Evolutionary Computation meets Stream Processing

Vincenzo Gulisano[1][0000−0002−2136−9179] and Eric Medvet[2][0000−0001−5652−2113]

[1] Department of Computer Science and Engineering, Chalmers University of Technology, Sweden
[2] Department of Engineering and Architecture, University of Trieste, Italy
vincenzo.gulisano@chalmers.se, emedvet@units.it

Abstract. Evolutionary computation (EC) has a great potential of exploiting parallelization, a feature often underemphasized when describing evolutionary algorithms (EAs). In this paper, we show that the paradigm of stream processing (SP) can be used to express EAs in a way that allows the immediate exploitation of parallel and distributed computing, not at the expense of the agnosticity of the EAs with respect to the application domain. We introduce the first formal framework for EC based on SP and describe several building blocks tailored to EC. Then, we experimentally validate our framework and show that (a) it can be used to express common EAs, (b) it scales when deployed on real-world stream processing engines (SPEs), and (c) it facilitates the design of EA modifications which would require a larger effort with traditional implementation.

Keywords: Parallellization · Design of EAs · Distributed computing

1 Introduction

Artificial intelligence (AI) has witnessed significant growth in recent decades, propelled by advancements in hardware and the establishment of de-facto standard frameworks with rich application programming interfaces (APIs). These frameworks play a pivotal role in decoupling aspects such as efficient implementation and interfacing with diverse hardware platforms. Despite this overall progress, not all AI-related techniques have advanced uniformly. Evolutionary computation (EC), the focus of this paper, has seen the emergence of numerous fragmented and specialized frameworks with limited customizability and in need of structural modifications to address crucial aspects like scalability and parallel/distributed execution [2].

To make a significant step towards enhancing the adaptability and performance of EC, we present here a novel approach to overcome these limitations by leveraging a state-of-the-art computing paradigm named stream processing (SP), widely adopted in the IoT-to-Cloud continuum [6, 11, 15, 24]. SP allows to describe forms of computation which occur over streams of items that flow over time. By connecting simple (or complex) SP building blocks, either stateless of

stateful, and organizing them in graphs called queries, one may describe complex workflows in an elegant way. Moreover, and more importantly, SP is more than a scientifically mature field [4, 30, 34]: there exist many widespread software frameworks which are used in real-world production-level applications [17] and nicely couple with different kinds of distributed computing systems [16, 29]. On the other side, most of the significant evolutionary algorithms (EAs) used in EC are population-based and iterative. In practice, this means that several candidate solutions exist during the execution of an EA and they are modified over time by re-iteratively applying a small set of operations (e.g., selection, variation, evaluation), combined in simple or more complex ways. The potential link between SP and EC is hence clear: candidate solutions, i.e., individuals in the EC jargon, are the items that can be processed by SP blocks in EC-aware queries that reflect the overall working principles of the EA.

In this work, we lay down this link and propose the first formal framework based on the SP paradigm for EC. We describe a number of SP blocks, called operators, tailored to EC which may be used to define different EAs, without imposing limitations on the kind of entities the EA can work on. In fact, one of the strong points which has favored the usage of EC in very different domains is its capability to work with different kinds of solutions (e.g., numerical formulae [18], Boolean functions for cryptography [31], security policies [20], robotic controllers [27]) and hence on different domains [5]. With most of these solution kinds, simply resorting on GPU-based parallelization would not be enough. In contrast, SP facilitates seamless adaptation of existing EAs to different domains while decoupling the support for efficient parallel/distributed execution on larger and more heterogeneous computing systems.

We validate our proposal experimentally, by using our SP framework to implement two EAs and applying them to five problems with two kinds of solutions (bitstrings and mathematical formulae). We show that the implementations of stream-based EAs are effective and we provide evidence of how simultaneous executions (jobs) can be customized without requiring alterations to the underlying implementation. We also showcase the advantages of decoupling EA definition and execution in SP with advanced intra- and inter-job customization options such as merging of populations from different jobs at runtime, to trade-off fitting performance and job completion time.

To our knowledge, this is the first attempt to port EAs into the realm of SP. However, the power of SP has already been harnessed for improving other AI-realated workflows [22, 23], mostly based on machine learning (ML). Conversely, AI and ML proved useful to tune and optimize SP tasks. Actually, most of the existing literature (e.g., [33]) focuses on the latter case, which is not relevant to our work. Concerning the first case, i.e., SP for AI, we note that Apache Flink provides a set of ML APIs [8], which, however, do not support any EA. As stated in [3], the integration of ML within stream processing is still at its early stage, with many systems that support efficient data distribution across ML jobs but that, under the hood, still rely on RPC calls to external frameworks for carrying out the actual learning processes.

For what concerns the parallelization of EAs [1], many previous works targeted specific hardware platforms, with an increasing interest in the last decade in those based on GPUs [19, 28]. However, despite the EC community acknowledges that scalability and exploitation of large computing infrastructures are key goals [14], the SP paradigm has not yet been applied to EC. Nevertheless, modern EC software [9, 21] is often designed to exploit concurrency.

2 Preliminaries: stream processing

2.1 Definitions

A *stream* S is an unbounded sequence of tuples defined over the attributes $A(S) = \{\tau, a_1, \ldots, a_p\}$, where each *attribute* a has a domain $V(a)$. τ is a special attribute called *timestamp* defined in a time domain $V(\tau) = \mathbb{T} \cup \mathbb{R}$, $\mathbb{T}$ being the time domain. A *tuple* t of a stream S is composed of $|A(S)| = p + 1$ attribute values, with each value $v(a, t) \in V(a)$, with $a \in A(S)$.

For the sake of brevity, we write $V(A(S))$ for the set of all possible tuples defined on the attributes $A(S)$ of a stream S, i.e., $V(A(S)) = \mathbb{T} \times V(a_1) \times \cdots \times V(a_p)$ with $A(S) = \{\tau, a_1, \ldots, a_p\}$. Moreover, we define $A_\tau(S) = A(S) \setminus \{\tau\}$ and, accordingly, we write $v(A_\tau(S), t)$ for the part of the tuple of t consisting of all the attributes without the timestamp, i.e., $v(A_\tau(S), t) = \langle v(a_1, t), \ldots, v(a_p, t)\rangle \in V(A_\tau(S))$.

In the literature [12], several models build on different assumptions about the ordering of tuples within a stream S based on their τ attribute. For generality, we do not impose a total order on τ, but only assume there can exist substreams $S_k \in S$ so that $\forall t_i, t_j \in S, k(v(A_\tau(S), t_i)) = k(v(A_\tau(S), t_j)) \implies v(\tau, t_i) \leq v(\tau, t_j)$, where $k(v(A\tau(S), t)$ is an arbitrary function applied on any attribute value of tuple t, implies t_i is observed before t_j in S.

A *stream processing query* (or simply query) is a directed graph where nodes are either sources, operators, or sinks and edges are streams. A query meets the following criteria: (a) sources have no incoming streams, (b) sinks have no outgoing streams, (c) and operators have at least one incoming and one outgoing stream. Sources generate tuples over time. Sinks consume tuples. Operators process tuples from input streams and produce tuples to output streams, not necessarily resulting in each tuple in the input stream becoming a tuple in the output stream. We describe operators in the next section.

2.2 Operators

Operators are either stateless or stateful. Stateless operators do not maintain a state that evolves based on the tuples they process, while stateful operators do. Since a comprehensive overview of common operators found in stream processing engines (SPEs) [12] is not within the scope of this contribution, we present next the ones we consider in our work.

Merger. This is a stateless operator with n input streams $S_{\text{in},1}, \ldots, S_{\text{in},n}$ and one output stream S_{out}, such that $A(S_{\text{in},1}) = \cdots = A(S_{\text{in},n}) = A(S_{\text{out}})$. We denote by M a Merger operator.

Merger outputs each input tuple of each input stream on the output stream, keeping unmodified the timestamp and each attribute.

Delayer. This is a stateless operator with one input stream S_{in} and one output stream S_{out}, such that $A(S_{\text{in}}) = A(S_{\text{out}})$ and is defined by a delay value $\delta \in \mathbb{T}$. We denote by D[δ] a Delayer operator with its parameter.

Delayer outputs each input tuple t_{in} on the output stream as a tuple t_{out}, increasing the timestamp by δ, i.e., $v(\tau, t_{\text{out}}) = v(\tau, t_{\text{in}}) + \delta$.

FlatMap. This is a stateless operator with one input stream S_{in} and n output streams $S_{\text{out},1}, \ldots, S_{\text{out},n}$ and is defined by n functions $f_1, \ldots, f_n$, each processing an input tuple into a bag of tuples of the i-th output stream. Formally, $f_i : V(A_\tau(S_{\text{in}})) \to \mathcal{P}^*(V(A_\tau(S_{\text{out},i})))$. We denote by FM[$f_1, \ldots, f_n$] a FlatMap operator with its parameters.

Intuitively, FlatMap maps one incoming tuple to zero or more output streams. Formally, for each input tuple t_{in} and each f_i, FM[$f_1, \ldots, f_n$] first computes the bag $T_{\text{out},i} = f_i(v(A_\tau(S_{\text{in}}), t_{\text{in}}))$, then it outputs one tuple $t_{\text{out},i}$ for each element in $T_{\text{out},i}$ to the i-th output stream, setting $v(\tau, t_{\text{out},i}) = v(\tau, t_{\text{in}})$.

Aggregate. This is a stateful operator with one input stream S_{in} and one output stream S_{out}. Aggregate is defined by: a key function $f_{\text{key}} : V(A_\tau(S_{\text{in}})) \to K$, with K being a discrete set of keys, that takes a tuple of S_{in} and returns a key; an output function $f_{\text{out}} : \mathcal{P}^*(V(A_\tau(S_{\text{in}}))) \to \mathcal{P}^*(V(A_\tau(S_{\text{out}})))$ that takes a bag of tuples of S_{in} and produces a bag of tuples of S_{out}; a window size $w_s \in \mathbb{R}^+$; and a window advance $w_a \in \mathbb{R}^+$. We denote by A[$f_{\text{key}}, f_{\text{out}}, w_s, w_a$] an Aggregate operator with its parameters.

Intuitively, Aggregate groups incoming tuples in sets (called *instances*), based on their key and timestamp, and transforms instances in outgoing tuples; the instances constitute the state of the operator: they are initially empty and are updated based on incoming tuples. Formally, A[$f_{\text{key}}, f_{\text{out}}, w_s, w_a$] works as follows for each input tuple t_{in} of S_{in}: (1) it computes the key $k = f_{\text{key}}(v(A_\tau(S_{\text{in}}), t_{\text{in}}))$; (2) it computes the *epochs* $e_1, \ldots, e_h \in \mathbb{N}$ such that $v(\tau, t_{\text{in}}) \in [e_i w_a, e_i w_a + w_s[$; (3) it adds t_{in} to each instance I_{k,e_i} associated with k and e_i. Finally, based on the assumption that tuples in S_k are timestamp sorted (see Section 2.1), for each $I_{k',e}$ such that $ew_a + w_s < v(\tau, t_{\text{in}})$ and $k' = k$, (1) it computes $T_{\text{out}} = f_{\text{out}}(I_{k',e})$ and (2) it outputs one tuple t_{out} for each element of T_{out} with $v(\tau, t_{\text{out}}) = \max_{t \in I_{k',e}} v(\tau, t)$. After having set the tuples of T_{out}, Aggregate removes the corresponding instance $I_{k',e}$ from the state.

Multiplexer. This is a specialized FM[$f_1, \ldots, f_n$] with one input stream S_{in} and n output stream $S_{\text{out},1}, \ldots, S_{\text{out},n}$, such that $A(S_{\text{in}}) = A(S_{\text{out},1}) = \cdots = A(S_{\text{out},n})$. All the $f_i : V(A(S_{\text{in}})) \to \mathcal{P}^*(V(A(S_{\text{out}})))$ functions are the same: they return a one-element-bag containing the input. Hence, this operator forwards each input tuple on each output stream. We denote by X a Multiplexer.

Filter. This is a specialized FM[f_1] with one input stream S_{in} and one output stream S_{out}, such that $A(S_{\text{in}}) = A(S_{\text{out}})$. The only $f_1 : V(A(S_{\text{in}})) \to \mathcal{P}^*(V(A(S_{\text{out}})))$ applies a predicate $\pi : V(A(S_{\text{in}})) \to \{\text{true}, \text{false}\}$ to the input $v(A(S_{\text{in}}), t_{\text{in}})$ and returns an empty bag if $\pi(v(A(S_{\text{in}}), t_{\text{in}}))$ is false or a one-element-bag containing only the input otherwise. We denote by F[π] a Filter with its parameter.

3 EAs as queries

We consider optimization problems defined by a search space P and a fitness function $q : P \to \mathbb{R}$. We assume, without loss of generality, that q has to be minimized, i.e., the goal is to find $p^\star = \arg\min_{p \in P} q(P)$.

We employ an EA for solving the optimization problem. We do not enforce any specific constraint on the EA. We only assume it is iterative and population-based, i.e., that it evolves a population (formally, a bag) of individuals iteratively until some predefined termination criterion is met. We call *individual* a triplet given by an *genotype* $g \in G$, a *phenotype* $p \in P \cup \varnothing$, which is a candidate solution to the optimization problem, and its fitness, which can be either $q(p)$ or $\varnothing$; for both the phenotype and the fitness, $\varnothing$ represents the case when they are not yet been evaluated for g. We call genotype-phenotype *mapping* a function $\phi : G \to P$ that allows to obtain a phenotype $p = \phi(g)$ from a genotype g: in EC terms, ϕ (together with its domain G and co-domain P) defines the representation of solutions. We denote by $\mathcal{I} = G \times P \times \mathbb{R}$ the set of all possible individuals for a problem defined over P and tackled with a $\phi : G \to P$ representation.

An EA has some parameters and is, in general, stochastic. We call *job* an execution of an EA with some predefined parameter values: the outcome of a job is one solution p^* corresponding to the best individual, i.e., the one with the best fitness in the population at the last iteration of the EA.

In the following sections, we describe how to use stream processing to describe EAs, namely, how to express EAs as queries. To this aim, we introduce a number of operators, defined as specializations of the FlatMap and Aggregate operators described in Section 2.2, with names and functionalities which are familiar to the EC community.

In a query representing an EA there are three kinds of streams: (a) streams S_J of jobs, where $A(S_J) = \{\texttt{jobId}, \dots\}$, $V(\texttt{jobId}) = \mathbb{N}$, and the other attributes describe possible other parameters of the EA; (b) streams S_I of individuals, where $A(S_I) = \{\texttt{jobId}, \texttt{individual}\}$ and $V(\texttt{individual}) = \mathcal{I}$; (c) streams S_{I^*} of bags of individuals, where $A(S_{I^*}) = \{\texttt{jobId}, \texttt{individuals}\}$ and $V(\texttt{individuals})$ is $\mathcal{P}^*(\mathcal{I})$.

Finally, in a query representing an EA the time domain $\mathbb{T}$ is $\mathbb{N}$ and $v(\tau, t)$ represents the iteration of "birth" of an individual.

IndividualFactory. This is a specialized FM[f_1] with one input stream of jobs $S_{J,\text{in}}$ and one output stream of individuals $S_{I,\text{out}}$. The only $f_1 : \mathcal{I} \to \mathcal{P}^*(\mathcal{I})$ takes a job and returns a bag of n individuals where only the genotype is set, according to the parameters of the input job. In EC terms, f_1 represents the population

initialization procedure. We denote by IF[n] an IndividualFactory operator with its parameter.

FitnessEvaluator. This is a specialized FM[f_1] with one input stream of individuals $S_{I,\text{in}}$ and one output stream of individuals $S_{I,\text{out}}$. The only $f_1 : \mathcal{I} \to \mathcal{P}^*(\mathcal{I})$ "fills" the phenotype and fitness of the individual, if they are $\varnothing$. Note that, f_1 always outputs bags of one element, i.e., $\forall i \in \mathcal{I}, |f_1(i)| = 1$. We denote by FE a FitnessEvaluator operator.

GeneticOperator. This is a specialized FM[f_1] with one input stream of bags of individuals $S_{I^*,\text{in}}$ and one output stream of individuals $S_{I,\text{out}}$. The only $f_1 : \mathcal{I} \to \mathcal{P}^*(\mathcal{I})$ takes the input individuals and applies a genetic operator $o : G^* \to G$ to their genotypes: the resulting genotype g is set as the genotype of the output individual with $\phi(g)$ as phenotype and $\varnothing$ as fitness. As for FE, here f_1 always outputs bags of one element. We denote by GO[o] a GeneticOperator with its parameter.

Selector. This is a specialized A[$f_{\text{key}}, f_{\text{out}}, 1, 1$] (i.e., with $w_s = w_a = 1$) with one input stream of individuals $S_{I,\text{in}}$ and one output stream of bags of individuals $S_{I^*,\text{out}}$. The key function f_{key} returns the `jobId` of the individual, hence instances contain individuals of the same iteration and of the same job—this is the default behavior for the Selector operator; later in the paper, we explore different alternatives. The output function f_{out} works as follows: let $\Sigma : \mathcal{P}^*(V(A_\tau(S_I))) \to \mathcal{P}^*(V(A_\tau(S_I)))$ be a stochastic function that takes a bag of individuals and returns a subbag of those individuals, then, given an instance I, f_{out} applies Σ to I for n_{sel} times, hence obtaining n_{sel} bags. We denote by S[Σ, n_{sel}] a Selector with its parameters.

IndividualWrapper. This is a specialized FM[f_1] with one input stream of individuals $S_{I,\text{in}}$ and one output stream of bags of individuals $S_{I^*,\text{out}}$. The only $f_1 : \mathcal{I} \to \mathcal{P}^*(\mathcal{P}^*(\mathcal{I}))$ takes an individual and returns a bag containing a one-element-bag containing that individual. Hence, this operator "wraps" input individuals in bags. We denote by IW a IndividualWrapper.

IndividualUnwrapper. This is a specialized FM[f_1] with one input stream of bags of individuals $S_{I^*,\text{in}}$ and one output stream of individuals $S_{I,\text{out}}$. The only $f_1 : \mathcal{P}^*(\mathcal{I}) \to \mathcal{P}^*(\mathcal{I})$ is the identity. Hence, this operator "unwraps" an input bag of individuals to its elements, which are sent on the output stream. We denote by IU a IndividualUnwrapper.

3.1 Example: genetic algorithm (GA) query

We here show an example of an EA expressed as a query.

We consider the case of a rather standard GA, mostly agnostic with respect to the solution representation ϕ—in Section 4 we discuss the experiments we performed with a bitstring representation and a tree-based representation for mathematical formulae.

This EA works as follows. Initially, it builds a population of n_{pop} individuals, according to a representation-specific procedure. Then it iterates the following

steps. First, it builds an offspring of n_{pop} individuals by generating $0.8n_{\text{pop}}$ individuals with crossover followed by mutation and the remaining $0.2n_{\text{pop}}$ with just mutation—in both cases, it selects parents with tournament selection. Then it merges the parents with the offspring and selects the best n_{pop} individuals that will constitute the population at the next iteration. The EA keeps iterating for n_{iter} times. Figure 1 shows the query corresponding to this EA.

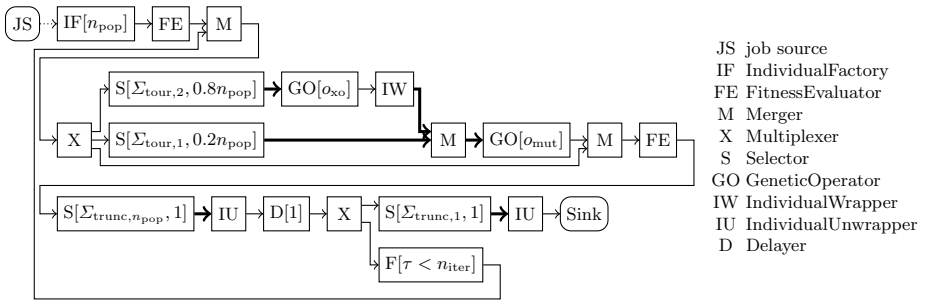

Fig. 1: The query for a standard GA. Arrow types indicate the stream types: dotted ⇢ for streams of jobs S_J, solid → for streams of individuals S_I, solid thick ➔ for for streams of bags of individuals S_{I^*}.

$\Sigma_{\text{tour},n}$ represents tournament selection: given a bag of individuals, it repeats n times the following steps: it first selects n_{tour} individuals—n_{tour} being a parameter of tournament selection—from the bag (randomly with repetition), then it selects the best individual in the subbag; the output is a bag of n individuals, consistently with the parameters needed by the $\text{S}[\Sigma, n_{\text{sel}}]$ operator. Note that in the query for this EA $\Sigma_{\text{tour},n}$ is used two times, once for generating $n_{\text{sel}} = 0.8n_{\text{pop}}$ bags of $n = 2$ individuals (that will be the parents of a new individual built with crossover followed by mutation), once for generating $n_{\text{sel}} = 0.2n_{\text{pop}}$ bags of $n = 1$ individual (that will be the parent of a new individual built with mutation).

$\Sigma_{\text{trunc},n}$ represents truncation selection: given a bag of individuals, it returns the n best individuals in the subbag—$\Sigma_{\text{trunc},n}$, differently from $\Sigma_{\text{tour},n}$, is hence deterministic. There are two operators in the query based on this selection function: $\text{S}[\Sigma_{\text{trunc},n_{\text{pop}}}, 1]$ takes input individuals of one iteration (recall that S is an Aggregate with $w_s = w_a = 1$ and that τ is the iteration number) and outputs one bag of the n_{pop} best ones. $\text{S}[\Sigma_{\text{trunc},1}, 1]$ just outputs the best individual of each iteration: this one is then unwrapped and sent to the sink.

The operators D[1] and $\text{F}[\tau < n_{\text{iter}}]$ govern the iterations of the EA. The former increases the iteration number; the latter stops sending back individuals to the first Multiplexer when n_{iter} iterations occurred: that is, it acts as a termination criterion.

The job source JS and the Sink represent the "start" and "end" of the evolutionary optimzation. Namely, we assume the source emits one job tuple upon some user action as, e.g., the submission of an optimization task to the SPE running the query representing the EA. On the other end, the arrival of one individual (which is the best individual at each iteration) to the Sink might trigger

the storing or logging of the individual (i.e., the solution and its fitness) for later analysis.

3.2 Example: random walk (RW) query

Here we show an example of a query corresponding to another, much simpler EA. This EA is a form of RW where the population is constituted by one single individual.

In detail, RW works as follows. Initially, it builds the first individual according to a representation-specific procedure. Then, at each iteration, it mutates the individual, used as parent, and compares the obtained offspring against the parent. If the offspring is better than the parent, it keeps it as the parent for the next iteration; otherwise, it keeps the parent. The EA keeps iterating for n_{iter} times. Figure 2 shows the query corresponding to RW.

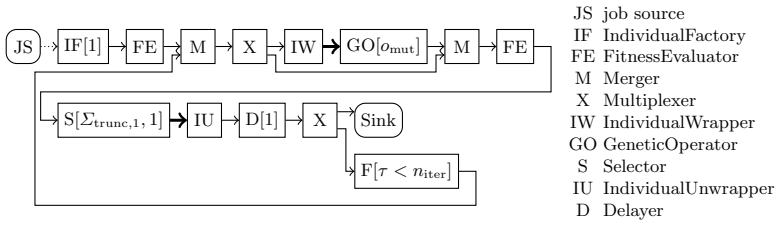

Fig. 2: The query for RW. Arrow types indicate the stream types, as in Figure 1.

3.3 Streaming-based implementation details

For ease of exposition, Sections 3.1 and 3.2 discuss the steps performed by the operators while processing the individuals of a single job. One of the key advantages of stream processing, though, is the possibility of leveraging task/operator pipelining and data parallelism while processing the individuals of one or more jobs in a concurrent, parallel, and distributed fashion. In this section, we provide further insights about how stream processing achieves this while efficiently handling the tuples flowing through operators.

Watermarks and result production. In Section 2.2, we stated that the A operator (i.e., the stateful operator specialized by the Selector operator S—see Section 3) produces a result for $I_{k,e}$ upon the reception of a tuple $t_{\text{in}} \in S_{\text{in}}$ so that $ew_a + w_s < v(\tau, t_{\text{in}})$ and $f_{\text{key}}(v(A_\tau(S_{\text{in}}), t_{\text{in}})) = k$.

In the examples from Sections 3.1 and 3.2, we note all the individuals of a job belonging to the first iteration are eventually fed to the S operator. For such individuals to be fed to f_{out}, nonetheless, S needs to receive at least a tuple with the timestamp and key required to trigger the invocation of f_{out}. Under the hood, such triggering is based on special tuples called *watermarks* [12] that only carry a timestamp and a key. In SPEs, watermarks are automatically generated, forwarded, and processed by operators to trigger results production while correctly enforcing operators' semantics.

Concurrent, parallel, and distributed EA job execution. Two important aspects must be taken into account when multiple EA jobs are carried out at the same time by a given query. First, due to the asynchronous analysis of streaming operators, individuals of one or more jobs could be processed at different paces, thus advancing their iterations at different rates. Second, depending on the f_{key} used by the S operator, two bags of individuals from the same or different jobs that are not jointly processed at the i-th iteration could later be expected to be jointly processed at the j-th iteration, with $j > i$ (this holds true also for their offspring individuals).

Accounting for the first aspect alone, the watermarks described in the previous section are sufficient to enforce correct semantics even when intra- or inter-job iterations advance at different paces, since the watermark triggering the invocation of f_{out} on such individuals is received after all such individuals. To handle the second aspect, though, watermarks from different keys need to be merged, using only the minimum of the latest received values [12] as a trigger for result production. By doing this, being P_i and P'_i two bags of individuals associated to two different keys k, k' at the i-th iteration, and so that individuals in P_i and P'_i (or their offspring) are to be jointly processed in bag P_j at the j-th iteration, with $j > i$, the watermark forwarded after invoking f_{out} on P_i cannot prematurely trigger the invocation of f_{out} on P_j before P'_i individuals/offspring are added to P_j (and vice-versa). This mechanism is transparently handled by SPEs, as we also exemplify in Section 4.3.

Query optimizations. SPEs usually compile the queries defined by users (referred to as *logical*) into *physical* queries by automatically applying optimizations such as operator *chaining* and parallelization to boost performance while preserving the semantics of the logical query [10, 25]. Such a conversion is also applied to the queries considered in this work. During such conversion, UW operators are chained with their upstream peers. An S followed by a UW, for instance, performs directly UW's unwrapping upon the production of a bag of individuals, thus avoiding unnecessary tuple communication costs between S and UW. IW wrapping, moreover, is transparently handled by the stream connecting IW to its preceding/subsequent operator, avoiding also in this case extra communication overheads to/from the IW operator itself.

4 Experimental evaluation

Our experiments aim at answering the following research questions: (RQ1) does an EA implemented as an SP query deliver the same search effectiveness of its "classic" implementation? (RQ2) does an EA query scale, in terms of search efficiency, with the degree of parallelism available to the SPE? (RQ3) is it possible to express new EAs conveniently in the form of queries?

We performed the experiments considering the EAs presented in Sections 3.1 and 3.2, each one tailored to two different representations and used to solve two different problems. Unless otherwise specified, we set $n_{\text{pop}} = 100$, $n_{\text{tour}} = 5$, and $n_{\text{iter}} = 100$ for GA and $n_{\text{iter}} = 10\,000$ for RW. Note that this way, both EAs

generate 10 000 new individuals for each job. For easing the comprehension, in the following we present the results of the experiments in terms of the number n_{evals} of fitness function evaluations, rather than of n_{iter}.

Problems and representations. Concerning the problems, we considered two cases. First, we considered the classic one-max (OM) problem, in which the goal is to find an ℓ-long bitstring of ones: we took $\ell \in \{100, 1000\}$. The fitness function q gives the rate of bits in the string set to zero, to be minimized. For OM, $G = P = \{\mathtt{0}, \mathtt{1}\}^\ell$, i.e., genotypes and phenotypes are bitstrings and the mapping function ϕ is the identity. We used as genetic operator o_{mut} the bitflip mutation and as o_{xo} the uniform crossover followed by a bitflip mutation, both mutations with probability 0.01. We recall that RW uses only o_{mut}, while GA uses both genetic operators. Finally, when building the initial population for GA and the initial genotype for RW, we simply sampled ℓ bits for each genotype with uniform probability.

Second, we considered symbolic regression (SR), in which the goal is to find a mathematical expression which minimizes the prediction error on a dataset $\{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i=1}^{i=n}$, with $\boldsymbol{x} \in \mathbb{R}^p$ and $y \in \mathbb{R}$. In particular, we considered the datasets corresponding to three popular benchmarks [36]: *Keijzer-6*, where $y = \sum_{j=1}^{j=\lfloor x_1 \rfloor} \frac{1}{j}$ and the dataset contains $n = 50$ observations with x_1 evenly spaced in $[1, 50]$; *Nguyen-7*, where $y = \ln(x_1{+}1){+}\ln(x_1^2{+}1)$ and the dataset contains $n = 20$ points with x_1 randomly distributed in $[0, 2]$; *Pagie-1*, where $y = \frac{1}{1+x_1^{-4}} + \frac{1}{1+x_2^{-4}}$ and the dataset contains $n = 625$ points with both x_1 and x_2 evenly spaced in $[-5, 5]$. In the three cases, the fitness function q is given by the mean squared error (MSE) of a candidate solution on the dataset—we remark that we did not use linear scaling while evaluating individuals [35]. For SR we adopted a tree-based representation for the individuals—when coupled with GA, they correspond to a standard form of genetic programming (GP). In detail, the genotype space G is the set of all the trees with a depth in $[3, 8]$ in which the non-terminal nodes are $\bullet + \bullet$, $\bullet - \bullet$, $\bullet \times \bullet$, $\bullet /^* \bullet$, or $\ln^* \bullet$ (where $\bullet$ represents child nodes and $/^*$, $\ln^*$ are the protected versions of the corresponding operations) and terminal nodes are the problem independent variables x_i or the constants 0.1, 1, 10. The phenotype space P is the set of mathematical expressions corresponding to the trees in G. With this representation, we used the standard tree mutation and standard tree crossover as genetic operators.

Implementation and baseline. We implemented the queries corresponding to GA and RW (and all the stream operators adopted by them) using Flink 1.15.2 [7], a popular and well-established SPE also offered by Cloud providers such as AWS. For the experiment related to (RQ1), we used JGEA [21] as the classic implementation of the two EAs. We remark that both implementations are based on Java.

We run the experiments on an Intel Xeon E5-2637 v4 @ 3.50 GHz (4 cores, 8 threads) server with 64 GB of RAM with Ubuntu 18.04. In general, the two implementations exhibited similar performance in terms of running time: however, we remark that a thorough comparison of the computational efficiency of JGEA against our prototypical EA queries was not a goal of this study.

4.1 (RQ1): equivalence of search effectiveness

We performed 30 jobs, i.e., evolutionary runs, for each combination of problem, EA, and implementation (i.e., query or classic). We report the results in Figures 3 and 4 for OM and in Figures 5 and 6 for SR. Namely, Figures 3 and 5 show the fitness $q(p^*)$ of the best individual during the evolution, while Figures 4 and 6 detail the distribution of the best fitness at two stages of the evolution, at $n_{\text{evals}} = 1000$ and at $n_{\text{evals}} = 10\,000$, i.e., at the end of the evolution.

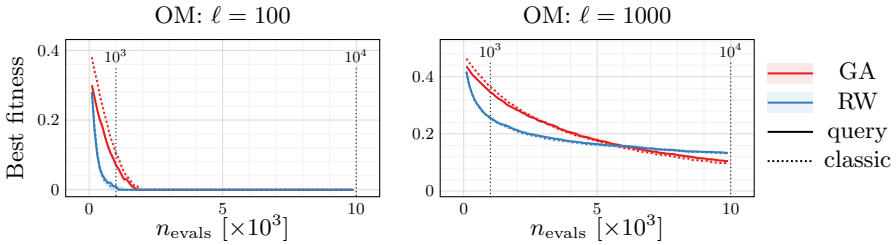

Fig. 3: Best fitness (median and interq. range) during the evolution.

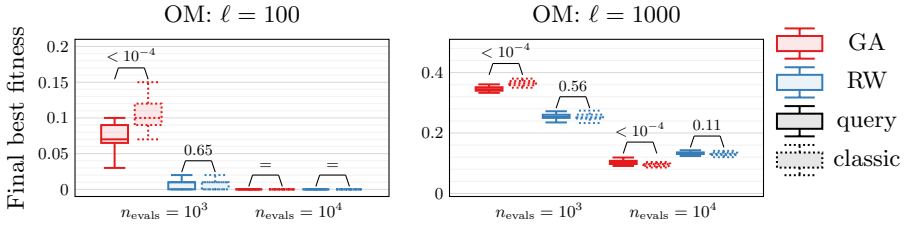

Fig. 4: Distribution of the best fitness $n_{\text{evals}} = 10^3$ and 10^4. Over the pairs of boxplots, p-value of the Wilcoxon test: = means that all the samples are 0.

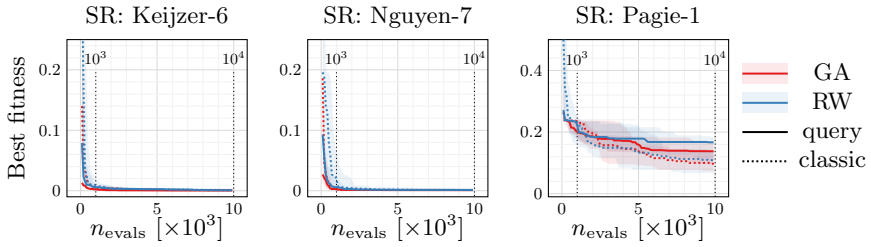

Fig. 5: Best fitness (median and interq. range) during the evolution.

By observing the figures, we note that the general trend of the lines (for Figures 3 and 5) is similar for the two implementations, regardless of the problem and of the EA. This finding suggests that an EA implemented as a query is

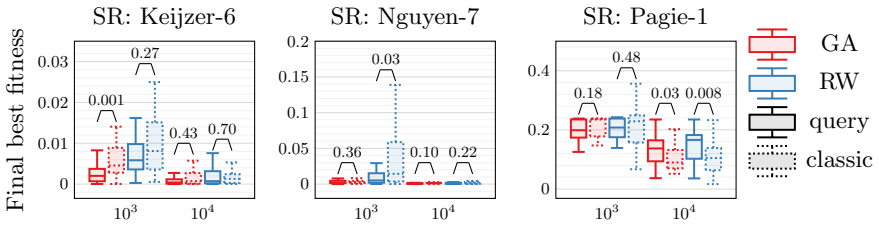

Fig. 6: Distr. of the best fitness at $n_{\text{evals}} = 10^3$ and 10^4; p-values as in Figure 4.

"functionally" similar to its counterpart implemented in a classic way: solutions found with a query are as good as those found with a classic implementation.

For each combination of problem and EA, we performed a statistical significance test (Wilcoxon signed rank, after having verified the proper hypotheses) with the null hypothesis of equality of the means of the best fitness, at the two stages of the evolution corresponding to $n_{\text{evals}} = 1000$ and $n_{\text{evals}} = 10\,000$. Figures 4 and 6 report the p-values (and the underlying distributions, in form of boxplots): in most of the cases the differences are not statistically significant.

4.2 (RQ2): scalability

SPEs are highly optimized to exploit parallelism. We wanted to verify that this capability holds also when the SPE is executing a query corresponding to an EA. For this experiment, we considered the Pagie-1 problem and GA, because they correspond to the most computationally intensive combination. We submitted to the query a number n_{jobs} of concurrent jobs (with different random seeds), with $n_{\text{jobs}} \in \{1, 3, \ldots, 29\}$, and we measured the average completion time (wall time) for each job. We repeated the experiment four times setting Flink parallelism to 1, 2, 4, or 8 (with 4 and 8 being the parallelism degrees for which all cores are used). Figure 7 presents the salient results of this experiment, i.e., the average wall time vs. n_{jobs} for different parallelism degrees.

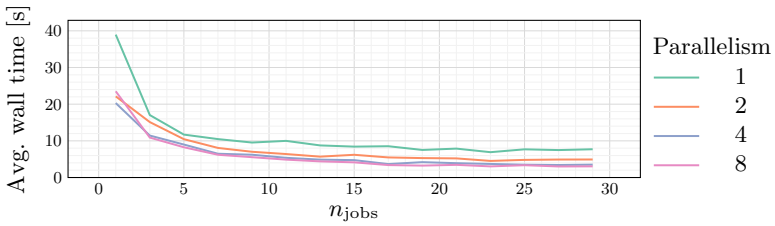

Fig. 7: Average job completion time (wall time) when performing n_{jobs} concurrent jobs using different degrees of parallelism.

It can be seen that, provided a large enough number of jobs (here, ≈ 10) is submitted to the query, the SPE can keep the average wall time constant. Also, by comparing the different lines, one can observe that the greater the parallelism, the lower the average time: namely, it corresponds to 9.5 s, 7.0 s, 6.2 s, and 5.5 s

for 1, 2, 4, and 8 parallelism, respectively, for $n_{\text{jobs}} = 9$ and to 7.7 s, 4.9 s, 3.5 s, and 3.0 s for $n_{\text{jobs}} = 29$. Note that, the gains of higher parallelism degrees are less pronounced as Flink's parallelization grows, especially once 5 or more jobs are run in parallel. This is expected since the 4 available cores are fully utilized for a parallelism degree higher than 4 or when 5 or more jobs run in parallel.

4.3 (RQ3): expressive power of query-based EAs

Quantifying the expressive power of queries for EAs, i.e., "counting" how many sound EAs can be expressed as queries is an hard task which is beyond the scope of this study. Nevertheless, we attempted to show concretely that a practitioner would easily be capable of modifying an EA by acting on the query.

Beside trivial modifications involving single operators (e.g., changing the selection function Σ of the S operators in Figure 1 from tournament to roulette wheel), and simple modifications of the query (e.g., removing the "lowest" path exiting from the first X operator in Figure 1, hence making the generational model without overlapping), we considered a modification showcasing the ease for rich customization enable by SP-based EA queries. In particular, we put ourselves in the perspective of a user who does not know how to set some hyperparameter of an EA for a given problem, considering GA applied to Pagie-1 and the set C of constants defining the genotype space of trees as hyperparameter.

A straightforward approach would be to consider n_p candidates for C and then run n_p jobs. Eventually, one candidate would turn out to be the best one, i.e., the one delivering the best solution. However, this approach would be computationally heavy, because all jobs would be executed entirely, including those corresponding to bad values of C. A smarter approach is to start n_p jobs in parallel with n_p candidate values for C and to merge them after a number n_{merge} of *bootstrap* iterations, on the assumption that in the merged population the individuals built with the best C will likely score better.

While merging jobs is not straightforwardly supported in a traditional EC software (e.g., in JGEA it would imply a rewriting of the full GA), it is trivial in the context of a streaming query, since all individuals (from any number of jobs) are being processed within the same instance of the query. More concretely, this can be achieved by customizing the f_{key} of the S operator, returning the same key for different jobs after the given number of bootstrap iterations.

We realized this modification and experimentally compared it against the expensive approach where all the candidate values are tested entirely—we remark, however, that our goal was not to propose a novel and effective meta-EA, but to show that realizing it is easy if it is an SP-based EA. Namely, we tested two cases: one ("at once") in which we started n_p jobs and merged all the jobs in a single job after n_{merge} iterations and one ("continuous") in which, starting at the n_{merge}-th iteration, we merged two random jobs at each generation until obtaining one single job.

Figure 8 shows the results of the comparison of this meta-EA against the trivial approach ("baseline") where n_p jobs are executed all together and entirely,

all terminating after 100 iterations. We executed 10 times each of the five meta-EA and reported the best fitness across all their n_p. We set $n_p = 30$ and built the 30 candidates for C by randomly sampling 3 values in $[0, 10]$ for each one.

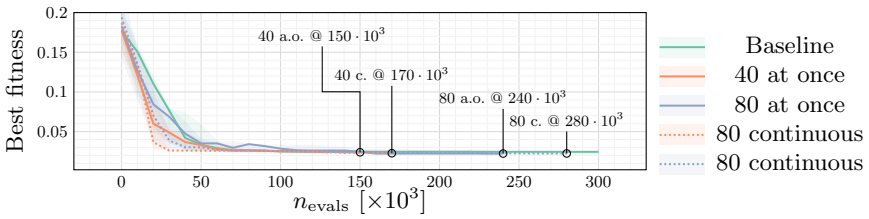

Fig. 8: Best fitness (median and interq. range) during the evolution w meta-EAs.

By observing Figure 8, we see that the experiment highlights the differences between meta-EAs. In particular, all the variants of the smart approach are cheaper in terms of the overall number of fitness evaluations. Moreover, they also appear to be faster in convergence.

5 Concluding remarks

We proposed a formal framework based on stream processing (SP) for evolutionary algorithms (EAs) in the context of evolutionary computation (EC). We implemented two EAs in the form of SP queries, i.e., graphs of SP operators, and experimentally showed they are as effective as their non-SP counterparts, i.e., the EAs implemented in a traditional way. Also, we showed that using our framework, one may easily define new EAs or modify existing ones to explore new algorithmic possibilities or to streamline complex experimental procedures.

We believe our work may indicate a path towards a democratized EC. SP, a paradigm widely used in the IoT-to-Cloud continuum and offered as a service by, e.g., AWS, may enable the effective use of EAs beyond the communities of researchers and of specialists of specific implementations, without, remarkably, diminishing the wide applicability of EC to very different domains. Moreover, we think that by leveraging the existing literature aimed at characterizing [13] and explaining [26] the execution of SP queries, EC might stand out, in the broad family of artificial intelligence (AI), in consistency and explainability [32].

Acknowledgements

This study was carried out within the PNRR research activities of the consortium iNEST (Interconnected North-Est Innovation Ecosystem) funded by the European Union Next-GenerationEU (Piano Nazionale di Ripresa e Resilienza (PNRR) - Missione 4 Componente 2, Investimento 1.5 – D.D. 1058 23/06/2022, ECS_00000043), and by the Marie Skłodowska-Curie Doctoral Network project RELAX-DN, funded by the European Union under Horizon Europe 2021-2027 Framework Programme Grant Agreement number 101072456.

References

[1] Alba, E., Luque, G., Nesmachnow, S.: Parallel metaheuristics: recent advances and new trends. International Transactions in Operational Research **20**(1), 1–48 (2013)

[2] Bartoli, A., Manzoni, L., Medvet, E.: Commentary on "Jaws 30", by W. B. Langdon. Genetic Programming and Evolvable Machines (2023)

[3] Carbone, P., Fragkoulis, M., Kalavri, V., Katsifodimos, A.: Beyond analytics: The evolution of stream processing systems. In: Proceedings of the 2020 ACM SIGMOD international conference on Management of data, pp. 2651–2658 (2020)

[4] Cardellini, V., Lo Presti, F., Nardelli, M., Russo, G.R.: Runtime adaptation of data stream processing systems: The state of the art. ACM Computing Surveys **54**(11s), 1–36 (2022)

[5] De Lorenzo, A., Bartoli, A., Castelli, M., Medvet, E., Xue, B.: Genetic programming in the twenty-first century: a bibliometric and content-based analysis from both sides of the fence. Genetic Programming and Evolvable Machines **21**, 181–204 (2020)

[6] Duvignau, R., Gulisano, V., Papatriantafilou, M., Savic, V.: Streaming piecewise linear approximation for efficient data management in edge computing. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (2019)

[7] flink: Apache Flink. `https://flink.apache.org` (2023), accessed: 2023-1-27

[8] flinkml: Apache Flink ML Documentation. `https://nightlies.apache.org/flink/flink-ml-docs-stable/` (2023), accessed: 2023-11-14

[9] Fortin, F.A., De Rainville, F.M., Gardner, M.A.G., Parizeau, M., Gagné, C.: Deap: Evolutionary algorithms made easy. The Journal of Machine Learning Research **13**(1), 2171–2175 (2012)

[10] Frasca, F., Gulisano, V., Mencagli, G., Palyvos-Giannas, D., Torquati, M.: Accelerating stream processing queries with congestion-aware scheduling and real-time linux threads. In: Proceedings of the 20th ACM International Conference on Computing Frontiers, pp. 144–153 (2023)

[11] Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: An elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems **23**(12), 2351–2365 (2012)

[12] Gulisano, V., Palyvos-Giannas, D., Havers, B., Papatriantafilou, M.: The role of event-time order in data streaming analysis. In: Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems, p. 214–217, DEBS '20, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450380287, https://doi.org/10.1145/3401025.3404088, URL `https://doi.org/10.1145/3401025.3404088`

[13] Gulisano, V., Papadopoulos, A.V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: Performance modeling of stream joins. In: Proceedings of

the 11th ACM International Conference on Distributed and Event-based Systems, pp. 191–202 (2017)
[14] Harada, T., Alba, E.: Parallel genetic algorithms: a useful survey. ACM Computing Surveys (CSUR) **53**(4), 1–39 (2020)
[15] Havers, B., Duvignau, R., Najdataei, H., Gulisano, V., Koppisetty, A.C., Papatriantafilou, M.: Driven: a framework for efficient data retrieval and clustering in vehicular networks. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1850–1861, IEEE (2019)
[16] Hummer, W., Satzger, B., Dustdar, S.: Elastic stream processing in the cloud. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **3**(5), 333–345 (2013)
[17] Isah, H., Abughofa, T., Mahfuz, S., Ajerla, D., Zulkernine, F., Khan, S.: A survey of distributed data stream processing frameworks. IEEE Access **7**, 154300–154316 (2019)
[18] La Cava, W., Orzechowski, P., Burlacu, B., de França, F.O., Virgolin, M., Jin, Y., Kommenda, M., Moore, J.H.: Contemporary symbolic regression methods and their relative performance. arXiv preprint arXiv:2107.14351 (2021)
[19] Maitre, O., Baumes, L.A., Lachiche, N., Corma, A., Collet, P.: Coarse grain parallelization of evolutionary algorithms on gpgpu cards with easea. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 1403–1410 (2009)
[20] Medvet, E., Bartoli, A., Carminati, B., Ferrari, E.: Evolutionary inference of attribute-based access control policies. In: Evolutionary Multi-Criterion Optimization: 8th International Conference, EMO 2015, Guimarães, Portugal, March 29–April 1, 2015. Proceedings, Part I 8, pp. 351–365, Springer (2015)
[21] Medvet, E., Nadizar, G., Manzoni, L.: JGEA: a Modular Java Framework for Experimenting with Evolutionary Computation. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, p. 2009–2018 (2022)
[22] Najdataei, H., Gulisano, V., Tsigas, P., Papatriantafilou, M.: pi-lisco: parallel and incremental stream-based point-cloud clustering. In: Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, pp. 460–469 (2022)
[23] Najdataei, H., Nikolakopoulos, Y., Gulisano, V., Papatriantafilou, M.: Continuous and parallel lidar point-cloud clustering. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 671–684, IEEE (2018)
[24] Palyvos-Giannas, D., Havers, B., Papatriantafilou, M., Gulisano, V.: Ananke: a streaming framework for live forward provenance. Proceedings of the VLDB Endowment **14**(3), 391–403 (2020)
[25] Palyvos-Giannas, D., Mencagli, G., Papatriantafilou, M., Gulisano, V.: Lachesis: a middleware for customizing os scheduling of stream processing queries. In: Proceedings of the 22nd International Middleware Conference, pp. 365–378 (2021)

[26] Palyvos-Giannas, D., Tzompanaki, K., Papatriantafilou, M., Gulisano, V.: Erebus: explaining the outputs of data streaming queries. In: Very Large Data Base, vol. 16, pp. 230–242 (2023)
[27] Pigozzi, F., Medvet, E.: Evolving modularity in soft robots through an embodied and self-organizing neural controller. Artificial life **28**(3), 322–347 (2022)
[28] Pospichal, P., Jaros, J., Schwarz, J.: Parallel genetic algorithm on the cuda architecture. In: European conference on the applications of evolutionary computation, pp. 442–451, Springer (2010)
[29] Rathore, M.M., Son, H., Ahmad, A., Paul, A., Jeon, G.: Real-time big data stream processing using gpu with spark over hadoop ecosystem. International Journal of Parallel Programming **46**, 630–646 (2018)
[30] Röger, H., Mayer, R.: A comprehensive survey on parallelization and elasticity in stream processing. ACM Computing Surveys (CSUR) **52**(2), 1–37 (2019)
[31] Rovito, L., De Lorenzo, A., Manzoni, L.: Evolution of walsh transforms with genetic programming. In: Proceedings of the Companion Conference on Genetic and Evolutionary Computation, pp. 2386–2389 (2023)
[32] Rudin, C.: Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nature Machine Intelligence **1**(5), 206–215 (2019)
[33] Russo, G.R., Cardellini, V., Presti, F.L.: Reinforcement learning based policies for elastic stream processing on heterogeneous resources. In: Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems, pp. 31–42 (2019)
[34] Stephens, R.: A survey of stream processing. Acta Informatica **34**, 491–541 (1997)
[35] Virgolin, M., Alderliesten, T., Bosman, P.A.: Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In: Proceedings of the genetic and evolutionary computation conference, pp. 1084–1092 (2019)
[36] White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**, 3–29 (2013)