IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 14, 2020, accepted June 19, 2020, date of publication June 24, 2020, date of current version July 3, 2020.

*Digital Object Identifier 10.1109/ACCESS.2020.3004612*

# Pipeline-Based Linear Scheduling of Big Data Streams in the Cloud

**NICOLETA TANTALAKI[1], STAVROS SOURAVLAS[1], (Member, IEEE), MANOS ROUMELIOTIS[1], (Member, IEEE), AND STEFANOS KATSAVOUNIS[2]**

[1]Department of Applied Informatics, University of Macedonia, 54636 Thessaloniki, Greece
[2]Department of Production and Management Engineering, Democritus University of Thrace, 67100 Xanthi, Greece

Corresponding author: Nicoleta Tantalaki (nicoleta@uom.gr)

This work was supported by the project Algorithms and Applications in Social Networks and Big Data Systems which is funded by the Unified Insurance Fund of Independently Employed (ETAA), Greece.

**ABSTRACT** Nowadays, there is an accelerating need to efficiently and timely handle large amounts of data that arrives continuously. Streams of big data led to the emergence of several Distributed Stream Processing Systems (DSPS) that assign processing tasks to the available resources (dynamically or not) and route streaming data between them. Efficient scheduling of processing tasks can reduce application latencies and eliminate network congestions. However, the available DSPSs' in-built scheduling techniques are far from optimal. In this work, we extend our previous work, where we proposed a linear scheme for the task allocation and scheduling problem. Our scheme takes advantage of pipelines to handle efficiently applications, where there is need for heavy communication (all-to-all) between tasks assigned to pairs of components. In this work, we prove that our scheme is periodic, we provide a communication refinement algorithm and a mechanism to handle many-to-one assignments efficiently. For concreteness, our work is illustrated based on Apache Storm semantics. The performance evaluation depicts that our algorithm achieves load balance and constraints the required buffer space. For throughput testing, we compared our work to the default Storm scheduler, as well as to R-Storm. Our scheme was found to outperform both the other strategies and achieved an average of 25%-40% improvement compared to Storm's default scheduler under different scenarios, mainly as a result of reduced buffering ($\approx$ 45% less memory). Compared to R-storm, the results indicate an average of 35%-45% improvement.

**INDEX TERMS** Stream processing, scheduling, big data, pipelines, distributed systems.

## I. INTRODUCTION

Over past 20 years data has increased in a large scale and in various fields. The rapid growth of cloud computing and Internet of Things (IoT) promote the sharp growth of data further. Managing the produced data and gaining insights from it, is a challenge and possibly a key to competitive advantage. Many organizations' decisions rely heavily on processing and analyzing this data the time it arrives. Environmental monitoring, fraud detection, emergency response are just a few examples of applications that require continuous and timely processing of information [2], [3].

However, managing in real-time incoming data that arrives continuously at volumes and high velocity, far exceeds the capabilities of individual machines. Stream data processing requires continuous calculation without interruption, and high reliability requirements on resources. Messages should be processed in-stream without any requirement to store them to perform any operation or sequence of operations [4]. Cloud computing technology with superior computational power and high reliability rises as a promising solution for this kind of processing.

In-memory computing is based on using a distributed main memory system to process (and store) big data in real time and is used to meet performance related requirements like latency and throughput that are extremely important in streaming applications [3]. If streams are not managed carefully processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure high

The associate editor coordinating the review of this manuscript and approving it for publication was Ali Kashif Bashir.

throughput and increased performance. On the other hand, heavily used machines result in memory thrashing, node failures and increased network congestion. Overloaded and underutilized machines should be avoided as imbalances lead to increased computations and deteriorate system's performance [5]. Moreover, despite the dropping price of memory, using large amounts of RAM to run everything in-memory can be expensive, so proper mechanisms are needed to handle it in an effective way.

The demand of elaborate orchestration over a collection of machines becomes imperative to meet the required performance for streaming applications and reduce costs in the cloud. The process of scheduling the tasks into the cluster resources in a way that achieves minimization of task completion time and improves resource utilization is known as *task scheduling*. It focuses on which tasks to be placed on which previously obtained resources, controls the order of job execution [6], [7] and is an NP-hard problem [8], [9].

Several data stream processing systems (DSPSs) like Apache Storm [10], Spark Streaming [11], Samza [12] and Flink [13] have specifically emerged to address the challenges of processing high-volume, real-time data. They are designed to execute complex streaming applications as Directed Acyclic Graphs (DAGs) over tuples of a stream. They leverage data parallelism using mutiple threads of execution per task (replicas), in addition to pipelined and task-parallel execution of the desired DAG (see subsection II.A ''Preliminaries in Storm'') [14].

Although there is extended literature on task scheduling in batch systems like Hadoop [15], [16], the techniques used do not fit in real-time processing of streams mainly because of the difference in the computational model used in each case. In batch processing systems, computations are assigned to the nodes where the required data is stored, while in stream processing systems most communicating tasks have to be placed together on one node or rack. The available DSPSs' in-built scheduling techniques are far from optimal. For instance, round robin that is the strategy used as the default scheduler of Apache Storm does not take into account the cost of moving tuples through network links to let them traverse the correct sequence of tasks, defined in users' applications. Several heuristics have been proposed, taking decisions either offline (static solutions) or online (dynamic solutions), based on the applications' structure, tasks' characteristics, resources availability, workload and traffic conditions. These heuristics are based on a number of different assumptions, have different optimization goals and aim at minimizing different utility functions, like latency and network usage.

In this work, we extend our previous work [1], where we proposed a static, matrix-based, task allocation and scheduling scheme performed in a memory-efficient and well-balanced manner. We used a pipeline-based scheduling approach that reduces the required buffer size to effectively speed up processing. Our approach is inspired by the idea that the reduction of the required buffer space can minimize tuple losses (and possible re-submissions) and overhead delays that heavily affect system's performance. The use of pipelines provides an enhanced scheme for internal queuing of the incoming tuples and helps to process and forward them as they arrive at the target node. Moreover, load balance optimises the response time for each task, avoiding unevenly overloading nodes while others are left idle. To the best of our knowledge most of the existing solutions found in literature rarely consider memory consumption in their analysis and while they take into account the capability of the resources, they generally ignore their load.

This time, we prove that the task allocation scheme that forms the basis for our scheduler is periodic and thus, the number of necessary computations can be decreased. Moreover, taking into consideration that relevant tasks should better be assigned to the same or adjacent nodes, we provide a communication refinement algorithm that was only briefly described in our previous work. We also add another pipeline-based scheme to handle ''many-to-one'' assignments between tasks effectively and improve the system's performance. ''Many-to-one'' assignments led to delays in tuple processing in our previous work. We also conduct more experiments to validate our scheduler's efficiency and compare its performance with the round strategy, that is the default Apache Storm scheduler, and the R-Storm [17] scheduler from the state-of-the-art.

We are going to use the operator-based model and Apache Storm's semantics to describe our work as it is a mature project, with a very large community and popularity in cloud computing industry, due to its high reliability and good processing mode [3]. Our approach, though, is generic to any data flow system and suitable for deployment and use in large-scale clusters. While the proposed scheduler considers only static scheduling for now, it can be extended to dynamic scenarios but this is left for future work as described in Section VII.

Our contributions are described below. We provide a general topology-aware formulation of the task allocation and scheduling problem using matrix transformations, which introduces a scheme that:

- reduces the required buffer space. The queue size at each task does not grow unmanageably, as each task receives tuples from only one part of the stream at a time. Pipeline stalls are also used to increase throughput and reduce the number of tuple losses (tuples which are lost because they cannot be processed on time).
- reduces the inter-node communication cost, thus the total communication time, by placing to the maximum extent the communicating tasks to nearby nodes.
- is balanced; provides almost equal processing load to the cluster's nodes.
- is periodic; the allocation of a specific number of tasks is sufficient to complete the overall task allocation procedure.
- has linear complexity, determining faster computations.

The rest of this paper is organised as follows: Section II introduces the reader to Apache Storm and its default scheduler to present the mathematical background of our scheme based on Storm's semantics. Section III describes our task allocation and scheduling algorithms and includes the proof of our scheme's linear complexity. Section IV depicts the application of our algorithms using two motivating examples. Section V reports the experiments done on our approach. A discussion on the findings of our experiments is also included. Section VI presents the related work regarding both solutions already incorporated in prominent DSPSs and heuristics found in literature. Finally, Section VII provides conclusions and future directions.

## II. BACKGROUND AND PROBLEM FORMULATION

In this section we briefly describe the way Apache Storm represents and executes a stream processing application, as we are going to use its semantics to describe our work. Then, we present the mathematical background behind our prototype design in details.

### A. PRELIMINARIES IN STORM

A DSPS like Apache Storm represents a streaming application as a Directed Acyclic Graph (DAG), where the vertices show the operators that encapsulate processing logic (called *components* in Storm) and the edges show the data flow direction. Apache Storm uses the terms *topology* for a DAG while a *task* is a component's instance. The components of a DAG are divided to *spouts* and *bolts*. A spout is a source of streams in a topology and a bolt receives streams, processes them, and forwards them for further processing.

We distinguish between the logical and physical abstraction in Storm. Fig. 1 shows the intercommunication of tasks within a common topology in Storm (logical abstraction). There is one spout, and four bolts, and each component has four tasks. Links between components in the topology indicate how tuples are passed around.

Part of defining a topology includes specifying for each bolt which streams it should receive as input. A stream grouping defines how a stream should be partitioned among bolts' tasks. Shuffle grouping is the most commonly used grouping [18]. It distributes tuples in a uniform randomly across the tasks. An equal number of tuples should be processed by each bolt. It is ideal when the processing load needs to be distributed uniformly across the tasks and when there is no requirement of any data-driven partitioning. It can be useful for doing atomic operations such as a math operation but in case the operation can't be randomly distributed (e.g. in case of word count), it does not fit.

In the physical layer, there is a master node (Storm's default scheduler is a part of the Nimbus daemon on the master node) and a set of physical machines (worker nodes) that can host multiple operators. Nimbus communicates and cooperates with a Zookeeper service to maintain a consistent list of active worker nodes and to detect failures. Slots on each worker node indicate the number of workers (Java Virtual Machines-JVMs) that can run on this node. The number of slots are typical set to the number of cores and execute a part of the topology. Each worker running is launched and monitored to have possible failures handled by a supervisor executing on its worker node. Each operator's code is executed by *threads* or *executors* and multiple threads of the same component execute the same computation on different parts of the stream in parallel. The number of parallel instances of an operator (i.e. replicas) determine the operator's replication degree (also known as *parallelization degree*). In Apache Storm it is set when a topology is submitted by the user. Consequently, each component in a Storm topology has several executors and each executor can have several tasks. By default, Storm will run one task per thread. Each JVM can host a number of threads from different components of the same topology.

Storm's default scheduler distributes the tasks of bolts and spouts uniformly across all the nodes in the cluster in a round robin fashion, but in this way it is not possible to balance load. Tasks from a single bolt or spout will most likely be placed on different physical machines but the main consideration in this strategy is that the communication between tasks is not taken into account. It is a common assumption that nearby tasks would most probably communicate during processing, so high communication latencies can be improved, if we achieve task locality. Our approach looks at how components are interconnected within the topology to determine what are the executors (instances) that should be assigned to the same or nearby nodes. The key idea is to use communication patterns among executors, trying to place the most communicating executors as close as possible. Such a scheduling is executed before the topology is started, so neither the load nor the traffic are taken into account, and consequently no constraint about memory or CPU is considered. Not taking into account these points obviously limits the effectiveness of an offline scheduler but our pipeline-based scheduling technique tries to balance load and decrease queue waiting times, enabling a very simple implementation that provides a good performance.

Without loss of generality, we assume that the tuple processing time of all the tasks is almost the same (this is a logical assumption used also by shuffle grouping, see [8], [14]). Under this hypothesis, we can divide the overall processing into a set of well-defined *processing steps* and *stages*, terms that are defined in the following subsection.

### B. PROBLEM FORMULATION

Let us consider a cluster of $N$ nodes, and an application topology like the one Fig. 1, where the interconnection between the components is shown. In this figure, there are 4 bolts and 1 spout, each of them having 4 threads. Each thread executes one task, so we can refer to tasks and threads interchangeably from this moment on. We define the initial *matrix*, $M_{init}$, as a table that stores the tasks assigned to each node by the default round-robin Storm scheduler. This table can have two forms: in the first form, the tasks are indicated as letters and

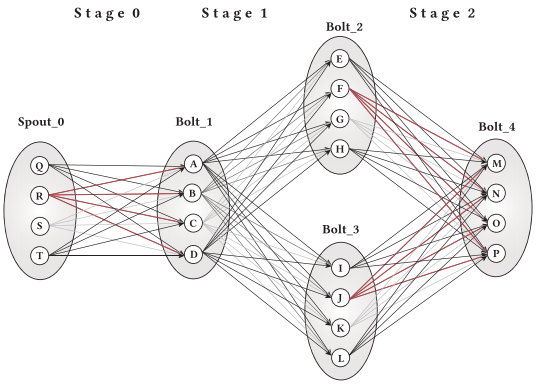


**FIGURE 1.** Intercommunication of tasks in a streaming application.

in the second they have been replaced by numbers.

$$M_{init} = \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 \\ \mathrm{Q} & \mathrm{R} & \mathrm{S} & \mathrm{T} & \mathrm{A} & \mathrm{B} \\ \mathrm{C} & \mathrm{D} & \mathrm{E} & \mathrm{F} & \mathrm{G} & \mathrm{H} \\ \mathrm{I} & \mathrm{J} & \mathrm{K} & \mathrm{L} & \mathrm{M} & \mathrm{N} \\ \mathrm{O} & \mathrm{P} & \Omega & \Omega & \Omega & \Omega \end{bmatrix} \equiv \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & \textcircled{20} & \textcircled{21} & \textcircled{22} & \textcircled{23} \end{bmatrix} \quad (1)$$

The tasks indicated by Ω are added by our model as ''dummies'' and they are used to avoid empty values in $M_{init}$. In the numbered representation, the dummy tasks are circled. A dummy task plays no role in the actual processing.

Without loss of generality, we assume that the tuple processing time of all the tasks is almost the same. Under this hypothesis, we can divide the overall processing into a set of well-defined *processing steps* and *stages*. In Definitions 1 and 2, we rigorously define these terms.

*Definition 1: In our context, a processing step defines a set of communications between nodes (which result in communications between tasks), such that each node receives stream parts from one node only. Once received, these stream parts are assigned to proper threads, which, in their turn, are executed on the data received.*

*Definition 2: In our context, a processing stage defines the points of a logical path of spouts and bolts that a stream has to follow, from its generation until the end of its processing.*

In the example of Fig. 1, there are 3 stages: Stage 0, where the stream parts move from the spout to Bolt 1, Stage 1, where stream parts move from Bolt 1 to Bolts 2 and 3, and Stage 2, where stream parts move from Bolts 2 and 3 to Bolt 4. Let us define such an initial task allocation as $\mathcal{A}(t, N)$, where $t$ is the number of tasks per spout/bolt and $N$ is the set of nodes in the cluster. Proposition 1 forms the basis of our task allocation. Algorithm 1 that is presented in subsection III.A derives directly from the proof of this proposition.

*Proposition 1: The initial matrix of Eq.1 can always be transformed into an intermediate task allocation matrix, $M'_{inter}$, where the rows of $M'_{inter}$ define a communication between the cluster's nodes, such that each node will be receiving stream parts from one node at a time.*

*Proof:* If we want to derive a mathematical formula describing the round-robin placement of tasks in $M_{init}$, this would be

$$M_{init}(i, j) = iN + j, \quad (2)$$

where $j$ is the column index, $j \in [0 \ldots N-1]$, $i$ is the row index, $i \in \left[0, \ldots, \left\lfloor \frac{\mathcal{T}}{N} \right\rfloor - 1\right]$, and $\mathcal{T}$ is the total number of tasks in the system. For our example, the application of Eq.2 would produce the arithmetic representation of Eq.1. By dividing $M_{init}$ by $t$, we obtain an intermediate matrix, $M'_{init}$, which represents tasks with their corresponding component IDs (assuming that the spout has ID = 0, and bolts 1-4 have IDs 1-4, respectively). The ID = 5 corresponds to the dummies:

$$M'_{init} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 4 & 4 \\ 4 & 4 & 5 & 5 & 5 & 5 \end{bmatrix} \equiv \begin{bmatrix} \mathrm{Q} & \mathrm{R} & \mathrm{S} & \mathrm{T} & \mathrm{A} & \mathrm{B} \\ \mathrm{C} & \mathrm{D} & \mathrm{E} & \mathrm{F} & \mathrm{G} & \mathrm{H} \\ \mathrm{I} & \mathrm{J} & \mathrm{K} & \mathrm{L} & \mathrm{M} & \mathrm{N} \\ \mathrm{O} & \mathrm{P} & \Omega & \Omega & \Omega & \Omega \end{bmatrix} \quad (3)$$

Actually, $M'_{init}$ shows the initial allocation of spout/bolt tasks into the system's nodes. Now, we set $G = \gcd(t, N)$, the greatest common divisor of $t$ and $N$. Thus, we can find integers $t'$ and $N'$, such that $t = t'G$ and $N = N'G$. Under this assumption, let us consider two rows of $M'_{init}$, one with row index $i$ and one with row index $i'$, where $i' = i + \mu t'$, for some integer $\mu$. Then,

$$M'_{init}(i, j) = \frac{iN + j}{t} \quad (4)$$

$$M'_{init}(i', j) = \frac{i'N + j}{t} = \frac{(i + \mu t')N + j}{t} \quad (5)$$

By subtracting (4) from (5), we get

$$\begin{aligned} M'_{init}(i', j) - M'_{init}(i, j) &= \frac{(i + \mu t')N + j - iN - j}{t} \\ &= \frac{iN + \mu t'N + j - iN - j}{t} \\ &= \frac{\mu t'N}{t} = \frac{\mu t'GN'}{t} \end{aligned}$$

All the above divisions are integer. The notion of a *class* will help us proceed with our proof.

*Definition 3: A class $k$ is defined as a group of the row indices of $M'_{init}$, such that $i \pmod{t'} = k$.*

Because $\mu t'GN'$ is divided by $N'$, in every column, all the elements of $M'_{init}$ with row indices that differ by $t'$ will produce the same modulo when divided by $N'$. These elements

are $G$ in total and can be brought together by following the steps below:

1. For all the row indices of $M'_{init}$, find all the $k$ values, such that $i \pmod{t'} = k$. These distinct $k$ values are called *classes*.
2. Take the first row $i$, such that $i \pmod{t'} = k$, that is $i$ belongs to class $k$.
3. All the row indices in every class will move to a new row index:

$$i_{new} = \left\lfloor \frac{i}{t'} \right\rfloor + kG \tag{6}$$

The row transpositions will generate a *transposed matrix* $M'_{trn}$. The transposed matrix $M'_{trn}$ has two properties: (1) For every column of $M'_{trn}$, each class's elements produce the same modulo when divided to $N'$, and (2) $M'_{trn}$ can be divided into a set of $t' \times N'$ sub-matrices of size $G \times G$. In other words, it is a matrix with $t'$ rows and $N'$ columns of square sub-matrices of size $G \times G$. We use the notation $\Delta$, to denote these square sub-matirces:

$$M'_{trn} = \begin{bmatrix} \Delta_{0,0} & \Delta_{0,1} & \cdots & \Delta_{0,N'-1} \\ \Delta_{1,0} & \Delta_{1,1} & \cdots & \Delta_{1,N'-1} \\ \vdots & \vdots & \vdots & \vdots \\ \Delta_{t'-1,0} & \Delta_{t'-1,1} & \cdots & \Delta_{t'-1,N'-1} \end{bmatrix} \tag{7}$$

each element of $M'_{trn}$ is described by the following indices:

- $i$ row index of $M'_{trn}$ as a whole, $i \in [0, \ldots \left\lfloor \frac{\mathcal{T}}{N} \right\rfloor - 1]$
- $j$ column index of $M'_{trn}$ as a whole, $j \in [0, \ldots, N-1]$
- $r$ row index of sub-matrices, $r \in [0, \ldots, t'-1]$
- $c$ column index of sub-matrices, $c \in [0, \ldots, N'-1]$
- $r'$ row index of an element within a sub-matrix, $r' \in [0, \ldots, G-1]$
- $c'$ column index of an element within a sub-matrix, $c' \in [0, \ldots, G-1]$

Now, since the elements of $M'_{trn}$ are, in groups, having the same modulo $N'$ values, we can express $M'_{trn}$ as a sum of a constant factor $M'_1$ and a matrix $M'_2$, where:

$$M'_1 = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 1 \times N' & 1 \times N' & \cdots & 1 \times N' \\ \vdots & \vdots & \vdots & \vdots \\ (G-1) \times N' & (G-1) \times N' & \cdots & (G-1) \times N' \end{bmatrix} \tag{8}$$

$$M'_2 = \left\lfloor \frac{rN' + c}{t'} \right\rfloor \tag{9}$$

Note that $M'_1$ is a matrix with $G$ in total rows whose elements are multiplied by $N'$, while the $M'_2$ derives by applying Eq. (2) on the elements of a sub-matrix of size $G \times G$, substituting $r$ for $i$, $N'$ for $N$, and $c$ for $j$, and dividing by $t'$ instead of $t$, $0 \le r \le t'-1$ and for $0 \le c \le N'-1$.

To complete the proof for Proposition 1, we need to show that the elements of $M'_2$ can be aligned in such a manner that all the rows contain different elements, since $M'_1$ is a constant.

Assume that two of the $G \times G$ sub-matrices in the same row of $M'_{trn}$ have the same elements. Because these sub-matrices are a sum of a constant part $M'_1$ and a variable part $M'_2$, we examine the variable part $M'_2$. Eq. (9) states that the row index $r$ of the sub-matrices $M'_{trn}$ has the same modulo when divided by $t'$ but $c \pmod{t'}$ is different for the sub-matrix columns. Therefore, to have a row of different sub-matrices, we need to move at most $N'-1$ sub-matrices from row $r$ to row $r_{new}$, using the formula:

$$r_{new} = (rN' + c) \pmod{t'} \tag{10}$$

Because $(rN' + c) \pmod{t'} = rN' \pmod{t'} + c \pmod{t'}$, it follows that the sub-matrices found in the same row have the same $rN' \pmod{t'}$ value, but their $c \pmod{t'}$ value changes between consecutive columns.

Finally, to separate the same-valued elements found in each row of a sub-matrix because of the constant factor of $M'_1$ (see Eq. 8), we have to circularly move these elements to different rows in the sub-matrix, so that they are put in diagonal positions. This can be done because these matrices are square, of size $G \times G$. Thus, every row element will move to a new row index $r'_{new}$ according to:

$$r'_{new} = (r' + c') \pmod{G} \tag{11}$$

Now, the elements of every row of $M'_{trn}$ are different between them. The transformations of Eq.(10) and Eq.(11) produce $M'_{inter}$. Thus, if we read each row element $M'_{inter}$ as a target node for the corresponding node label in every column (for example, nodes $N_0 - N_5$ in Eq. 1), then each row represents a processing step, where each node communicates with only one other node and Proposition 1 is proven. ■

Proposition 1.1 proves the periodicity of the transformations described and proved above.

*Proposition 1.1: The transformation procedure that leads to a task allocation matrix is periodical and its period is LCM(t, N), the least common multiple of t and N.*

*Proof:* Assume that two tasks $t_\lambda$ and $t_\mu$ are initially distributed in node $n$ and belong to the same class $k$. Then, $n = t_\lambda \mod N$ and $n = t_\mu \mod N$. From these two relationships, it follows that

$$(t_\lambda - n) \mod N = 0 \tag{12}$$

$$(t_\mu - n) \mod N = 0 \tag{13}$$

From Eq.(12) and (13), it follows that:

$$(t_\lambda - t_\mu) \mod N = 0 \tag{14}$$

Suppose that $t_\lambda$ is located at line $i_1$ and $t_\mu$ is located at line $i_2$. Since these two tasks belong to the same class $k$, from the definition of classes, we have: $i_1 \mod t' = k$ and $i_2 \mod t' = k$. From these two relation ships, we have

$$(i_1 - k) \mod t' = 0 \tag{15}$$

$$(i_2 - k) \mod t' = 0 \tag{16}$$

From Eq. (15) and (16), we get:

$$\begin{aligned} & (i_1 - i_2) \mod t' = 0 \\ \Rightarrow\ & G(i_1 - i_2) \mod Gt' = 0 \\ \overset{Gt'=t}{\Rightarrow}\ & G(i_1 - i_2) \mod t = 0 \end{aligned} \tag{17}$$

However, $i_1 = \frac{t_\lambda}{N}$ and $i_2 = \frac{t_\mu}{N}$, so Eq. (17) becomes

$$\begin{aligned} & \frac{G(t_\lambda - t_\mu)}{N} \mod t = 0 \\ \overset{N=GN'}{\Rightarrow}\ & \frac{G(t_\lambda - t_\mu)}{GN'} \mod t = 0 \\ & \frac{(t_\lambda - t_\mu)}{N'} \mod t = 0 \\ N' \textbf{ integer} \Rightarrow\ & \frac{N'(t_\lambda - t_\mu)}{N'} \mod t = 0 \quad \text{or} \\ & (t_\lambda - t_\mu) \mod t = 0 \end{aligned} \tag{18}$$

Let us rewrite Eq. (14) and (18):

$$\begin{aligned} (t_\lambda - t_\mu) \mod N = 0 \\ (t_\lambda - t_\mu) \mod t = 0, \end{aligned}$$

from which it follows that

$$(t_\lambda - t_\mu) \mod LCM(t, N) = 0 \tag{19}$$

Equation 19 states that the two tasks differ by the LCM of $t$ and $N$. Thus all tasks that differ by this quantity can be distributed in the same manner. ■

Proposition 2 forms the basis of our task scheduling. Algorithm 2, that is also presented in subsection III.A, derives directly from the proof of this proposition and along with Algorithm 3 constitute our task scheduling approach.

*Proposition 2: The task allocation defined by the intermediate task allocation matrix, $M'_{inter}$, can be refined to map to the specific application's DAG, so that the communicating tasks can be (to the maximum extent) placed in nearby nodes and produce the final task allocation-scheduling matrix, $M'_{fin}$.*

*Proof:* If we view the elements of rows of $M'_{inter}$ as component IDs (and further, as tasks) rather than target nodes, then we can show Proposition 2. Indeed, to see that Proposition 2 also holds, one can easily see that each of the $\Delta$ square sub-matrices, in its final form, has $G$ elements in each diagonal. In this context a diagonal is a group of G elements that belong to the same component.

By reading the DAG, we can easily see the interconnection between the components. Then, we can interchange the diagonals of the $N'$ sub-matrices over each row of $M'_{inter}$ in a proper way, so that each sub-matrix has component IDs that in fact are settled to communicate by the application. Because $M'_{inter}$ has $t'$ rows and $N'$ columns of square sub-matrices of size $G \times G$, it follows that, every diagonal of each row sub-matrix can be transferred to at most $N' - 1$ different sub-matrices over the row, indication that each diagonal can change position at most $N'-1$ times. This completes the proof of existence for Proposition 2. ■

## III. TASK ALLOCATION AND SCHEDULING APPROACH

Our scheme is divided into three parts: task allocation, communication refinement and task scheduling, which are described in the following paragraphs.

### A. TASK ALLOCATION

The task allocation strategy is a straight forward implementation of the proof presented for Propositions 1 and 2. In Algorithm 1, the proof of Proposition 1 is organized in steps, so that the initial task allocation is produced. Then, the diagonal movements discussed to prove the existence of Proposition 2 result in Algorithm 2, that constitutes the *refinement* of our task allocation. The communication refinement of the $M'_{inter}$, the matrix that derives from the application of Algorithm 1, uses as an input:

- a Bitmap matrix that represents the actual communications between the DAG's components (rows and columns refer to component IDs and "1" is used every time a communication between the corresponding components exists)
- the matrix $M'$ which contains the discrete elements of each $G \times G$ submatrix in each row in ascending order and
- an $R^-$ array that contains the non-communicating components in each $G \times G$ sub-matrix (each row represents a component ID and its elements depict the component IDs with which it does not communicate, in descending order).

The function *check_swap* is called to make the necessary diagonals' interchanges, and gives as an output the refined allocation table $M'_{fin}$.

**Algorithm 1:** Task Distribution

**input :** An application graph organized in $n$ spouts/bolts of $t$ tasks
A cluster of $N$ nodes

**output:** $M'_{inter}$ depicting a task allocation policy, such that each node receives stream parts from one node at a time

**begin**
Declare $M'_{init}$, such that $\forall i \in \left[0, \ldots \left\lfloor \frac{\mathcal{T}}{N} \right\rfloor - 1\right]$ and $\forall j \in [0, \ldots, N-1] : M'_{init} = \frac{iN+j}{t}$
Set $G = \gcd(t, N)$, $t' = \frac{t}{G}$, $N' = \frac{N}{G}$
Find all the classes $k$, such that $i \pmod{t'} = k$
**if** *a row index* $i \in k$ **then**
  $i_{new} = \frac{i}{t'} + kG$
**end**
Define the $t' \times N'$ sub-matrices of size $G \times G$
Move all the sub-matrices with similar values to different rows using Eq. (10)
Move all the similar values within each sub-matrix to different rows using Eq. (11) // $M'_{inter}$ has been generated

**Algorithm 2:** Communication Refinement

**input :** The task allocation matrix $M'_{inter}$ derived from the application of Algorithm 1 on an application graph
*Bitmap*: A matrix representing the existing communications between components
$M'$: An array consisting the elements of each $G \times G$ submatrix in each one of its rows in ascending order
$R^-$: An array containing the non-communicating components in each $G \times G$ submatrix in descending order
**output:** A refined allocation-scheduling table $M'_{fin}$

```
1  Function main()
2  num_of_swaps = 0
3  for i = 0 to N' − 2 do
4  |  for j = i + 1 to N' − 1 do
5  |  |  check_swap(i,j)
6  |  end
7  end
8  check_swap(i,j)
9  k = 0 //Index used in R^−
10 for x = 0 to G − 1 do
11 |  for y = 0 to G − 1 do
12 |  |  source= M'[i, x]
13 |  |  target= M'[j, y]
14 |  |  if Bitmap[source, target] == 1 then
15 |  |  |  if num_of_swaps < ⌊G/2⌋ then
16 |  |  |  |  swap(R^−[source, k], target)
17 |  |  |  |  k = k + 1
18 |  |  |  |  num_of_swaps = num_of_swaps + 1
19 |  |  |  |  Store to M'_fin
20 |  |  |  |  else
21 |  |  |  |  num_of_swaps = 0
22 |  |  |  |  GoTo Line 3
23 |  |  |  end
24 |  |  end
25 |  end
26 end
```

**Algorithm 3:** Task Scheduling

**input :** Number of stages, $s$, number of processing steps, $P$, time required to process a stream part, $h$
**output:** $M'_{fin}$ depicting a task scheduling policy with equal processing load per node

```
1  begin
2  δ = 0
3  while processing not complete do
4  {
5  |  time = δh;
6  |  for l = 0 to s − 1
7  |  {
8  |  |  if δ − l ≥ 0 then
9  |  |  |  stage_l → step_{δ−l} (mod P)
10 |  |  |  else stage_l → ∅
11 |  |  }
12 |  |  Execute communications defined by the processing step
13 |  |  δ = δ + 1;
14 }
```

### B. TASK SCHEDULING

To perform the task scheduling, we need to arrange the inter-node communications. In this context, we can view $M'_{fin}$ as a communication schedule, where each row corresponds to a processing step, as defined in Definition 1. An all to all communication between all the cluster nodes requires $N$ rows for $M'_{fin}$. In cases where $N > t$, we need to add another $\frac{N-t}{G}$ rows of sub-matrices. Such matrices are always available, since, $N' > t'$ (the number of sub-matrix columns is $>$ than the number of sub-matrix rows). The sub-matrices chosen to be added include the missing communications. This addition results in a square $N \times N$ $M'_{fin}$ matrix and is necessary before applying the scheduling approach of Algorithm 3.

To read the communications, we view the label on top of each column of $M'_{fin}$ as the sending node and the corresponding row elements as the receiving **nodes**. The intra-node communications that result from this scheme are very useful, as they perform task communications internally within a node, thus inter-node communication costs are reduced. Of course we *are also interested that the communication between neighboring nodes is mapped in the columns as well, to the larger possible extent*. This is guaranteed by the way this refinement is performed using Algorithm 2.

Each application runs in processing stages (defined in Definition 2). Our task scheduling approach organizes the communication between tasks in a pipeline-based fashion, such that: (1) At each processing step, each node receives stream parts which are delivered to the proper tasks for processing, from one node only, (2) The processing load is balanced between the nodes available.

The time is divided into time slots of duration $h$, where $h$ is the constant time required to process each stream part. We will use the notation ''a step occupies a stage'', to show that, from the set of all the stream parts that need to be transferred and processed by the tasks at this stage, our system transfers only the stream parts between the node pairs defined in the step. The rationale between our scheme is the following: At every time slot, different steps occupy different processing stages, thus, each node receives mostly streams processed at different stages of the application (thus, different tasks are occupied). This reduces the buffer space that would be required if a task had to process many stream parts arriving simultaneously to its node. Algorithm 3 gives

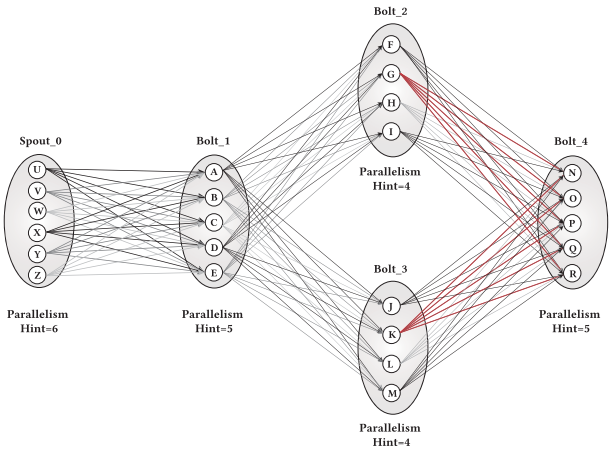


**FIGURE 2.** **Intercommunication of tasks within a random topology (DAG) with heavy communications.**

the pseudo-code of our task scheduling approach. The application of Algorithms 1,2 and 3 are depicted by the motivating examples of Section IV.

### C. OVERALL COMPLEXITY

The transformations required by our task allocation and scheduling scheme are of three types: 1) the transformations required to move the classes in the same part of $M'_{trn}$, 2) the transformations of Eq. (10-11), that will generate the processing steps, and (3) the transformations used for refinement. The first type is defined by Eq. (6) and at most $t'$ rows change position. The transformations described by Eq.(10) moves at most $N'-1$ sub-matrices and the transformations of Eq.(11) introduce another $G$ simultaneous moves. Finally, the refinement phase includes at most $N'-1$ moves of diagonal elements to a new sub-matrix in every row of sub-matrices. Since we have $t'$ such rows, the refinement requires at most $(N'-1)t'$ moves. Totally, our scheme requires $t'+N'-1+(N'-1)t'$ moves, so its cost is $O(t', N')$, a linear dependence on $t'$ and $N'$.

## IV. MOTIVATING EXAMPLES

We consider two different topologies; a random and a linear as motivating examples. Figs. 2 and 6 show the interconnection between the components that are used.

### A. RANDOM TOPOLOGY

In Fig.2 the topology consists of 4 bolts and 1 spout and the maximum number of threads $t$ per component, is 6. Each thread executes one task, so we can refer to tasks and threads interchangeably from this moment on. A cluster of $N=9$ nodes will be used in our case. We add additional dummy threads to components that have less than $t$ tasks to define an initial *matrix*, $M_{init}$, as a table that stores the tasks assigned to each node by the default round-robin Storm scheduler. This table can have two forms: in the first form, the tasks are indicated as letters and in the second they have been replaced by numbers.

$$M_{init} = \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \mathrm{U} & \mathrm{V} & \mathrm{W} & \mathrm{X} & \mathrm{Y} & \mathrm{Z} & \mathrm{A} & \mathrm{B} & \mathrm{C} \\ \mathrm{D} & \mathrm{E} & \textcircled{\mathrm{A}'} & \mathrm{F} & \mathrm{G} & \mathrm{H} & \mathrm{I} & \textcircled{\mathrm{F}'} & \textcircled{\mathrm{G}'} \\ \mathrm{J} & \mathrm{K} & \mathrm{L} & \mathrm{M} & \textcircled{\mathrm{J}'} & \textcircled{\mathrm{K}'} & \mathrm{N} & \mathrm{O} & \mathrm{P} \\ \mathrm{Q} & \mathrm{R} & \textcircled{\mathrm{N}'} & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{bmatrix}$$

$$\equiv \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 9 & 10 & \textcircled{11} & 12 & 13 & 14 & 15 & \textcircled{16} & \textcircled{17} \\ 18 & 19 & 20 & 21 & \textcircled{22} & \textcircled{23} & 24 & 25 & 26 \\ 27 & 28 & \textcircled{29} & \textcircled{30} & \textcircled{31} & \textcircled{32} & \textcircled{33} & \textcircled{34} & \textcircled{35} \\ \textcircled{36} & \textcircled{37} & \textcircled{38} & \textcircled{39} & \textcircled{40} & \textcircled{41} & \textcircled{42} & \textcircled{43} & \textcircled{44} \\ \textcircled{45} & \textcircled{46} & \textcircled{47} & \textcircled{48} & \textcircled{49} & \textcircled{50} & \textcircled{51} & \textcircled{52} & \textcircled{53} \end{bmatrix}$$

The tasks indicated by $\Omega$ are added by our model as "dummies", they are used to avoid empty values in $M_{init}$, and assure that all the desired nodes take part in the allocation procedure. In the numbered representation, the dummy tasks are circled. A dummy task plays no role in the actual processing. The resulting table is an $t \times N$ matrix. By dividing $M_{init}$ by $t$, we obtain an intermediate matrix, $M'_{init}$, which represents tasks with their corresponding component IDs (assuming that the spout has ID $=0$, and bolts 1-4 have IDs 1-4, respectively). The IDs 5, 6, 7 and 8 correspond to components containing dummies:

$$M'_{init} = \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ \hline 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

We also have $G=\gcd(6,9)=3$, thus $t'=2$ and $N'=3$. Because $t'=2$, there are two classes, $k=0$ and $k=1$. According to Definition 3, row indices 0, 2 and 4 belong to class $k=0$ and row indices 1, 3 and 5 belong to class $k=1$. According to Eq. (6), row $i=0$ is the first row of class 0 and as $\left\lfloor \frac{0}{2} \right\rfloor + 0 \times 3 = 0$, it will remain in the same position. The next row in class $k=0$ is the row with $i=2$. It will move to row $\left\lfloor \frac{2}{2} \right\rfloor + 0 \times 3 = 1$. The last row in class $k=0$ is the row with $i=4$. It will move to row $\left\lfloor \frac{4}{2} \right\rfloor + 0 \times 3 = 2$. For class $k=1$, we have row $i=1$, that will move to row $\left\lfloor \frac{1}{2} \right\rfloor + 1 \times 3 = 3$, row $i=3$, that will move to row $\left\lfloor \frac{3}{2} \right\rfloor + 1 \times 3 = 4$ and row $i=5$, that will stay in row $\left\lfloor \frac{5}{2} \right\rfloor + 1 \times 3 = 5$. To summarize, we have the following moves:

- Row 0 will stay in row 0
- Row 1 will move to row 3

- Row 2 will move to row 1
- Row 3 will move to row 4
- Row 4 will move to row 2
- Row 5 will stay in row 5

resulting into a transformed matrix $M'_{trn}$:

$$M'_{trn} = \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

The horizontal line separates the elements of the two classes. Class $k = 0$ elements are at the top rows and class $k = 1$ elements are at the bottom rows. Also, note that, in every column of $M'_{trn}$, each classe's elements produce the same modulo when divided to $N' = 3$. Next, we can note that $M'_{trn}$ can be divided into a set of $t' \times N'$ sub-matrices of size $G \times G$:

$$M'_{trn} = \left[\begin{array}{c|c|c} \Delta_{0,0} & \Delta_{0,1} & \Delta_{0,2} \\ \hline \Delta_{1,0} & \Delta_{1,1} & \Delta_{1,2} \end{array}\right] = \left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 5 \\ 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 \end{array}\right]$$

Now, by applying the row transformations of Eq. (10), $M'_{trn}$ becomes

$$M'_{trn} = \left[\begin{array}{ccc|ccc|ccc} 0 & 0 & 0 & 2 & 2 & 2 & 1 & 1 & 1 \\ 3 & 3 & 3 & 5 & 5 & 5 & 4 & 4 & 4 \\ 6 & 6 & 6 & 8 & 8 & 8 & 7 & 7 & 7 \\ \hline 1 & 1 & 1 & 0 & 0 & 0 & 2 & 2 & 2 \\ 4 & 4 & 4 & 3 & 3 & 3 & 5 & 5 & 5 \\ 7 & 7 & 7 & 6 & 6 & 6 & 8 & 8 & 8 \end{array}\right]$$

and by applying the row transformations of Eq. (11), we get the $M'_{inter}$:

$$M'_{inter} = \begin{array}{c} \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 6 & 3 & 2 & 8 & 5 & 1 & 7 & 4 \\ 3 & 0 & 6 & 5 & 2 & 8 & 4 & 1 & 7 \\ 6 & 3 & 0 & 8 & 5 & 2 & 7 & 4 & 1 \\ 1 & 7 & 4 & 0 & 6 & 3 & 2 & 8 & 5 \\ 4 & 1 & 7 & 3 & 0 & 6 & 5 & 2 & 8 \\ 7 & 4 & 1 & 6 & 3 & 0 & 8 & 5 & 2 \end{array}\right] \end{array}$$

To start the communication refinement, we define the matrices *Bitmap*, $M'$, and $R^-$, that are necessary for the implementation of Algorithm 2.

$$Bitmap = \begin{array}{c} \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right] \end{array}$$

$$M' = \left[\begin{array}{ccc} 0 & 3 & 6 \\ 2 & 5 & 8 \\ 1 & 4 & 7 \end{array}\right],$$

$$R^- = \left[\begin{array}{cc} 6 & 3 \\ 7 & 4 \\ 8 & 5 \\ 6 & 0 \\ 7 & 1 \\ 8 & 2 \\ 3 & 0 \\ 4 & 1 \\ 5 & 2 \end{array}\right]$$

Based on Algorithm 2, the first swap occurs for $i = 0$, $j = 2$, $x = 0$ and $y = 0$ as:

- $source = M'[0, 0] = 0$
- $target = M'[2, 0] = 1$
- $Bitmap[0, 1] = 1$ as there is a communication from component ID = 0 to component ID = 1

and this leads to a swap between elements 6 ($R^-[0, 0]$) and 1 (*target*) in matrix $M'_{fin}$. Moreover, for $i = 1, j = 2, x = 0$ and $y = 1$:

- $source = 2$
- $target = 4$
- $Bitmap[2, 4] = 1$ as there is a communication from component ID = 2 to component ID = 4

and this leads to a swap between elements 8 and 4 in matrix $M'fin$. Finally, $M'_{fin}$ becomes:

$$M'_{fin} = \begin{array}{c} \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 1 & 3 & 2 & 4 & 5 & 6 & 7 & 8 \\ 3 & 0 & 1 & 5 & 2 & 4 & 8 & 6 & 7 \\ 1 & 3 & 0 & 4 & 5 & 2 & 7 & 8 & 6 \\ 6 & 7 & 8 & 0 & 1 & 3 & 2 & 4 & 5 \\ 8 & 6 & 7 & 3 & 0 & 1 & 5 & 2 & 4 \\ 7 & 8 & 6 & 1 & 3 & 0 & 4 & 5 & 2 \end{array}\right] \end{array}$$

$$\equiv \begin{array}{c} \begin{array}{ccccccccc} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \end{array} \\ \left[\begin{array}{ccccccccc} \mathtt{U} & \mathtt{A} & \mathtt{J} & \mathtt{F} & \mathtt{N} & \Omega & \Omega & \Omega & \Omega \\ \mathtt{K} & \mathtt{V} & \mathtt{B} & \Omega & \mathtt{G} & \mathtt{O} & \Omega & \Omega & \Omega \\ \mathtt{C} & \mathtt{L} & \mathtt{W} & \mathtt{P} & \Omega & \mathtt{H} & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \mathtt{X} & \mathtt{D} & \mathtt{M} & \mathtt{I} & \mathtt{Q} & \Omega \\ \Omega & \Omega & \Omega & \Omega & \mathtt{Y} & \mathtt{E} & \Omega & \Omega & \mathtt{R} \\ \Omega & \Omega & \Omega & \Omega & \Omega & \mathtt{Z} & \Omega & \Omega & \Omega \end{array}\right] \end{array}$$

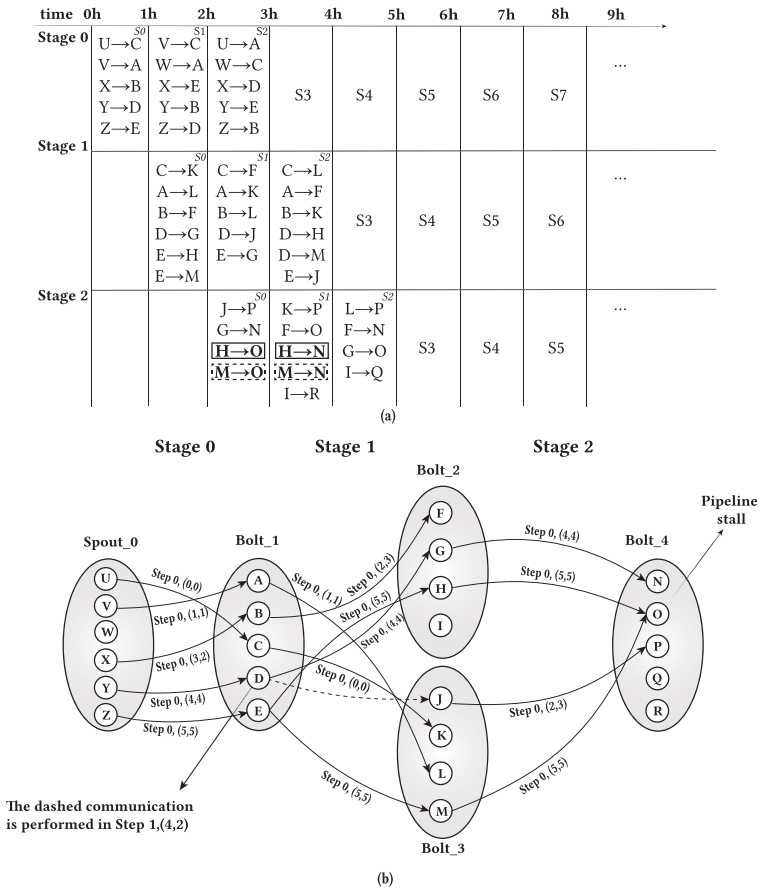


**FIGURE 3.** **(a) Pipeline-based scheduling for the random topology example (b)Task communications after implementing processing step 0, when the pipeline is already full.**

Now we can also view $M'_{fin}$ as a communication schedule. An all to all communication between all the cluster nodes requires $N$ rows for $M'_{fin}$. In our example, we add another $\frac{9-6}{3} = 1$ row of sub-matrices to $M'_{fin}$, to make it a square $9 \times 9$ matrix. The sub-matrices chosen to be added include the missing communications and our scheduling matrix becomes as follows:

$$M'_{fin} = \begin{array}{c|ccccccccc} & N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 & N_8 \\ \hline step\ 0 & 0 & 1 & 3 & 2 & 4 & 5 & 6 & 7 & 8 \\ step\ 1 & 3 & 0 & 1 & 5 & 2 & 4 & 8 & 6 & 7 \\ step\ 2 & 1 & 3 & 0 & 4 & 5 & 2 & 7 & 8 & 6 \\ step\ 3 & 6 & 7 & 8 & 0 & 1 & 3 & 2 & 4 & 5 \\ step\ 4 & 8 & 6 & 7 & 3 & 0 & 1 & 5 & 2 & 4 \\ step\ 5 & 7 & 8 & 6 & 1 & 3 & 0 & 4 & 5 & 2 \\ step\ 6 & 2 & 4 & 5 & 6 & 7 & 8 & 0 & 1 & 3 \\ step\ 7 & 5 & 2 & 4 & 8 & 6 & 7 & 3 & 0 & 1 \\ step\ 8 & 4 & 5 & 2 & 7 & 8 & 6 & 1 & 3 & 0 \end{array}$$

The pipeline that derives from the application of Algorithm 3 in the example of Fig. 2 is shown in Fig. 3(a). Fig.3(b) shows the task communications performed after implementing Step 0, when the pipeline is already full. In our context, the pipeline is *full once all the steps have gone through all the stages once* (unlike the usual interpretation, which states that all the stages are full). When this is the case, we have a communication pattern between the application's tasks, such that the number of multiple part streams loaded to a task is minimized, thus minimizing the need for buffer space. From Fig. 3(b), one can see that the communication from task J to P can only be implemented after the communication from D to J. The first implementation of step 1, results in a communication between the JVMs of nodes 4 and 2 transferring a stream part from D to J (task D has already received and can deliver a stream part). Then the implementation of step 0, transfers a stream part between the JVMs of nodes 2 and 3 i.e. from task J to task P. This is depicted by the dashed arrow connecting tasks D and J in Fig. 3(b) as it refers to Step 1.

Moreover, we see a case where possible buffering is required in Step 0; two tasks from node 5 (M and H) communicate with task O in the same node. In such cases (which sometimes are inevitable when multiple components forward tuples to a single one), we could monitor an increase in queue waiting times and tuple latencies. The burden can become even worse when the application workload is increased.

A dynamic scheme could adopt data parallelism and scale out the number of parallel instances for the operator that is overloaded and becomes a bottleneck and/or increase the

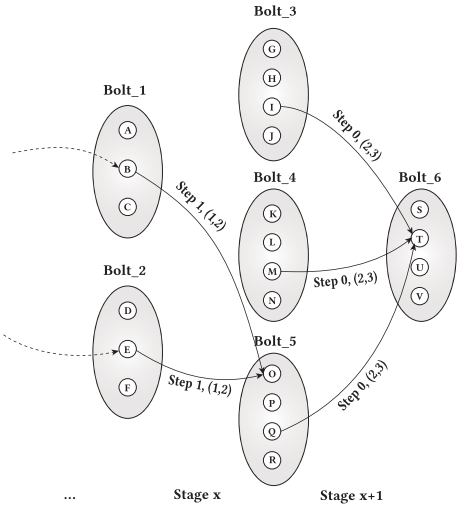


**FIGURE 4.** A complex scenario.

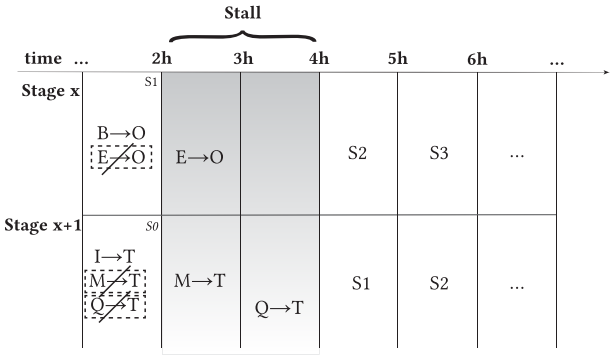


**FIGURE 5.** Pipeline stall.

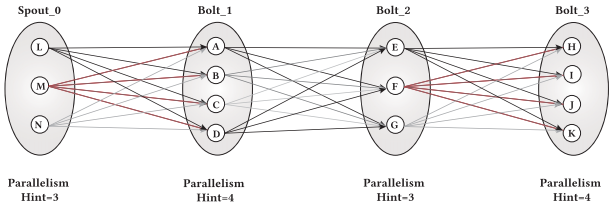


**FIGURE 6.** Intercommunication of tasks within a linear topology (DAG) with heavy communications.

number of VMs that run in the cluster [14]. Possible task migrations would be needed to reduce resource utilization imbalances between nodes. Elastic data parallelism during run-time, makes a system adaptive to changes in the execution environment but in systems like Apache Storm, the required reconfiguration and restart of the application also results in significant downtime [5], [9].

In a static scheme, when several tuples arrive simultaneously from different tasks to the same task, more than one tuples could fail to be processed or a tuple could be selected to be processed and let the remaining be processed in the next processing step, where the same communication occurs. In the first case, the tuple could be replayed later or could be missed based on the fault-tolerance guarantees that would be needed in the specific user's topology (at-least-once guarantee in contrast to at-most-once guarantee). This would either increase tuple latency or our system would provide the least desirable outcome in matters of reliability, as messages would be lost [3]. In case of keeping the tuples to be processed in the next appropriate step, the required buffer space would increase.

Our approach provides an efficient solution for the aforementioned scenario and implements a pipeline stall. The duration of this stall equals to the time required to process the remaining tuples by the corresponding tasks. In our example the stall's duration equals to $h$ as we just need to let task O process the tuple received from e.g. task M (in case the tuple received by task H was initially selected to be processed).

This solution can prove its value in more complex scenarios like the one presented in Fig. 4. During Step 0, three different tasks from Node 2 (tasks I, M, and Q) send stream parts to a single task of Node 3 (task T). Simultaneously, in Step 1, two different tasks from Node 1 send stream parts to be processed to a task (O) in Node 2. In this case, the duration of the pipeline stall equals the maximum time needed by a task to process all its stream parts as can be seen in Fig. 5. In this case this duration equals $t = 2h$ as O will take $t = h$ to process the remaining part from the second task, while T will need $t = 2h$ to process the remaining parts from the other two tasks that wanted to communicate with it.

### B. LINEAR TOPOLOGY

In Fig.6 a linear topology with 3 bolts and 1 spout is represented. The maximum number of threads $t$ per component, is 4. A cluster of $N = 8$ nodes will be used in this example. The resulting initial *matrix*, $M_{init}$, is the following:

$$M_{init} = \begin{bmatrix} N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \mathrm{L} & \mathrm{M} & \mathrm{N} & \textcircled{\mathrm{L}}' & \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} \\ \mathrm{E} & \mathrm{F} & \mathrm{G} & \textcircled{\mathrm{E}}' & \mathrm{H} & \mathrm{I} & \mathrm{J} & \mathrm{K} \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \\ \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{bmatrix}$$

By dividing $M_{init}$ by $t$, we obtain an intermediate matrix, $M'_{init}$. The IDs 4, 5, 6 and 7 correspond to components containing dummies:

$$M'_{init} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 \end{bmatrix}$$

In this example we have $G = \gcd(4, 8) = 4$, thus $t' = 1$ and $N' = 2$. Because $t' = 1$, there is only one class and consequently, there is no need to transpose lines. The transposed matrix $M'_{trn}$ is equivalent to $M'_{init}$ and can be

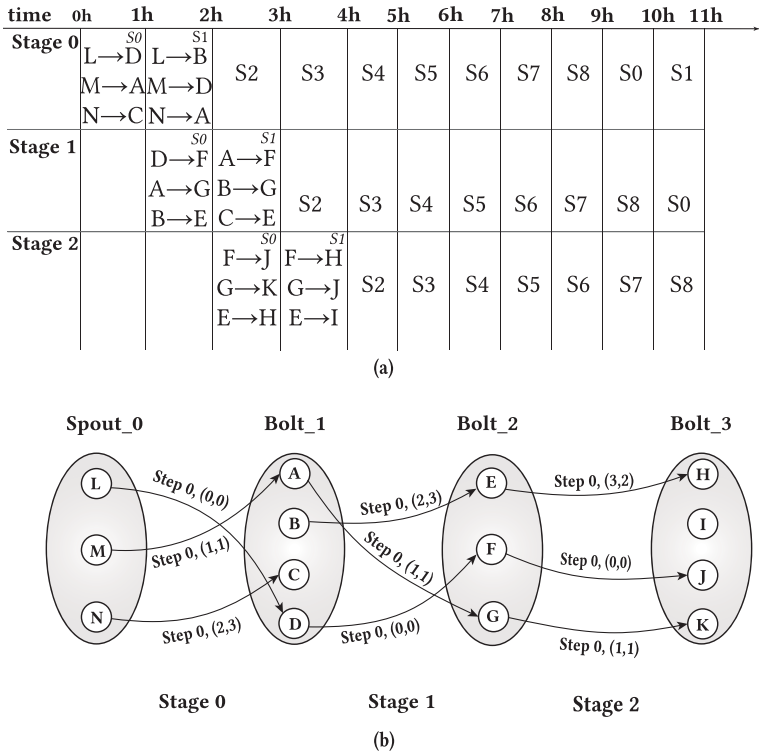


**FIGURE 7.** (a) Pipeline-based scheduling for the linear topology example (b)Task communications after implementing processing step 0, when the pipeline is already full.

divided into a set of $t' \times N'$ sub-matrices of size $G \times G$:

$$M'_{trn} = \left[\, \Delta_{0,0} \,\middle|\, \Delta_{0,1} \,\right] = \left[\begin{array}{cccc|cccc} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 \\ 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 \end{array}\right]$$

Eq. (10) does not result in row transformations in $M'_{trn}$ (there is only one row), but Eq. (11), results in the following allocation matrix $M'_{inter}$:

$$M'_{inter} = \begin{array}{c|cccccccc} & N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline & 0 & 6 & 4 & 2 & 1 & 7 & 5 & 3 \\ & 2 & 0 & 6 & 4 & 3 & 1 & 7 & 5 \\ & 4 & 2 & 0 & 6 & 5 & 3 & 1 & 7 \\ & 6 & 4 & 2 & 0 & 7 & 5 & 3 & 1 \end{array}$$

The communication refinement algorithm leads to two swaps; 6 with 1, and 4 with 3 that result in the following $M'_{fin}$:

$$M'_{fin} = \begin{array}{c|cccccccc} & N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline & 0 & 1 & 3 & 2 & 6 & 7 & 5 & 4 \\ & 2 & 0 & 1 & 3 & 4 & 6 & 7 & 5 \\ & 3 & 2 & 0 & 1 & 5 & 4 & 6 & 7 \\ & 1 & 3 & 2 & 0 & 7 & 5 & 4 & 6 \end{array}$$

$$\equiv \begin{array}{c|cccccccc} & N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline & L & A & H & E & \Omega & \Omega & \Omega & \Omega \\ & F & M & B & I & \Omega & \Omega & \Omega & \Omega \\ & J & G & N & C & \Omega & \Omega & \Omega & \Omega \\ & D & K & \Omega & \Omega & \Omega & \Omega & \Omega & \Omega \end{array}$$

Now we can also view $M'_{fin}$ as a communication schedule that after adding 1 row of sub-matrices to include the missing communications, becomes as follows:

$$M'_{fin} = \begin{array}{c|cccccccc} & N_0 & N_1 & N_2 & N_3 & N_4 & N_5 & N_6 & N_7 \\ \hline step\ 0 & 0 & 1 & 3 & 2 & 6 & 7 & 5 & 4 \\ step\ 1 & 2 & 0 & 1 & 3 & 4 & 6 & 7 & 5 \\ step\ 2 & 3 & 2 & 0 & 1 & 5 & 4 & 6 & 7 \\ step\ 3 & 1 & 3 & 2 & 0 & 7 & 5 & 4 & 6 \\ step\ 4 & 6 & 7 & 5 & 4 & 0 & 1 & 3 & 2 \\ step\ 5 & 4 & 6 & 7 & 5 & 2 & 0 & 1 & 3 \\ step\ 6 & 5 & 4 & 6 & 7 & 3 & 2 & 0 & 1 \\ step\ 7 & 7 & 5 & 4 & 6 & 1 & 3 & 2 & 0 \end{array}$$

The pipeline that derives from the application of Algorithm 3 in the example of Fig. 6 is shown in Fig. 7(a). Fig.7(b) shows the task communications performed after implementing Step 0, when the pipeline is already full.

## V. EXPERIMENTAL RESULTS

In this section, we discuss how we evaluate the performance of our proposed scheduling approach. We present two sets of

experiments, as will be described in the following paragraph: In the first set, we examine the *average latency*, the *percentage of buffer memory used*, the *load balancing per node*, and finally the *throughput* to check our system's performance against the default Apache Storm's scheduler using a random and a linear topology. In the second set, we compared our system with one more scheduler, Peng's *et al.* [17] scheduler, named R-Storm, to examine their throughput using a diamond and a linear topology. R-Storm was chosen as it has been included as an alternative scheduler for Apache Storm, as of v1.0.1., it tries to reduce the inter communication latency between adjacent tasks just like our approach and takes into consideration memory usage. It is a resource-aware strategy that tries to maximize resources utilization (also discussed in subsection VI.B "HEURISTIC APPROACHES".)

### A. EXPERIMENTAL SETUP

Our scheduling strategy is evaluated using a simulation environment, which provides researchers with a wide range of choices to develop, debug, and evaluate their experimental system. In our experimental setup, the Storm cluster consists of nodes that run Ubuntu 16.04.3 LTS with an Intel Core i7-8559U Processor system and clock speed at 2.7GHz, 1 Gb RAM per node. Further, there is all-to-all communication between the nodes, which are interconnected at a speed of 100 Mbps. Also, we assume that the data transfer rates between the cluster nodes are equal, but their proximity differs (nodes with smaller index difference are consider to be located at lower distances between them). The tuples generated are assumed to have equal size, 8Kb.

For our experiments we ran two topologies: (a) A random topology with four bolts and one spout, where the maximum number of threads per component, is 6 (see Fig.2). (b) A linear topology with three bolts and one spout. The maximum number of threads $t$ per component, is 4 (see Fig.6). In the case of random topology, we used a cluster with $N = 9$ worker nodes, each with 4 slots and in the case of linear topology we used a set of $N = 8$ worker nodes, each with 4 slots. One extra node, designated as the master node to host the Nimbus and Zookeeper services was also used in both cases. For our comparisons, we chose the default Storm scheduler, which is the most widely used comparison candidate in the literature.

**TABLE 1. Experimental environment.**

| | | |
|---|---|---|
| **Hardware** | **CPU**<br>**Memory**<br>**Network Speed** | Intel Core i7-8559U 2.7GHz<br>1Gb<br>100 Mbps |
| **Software** | **Operating System** | Ubuntu 16.04.3 LTS |

### B. AVERAGE TOTAL LATENCY

This set of experiments focus on the comparison of the overall runtime behavior of our scheduling strategy against the default Storm scheduler. The average latency refers to the time needed by tuples to traverse the entire topology. To fairly

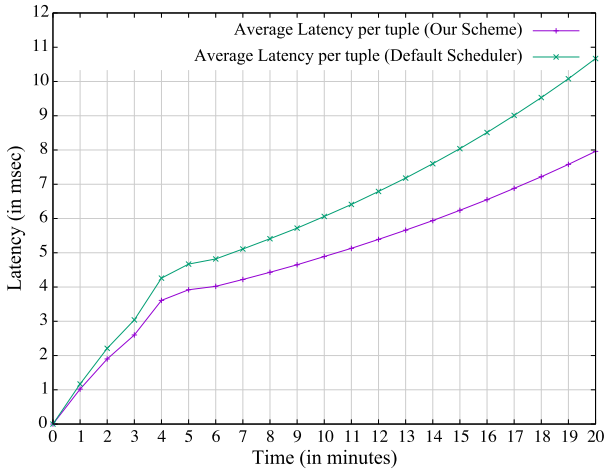


**FIGURE 8. Average latency comparisons between our strategy and the default scheduler: Random topology.**

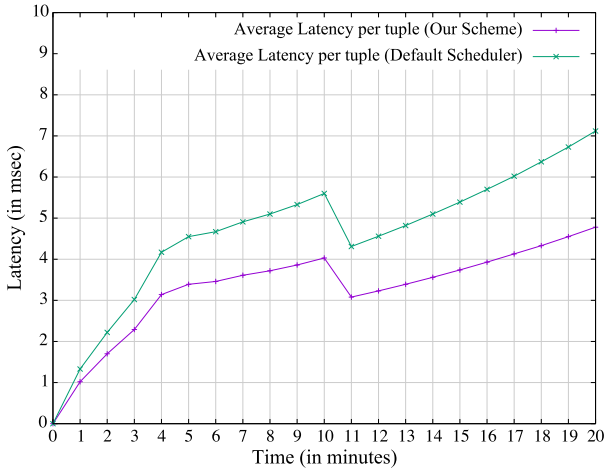


**FIGURE 9. Average latency comparisons between our strategy and the default scheduler: Linear topology.**

estimate the overall latency we worked as follows: (a) Each tuple has to "travel" some distance (nodes with close ID numbers are considered to be placed more closely) from the node that has processed it until the node that will continue the processing. (b) A tuple can either be buffered or directly be processed. However, in cases where there is no buffer space available, the tuple is not omitted. Instead, it is resent after a short period. Although this is not always the case (some systems prefer to omit such tuples), this type of policy can be helpful in examining the overall latency.

In the experiments presented, we assigned a buffer space of 32 Mb per task, which is enough to accommodate about 4K tuples. The other settings (number of threads and nodes) are as described in the previous subsection. Fig. 8 shows the average tuple latency for the random topology, while Fig. 9 shows the average tuple latency for the linear topology. In both cases, our strategy presents lower latencies, but the average gain is higher when the linear topology is

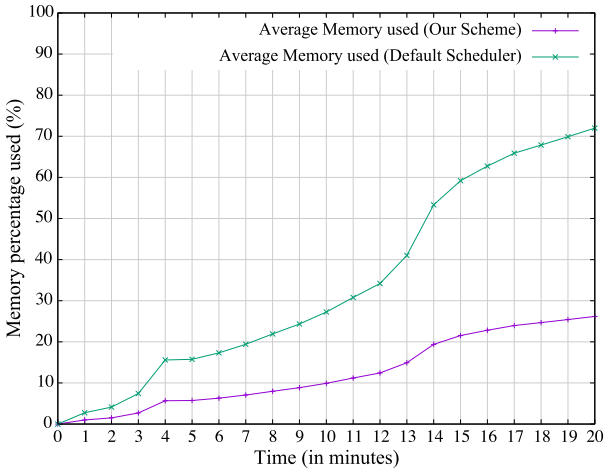


**FIGURE 10.** Average memory use comparisons between our strategy and the default scheduler: Random topology.

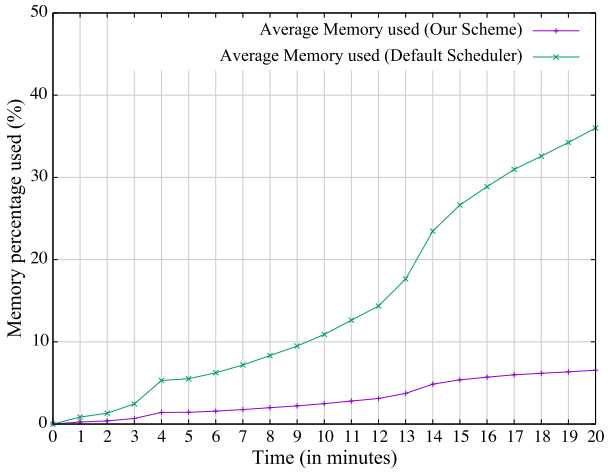


**FIGURE 11.** Average memory use comparisons between our strategy and the default scheduler: Linear topology.

used (25% and 40% respectively). There is a combination of reasons behind this result: First, the linear topology has even smaller inter-node traffic when executed in a pipeline fashion, compared to the random topology. However, reducing the inter-node communication cost is not always sufficient to guarantee lower latencies. It is also important to consider that, our strategy avoids having intensively loaded paths between nodes, and this is especially true in the linear case. In the random topology, there are cases where a task may receive large loads (see the example of Fig. 3). Therefore, our strategy pays-off specifically for linear applications, in that it reduces the overall latency to almost 40% compared to the default scheduler.

### C. PERCENTAGE OF BUFFER MEMORY USED

Although there are 3 main resources involved in the overall evaluation of a stream scheduling strategy (network links, CPU and memory), in this paragraph, we show the fact that our strategy requires much less memory space compared to the default scheduler. In the plots of Figs. 10 and 11, we show the average memory usage for the random and linear topology, respectively. Specifically, when the random topology is executed, the results have indicated that the default scheduler uses, in the worst case, about 70% of the available memory space available. This is due to the fact that some nodes become overloaded for some periods of time and require more buffering. Our strategy uses at most 26% of the memory space, in cases where pipeline stalls may occur (thus buffering is required). For the linear case, the memory space required is reduced for both strategies, however, the improvement offered by our strategy is higher, primarily because in this case our strategy consumes only about 7% of the memory resources in this case (some tuples needed to be bufferred due to network flaws, so re-transmissions were necessary). In fact, in a linear topology, our strategy can have each task receive a tuple at a time thus buffering is not generally required.

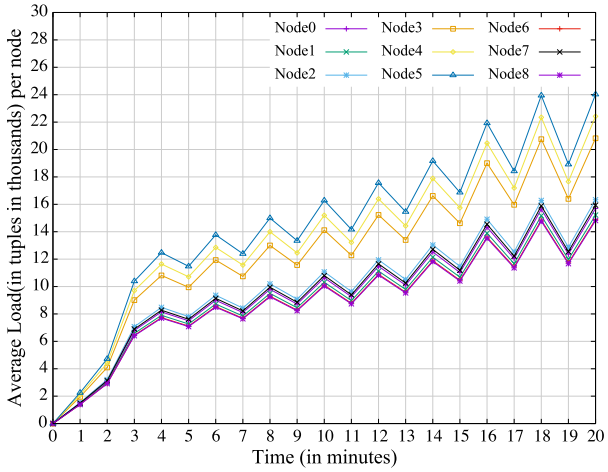


**FIGURE 12.** Load balancing in a random topology application.

### D. LOAD BALANCING

In this part of our study we examine if our strategy offers indeed satisfactory balancing between the nodes. We reduced the tuple size to 1KB, to have faster processing time per tuple and we measured the tuples processed at each node. The results have shown that, for both topologies, the 9 nodes receive almost balanced processing load (see Fig. 12 and 13). Some divergences (an average of 7%) appear in the random topology scenario, where nodes 3 to 5 appear to process more load compared to the others. This can be explained in two ways: (1) these nodes have processed more pipeline stalls (cases where tasks inside these nodes receive more tuples, which are buffered and pipeline stalls are used), (2) these nodes suffer less data losses during transmissions. When the topology is linear, the load delivered to the nodes is more balanced, as seen in Fig. 13: there is a "one-to-one" component communication and "one-to-one" inter-node communication and the small imbalances that appear can be explained by the fact that not all the task communications defined by our

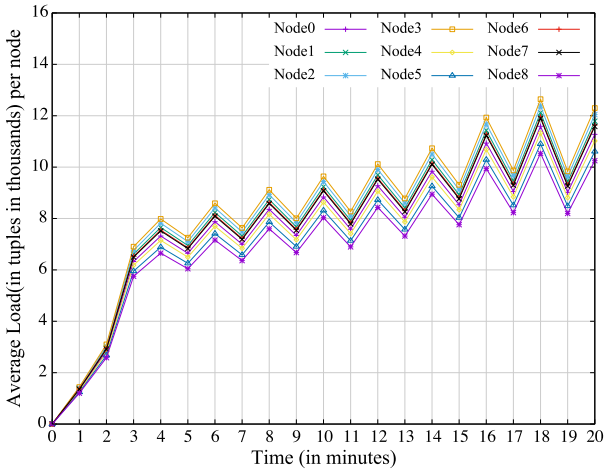


**FIGURE 13.** Load balancing in a linear topology application.

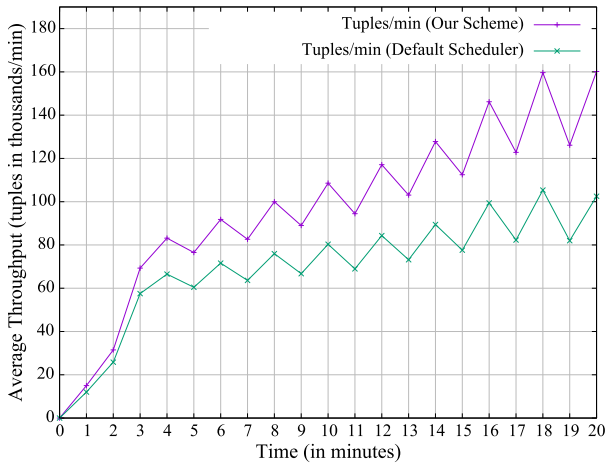


**FIGURE 14.** Throughput comparisons between our strategy and the default scheduler: Random topology.

schedule are actually defined in the application. The default Storm scheduler does not provide any mechanism for handling the communication between tasks in a stepwise manner, so generally it achieves no balancing.

### E. THROUGHPUT

This section provides the results of two different sets of experiments: (i) First, we compared the average throughput (tuples/min) between our strategy and the default round-robin scheduling strategy. (ii) We then compared our system with the default scheduler and R-Storm under different scenarios, to verify the fact that our strategy can achieve better throughput performance.

(i) The average throughput is defined as the rate of tuples being processed by the topology's bolts. For the random topology, we varied the number of threads from 3 to 6 and we averaged the throughput values. We kept processing tuples over a period of 20 minutes. Throughput is mainly affected by the inter-node communication required, and the possible delay, when a tuple is buffered to be processed at a later time.

The results shown in Fig. 14 indicate that our strategy outperforms the default round robin strategy. Our strategy offers an average of 25% improvement in throughput. There are two main reasons for this improvement: (i) As the total time increases, the round-robin strategy suffers large numbers of tuples, which are not processed in-time, due to node over-utilization (for example, when multiple tuples are submitted from different tasks to a certain task). These tuples are either buffered, or omitted (in such a case they are re-submitted for processing), and (ii) Our strategy places the relevant tasks to the same or nearby nodes (refinement process). This type of placement decreases the total tuple processing time, as the inter-node communication cost is reduced. For larger number of threads and a maximum runtime of 20 minutes, the overall improvement approached 35%.

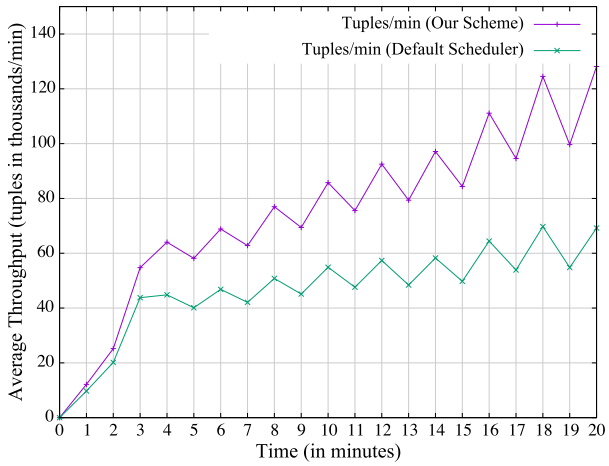


**FIGURE 15.** Throughput comparisons between our strategy and the default scheduler: Linear topology.

For the linear topology, we varied the number of threads from 2 to 4 and we averaged the results. Again, our strategy outperforms the default scheduler by an average of 40%. We noticed larger throughput increases compared to the random topology. This is explained by the fact that our approach buffers fewer tuples in the linear topology case, compared to the random one. Figure 15 shows the experimental results for the linear topology case.

(ii) In the second set of experiments (regarding the throughput of our strategy), we compared our work with the default scheduler and R-Storm. We used 8 worker nodes and 1 node designated as the master node, running Nimbus and Zookeeper and ran two different topologies; a diamond (Fig.16 (a)) and a linear (Fig.16 (b)). The diamond topology, consists of one spout, two intermediate bolts one sink bolt. Each component consists of 10 tasks. The linear topology has one spout and four bolts and the number of tasks is 16, 16, 8, 4 and 1 respectively.

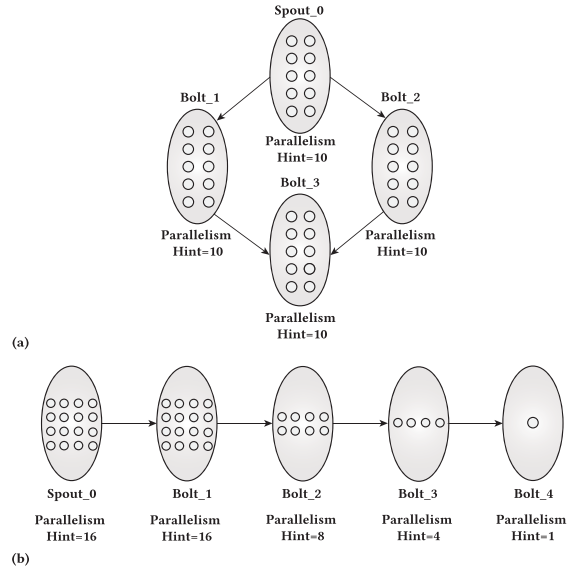


**FIGURE 16.** Experimental topologies.

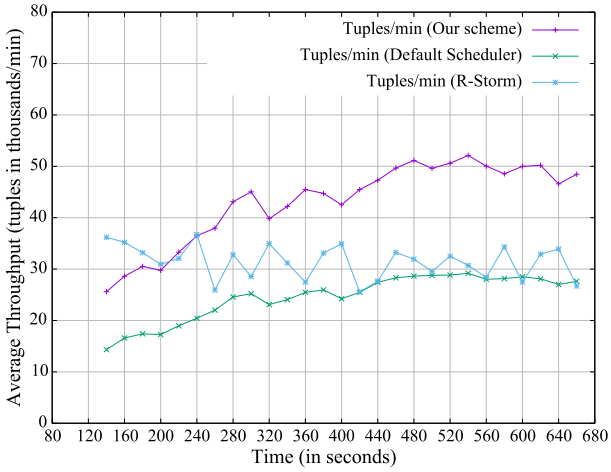


**FIGURE 17.** Throughput comparisons between our strategy, R-Storm and the default scheduler: Diamond topology.

The results obtained justify our claim, that when there is a special care on the buffering of incoming tuples, that is, buffering is reduced and thus the highest percentage of tuples are processed as they arrive to the proper target node, then the average throughput increases. The R-Storm does not have any special mechanism for reducing the buffer space required. Instead, it expects the user to provide the node capacities and the task requirements, in order to perform allocation of threads and manage the available resources. From Fig. 17 it can be seen that our strategy offers an improvement of $\approx 35\%$ compared to R-Storm as the time increases for the diamond topology.

For the linear topology the results indicate that the improvement approaches almost 45% compared to R-Storm. This is because that our strategy generally buffers less tuples when running a purely linear topology compared to

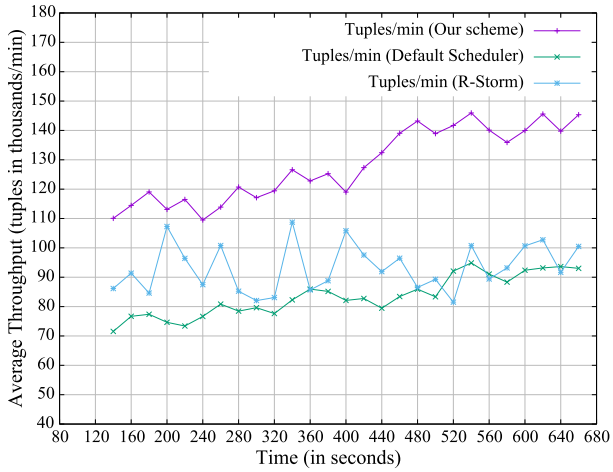


**FIGURE 18.** Throughput comparisons between our strategy, R-Storm and the default scheduler: Linear topology.

diamond or random topologies. In such topologies, tasks may receive multiple tuples from other tasks, which can't be processed simultaneously, thus they are bufferred. Then, all bufferred tuples are processed in an extra step (pipeline stall), as explained in the text.

One final observation is that, there is a period of time (in the first seconds of execution) that R-Storm outperforms our strategy. This is explained by the fact that our strategy needs to execute once all the communicating steps and have the pipelines full, in order to start performing more efficiently. When this occurs, our scheme clearly ourperforms R-Storm.

## VI. RELATED WORK

In this section, we review efforts related to scheduling strategies in DSPSs. We initially focus on available solutions incorporated in the most prominent DSPSs [3] and then present different heuristic approaches.

### A. DSPSs

We distinguish two different execution models in stream processing; the operator-based approach where a task encapsulates the logic of a predefined operator and records are processed as they arrive, and the micro-batch approach where a streaming computation is treated as a sequence of transformations on bounded sets by discretizing a distributed data stream into batches [19].

Historically, Aurora [20] was an early implementation that was used to parallelize streaming computations including rich operation and windowing semantics. Initially, it was designed as a single site stream-processing engine. Its predecessor, Borealis [21] was a stream processing engine that focused on balancing load on individual machines and distributing load shedding in static environments. Borealis was based on ROD (resilient operator distribution) to determine the best operator distribution plan, trying to be closest to an ''ideal'' feasible set, having a maximum set of machines underloaded. At the aftermath of Big Data and the Internet

of Things (IoT), new challenges were posed to traditional stream processing engines. These challenges arose from the need to work with huge amount of data, requiring massive parallel stream processing capabilities. New solutions were implemented to do for real-time processing, what Hadoop did for batch processing, using in-built scheduling techniques.

One of the most prominent representatives is the open source engine Apache Storm [10]. Storm uses a round-robin strategy to assign tasks to nodes' slots equally. In this way, though, logical links between tasks are not taken into consideration and inter-node communication costs may increase. This simplistic scheduling method frequently leads to low efficiency in load balancing among the available worker nodes. As a tuple is processed as it arrives, Storm follows the operator-based model.

Apache Samza [12] provides a unified programming API for both batch and stream processing. It is based on the publish/subscribe model that listens to a data stream and processes messages (tuples in Storm) as they arrive, one at a time. It is tightly tied to the Apache Kafka messaging system [22] for streaming data between tasks and Apache YARN [23] the for distribution of tasks among nodes in a cluster. YARN is configured to use Fair Scheduler with continuous-scheduling enabled.

Apache Flink [13] relies on a streaming execution model but can process both bounded and unbounded data, with two APIs running on the same distributed streaming execution. Its core is built on a data flow streaming engine, whose fundamental functionality is pipelining. A slot in a machine runs one pipeline which consists of multiple successive (communicating) tasks. Its system defines which tasks may share a slot and which tasks must be strictly placed into the same slot. It employs a schedule-once, long-running allocation of tasks and uses an immediate scheduling and a queued scheduling algorithm that work in an arbitrary fashion. The first one returns a slot immediately when there is a request, while the second one queues the request and returns the slot whenever it is available.

Apache Spark [11] batches up events within a short frame before processing the arrived data, offering full in-memory computations. Its scheduler runs jobs in FIFO fashion and each application tries to use all available nodes. Although job dependencies can be captured with FIFO, this approach can result in increased latency when a long running job delays jobs behind it [3].

The aforementioned DSPSs are presented in Table 2. The scheduling approaches incorporated in these systems are not optimal and do not take into consideration the application's structure that is about to be executed or prior knowledge of the cluster's condition.

**TABLE 2. Big data DSPSs and their scheduling techniques.**

| DSP Frameworks | Execution model | Scheduling decisions | Key Feauture |
|---|---|---|---|
| **Storm [10]** | Operator-based | Offline | Round-robin |
| **Samza [12]** | Operator-based | Offline | Fair Scheduler |
| **Flink [13]** | Hybrid | Online | Task locality, Immediate and Queued scheduling |
| **Spark [11]** | Micro-batches | Offline | FIFO |

### B. HEURISTIC APPROACHES

Most of the available DSPSs, allow users to specify customized scheduling for tasks. Several heuristic algorithms, that attempt to choose the optimal task allocation and scheduling technique to maximize their performance have been proposed. Most of them are implemented based on Apache Storm and its semantics.

Eidenbenz and Locher [8] proved that the task allocation problem is NP-hard and focused on stream topologies expressed as directed serial parallel decomposable graphs. In their work, they established a theoretical foundation by formally defining the allocation problem using a fixed set of resources with uniform capacities and bandwidths.

Aniello *et al.* [24] proposed a topology-aware scheduler for Storm. Their approach identifies possible sets of operators' threads to be scheduled on the same slot by looking how components are interconnected within the topology (DAG) and finally assigns slots to nodes in a round robin fashion. Their approach tries to balance the total CPU demand of each worker. As load imbalances are possible due to workload fluctuations, monitoring is used by their dynamic adaptive scheduler to handle these cases. Our approach is also a topology-aware approach but does not adapt to workload fluctuations. It improves system's performance by balancing load, reducing the inter-node communication cost and the required buffer space (for each task and thus for each worker node).

Peng *et al.* [17] implemented R-Storm, a topology and resource-aware scheduling approach that also schedules adjacent components' tasks as close as possible to reduce communication latency using breadth first traversal (BFS). It also tries to maximize the resource usage in a slot to minimize resource waste in nodes, using a resource-aware distance function. R-Storm's scheduler yields better performance than the default round-robin Storm's scheduler but it cannot control the performance when CPU sharing occurs. Towards this direction, De Xiang *et al.* [25] implemented a scheduling algorithm to minimize the resources wastage with the consideration of the worker nodes' load, named RB-storm. To do so, they applied a Resource Imbalance Vector (RIV) to represent the imbalance of resource utilization in tasks and work nodes. Both Peng *et al.* [17] and De Xiang *et al.* [25] worked with homogeneous clusters and considered memory as a constraint in their analysis. Resource waste is minimized with respect to the knowledge about the node capacities and the task requirements provided by user. Our solution does not require users to provide any further information but their topology, to achieve load balance and thus high performance.

Smirnov *et al.* [26] proposed another topology-aware strategy that also takes into consideration system's performance. Their approach is based on a genetic algorithm (GA) and uses performance models of executors on specified nodes to

estimate their throughput. In their experiments, they allowed CPU-sharing between tasks and they proved that maximum tasks' performance can be achieved via minimum CPU sharing, consuming though the maximum number of cores. In their experiments, GA scheduler was better in handling the high workload in several topologies, whereas R-Storm's and Storm's performance remained almost equal and low. Their performance models demand either history data or data collected during runtime. Our scheduler works with no prior knowledge. Memory saving is not considered in their analysis, whereas minimum CPU sharing that is suggested requires a cluster with large CPU-capacity.

Taking advantage of prior knowledge, Eskandari *et al.* [27] presented P-scheduler. This scheduler uses data transfer rates between tasks and topology workload obtained by running the known topology a priori. It places highly-communicating task pairs to the same working node applying hierarchical graph partitioning. Their work assumes that the cluster is homogeneous. Later, they extended their work [28] in heterogeneous clusters and proposed I-Scheduler, an iterative graph partitioning-based heuristic algorithm. This approach finds partitions of highly communicating tasks, sized according to node capacities and fuses each partition into a single task. A node's capacity is defined as the sum of the CPU speed for all cores within a node. While these schedulers can estimate the necessary number of nodes for an application and maximize resource utilization, memory resources and their consumption are not taken into account. Our approach does not not run the given topology a priori and cannot estimate the number of necessary nodes for an application. It uses all the nodes provided and tries to balance the load between them.

Shukla and Simmhan [14] also utilized a priori knowledge of the tasks' performance (using micro-benchmarks) to get a predictable scheduling behavior given a fixed input stream rate. Apart from assigning threads to the working nodes to ensure an expected performance, they also examined the matter of allocation of threads and resources for a DAG. That means that their scheduler determines the appropriate number of replicas per task and quanta of computing resources. Our algorithm cannot make such predictions. Their analysis is based on CPU and memory resources and offer lower resource requirements and VM cost compared to R-Storm. Benchmarking of application on a particular cluster prior to its run in production was also used by Rychly's *et al.* [7] resource and performance aware strategy. Their scheduling algorithm works on heterogeneous clusters and employs design-time knowledge (a tagging process is required) and benchmarking to take decisions. Unlike our work, both Rychly and Shukla use benchmarking techniques to get nodes' performance and tasks' computations characteristics. In this way they improve system's performance and balance the workload in the cluster optimally.

Linear programming is widely used in task scheduling approaches found in literature (e.g. [14], [17], [25], [26]). In such cases, the allocation problem is considered as a linear programming problem. Taking advantage of linear programming privilleges, Cardellini *et al.* [29] provided a solution that takes into consideration the heterogeneity of computing and network resources to optimize different QoS requirements. The proposed formulation considers user-oriented QoS attributes like end-to-end latency and application availability and network-related attributes like network usage, inter-node traffic and elastic energy. Memory consumption is not considered in their QoS metrics but it could be in a possible extension. Workload balancing and distribution are also not considered.

Towards this direction, Al-Sinayyid and Zhu [30] proposed MT-scheduler that uses a dynamic programming technique to efficiently map a DAG onto heterogeneous systems. They proposed a polynomial-time heuristic solution that is based on computing and data transfer requirements, and the capacity of the underlying cluster resources. Memory is not considered in node attributes in this work and unlike our work, reducing its consumption is not a matter of interest. Their system performance optimization is realized by estimating and minimizing the time incurred at the computing and transfer bottlenecks. To handle load and avoid system overloading, Nimbus periodically calls the scheduler to update the mapping process.

Janßen *et al.* [31] also assumed a static environment with heterogeneous resources. In their work, QoS metrics like response time, bandwidth congestion and resource fitting are combined into an optimization function to implement metaheuristic methods for near-optimal task placements. Unlike most approaches that work with Apache Storm, they extended Apache Flink's scheduling workflow.

Mortazavi-Dehkordi & Zamanifar [32] combined linear programming with queries' attributes. They proposed Bframework that examines the topology structure of a query to estimate the size of output stream flow of its operators to profile and partition them. Different scheduling strategies are applied to each partition. At first, operators are assigned to thread computing units and the identified threads are assigned to processes. Its off-line scheduler (they also extended their work to dynamic environments) finally assigns the processes to a set of available computing nodes. The aim is to minimize the inter-operator traffic load and thus the tuple latency of the accepted queries, distribute and balance the operators' workload. The complexity of their solution is logarithmic and memory is not considered in cluster resources.

The aforementioned approaches are static; they work off-line and are presented in Table 3. A review of these works is presented in [3] but the diversity of the clusters, on which the evaluation of these schemes took place, and the applications used do not allow safe comparisons between them.

Static scheduling fails to consider the workload variability and the variability in system conditions. This can lead to overutilization and under-utilization of the cluster resources and create a situation for job execution failure ( [5], [6]). In the context of DSPSs the relationships are very complicated and the problem requires strong assumptions to be solved [33].

**TABLE 3.** An overview of scheduling heuristics over DSPSs.

| Heuristic Approaches | Awareness | Based on | DSPS used |
|---|---|---|---|
| **Eidenbenz and Locher [8]** | Topology, Resources | Graph Theory | - |
| **Aniello et al. [24]** | Topology | Round-robin | Storm |
| **R-Storm [17]** | Topology, Resources, Resources Requirements | BFS, Euclidean distance, Linear Programming | Storm |
| **RB-storm [25]** | Resources, Resources Requirements | Resource Imbalance Vector, Linear Programming | Storm |
| **GA Storm [26]** | Topology, Performance | A priori knowledge, Genetic algorithm, Linear Programming | Storm |
| **P-Scheduler [27]** | Topology, Traffic, Workload | A priori knowledge, Graph partitioning | Storm |
| **I-Scheduler [28]** | Topology, Resources, Traffic, Workload | A priori knowledge, Graph partitioning | Storm |
| **Rychly et al. [7]** | Resources, Performance | A priori knowledge | Storm |
| **Shukla and Simmhan [14]** | Topology, Resources, Workload, Performance | A priori knowledge, Linear Programming, Queueing theory | Storm |
| **Cardellini et al. [29]** | Topology, Resources, Resources Requirements QoS attributes | Linear Programming | Storm |
| **MT-scheduler [30]** | Topology, Resources, Resources Requirements | Dynamic Programming, Linear Programming | Storm |
| **Janßen et al. [31]** | Topology, Resources, Resources Requirements QoS attributes | Linear Programming | Flink |
| **Bframework [32]** | Topology, Resources, Workload | Linear Programming, Queries attributes | Storm |

Dynamic scheduling techniques ( [5], [9], [24], [32]–[41]) monitor the queue waiting times and performance parameters (e.g. workload, traffic load, system's latency and throughput) during runtime and update tasks' replication degree and their placement (the MAPE-monitor, analyze plan,and execute- reference model is commonly used for implementing dynamic systems [5], [9], [14]). Decisions are taken online.

Elasticity is a matter of crucial importance in online environments as the input rate can vary drastically in streaming applications and operators' replication degree needs to be configured to maintain system's performance. Unfortunately, most of the available solutions require users to manually tune the number of replicas per operator but users usually have limited knowledge about the runtime behavior of the system [3], [9]. Several approaches (e.g. [9], [34], [39]–[43]) try to deal with replication runtime decisions in stream processing.

Dynamic techniques, while advantageous, can lead to local optima for individual tasks without regard to global efficiency of the dataflow. This introduces latency and cost overheads or offer weaker guarantees for the QoS. The application's reconfiguration and re-balancing, consisting of migrations and scaling operations may also be time-consuming (e.g. $\approx$ 200 secs in Storm [14]), and entail a significant service interruption when it comes to real-time stream processing. Recent works try to develop techniques to deal with application's downtime (e.g. [14], [33], [34], [43]).

In summary, our work presents a static and topology-aware formulation that well represents the task allocation and scheduling problem. The operator-based execution model and Apache Storm's semantics are used as in most systems mentioned in literature. DAGs are used to represent streaming applications. Our policy uses linear algebra and matrix transformations for our processes, while linear programming seems to be the dominant strategy in most of the aforementioned solutions.

Most of the static topology and resource-aware policies try to improve system's performance by considering the topology's structure and the capability of the resources, but they generally ignore the resource load. Our scheduler improves system's performance using an algorithm of linear complexity on a given topology's structure and taking advantage of pipelines. The required buffer space is reduced, while memory consumption is not considered in most cases of the state-of-the-art.

Traffic and workload-aware policies demand either prior knowledge to improve system's performance or users to provide design time information. However, users usually ignore the application's run-time resource demands. Of course, dynamic approaches can adapt to run-time needs but this requires monitoring and re-scheduling. While our proposed scheme considers only static scheduling for now, it handles queue waiting times efficiently. Rather than re-configuring online the tasks' allocation to cope with changes in the stream rates as dynamic techniques do, it tries to maintain a stable and robust configuration by balancing load between the cluster's nodes.

Of course an adaptive version of our scheme would increase its performance, so this extension is left for future work. The main idea behind this extended work is to model the possible system changes (for example, different number of tasks per executor or different number of nodes) as a task redistribution problem, formed by sets of linear Diophantine equations (more details are found in Section VII. Conclusions-Future Work). Such a strategy would be more comparable to the known dynamic schemes in the literature (e.g. [39]–[41]).

## VII. CONCLUSION-FUTURE WORK

This work presented a task allocation and scheduling approach to handle applications that require hefty communication between nodes and tasks. Our approach is organized in a set of communication steps, where there is an one-to-one communication between the system's nodes. This approach offers a set of advantages.

The buffer space required per task is reduced, resulting in higher throughput, as the largest percentage of tuples are

processed as they arrive to the target node (no buffering is required). In case of bufferred tuples resulting from each communication step, these are processed in a single step, where the pipeline is stalled. This reduces the extra processing time that would be necessary, if these tuples were processed at random times. In addition, the refinement phase employed by our strategy reduces the inter-node communication costs, thus it reduces communication latencies. Moreover, there is almost complete load balancing in the network resulting in reduced latencies and as the scheduler itself has linear complexity, it determines the communication steps very fast. Finally, the scheduler is proven to be periodic, with a period equal to $LCM(t, N)$. This means, that once the first LCM(t,N) communications are arranged, the same communication pattern can follow, in case of a bigger problem, where the number of nodes or tasks is multiplied by an integer factor.

The experimental results have verified the advantages mentioned above. Our strategy offers reduced average latency and percentage of buffed memory used compared to the default scheduler. Also, it offers good load balancing. For throughput testing, we compared our work to the default scheduler as well as to R-Storm. Our scheme was found to outperform both the other strategies and achieved higher throughput (tuples/min) under different scenarios, mainly as a result of reduced buffering.

In this paper, we assumed a static environment, in which bandwidth capacities and the other resources do not change over time, and also static ingestion rates, which in reality is often not the case due to possible changing network topologies and load fluctuations. In the future, we plan to implement a dynamic version, where task replicas will be introduced when necessary and the number of nodes may need to change during execution. Continuous monitoring and adaption of schedules would considerably improve solutions for real environments and applications.

One drawback of our current work is that, when $G = \gcd(t, N) = 1$, it needs to add a minimum number of tasks, so that the $G$ value becomes $\neq 1$. This needs to be done before scheduling. Currently, we are working on a dynamic strategy that will resolve this issue. Our dynamic approach models the system changes in the form of redistribution from $R$ to $R'$, where $R$ is the initial task distribution between nodes and $R'$ is the next task distribution derived from the system changes. Both $R$ and $R'$ are modeled via linear Diophantine equations (which are ideal for round robin distributions) and the task redistribution is determined by the solutions to the set of linear Diophantine equations, $R = R'$.

## REFERENCES

[1] N. Tantalaki, S. Souravlas, M. Roumeliotis, and S. Katsavounis, "Linear scheduling of big data streams on multiprocessor sets in the cloud," in *Proc. IEEE/WIC/ACM Int. Conf. Web Intell. (WI)*. New York, NY, USA: ACM, 2019, pp. 107–115.

[2] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The rise of 'big data' on cloud computing: Review and open research issues," *Inf. Syst.*, vol. 47, pp. 98–115, Jan. 2015.

[3] N. Tantalaki, S. Souravlas, and M. Roumeliotis, "A review on big data real-time stream processing and its scheduling techniques," *Int. J. Parallel, Emergent Distrib. Syst.*, p. 30, Mar. 2019.

[4] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *ACM SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, Dec. 2005.

[5] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3553–3569, Dec. 2017.

[6] K. Govindarajan, S. Kamburugamuve, P. Wickramasinghe, V. Abeykoon, and G. Fox, "Task scheduling in big data–review, research challenges, and prospects," in *Proc. 9th Int. Conf. Adv. Comput. (ICoAC)*, Dec. 2017, pp. 165–173.

[7] M. Rychlý, P. Škoda, and P. Smrž, "Heterogeneity-aware scheduler for stream processing frameworks," *Int. J. Big Data Intell.*, vol. 2, no. 2, pp. 70–80, 2015.

[8] R. Eidenbenz and T. Locher, "Task allocation for distributed stream processing," in *Proc. IEEE INFOCOM-35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[9] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency Comput., Pract. Exper.*, vol. 30, no. 9, p. e4334, May 2018.

[10] Apache Software Foundation. (2019). *Apache Storm*. Accessed: Jun. 5th, 2019. [Online]. Available: http://storm.apache.org/

[11] Apache Software Foundation. (2019). *Spark Streaming-Apache Spark*. Accessed: Jun. 5th, 2019. [Online]. Available: http://spark.apache.org/streaming/

[12] Apache Software Foundation. (2019). *Apache Samza-A Distributed Stream Processing Framework*. Accessed: Jun. 5th, 2019. [Online]. Available: https://samza.apache.org

[13] Apache Software Foundation. (2019). *Apache Flink-Stateful Computations Over Data Streams*. Accessed: Jun. 5th, 2019. [Online]. Available: https://flink.apache.org/

[14] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *J. Parallel Distrib. Comput.*, vol. 117, pp. 98–114, Jul. 2018.

[15] W. Zhang, S. Rajasekaran, T. Wood, and M. Zhu, "MIMP: Deadline and interference aware scheduling of Hadoop virtual machines," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, May 2014, pp. 394–403.

[16] Y. Wang and W. Shi, "Budget-driven scheduling algorithms for batches of MapReduce jobs in heterogeneous clouds," *IEEE Trans. Cloud Comput.*, vol. 2, no. 3, pp. 306–319, Jul. 2014.

[17] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proc. 16th Annu. Middleware Conf.* New York, NY, USA: ACM, 2015, pp. 149–161.

[18] G. Eisbruch, J. Leibiusky, and D. Simonassi, *Continuous Streaming Computation With Twitter's Cluster Technology*. Newton, MA, USA: O'Reilly Media, 2012.

[19] P. Carbone, G. E. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi, *Large-Scale Data Stream Processing Systems*. Cham, Switzerland: Springer, 2017, pp. 219–260.

[20] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *VLDB J. Int. J. Very Large Data Bases*, vol. 12, no. 2, pp. 120–139, Aug. 2003.

[21] N. Tatbul, Y. Ahmad, U. Çetintemel, J.-H. Hwang, Y. Xing, and S. Zdonik, *Load Management and High Availability in the Borealis Distributed Stream Processing Engine*. Berlin, Germany: Springer, 2008, pp. 66–85.

[22] Apache Software Foundation. (2019). *Apache Kafka-A Distributed Streaming Platform*. Accessed: Jun. 5th, 2019. [Online]. Available: https://kafka.apache.org/

[23] V. K. Vavilapalli, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, E. Baldeschwieler, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, and H. Shah, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput. (SOCC)*, 2013, pp. 1–16.

[24] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proc. 7th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*. New York, NY, USA: ACM, 2013, pp. 207–218.

[25] D. Xiang, Y. Wu, P. Shang, J. Jiang, J. Wu, and K. Yu, "RB-storm: Resource balance scheduling in Apache storm," in *Proc. 6th IIAI Int. Congr. Adv. Appl. Informat. (IIAI-AAI)*, Jul. 2017, pp. 419–423.
[26] P. Smirnov, M. Melnik, and D. Nasonov, "Performance-aware scheduling of streaming applications using genetic algorithm," *Procedia Comput. Sci.*, vol. 108, pp. 2240–2249, Jun. 2017.
[27] L. Eskandari, Z. Huang, and D. Eyers, "P-scheduler: Adaptive hierarchical scheduling in Apache storm," in *Proc. Australas. Comput. Science Week Multiconf.* New York, NY, USA: ACM, 2016, pp. 26:1–26:10.
[28] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "Iterative scheduling for distributed stream processing systems," in *Proc. 12th ACM Int. Conf. Distrib. Event-Based Syst.* New York, NY, USA: ACM, Jun. 2018, pp. 234–237.
[29] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proc. 10th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*. New York, NY, USA: ACM, 2016, pp. 69–80.
[30] A. Al-Sinayyid and M. Zhu, "Job scheduler for streaming applications in heterogeneous distributed processing systems," *J. Supercomput.*, p. 20, Mar. 2020.
[31] G. Janssen, I. Verbitskiy, T. Renner, and L. Thamsen, "Scheduling stream processing tasks on geo-distributed heterogeneous resources," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2018, pp. 5159–5164.
[32] M. Mortazavi-Dehkordi and K. Zamanifar, "Efficient resource scheduling for the analysis of big data streams," *Intell. Data Anal.*, vol. 23, no. 1, pp. 77–102, Feb. 2019.
[33] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," *Proc. VLDB Endowment*, vol. 11, no. 6, pp. 705–718, 2018.
[34] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy, "Dhalion: Self-regulating stream processing in Heron," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1825–1836, Aug. 2017.
[35] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "Distributed QoS-aware scheduling in storm," in *Proc. 9th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*. New York, NY, USA: ACM, 2015, pp. 344–347.
[36] C. Meng-Meng, Z. Chuang, L. Zhao, and X. Ke-Fu, "A task scheduling approach for real-time stream processing," in *Proc. Int. Conf. Cloud Comput. Big Data*, Nov. 2014, pp. 160–167.
[37] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Proc. IEEE 34th Int. Conf. Distrib. Comput. Syst.*, Jun. 2014, pp. 535–544.
[38] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "DRS: Dynamic resource scheduling for real-time analytics over fast streams," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, Jun. 2015, pp. 411–420.
[39] D. Sun, H. Yan, S. Gao, X. Liu, and R. Buyya, "Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams," *J. Supercomput.*, vol. 74, no. 2, pp. 615–636, Feb. 2018.
[40] M. Mortazavi-Dehkordi and K. Zamanifar, "Efficient deadline-aware scheduling for the analysis of big data streams in public cloud," *Cluster Comput.*, vol. 23, no. 1, pp. 241–263, Mar. 2020.
[41] D. Sun, S. Gao, X. Liu, F. Li, X. Zheng, and R. Buyya, "State and runtime-aware scheduling in elastic stream computing systems," *Future Gener. Comput. Syst.*, vol. 97, pp. 194–209, Aug. 2019.
[42] L. Xu, B. Peng, and I. Gupta, "Stela: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, Apr. 2016, pp. 22–31.
[43] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic, "Enabling elastic stream processing in shared clusters," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2016, pp. 108–115.

**NICOLETA TANTALAKI** received the Diploma degree in applied informatics and the first M.S. degree in business informatics from the University of Macedonia (UOM), Greece, and the second M.S. degree in information and communication technology in education from the Aristotle University of Thessaloniki, Greece. She is currently pursuing the Ph.D. degree with the University of Macedonia. She is also a Research Associate with the University of Macedonia. Her research interests are in the field of parallel and distributed computing systems, with a special emphasis on big data systems and scheduling algorithms for big data stream processing. She received over ten research and education grants from various agencies, such as the State Scholarship Foundation, the Onassis Foundation, and Google Company.

**STAVROS SOURAVLAS** (Member, IEEE) is currently an Assistant Professor of computer architecture and digital logic design with the Department of Applied Informatics, School of Information Sciences, University of Macedonia, where he joined in 2014. His research interests include computer architecture and performance evaluation, parallel and distributed systems, big data stream scheduling, cloud computing, and systems modeling and simulation.

**MANOS ROUMELIOTIS** (Member, IEEE) received the Diploma degree in electrical engineering from the Aristotle University of Thessaloniki, Greece, in 1981, and the M.S. and Ph.D. degrees in computer engineering from the Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, in 1983 and 1986, respectively. At VPI&SU, he taught as a visiting Assistant Professor, in 1986. From 1986 to 1989, he was an Assistant Professor with the Department of Electrical and Computer Engineering, West Virginia University. From 1995 until 2008, he was an Assistant Professor with the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. Since April 2008, he has been a Professor with the Department of Applied Informatics, University of Macedonia. He is currently the Director of the CNST Lab. He has published more than 120 articles in refereed journals and at conference proceedings. His research interests include digital logic simulation and testing, computer architecture and parallel processing, VR and serious gaming, and ancient technology.

**STEFANOS KATSAVOUNIS** is currently an Associate Professor with the Department of Production Engineering and Management, Democritus University of Thrace, Greece. His scientific interests revolve around scheduling, RCMPSP, project management, graph theory and modeling, heuristics for NP-hard problems in social networks, transportation and supply chain management, grey analysis, and data processing in material science.