Computational Management Science (2023) 20:40
https://doi.org/10.1007/s10287-023-00474-y

ORIGINAL PAPER

Mathematical modeling for further improving task scheduling on Big Data systems

Stavros Souravlas[1,2] · Sofia Anastasiadou[2] · Angelo Sifaleras[1]

Received: 30 June 2023 / Accepted: 21 August 2023

Abstract

In the big data era which we have entered, the development of smart scheduler has become a necessity. A Distributed Stream Processing System (DSPS) has the role of assigning processing tasks to the available resources (dynamically or not) and route streaming data between them. Smart and efficient task scheduling can reduce latencies and eliminate network congestions. The most commonly used scheduler is the default Storm scheduler, which has proven to have certain disadvantages, like the inability to handle system changes in a dynamic environment. In such cases, rescheduling is necessary. This paper is an extension of a previous work on dynamic task scheduling. In such a scenario, some type of rescheduling is necessary to have the system working in the most efficient way. In this paper, we extend our previous works Souravlas and Anastasiadou (Appl Sci 10(14):4796, 2020); Souravlas et al. (Appl Sci 11(1):61, 2021) and present a mathematical model that offers better balance and produces fewer communication steps. The scheduler is based on the idea of generating larger sets of communication steps among the system nodes, which we call superclasses. Our experiments have shown that this scheme achieves better balancing and reduces the overall latency.

Keywords Task scheduling · Big data streams · Task redistribution · Scheduling

✉ Angelo Sifaleras
sifalera@uom.gr

Stavros Souravlas
sourstav@uom.edu.gr

Sofia Anastasiadou
sanastasiadou@uowm.gr

[1] Department of Applied Informatics, University of Macedonia, 156 Egnatias Str., Thessaloniki 54636, Thessaloniki, Greece

[2] Faculty of Health Sciences, Department of Midwifery, University of Western Macedonia, KEPTSE Area, Prolemaida 50200, Western Macedonia, Greece

Published online: 06 September 2023

1 Introduction

During the last 15 years, data have increased on a large scale in almost all fields. Therefore, efficient ways of collecting and processing data from distributed resources must be implemented in order to gain insight from it. The term "big data" simply refers to the explosion of data volumes that are difficult to store, process, and analyze using traditional database technologies.

Managing large data volumes that arrive continuously many times can exceed the capabilities of individual machines. Stream data processing requires continuous calculation without interruption and high reliability requirements on resources. In this regard, it is important to develop efficient task scheduling algorithms. By *task scheduling*, we refer to the process we organize the resources in a way that the task completion time is reduced and the system resources are utilized in an improved way. Task scheduling focuses on which tasks to be placed on which previously obtained resources, controls the order of job execution, and is an $\mathcal{NP}$-hard problem, in the sense that no solution has been found to be optimal for all the possible topologies. Therefore, we cannot suggest that a proposed scheme is optimal for a certain topology or for certain loads. The works proposed in the literature are well-defined theoretically; however, none of them claims optimality. Simulation experiments are widely used to compare similar schedules (in terms of their goals), in approximately the same-sized clusters and over approximately similar datasets.

Naturally, responsive schedules are required to keep up with the transmission of massive data among large-scale tasks, and this aggravates the difficulty of the workflow scheduling problem (Tantalaki et al. 2020; Cardellini et al. 2016; Floratou et al. 2017). An important challenge presents itself when the system parameters (the number of available nodes and the execution tasks) need to change during runtime. The dynamic strategies found in the literature generally deal with the following two important issues: (1) *Task migrations*: They must be implemented during run-time. However, the tasks have to migrate along with their state and latencies can increase, depending on the processing load, and (2) *Load balancing*: which is important for the system's performance. Generally, imbalances reduce the performance of the system. In the remainder of this section, we briefly describe some of the most representative dynamic scheduling strategies with regard to the issues just mentioned.

Several distributed stream processing systems (DSPSs) that take advantage of the inherent characteristics of parallel and distributed computing such as Apache Storm, Spark Streaming, Samza and Flink have specifically emerged to address the challenges of processing high-volume, real-time data. Such systems are designed to execute complex streaming applications such as directed cyclic graphs (DAGs) over tuples of a stream. They leverage data parallelism using multiple threads of execution per task (replicas). The default Storm scheduler has become the point of reference for most of the researchers, who compare their proposed schemes against this simplistic scheduling algorithm. The main drawbacks of the Storm scheduler are:

1. It does not offer optimality in terms of throughput

2. It does not take into account the resource (memory, CPU, bandwidth) requirements/availability when scheduling
3. It is unable to handle cases where system changes incur.

In this work, we propose a reduced overhead modular arithmetic-based approach, RO-MOD scheduler, which is based on the idea of having each node receiving tuples for processing only from one other node at a time, whereas the number of such communications is reduced (tuples are grouped into reduced number of transmissions through superclasses), thus overheads are reduced. Our scheme has the following contributions:

1. *Reduced communication latencies:* The RO-MOD scheduler includes a mechanism which aims at optimizing the total inter-node communication costs.
2. *Good load balancing within the network:* As the RO-MOD scheduler is organized in communication steps, where one node communicates with only one node at a time, it can offer quite good balancing especially for linear topologies.
3. *Reduced communication overheads:* As fewer transmissions are scheduled by the RO-MOD scheduler, the overheads, and thus overall performance improve.
4. *Improves overall performance:* Our proposed scheme, while not optimal, outperforms existing strategies and is tailored to handle real-time stream processing efficiently in terms of system throughput, load balancing and average total latency.

In its current form, the proposed scheduler can be used with applications that can use relatively large datasets over small or medium sized clusters in terms of number of nodes and capacity (i.e., the number of tasks to be executed). A potential example could be a relatively small/medium sized retail/warehouse inventory for inventory management across all channels and locations, or the familiar word count application, which has been used for the experiments in this research work. However, in the future, the proposed scheduler could be applied to extremely large datasets and networks (for example, applications such as environmental monitoring and fraud detection).

The remainder of this work is organized as follows. All the notation used in this work is introduced in Table 1. Section 2 briefly summarizes some important scheduling approaches and describes the methodology on which their scheduling strategy is based. Section 3 describes the mathematical model of the RO-MOD scheduler. For completeness, we give a few details of our previous MOD scheduler. In Section 4, we present a few experimental results, and Section 5 concludes the paper and offers aspects for future work.

2 Related work

In this section, we describe some of the most important dynamic strategies found in the literature. Static strategies work offline and aim at placing the tasks to the most suitable nodes, in order to minimize the communication latencies. Dynamic

Table 1 Definitions and notations

Term	Definition
Component	A processing module that is either a Bolt or Spout.
Topology	Shows the interconnection between the tasks assigned to the spouts and bolts, for a given application
Spout	A type of component, which is a source of data streams and forwards an unlimited number of tuples in the topology.
Bolt	A component that receives, processes, and potentially forwards the processed data streams.
Task	A job that is assigned to a spout or bolt and it is designed to process selected data streams coming from other tasks.
Executor	A thread that is spawned in a worker process that executes one or more tasks (Peng et al. 2015)
Slot	An indication of the number of workers (see this definition later in this table) that can be run on a node.
Time Slot	A time slot is a division of time required to process a data stream.
Operator	A processing vertex
Processor	A physical unit that performs processing
Master Node	The node, which is acts as the scheduler for the tasks assigned to the worker nodes (see the next definition)
Worker Node	A node which is responsible for certain task execution.
Process	A contained for tasks to be executed.
Communication Class	A set of communicating processor pairs that derives from the solution of a linear Diophantine equation.
Homogeneous Classes	Classes, which include processor pairs that produce the same number of solutions for a certain linear Diophantine equation.

strategies monitor performance during run-time and may change the task assignment. Decisions are made online. However, rebalancing can be time consuming, e.g., ≈ 200secs in Storm (Shukla and Simmhan 2018; Tom et al. 2015). Moreover, several existing works employ the CPU without considering memory constraints (Tom et al. 2015; Xu et al. 2014; Shukla and Simmhan 2018) and this can lead to memory overflow.

Aniello et al. (2013) developed a dynamic online scheduler that reduces inter-node and inter-slot traffic on the basis of the communication patterns among executors observed at run-time. The goal of the online scheduler is to re-allocate executors to nodes so as to limit the number of workers on which a topology has to run, the number of slots available on each worker node, and the computational power of each node. The scheduler places pairs of communicating executors of each topology in descending order according to the rate with which they communicate data streams. If both executors have not yet been assigned, they are assigned to the least loaded worker. Otherwise, a set is generated by putting the least loaded worker together

with the workers where either executor of the pair is assigned. The assignment decision is based on the criterion of the lowest inter-worker traffic.

Shukla and Simmhan (2018, 2018) developed two techniques: their first approach (Shukla and Simmhan 2018) utilized benchmarks to develop performance model functions. Tasks are scheduled in such a way that the resources used are minimized and the performance offered is predictable. Also, they examined the matter of allocation of threads and resources for an application. Their second approach (Shukla and Simmhan 2018) tries to achieve load balancing by employing task migration. Tom et al. (2015) designed DRS, a dynamic resource scheduler that considers the number of operators in an application and the maximum number of processors available that can be allocated to them. Then, it finds an optimal assignment of processors that results in the minimum expected total sojourn time. They estimated the total sojourn time of an input by modeling the system as an Open Queueing Network (OQN). The system monitors the actual total sojourn time and checks if the performance falls, or if the system can satisfy the constraint with fewer resources, and reschedules if necessary. It repeatedly adds one processor to the operator with the maximum marginal benefit, until the estimated total sojourn time is no larger than a real-time constraint parameter. Generally, DRS' overhead is less than milliseconds in most of the tested cases, resulting in a small impact on system's latency.

Meng-Meng et al. (2014) proposed a dynamic task scheduling approach that considers links between tasks and reduces the cost of internode traffic by assigning tasks that communicate with each other to the same node or adjacent nodes. Node workload and internode communication traffic are examined a priori through switches. The T-Storm scheduler developed by Xu et al. (2014) also attempts to minimize internode traffic. The load information is collected during run-time by load monitors. The future load is estimated using a machine learning prediction method. The schedule generator periodically reads the above information from the database, sorts the executors in descending order of their traffic load, and assigns executors to slots. Executors are assigned to the same or nearby slots to reduce inter-process traffic. *Elasticity* an important issue in online environments, as input rate can vary in streaming applications, and it is necessary to configure the degree of operator replication, to maintain system performance. 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. Several approaches (e.g., (Cardellini et al. 2016; Floratou et al. 2017) 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 the global efficiency of the dataflow. This introduces latency and cost overhead. Rebalancing the application's reconfiguration and regular task migrations may also be time consuming. In this work we extend our load balanced dynamic scheme, which reduces buffering memory requirements, and we introduce a mathematical model that reduces the number of communications and thus the overheads. In addition, the reallocation or task is completely avoided. To explain the importance of this feature, let us consider dynamic task migration as a procedure that has to be performed during run-time. This means that, some tasks have to be assigned to a node other than the one they are being executed. However, this task

has to migrate along with its context (for example, all the data having been processed, assigned variable values, etc.). This cost can increase, depending on the processing load. On the other hand, task migration has the advantage that it can move communicating tasks to nearby nodes. This offers some efficiency. By narrowing the problem and avoiding this migration, the proposed RO-MOD scheduler avoids context overhead. Additionally, our strategy implements a stepwise all-to-all communication strategy, where every source node submits data streams to every target. As this strategy leads to an optimal in terms of cost communication schedule, the data streams are in any case transferred in minimized time, with no need for migration and unnecessary context switches. Since this is the case, our scheme eliminates the advantage offered by placing tasks at nearby nodes.

3 Mathematical model of communication

In this section, we present the mathematical notation required to implement our scheduler. These notation was introduced in our previous work (Souravlas and Anastasiadou 2020), but we will briefly describe them to make this work self-contained. Then, we present our extensions. The main idea behind what follows is to organize all the communications in groups of homogeneous in terms of communicating pairs, which will be used to achieve a schedule with minimized inter-node communication, while the load is equally balanced among the system's nodes. Initially, we assume that there is an initial distribution, where the tasks are equally distributed among the system's nodes. This is not a narrowing approach, since most scheduling strategies try to keep equal numbers of tasks among the nodes. Initially, let us define an equation that describes the round-robin placement of t consecutive tasks into a set of nodes. This equation will describe the initial task distribution, which are evenly distributed among the system's nodes.

$$n = \lfloor i/t \rfloor \mod N, \tag{1}$$

where N is the number of nodes in the initial distribution, n is the node where a task indexed i is placed and t is the number of tasks assigned per node. The range of i ranges from 0 to $N \times t$. For example, if there are $t = 4$ tasks per node and $N = 6$ our model assumes 24 tasks.Then, tasks $i = 0, 1, 2$ and 3 will be located at node $n = 0$, tasks $i = 4, 5, 6$ and 7 will be located at node $n = 1$, tasks $i = 8, 9, 10$ and 11 will be located at node $n = 2, \cdots$, and tasks $i = 20, 21, 22$ and 23 will be located at node $n = 5$. From Eq. 1, for some integer L we get the following:

$$\lfloor i/t \rfloor = LN + n. \tag{2}$$

Now, if we set an integer x, such that $x = i \mod t, 0 \leq x < t$, Eq. 2 becomes:

$$i = (LN + n)t + x \tag{3}$$

Eq. 3 describes the initial task distribution. We use $R(i, n, L, x)$ to symbolize this distribution. Now, if we wish to describe a different scenario, where the number of tasks or nodes changes, we need a second equation. This equation is derived in a

similar way. Now, if we assume that the number of nodes changes from N to Q, then Q is now the number of nodes, q is the node where a task indexed with j will be placed, and s is the new number of tasks. Thus, we get:

$$j = (MQ + q)s + y \tag{4}$$

where the integers M, y are defined similarly to L and x in Eq. 3. For y, we have $0 \leq y < s$. We use $R'(j, q, M, y)$ to symbolize a distribution that would arise in the event of the system changes described above.

We need to define sets of homogeneous communications between nodes. As will be described in the next subsection, these communications will be used to quickly produce an efficient communication schedule with reduced communication latencies. The idea is to equate the two distributions defined in Eq.3 and Eq.4 and generate a linear Diophantine equation. From modular arithmetic we know that the linear Diophantine equations can have solutions divided into classes (the term is defined later; see Eq. 8). These classes will be the basis for our homogeneous communications. The linear Diophantine equation required by our schedule is provided as follows:

$$R = R' \Rightarrow (LN + n)t + x = (MQ + q)s + y \tag{5}$$

or

$$nt - qs + (x - y) = MQs - LNt \tag{6}$$

Such linear Diophantine equations are solved using the extended Eucidean algorithm in logarithmic time, which is perfectly suitable for our scheduler.

Now, we set $g = gcd(Nt, Qs)$, so $Nt = Lg$ and $Qs = Mg$ for arbitrary integers L, M. Thus, $LNt = L^2 g$ and $MQs = M^2 g$. It follows that $LNt - MQs = g(L^2 - M^2)$, therefore $LNt - MQs$ a multiple of g. This means that there is an integer λ, such that: $LNt - MQs = \lambda g$. If we also set $z = x - y$, then (6) is rewritten as:

$$\lambda g - z = nt - qs \tag{7}$$

From modular arithmetic, we are aware that for linear Diophantine equations, a pair (n, q) belongs to a communication class k if:

$$(nt - qs)\, mod\, g = k \tag{8}$$

and it can be proven that all node pairs (p, q) that belong to a class produce the same number of solutions for Eq. 7. The number of such solutions is c. These node pairs will be named "homogeneous". For a proof, see (Souravlas and Anastasiadou 2020; Souravlas et al. 2021).

Apparently, the pairs (p, q) in each class define communications between pairs of nodes. Classes that produce the same number of solutions for a certain linear Diophantine equation are called *homogeneous*. There may be two or more homogeneous classes.

Our previous scheme was based on the idea of mixing pairs of communicating nodes from different classes to achieve communication steps, so that, during each

step, the data volumes transmitted between nodes were equal. However, the communication steps themselves were unequal; in other words, not equal data volumes were transmitted between different steps. Now, we extend these ideas to present a novel communication scheme, in which all communication steps carry equal data volumes.

3.1 Extensions to our previous scheme

Initially, let us define the function $\mathcal{D}(k_1, k_2)$ that computes the distance between two classes k_1 and k_2 such that:

$$\mathcal{D}(k_1, k_2) = \begin{cases} k_2 - k_1, & \text{if } k_2 \geq k_1 \\ g - k_1 + k_2, & \text{otherwise} \end{cases} \tag{9}$$

A group of classes $V = k_1, k_2 ... k_{n-1}, k_n$ that differ by $r \, mod \, g$, that is: $\mathcal{D}(k_1, k_2) = \mathcal{D}(k_2, k_3) = \ldots \mathcal{D}(k_{n-1}, k_n) = r \, mod \, g$ is called *superclass*. Proposition 1 provides a starting point for our streaming communication:

Proposition 1 *There exists a set of node pairs* (n, q) *within* t *in total classes* $\Theta = \{k_0, k_1, k_2 ... k_{r-1}\}$ *that satisfy*:

$$-x \mod g = k, x \in [0 \ldots t-1]. \tag{10}$$

Proof We rewrite Equation (5) as:

$$MQs - LNt = nt - qs + (x - y) \tag{11}$$

We know that $g = gcd(Nt, Qs)$, making $MQs - LNt$ a multiple of g. This means that there is an integer λ, such that: $MQs - LNt = \lambda g$. If we also set $\xi = x - y$, Equation (6) is rewritten as:

$$\lambda g - \xi = nt - qs \tag{12}$$

If we divide both parts of Equation (12) with g, we get:

$$\begin{aligned} (\lambda g - \xi) \mod g &= (nt - qs) \mod g \Rightarrow \\ (\lambda g \mod g) - (\xi \mod g) &= (nt - qs) \mod g \Rightarrow \\ -\xi \mod g &= (nt - qs) \mod g. \end{aligned}$$

Since $\xi = x - y$, it is obvious that $-\xi = y - x$. Therefore, we obtain the following equation:

$$(y - x) \mod g = (nt - qs) \mod g \Rightarrow (y - x) \mod g = k \tag{13}$$

By setting $y = 0$ (the indices of the first tasks in distribution R' (which are incurred when the system parameters change), we get the result. Thus, we have obtained a set of starting communication node pairs. □

Proposition 2 provides a simple way of finding all the pairs of communicating nodes when the system parameters change, based on the idea of node classes.

Proposition 2 *During a task redistribution problem, two neighboring nodes n_γ, n_δ send data streams to perform the tasks found on the same target node q if the processor pairs (n_γ, q) and (n_δ, q) belong to the same superclass.*

Proof We need to show that if $n_\gamma, q \in k_1$ and $n_\delta, q \in k_2$, then the distance between the classes k_1 and k_2 is equal to $r \mod g$. Assume that processors n_γ and n_δ send data streams to q and $(p_\gamma, q) \in k_1$, $(n_\delta, q) \in k_2$. For processor pair (n_γ, q) Equation (13) is rewritten as: $(y - x) \mod g = (n_\gamma r - qs) \mod g$. Similarly, for (n_δ, q) we have $(y - x) \mod g = (n_\delta r - qs) \mod g$. Without loss of generality, we assume that the indices of two neighboring source nodes n_γ, n_δ differ by 1 (the proof is similar for any other integer value). Thus, $(y - x) \mod g = (n_\delta r - qs) \mod g$ becomes: $(y - x) \mod g = (n_\gamma r + r - qs) \mod g$. We summarize the set of equations for the two processor pairs:

$$(y - x) \, mod \, g = \begin{cases} (n_\gamma r - qs) \mod g = k_1, \text{for } (n_\gamma, q) \\ (n_\gamma r + r - qs) \mod g = k_2, \text{for } (n_\delta, q) \end{cases} \tag{14}$$

There are three cases that need to be examined. Here, we prove the first one, and the remaining cases can be proven in a similar way.

1. $n_\gamma r + r - qs < g$ and $r < g$
2. $n_\gamma r + r - qs < g$ and $r \geq g$
3. $n_\gamma r + r - qs > g$

Case 1: $n_\gamma r + r - qs < g$ and $r < g$:

1.1: $n_\gamma r + r - qs > 0, n_\gamma r - qs > 0$: In this case $k_2 > k_1$ therefore (see Equation 9): $\mathcal{D}(k_1, k_2) = k_2 - k_1 = n_\gamma r + r - qs - n_\gamma r + qs = r$. Because $r < g \Rightarrow r = r \mod g$.

1.2: $n_\gamma r + r - qs < 0, n_\gamma r - qs < 0$: Same as in case 1.1.

1.3: $n_\gamma r + r - qs > 0, n_\gamma r - qs < 0$: In this case $(n_\gamma r - qs) \mod g = \lambda g + n_\gamma r - qs = k_1$, where λ is an arbitrary integer. Also, $k_2 = n_\gamma r + r - qs$ (recall that $0 < n_\gamma r + r - qs < g$). If $k_1 > k_2$, $\mathcal{D}(k_1, k_2) = g - k_1 - k2 = g - (\lambda g + n_\gamma r - qs) + n_\gamma r + r - qs = r - (\lambda - 1)g = r \mod g$ (since $(\lambda - 1)g$ divides g). If $k_1 \leq k_2$, then $\mathcal{D}(k_1, k_2) = k_2 - k_1 = n_\gamma r + r - qs - \lambda g + n_\gamma r - qs = r - \lambda g = r \mod g$. □

The following Tables 2, 3, and 4 provide an example of communication classes and superclasses.

Table 2 Classes and their costs for $N = Q = 16, t = 7, s = 11$

Class	Communication cost C_k	Class	Communication cost C_k
$k = 0$	7	$k = 8$	3
$k = 1$	7	$k = 9$	2
$k = 2$	7	$k = 10$	2
$k = 3$	7	$k = 11$	2
$k = 4$	7	$k = 12$	2
$k = 5$	6	$k = 13$	4
$k = 6$	5	$k = 14$	5
$k = 7$	4	$k = 15$	6

3.2 Communication scheduling

To schedule the communications among the various nodes (and thus among tasks), we need a stepping function that arranges the sequence of transmissions. We name this *stepping* function S and define it as follows:

$$S(k) = \begin{cases} r \bmod g, & \text{if } (r \bmod g) + k < g \\ (r \bmod g) - g, & \text{if } (r \bmod g) + k \geq g. \end{cases} \tag{15}$$

The following steps are necessary to form the target blocks in the generator nodes' memory.

Step 1 Start with a class k_0 that belongs to Θ, as defined in Proposition 1.

Step 2 All nodes n, ($n \in [0..N-1]$) assign data streams of cost $s' = vol(p, q)$ to the destination nodes q ($q \in [0..Q-1]$). We use the *vol* function to compute the communication cost for our model. This computation has been defined in Souravlas and Anastasiadou (2020); Souravlas et al. (2021), but we provide it here for completeness. Specifically, the function *vol* returns the number of quadruples (L, M, x, y) that satisfy the redistribution Equation (5):

$$vol(p, q) = \{(L, M, x, y) : mQs - LNr = nr - qs + (x - y)\} \tag{16}$$

Step 3 If $s' = s$, we pick another class and move to STEP 1. If $s' < s$, the transmission is incomplete, and we move to STEP 4. *Step 4* Use the stepping function S to get the next class member of the superclass k_1:$k_1 = k_0 + S(k_0)$ and move to STEP 2.

Repetition of these steps allows us to organize data streams with cost s among all nodes and their tasks. This approach has two advantages:

- The communication steps have the same cost, s.
- This scheme requires fewer steps compared to our previous class-based approach despite the fact that larger data stream volumes are carried. This is due to the fact that our previous scheme was based on selecting communication pairs from just one class. With the use of superclasses, we have managed to merge more communication pairs in each step, thus fewer steps are required.

Table 3 Superclasses 0,15,14,13 for $N = Q = 16, t = 7, s = 11$

Superclass 0			Superclass 15			Superclass 14			Superclass 13		
Classes	$k = 7$ (p,q)	$k = 0$ (p,q)	Classes	$k = 6$ (p,q)	$k = 15$ (p,q)	Classes	$k = 5$ (p,q)	$k = 14$ (p,q)	Classes	$k = 4$ (p,q)	$k = 13$ (p,q)
7,0	(1,0)	(0,0)	6,15	(4,2)	(3,2)	5,14	(7,4)	(6,4)	4,13	(10,6)	(9,6)
	(12,7)	(11,7)		(15,9)	(14,9)		(2,11)	(1,11)		(5,13)	(4,13)
	(7,14)	(6,14)		(10,0)	(9,0)		(13,2)	(12,2)		(0,4)	(15,4)
	(2,5)	(1,5)		(5,7)	(4,7)		(8,9)	(7,9)		(11,11)	(10,11)
	(13,12)	(12,12)		(0,14)	(15,14)		(3,0)	(2,0)		(6,2)	(5,2)
	(8,3)	(7,3)		(11,5)	(10,5)		(14,7)	(13,7)		(1,9)	(0,9)
	(3,10)	(2,10)		(6,12)	(5,12)		(9,13)	(8,13)		(12,15)	(11,15)
	(14,1)	(13,1)		(1,0)	(0,0)		(4,5)	(3,5)		(7,7)	(6,7)
	(9,8)	(8,8)		(12,10)	(11,10)		(15,12)	(14,12)		(2,14)	(1,14)
	(4,15)	(3,15)		(7,1)	(6,1)		(10,3)	(9,3)		(13,5)	(12,5)
	(15,6)	(14,6)		(2,8)	(1,8)		(5,10)	(4,10)		(8,12)	(7,12)
	(10,13)	(9,13)		(13,15)	(12,15)		(0,1)	(15,1)		(3,3)	(2,3)
	(5,4)	(4,4)		(8,6)	(7,6)		(11,8)	(10,8)		(14,10)	(13,10)
	(0,11)	(15,11)		(3,13)	(2,13)		(6,15)	(5,15)		(9,1)	(8,1)
	(6,9)	(5,9)		(14,4)	(13,4)		(1,6)	(0,6)		(4,8)	(3,8)
	(11,2)	(10,2)		(9,11)	(8,11)		(12,13)	(11,13)		(15,15)	(14,15)

Table 4 Superclasses 12,11,10 for $P = Q = 16, r = 7, s = 11$

Superclass 12				Superclass 11				Superclass 10			
Classes	$k = 10$	$k = 3$	$k = 12$	Classes	$k = 9$	$k = 2$	$k = 11$	Classes	$k = 8$	$k = 1$	$k = 10$
	(p,q)	(p,q)	(p,q)		(p,q)	(p,q)	(p,q)		(p,q)	(p,q)	(p,q)
10,3,12	(3,1)	(2,1)	(1,1)	9,2,11	(6,3)	(5,3)	(4,3)	8,1,10	(9,5)	(8,5)	(7,5)
	(14,8)	(13,8)	(12,8)		(1,10)	(0,10)	(15,10)		(4,12)	(3,12)	(2,12)
	(9,15)	(8,15)	(7,15)		(12,1)	(11,1)	(10,1)		(15,3)	(14,3)	(13,3)
	(4,6)	(3,6)	(2,6)		(7,8)	(6,8)	(5,8)		(10,10)	(9,10)	(8,10)
	(15,13)	(14,13)	(13,13)		(2,15)	(1,15)	(0,15)		(5,1)	(4,1)	(3,1)
	(10,4)	(9,4)	(8,4)		(13,6)	(12,6)	(11,6)		(0,8)	(15,8)	(14,8)
	(5,11)	(4,11)	(3,11)		(8,12)	(7,12)	(6,12)		(11,14)	(10,14)	(9,14)
	(0,2)	(15,2)	(14,2)		(3,4)	(2,4)	(1,4)		(6,6)	(5,6)	(4,6)
	(11,9)	(10,9)	(9,9)		(14,11)	(13,11)	(12,11)		(1,13)	(0,13)	(15,13)
	(6,0)	(5,0)	(4,0)		(9,2)	(8,2)	(7,2)		(12,4)	(11,4)	(10,4)
	(1,7)	(0,7)	(15,7)		(4,9)	(3,9)	(2,9)		(7,11)	(6,11)	(5,11)
	(12,14)	(11,14)	(10,14)		(15,0)	(14,0)	(13,0)		(2,2)	(1,2)	(0,2)
	(7,5)	(6,5)	(5,5)		(10,7)	(9,7)	(8,7)		(13,9)	(12,9)	(11,9)
	(2,12)	(1,12)	(0,12)		(5,14)	(4,14)	(3,14)		(8,0)	(7,0)	(6,0)
	(13,3)	(12,3)	(11,3)		(0,5)	(15,5)	(14,5)		(3,7)	(2,7)	(1,7)
	(8,10)	(7,10)	(6,10)		(11,12)	(10,12)	(9,12)		(14,14)	(13,14)	(12,14)

Due to the better balancing achieved (the cost of all the communications is s) and since the number of steps is reduced (thus overheads are reduced), the proposed scheme manages to reduce the overall latency, as will be seen in the the following Sect. 4.

For example, all communications defined in Superclass 0 (see Table 3), are formed by Classes 7 and 0 and the total cost is $7 + 4 = 11 = s$. The communications defined in Superclass 12 (see Table 4) are formed by the classes 10, 3 and 12 and the total cost is $2 + 7 = 2 = 11 = s$. Our previous approach, which was based on single classes, would require three communication steps instead of the one we use here. Our overhead in this simple case is reduced by 66%.

4 Simulation results and discussion

Our scheduling strategy was 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. Furthermore, 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 differences are considered to be located at lower distances between them). The tuples generated are assumed to have equal size, 16Kb, The tuples are associated to simple text datasets. The application we used for our simulations is the typical word count example. For example, one task processes a tuple and seeks all words starting from a selected letter. Then, it passes the processed tuples to a next task, which in turn, seeks a word that starts with a combination of the selected letter and a few more letters. Proceeding in this way, the application can perform word counting on large datasets.

For our experiments, we ran two topologies: (1) A random topology with four bolts and one spout, where the number of tasks per component is initially 4 and then changes to 5. (2) A linear topology with three bolts and one spout. In a linear topology, the bolts and spouts are linearly connected. In both topologies, there is an all-to-all connection between the tasks. We have chosen these topologies to fairly compare our work with similar schemes that we chose for comparisons. These schemes work on similar topologies. For both topologies, we used a cluster with $N = 5$ worker nodes, each with 4 slots. In both cases, an additional node, designated as the master node to host Nimbus and Zookeeper services (services used to control processes), was also used. Each part of the stream is considered as a small group of 100 tuples.

For our comparisons, we chose the default Storm scheduler, Meng's et al. strategy Meng-Meng et al. (2014), and two more recent approaches, the approaches of Shukla and Simmhan (2018) and the MT-scheduler (Maximum Throughput scheduler) presented by Al-Sinayyid and Zhu (2020). The Storm scheduler is the point of reference for a large percentage of strategies developed in the literature. Just like our scheduler, Meng's et al. strategy is also based on the idea of using a matrix model

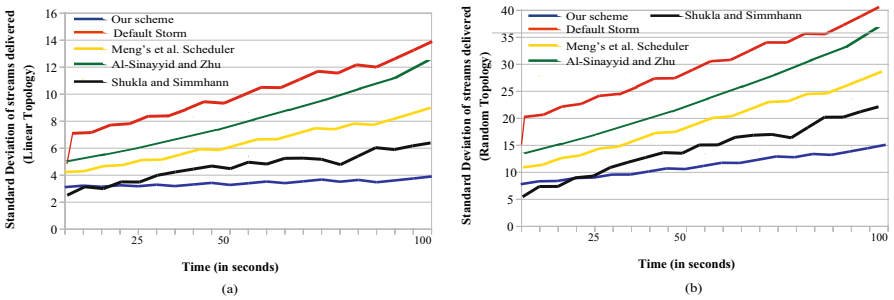

Fig. 1 Load balancing comparisons

for task scheduling. Moreover, it is based on task migrations, a strategy that can be opposed to our stepwise scheme. The approach found in Shukla and Simmhan (2018) also focuses on using task migration to balance network load. Finally, the MT-scheduler is the more recent approach, which tries to maximize throughput by trying to minimize the transfer times, so it is somehow comparable to our scheduler. The selected schemes will be useful in comparing our work with strategies that use some similar ideas (like matrices) and strategies that have "opposing" ideas (task migrations vs. stepwise communications).

4.1 Load balancing comparisons

In the first set of experiments, we compare the load balancing achieved by the compared strategies. To do so, we regularly (every five seconds) computed the average standard deviation of the load being delivered to each node (see Fig. 1) for both topologies considered. An increase in the standard deviation value indicated less balancing between nodes.

For the default Storm scheduler, it is obvious that the lack of a load balancing causes high imbalances as the time proceeds. Specifically, the default Storm scheduler does not care about the current load of the communicating tasks; it just handles the tasks as independent entities. Meng's scheme pays full attention to the links that connect the communicating tasks and effectively reduces traffic between these nodes through switches. This balances the load of the links, but the processing load is not balanced: The approach assigns tasks that communicate to each other to the same node or to adjacent nodes, which are selected via the current link information. This means that, when the link state is such that one or a few target nodes are chosen to accommodate the new tasks, then imbalances occur. The approach of Shukla and Simmhan is also based on using the link information through dataflow checkpoints. In this regard, there is no fear that in-flight messages will be lost. A timeout period can be used where no data is transmitted. During this period, the tasks to be migrated are paused, and the in-flight messages can be transmitted without contentions. This policy can reduce the imbalances that have occurred, but diminishes overall performance of the system. Moreover, it is based on link information, not on actual processing performed on each node. In our simulations, this regulation

helps Shukla's and Simmhan's approach to have somehow better balancing results compared to Meng's et al. scheme, as Fig. 1a indicates. The regulations performed are indicated with some slight peaks displayed on the line. Finally, Fig. 1b shows the load per node when we run the random topology.

The MT-Scheduler proposed by Sinayyid and Zhu, the bottlenecks determine the mapping and remapping procedures. The only regulation policy used is that users are allowed to configure and regulate the data locality, in order to maintain execution of the tasks as close to the data. This minimizes the transfer cost, but by no means guarantees the load balance. As Fig. 1 indicates, this strategy suffers from higher imbalances compared to the other schemes (excluding the default Storm scheduler. Our scheme was found to have smaller standard deviation values compared to the other schemes, thus better balancing. The reason is apparent: at each communication step, there is a fixed in terms of cost communication among the system nodes. Thus, the curve that shows the results of the load balance of our strategy seems to be gradually increasing. Our strategy is not as heavily affected by the growing number of tuples added to the nodes, as this is done in a balanced way (especially for the linear topology).

Our experiments have shown that the five lines showed quite similar behavior when we changed the topology from linear to random, but their slope appears larger, indicating that the standard deviation values are more affected (increasing with time). Thus, as the standard deviations computed suggest, higher imbalances occur when random (and generally more complex) topologies are used.

4.2 Throughput comparisons

In this set of experiments, we compared the overall throughput of the five strategies, that is, the number of streams being processed. Because our strategy implements its task migrations whenever they are required using minimum in terms of cost communication steps, it outperforms the compared strategies. Apart from the context switch overheads, task migrations require some more procedures, which add extra cost; killing of the migrated tasks from their original nodes to complete the migration process or possible recoveries of messages that were lost and after the migration process due to killing the dataflows or due to timeout policies being employed, like the one described for the second approach of Shukla and Simmhan. Thus, our careful stepwise implementation policy manages to calm down the effects of task migrations to the maximum possible extent (see Fig. 2).

4.3 Average latency

Figure 3 plots the average latency for the four works that were found to be dominant in terms of load balancing and throughput, that is, our scheme, Shukla's and Simmhan's scheme, and the MT-Scheduler. The proposed scheme takes advantage of the way it migrates the tasks, it has better load balancing and reduced communication steps (overheads) and manages to reduce the overall latency. In our work, the average latency seems to be changing quite smoothly and the slight peaks indicate the existence of

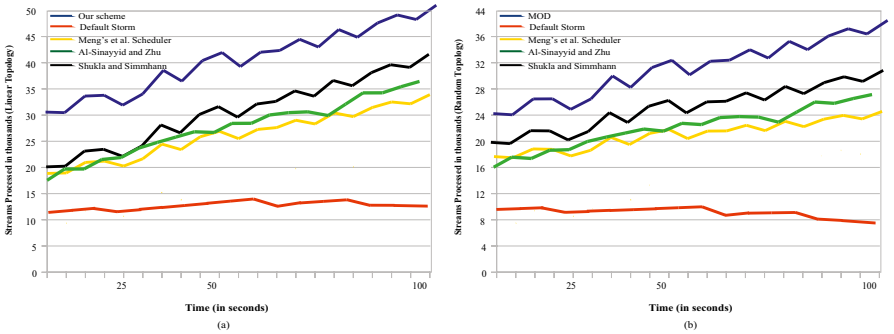

Fig. 2 Throughput comparisons

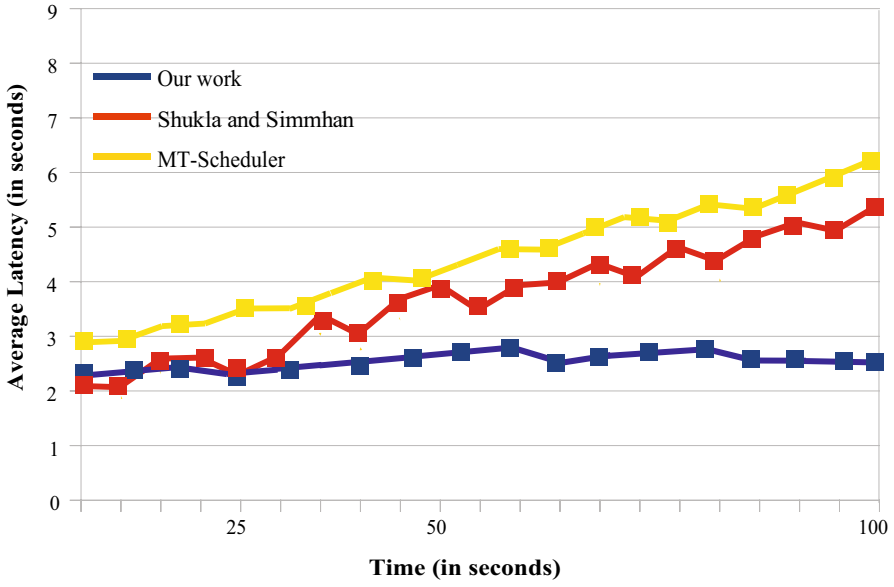

Fig. 3 Latency comparisons

migrations from time to time. Shukla's and Simmhan's scheme appears to have larger peaks, and this can be explained by the regular timeouts employed, which increase the average latencies. The MT scheduler has the highest latencies, as a result of the lack of a serious policy on load balancing.

5 Conlusions and future work

This work has presented a dynamic task scheduling approach that handles system changes (number of tasks or nodes) for applications that require heavy (and sometimes all-to-all) communications between the system's nodes and tasks. Our approach extends our previous work and organizes communication in a set of well-defined steps based on the idea of larger groups of communication classes called superclasses. This approach has the advantage of generating fewer communication steps and thus smaller latencies.

The simulation results have shown that our scheduler offers better load balancing and throughput compared to a number of other schemes chosen for comparison. It also reduces the overall latency, due to the way that the task migrations are implemented using the minimum number of steps, as they are determined by our communication scheduling policy. An advantage of the proposed model is that its computational complexity is the complexity of the Extended Euclidean algorithm, which is logarithmic. Therefore, its application cannot be considered as a burden. The cost of communication in every step is the same and is dictated by the s parameter, that is, the new number of tasks assigned to each node after task redistribution. Compared to other task distribution schemes, one can claim that, under certain task redistributions, the selected value of s is too large, so the nodes are overloaded. However, the trade-off here is that balancing is guaranteed, no matter what the value of s is.

Apparently, our scheme can theoretically be adapted to any workload size under the hypothesis that the number of tasks within each machine is adequate to handle this load. However, the examination of extremely large datasets (such as sensor applications) in very large networks is the subject of our future work. Perhaps, the proposed model would have to be subject to various changes in order to deal with aspects such as the communication costs and the adaptability under extremely large networks. Also, certain limitations on the value of s may need to be imposed as the datasets grow larger and larger.

On the other hand, we can consider that for very small-scale workloads (for example, applications that may require relatively small datasets), perhaps a straightforward round-robin approach like the default Storm scheme may be necessary. In our comparison results, we used the typical word count application with large datasets to serve our comparison purposes.

Different scheduling scenarios may appear not only depending on the application, but also on the cluster topology. Regularly, linear topology is preferred in terms of efficiency. However, there may be cases where an irregular topology can reduce the communication cost between certain nodes. In any case, the reduction of the inter-node communication cost does not always suffice to guarantee lower latencies. Our scheme manages to avoid imbalances in terms of data loads transferred in the paths among the system nodes. This reduces the overall latency.

In the future, we wish to extend this work for larger networks with larger numbers of nodes and suggest mathematical models (or change the existing one) targeted for specific topologies.

Funding Open access funding provided by HEAL-Link Greece.

References

Al-Sinayyid A, Zhu M (2020) Job scheduler for streaming applications in heterogeneous distributed processing systems. J Supercomput 20:9609–9628

Aniello L, Baldoni R, Querzoni L (2013) Adaptive online scheduling in Storm. In: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS '13), pp. 207–218

Cardellini V, Grassi V, Presti L, Nardelli M (2016) Optimal operator placement for distributed stream processing applications. In: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems. DEBS '16, pp. 69–80

Floratou A, Agrawal A, Graham B, Rao S, Ramasamy K (2017) Dhalion: self-regulating stream processing in Heron. Proc VLDB Endow 10(12):1825–1836

Meng-Meng C, Chuang Z, Zhao L, Ke-Fu X (2014) A task scheduling approach for real-time stream processing. In: Proceedings of the International Conference on Cloud Computing and Big Data, pp. 160–167

Peng B, Hosseini M, Hong Z, Farivar R, Campbell R (2015) R-Storm: Resource-aware scheduling in storm. In: Proceedings of the 16th Annual Middleware Conference. Middleware '15, pp. 149–161. ACM, Vancouver, BC, Canada

Shukla A, Simmhan Y (2018) Model-driven scheduling for distributed stream processing systems. J Parall Distrib Comput 117:98–114

Shukla A, Simmhan Y (2018) Toward reliable and rapid elasticity for streaming dataflows on clouds. In: Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1096–1106

Souravlas S, Anastasiadou S (2020) Pipelined dynamic scheduling of big data streams. Appl Sci 10(14):4796

Souravlas S, Anastasiadou S, Katsavounis S (2021) More on pipelined dynamic scheduling of big data streams. Appl Sci 11(1):61

Tantalaki N, Souravlas S, Roumeliotis M (2020) A review on big data real-time stream processing and its scheduling techniques. Int J Parall Emerg Distrib Syst 35(5):571–601

Tom ZJ, Fu JD, Richard TBM, Winslett M, Yin Y, Zhang Z (2015) DRS: Dynamic resource scheduling for real-time analytics over fast streams. In: Proceedings of the 35th IEEE International Conference on Distributed Computing Systems, pp. 411–420

Xu J, Chen Z, Tang J, Su S (2014) T-Storm: Traffic-aware online scheduling in Storm. ICDCS '14, pp. 535–544