Figure 10: Throughput results of I/O-intensive for linear, diamond and star micro-benchmarks, using I-Scheduler, R-Storm and OLS

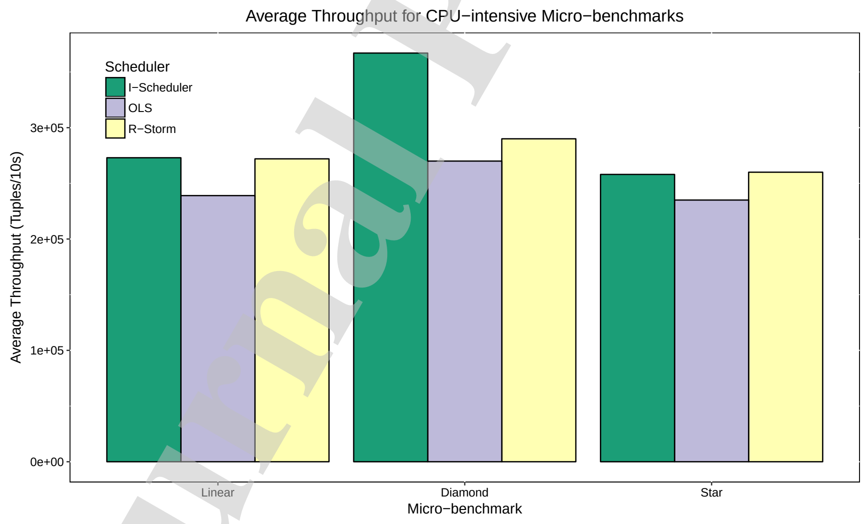


Figure 11: Throughput results of CPU-intensive for linear, diamond and star micro-benchmarks, using I-Scheduler, R-Storm and OLS

6.1. Micro-benchmarks

For this evaluation, we configure each of the operators in the linear and diamond micro-benchmarks to have eight tasks. The star micro-benchmark is configured with four and eight tasks for the source and computational operators respectively. Each micro-benchmark is run in two different configurations: I/O-intensive and CPU-intensive, described as follows.

In I/O-intensive configuration, the throughput of the system is limited by the amount of communication between the nodes. We reduce the workload of each computational operator by slowing the rate of the source operator, so that each computational operator has little processing to do, causing processing time to be limited by the network latency. In CPU-Intensive configuration, the throughput of the system is limited by the CPU utilisation of each node. We increase the workload of each computational operator by supplying tuples at a faster rate, ensuring the computational operators are fully loaded, resulting in a high CPU load.

I/O-intensive: We evaluate the throughput of I-Scheduler for the I/O-intensive micro-benchmarks, which are run on a small heterogeneous cluster, consisting of one high capacity node and two low capacity nodes. This configuration ensures each of the three schedulers uses both high and low capacity nodes, and is not impacted by node consolidation. The results for the micro-benchmarks, run by I-Scheduler, R-Storm and OLS are shown in Figure 10. As can be seen in the figure, I-Scheduler outperforms R-Storm by 25% for the diamond micro-benchmark and achieves a similar throughput for linear and star. In comparison, I-Scheduler outperforms OLS for linear, diamond and star micro-benchmarks by 8–46%, as OLS has a lower average throughput.

CPU-intensive: We then run each of the CPU-intensive micro-benchmarks on a heterogeneous cluster with two high capacity nodes and four low capacity nodes. From the results shown in Figure 11, it can be seen that I-Scheduler outperforms OLS by 10–35% for all micro-benchmarks, with similar results to R-Storm for linear and star micro-benchmarks. Again, I-Scheduler has an average throughput 26% higher than R-Storm for diamond, as a result of a better task placement for I-Scheduler. The average throughput for each CPU-intensive micro-benchmark is higher than that of the I/O-intensive micro-benchmarks, due to the higher load placed on the CPU.

Overall, these results show that I-Scheduler is better able to place highly communicating tasks on nodes, reducing inter-node communication. In comparison, R-Storm is unable to find an efficient schedule for diamond shaped layouts, while the greedy best fit approach of OLS has a lower average throughput for each micro-benchmark.

6.2. Load prediction application for smart homes

This application predicts the future power load of a collection of smart plugs, and the total load for each of the 40 houses in the dataset. This query is based on the first query of the DEBS 2014 grand challenge.[8] The future load predictions are based on a slice-based model which uses the current load measurements and historical data. The streaming application implementing this query has a diamond shape, with one source operator which emits the data to the two intermediate computational operators, which either predict the house or smart plug load, before the final predictions are emitted to the sink operator. Our application is configured with a time slice of 15 minutes, with the source and each of the computational operators having 10 tasks. We use a Redis server to store the smart plug records. We evaluate I-Scheduler by running the load prediction application on a homogeneous and heterogeneous cluster, which is compared with R-Storm and OLS. The homogeneous cluster is configured with 8 high capacity nodes, while the heterogeneous cluster has 2 high capacity and 4 low capacity nodes.

Figure 12 shows the experimental results for each of the three schedulers run on the homogeneous cluster. As can be seen from the figure, OLS and R-Storm have an average throughput of 3,700 and 5,200 respectively, while I-Scheduler with θ set to 1 has an average throughput of 6,900, representing an improvement of 86% and 30%, respectively. The lower throughput of OLS is because of the greedy approach used, which assigns tasks to the least loaded node. The task selection algorithm of R-Storm is unable to place any of the Plug Load component's tasks with other communicating tasks, separating this component from others, lowering the throughput. Further, with R-Storm's scheduling being offline, it cannot see the data transfer rates between tasks and cannot prioritise highly communicating tasks. As a result of OLS assigning tasks to the least loaded nodes, it uses all 8 nodes while R-Storm and I-Scheduler use 4 out of 8 nodes on average. As a re-

[8]`http://debs.org/debs-2014-smart-homes/`

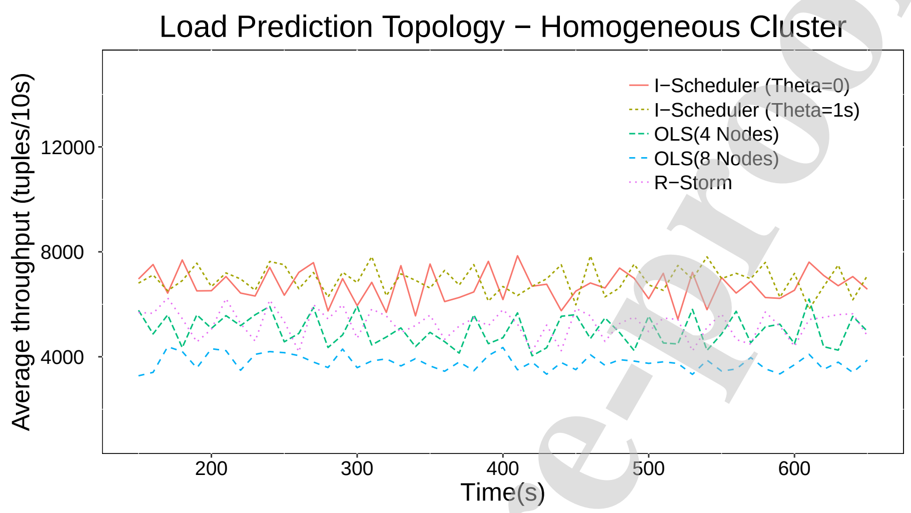


Figure 12: Throughput results of smart home load prediction application in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

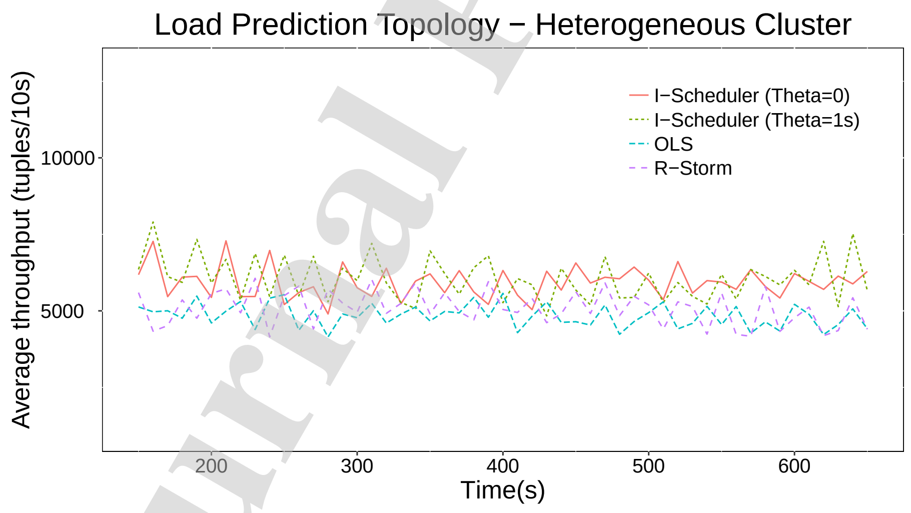


Figure 13: Throughput results of smart home load prediction application in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

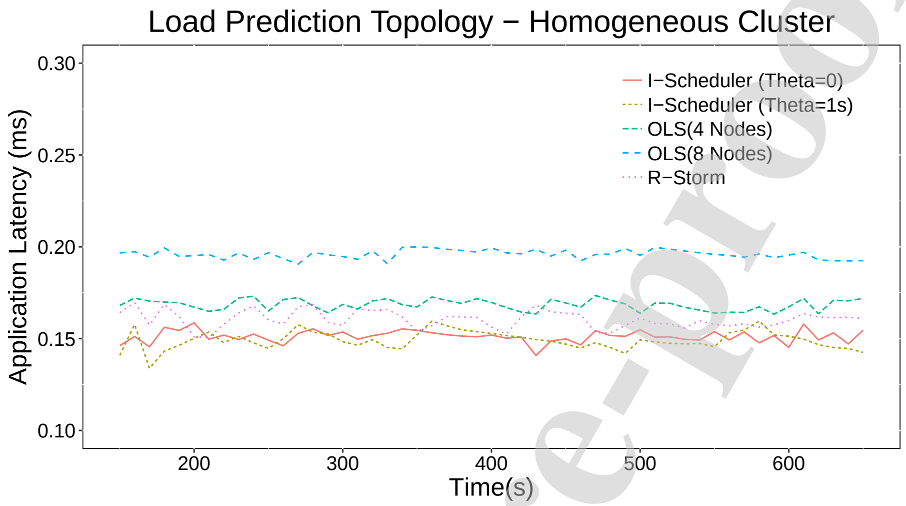


Figure 14: Latency results of smart home load prediction application in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

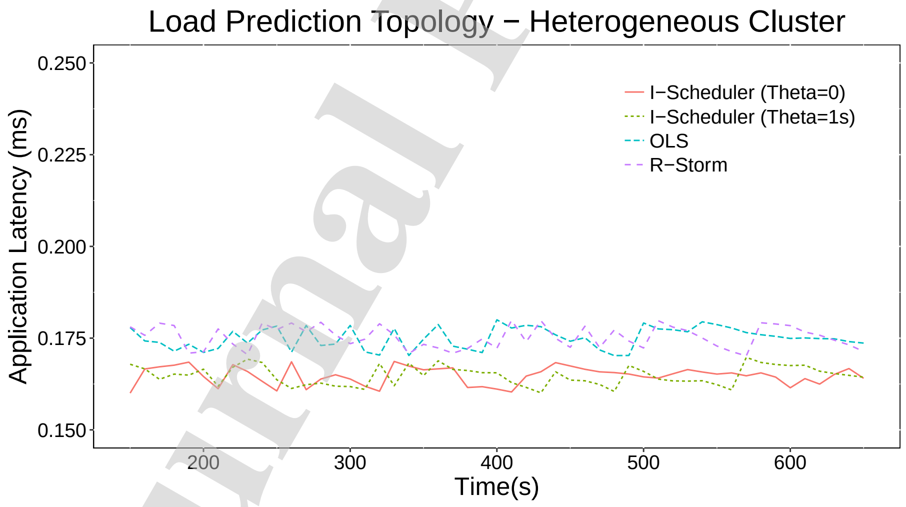


Figure 15: Latency results of smart home load prediction application in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

sult, R-Storm and I-Scheduler consolidate the cluster, reducing inter-node communication. For a fairer comparison which removes the impact of node consolidation, we rerun OLS on 4 nodes, similar to the number of nodes used by R-Storm and I-Scheduler. As can be seen in Figure 12, the throughput of OLS increases when run on fewer nodes, however it is still lower than the other two schedulers. From the figure, it can be seen that I-Scheduler with θ set to zero performs similarly to I-Scheduler with a θ of 1. This demonstrates the efficiency of the graph partitioning algorithms in finding groups of highly communicating tasks.

We then run all three schedulers on a heterogeneous cluster, consisting of 2 high capacity nodes and 4 low capacity nodes. This configuration evaluates which scheduler is better able to reduce inter-node communication by more efficiently assigning highly communicating tasks to the limited number of high capacity nodes. As can be seen in Figure 13, both settings of I-Scheduler can outperform R-Storm and OLS because of its task selection approach. It is worth noting that the throughput is lower for all schedulers than the homogeneous configuration, as there is more inter-node communication. Figures 14 and 15 show the execution latency for the application run by the three schedulers. From the figures, it can be seen that as the execution latency is reduced, we can achieve higher throughput, however the overall improvement in latency is not significant. In summary, these experimental results show that I-Scheduler can more efficiently place highly communicating tasks within the same node, resulting in a more efficient task assignment.

6.3. Top frequent routes in NYC taxi data

This application runs a query to find the top 10 most frequent routes of New York taxis for the last 30 minutes using the 2013 dataset, and is based on the first query of the DEBS 2015 grand challenge.[9] The layout of this application has a linear shape with one source operator which loads the data into the application and four computational operators, which are responsible for data pre-processing, performing a rolling count of routes, a first stage intermediate ranking of routes and final aggregation and ranking respectively. The numbers of tasks for each of the source, pre-process, rolling count, intermediate rank and final rank operators are 16, 16, 8, 4 and 1 respectively. We use a Redis server to store the trip records.

[9] `http://www.debs2015.org/call-grand-challenge.html`

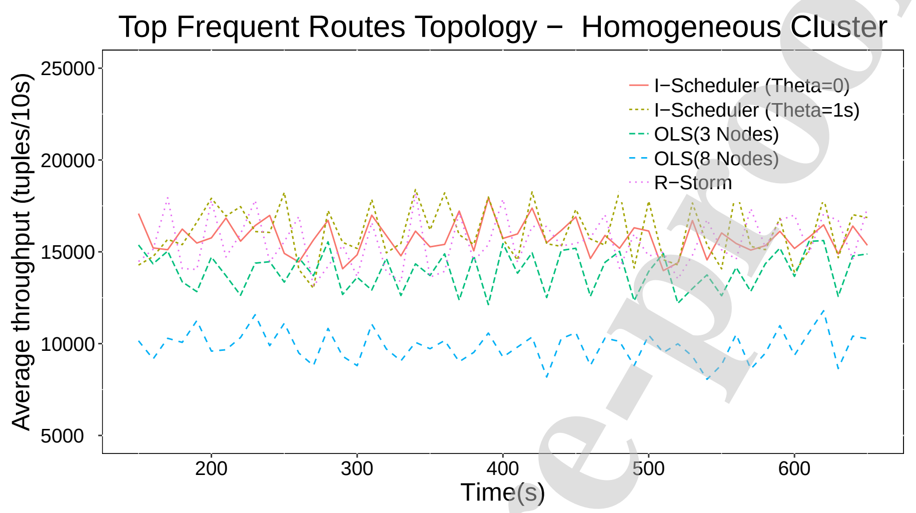


Figure 16: Throughput results of top frequent routes application in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

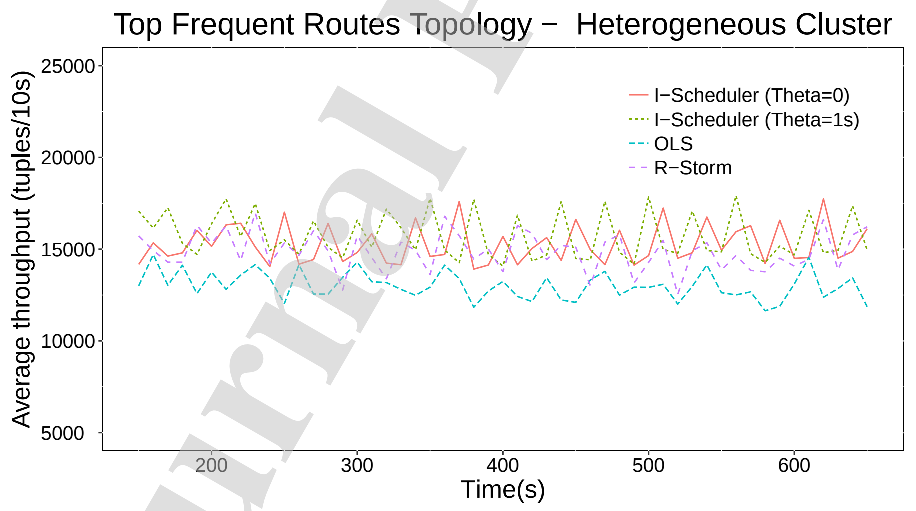


Figure 17: Throughput results of top frequent routes application in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

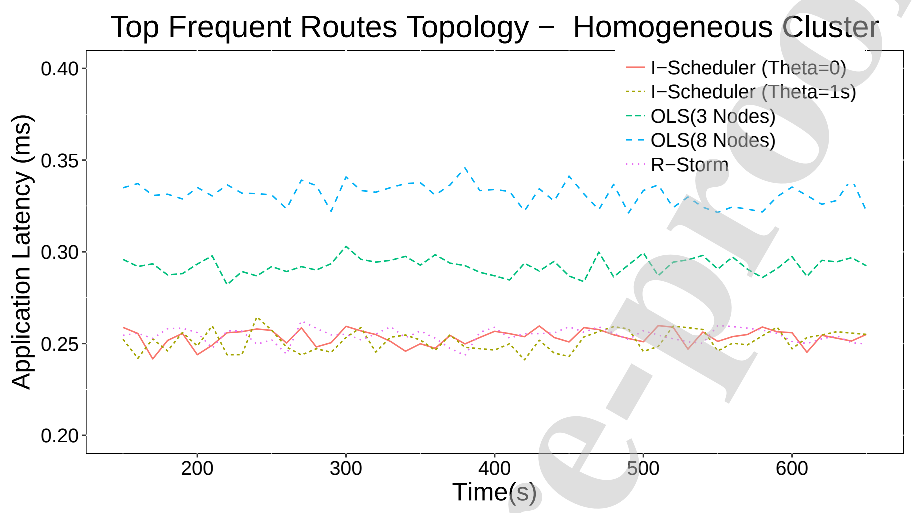


Figure 18: Latency results of top frequent routes application in a homogeneous cluster, using I-Scheduler, R-Storm and OLS

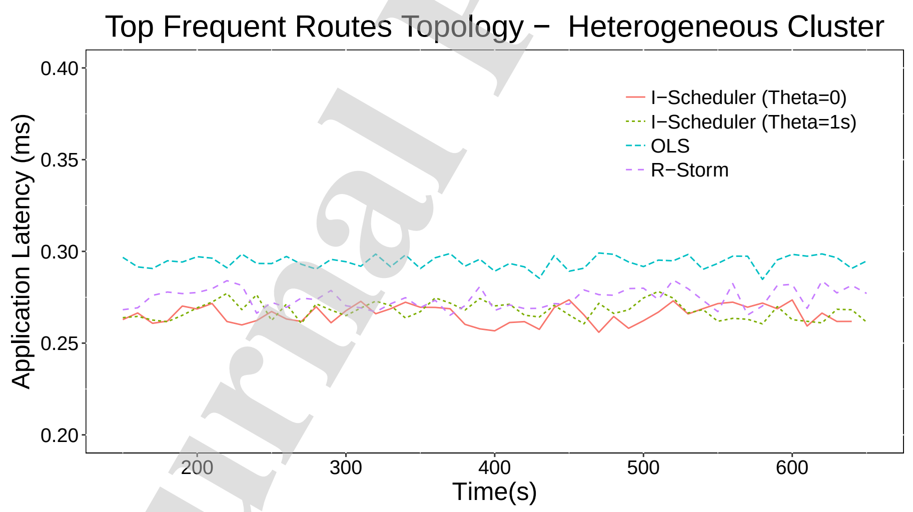


Figure 19: Latency results of top frequent routes application in a heterogeneous cluster, using I-Scheduler, R-Storm and OLS

Figure 16 shows the throughput for each of the three schedulers, running the top frequent routes application on an 8 node homogeneous cluster. As can be seen from the figure, the average throughput for I-Scheduler, R-Storm and OLS is 16,000, 15,500 and 9,800 respectively. The results show that both I-Scheduler and R-Storm are able to efficiently place highly communicating tasks together on the same node, reducing inter-node communication. It is worth noting again that R-Storm's scheduling operates offline, and required substantial fine tuning to be able to achieve the results presented, while I-Scheduler operates online without requiring such tuning. The lower average throughput of OLS is due to tasks being spread across nodes. OLS cannot see the whole communication pattern and co-locate the highly communicating tasks efficiently, because of its task pair view and greedy best fit approach. By rerunning OLS on a 3-node cluster, a higher throughput of 13,900 was achieved. 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 17, it can be seen that I-Scheduler has an average throughput of 15,700 while R-Storm and OLS have an average of 14,800 and 13,000 respectively. Finally, Figures 18 and 19 show the latency for each scheduler running on the homogeneous and heterogeneous cluster configurations.

In summary, the experimental results show that I-Scheduler is able to efficiently assign tasks for each of the streaming application layouts, increasing the throughput and reducing execution latency, when compared to R-Storm and OLS. This is important as many real-world streaming applications have either a linear, diamond, star or a combination of these layouts. By efficiently handling each layout, we have demonstrated that I-Scheduler is capable of handling a variety of possible traffic patterns. As fast responses are a critical requirement of many streaming applications, our ability to reduce the execution latency and increase throughput will help improve the performance in such real-world use cases. Further, by fully utilising and consolidating cluster resources we are able to reduce the cost of running such applications in the cloud, where the resources are pay-as-you-go, and it can be costly to keep underutilised resources running.

The oscillating fluctuations in throughput that can be seen in the experimental results are similar to the fluctuations seen in previous work [9]. From the detailed log files, there is no unexpected characteristics, so we do not believe that there is any cause for concern regarding this behaviour.

7. Conclusions and Future Work

In this paper, we have presented I-Scheduler, which iteratively uses graph partitioning to efficiently fuse groups of highly communicating tasks into a single task, in order to reduce the task size of the application graph. Once the task graph is sufficiently small, optimisation software is able to determine the task assignment within a user defined time tolerance. In cases where the task graph is too large to be solved by the optimisation software or the user cannot tolerate any delays in finding a schedule, a fallback heuristic method is used, where each fused task is assigned to a node with a relative capacity. I-Scheduler performs a second level of graph partitioning on each node, to minimise the inter-worker communication. The experimental results have shown that I-Scheduler can outperform R-Storm by 3–30% and OLS by 20–86%, when run on two real world applications.

However, there are some limitations to our proposed scheduler. I-Scheduler relies on the efficiency of the K-way graph partitioning algorithm and optimisation software such as CPLEX. It is also dependent on choosing the right partitions for fusion during each iteration, which was reflected in Section 5 for task size 30 for diamond and star layout where I-Scheduler was unable to find the optimal solution, but did achieve near optimal results. Moreover, I-Scheduler only considers the CPU capacity when assigning tasks to compute nodes, which leaves the potential for bottlenecks to occur when assigning either memory or network bandwidth intensive tasks. By including tasks' memory and bandwidth requirements in our future work, we can mitigate any bottlenecks that may arise when scheduling a variety of applications. Similarly, by considering additional resource requirements, we can extend I-Scheduler to include QoS as an optimisation goal. This will involve evaluating I-Scheduler on additional micro-benchmarks and real-world applications which have different communication and congestion patterns to those already implemented.

A further challenge is determining when rescheduling should occur, in response to changes in the workload. Currently, rescheduling is based upon recent traffic patterns, without the use of prediction. While this approach is common, it can result in inefficient schedules when an application has bursty traffic patterns. To handle this worst case scenario, workload prediction could be included in the task scheduler, similar to [42, 43]. It is worth noting that this worst case scenario did not occur in this paper as the experimental applications used had a stable traffic pattern. The use of prediction will help guide the scheduler in determining when the workload traffic pattern is likely to change, making it more proactive. Being able to make accurate, longer term predictions will help avoid the potential situation where a scheduler is repeatedly changing in response to bursty traffic. It will allow for an evaluation of the tradeoff between the improved performance of a new schedule, and the cost incurred by rescheduling. Our future work will also investigate improved methods for state migration of tasks, as surveyed in [44], which would allow for more efficient rescheduling. Currently, Apache Storm's state migration stops all tasks from executing to allow them to be moved, before being restarted. This is inefficient as it interrupts execution, and causes significant overhead. There is great interest in finding a solution to this problem, which is an area of ongoing research. By implementing an improved method of state migration, we would be able to have a more efficient scheduler, with lower overhead, enabling more frequent rescheduling to be performed in response to bursty traffic patterns.

References

References

[1] L. Eskandari, J. Mair, Z. Huang, D. Eyers, Iterative scheduling for distributed stream processing systems, in: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, ACM, 2018, pp. 234–237.

[2] N. Marz, J. Warren, Big Data: Principles and best practices of scalable realtime data systems, Manning Publications Co., 2015.

[3] 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.

[4] 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.

[5] U. Srivastava, K. Munagala, J. Widom, Operator placement for in-network stream query processing, in: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, 2005, pp. 250–258.

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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 9th ACM/IFIP/USENIX International Conference on Middleware, Springer, 2008, pp. 306–325.

[16] 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.

[17] Y. Jiang, Z. Huang, D. H. Tsang, Towards max-min fair resource allocation for stream big data analytics in shared clouds, IEEE Transactions on Big Data.

[18] Y. Wang, Z. Tari, M. R. HoseinyFarahabady, A. Y. Zomaya, Qos-aware resource allocation for stream processing engines using priority channels, in: Proceedings of the 16th International Symposium on Network Computing and Applications (NCA), IEEE, 2017, pp. 1–9.

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

[20] 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.

[21] 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: Proceedings of the ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer, 2009, pp. 308–327.

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

[23] 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.

[24] 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.

[25] 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.

[26] D. Sun, R. Huang, A stable online scheduling strategy for real-time stream computing over fluctuating big data streams, IEEE Access (2016) 8593–8607.

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

[28] X. Liu, R. Buyya, D-Storm: Dynamic resource-efficient scheduling of stream processing applications, in: Proceedings of the 23rd International Conference on Parallel and Distributed Systems (ICPADS), IEEE, 2017, pp. 485–492.

[29] P. Basanta-Val, N. Fernández-García, A. Wellings, N. Audsley, Improving the predictability of distributed stream processors, Future Generation Computer Systems 52 (2015) 22–36.

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

[31] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, R. Arif, Tcep: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms, in: Proceedings of the 12th ACM International Conference on Distributed Event-Based Systems, 2018, pp. 136–147.

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

[33] 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.

[34] A. Benoit, A. Dobrila, J.-M. Nicod, L. Philippe, Scheduling linear chain streaming applications on heterogeneous systems with failures, Future Generation Computer Systems 29 (5) (2013) 1140–1151.

[35] L. Eskandari, J. Mair, Z. Huang, D. Eyers, T3-scheduler: A topology and traffic aware two-level scheduler for stream processing systems in a heterogeneous cluster, Future Generation Computer Systems 89 (2018) 617–632.

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

[37] L. Xu, B. Peng, I. Gupta, Stela: Enabling stream processing systems to scale-in and scale-out on-demand, in: Proceedings of 2016 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2016, pp. 22–31.

[38] A. Shukla, Y. Simmhan, Model-driven scheduling for distributed stream processing systems, Journal of Parallel and Distributed Computing 117 (2018) 98–114.

[39] M. Nardelli, V. Cardellini, V. Grassi, F. L. Presti, Efficient operator placement for distributed data stream processing applications, IEEE Transactions on Parallel and Distributed Systems 30 (8) (2019) 1753–1767.

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

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

[42] N. Hidalgo, D. Wladdimiro, E. Rosas, Self-adaptive processing graph with operator fission for elastic stream processing, Journal of Systems and Software 127 (2017) 205–216.

[43] T. Li, J. Tang, J. Xu, Performance modeling and predictive scheduling for distributed stream data processing, IEEE Transactions on Big Data 2 (4) (2016) 353–364.

[44] Q.-C. To, J. Soto, V. Markl, A survey of state management in big data processing systems, The International Journal on Very Large Data Bases 27 (6) (2018) 847–872.

Leila Eskandari received her PhD in Computer Science from the University of Otago, New Zealand. She previously completed an MSc degree from Sharif University of Technology in Network Engineering and a BSc degree from Ferdowsi University of Mashhad in Computer Engineering, Iran. Her research interests include big data processing, distributed and cloud 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, distributed and cloud 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 an Associate Professor 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. He has broad research interests including green computing, information flow control, network security, and distributed and cloud computing.

Author Photo

Click here to access/download;Author Photo;Leila_Eskandari.jpg

Author Photo

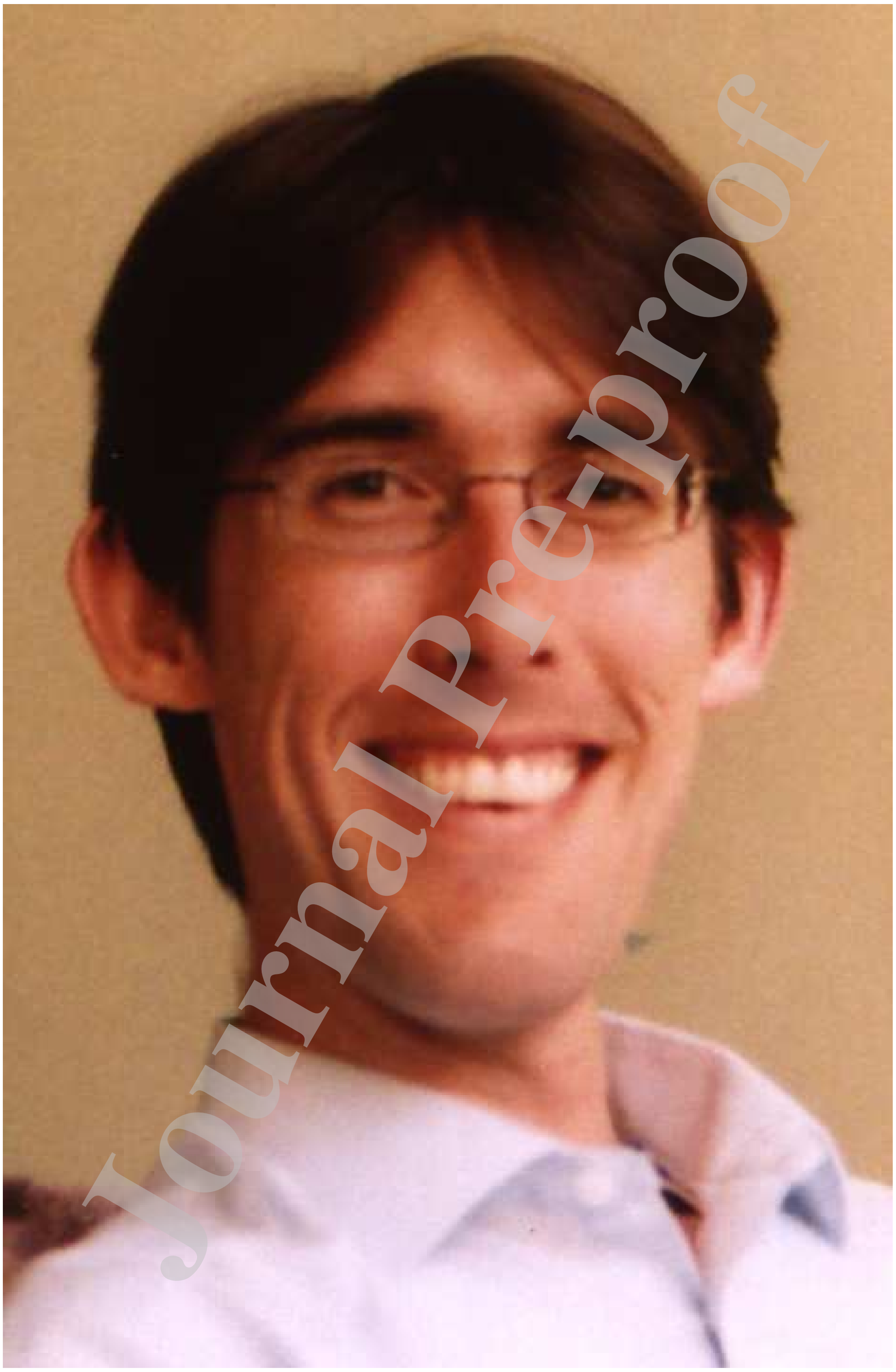
Click here to access/download;Author Photo;Jason_Mair.jpg

Author Photo

Click here to access/download;Author Photo;Zhiyi_Huang.jpg

Author Photo

Click here to access/download;Author Photo;David_Eyers.jpeg



Author Statement

This work was completed as a part of Leila Eskandari’s PhD thesis, under the supervision of David Eyers and Zhiyi Huang with collaboration of Jason Mair.
Leila Eskandari: Conceptualization, Writing - Original Draft **Jason Mair:** Writing - Review & Editing **Zhiyi Huang:** Writing - Review & Editing, Supervision **David Eyers:** Writing - Review & Editing, Supervision

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: