Island Model in ActoDatA: an actor-based Implementation of a classical Distributed Evolutionary Computation Paradigm

Giuseppe Petrosino
Federico Bergenti
giuseppe.petrosino@studenti.unipr.it
federico.bergenti@unipr.it
Dept. of Mathematical, Physical and Computer Sciences
University of Parma
Parma, Italy

Gianfranco Lombardo
Monica Mordonini
Agostino Poggi
gianfranco.lombardo@unipr.it
monica.mordonini@unipr.it
agostino.poggi@unipr.it
Dept. of Engineering and Architecture
University of Parma
Parma, Italy

Michele Tomaiuolo
Stefano Cagnoni
michele.tomaiuolo@unipr.it
stefano.cagnoni@unipr.it
Dept. of Engineering and Architecture
University of Parma
Parma, Italy

ABSTRACT

In this paper, we make a first assessment of the performance of ActoDatA, a novel actor-based software library for distributed data analysis and machine learning in Java that we have recently developed. To do so we have implemented an evolutionary machine learning application based on a distributed island model. The model implementation is compared to an equivalent implementation in ECJ, a popular general-purpose evolutionary computation library that provides support for distributed computing.
The testbed used for comparing the two distributed versions has been an application of Sub-machine code Genetic Programming to the design of efficient low-resolution binary image classifiers. The results we have obtained show that the ActoDatA implementation is more efficient than the corresponding ECJ implementation.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; • **Computing methodologies** → **Multi-agent systems**; **Genetic programming**.

KEYWORDS

genetic programming, island model, actor model, java

ACM Reference Format:
Giuseppe Petrosino, Federico Bergenti, Gianfranco Lombardo, Monica Mordonini, Agostino Poggi, Michele Tomaiuolo, and Stefano Cagnoni. 2021. Island Model in ActoDatA: an actor-based Implementation of a classical Distributed Evolutionary Computation Paradigm. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3449726.3463210

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463210

1 INTRODUCTION

Parallel and distributed implementations of evolutionary algorithms (EAs) are common in the literature. The intrinsic parallelism of EAs makes their distributed implementation very natural, especially as concerns the fitness function, which can be evaluated independently for each individual in the population. GAs and PSO, where individuals share the same structure and thus the same fitness function can be evaluated on different data instances of the same kind, can take great advantage of GPUs SIMD paradigm [17], which makes such a solution generally preferable to distributing the computation on different machines.

Instead, distributed computing on several homogeneous or heterogeneous nodes is often advantageous when the individuals' evaluation requires the execution of different code, as happens in Genetic Programming (GP), even if single-host environments optimized for such an EA have also been proposed based on the MIMD paradigm [5]. This is also true when the solution to the optimization problem follows a Divide-and-Conquer approach in which different populations independently explore different regions of the search space. This means that each population is evaluated on different fitness cases and no data sharing among islands is required. This is the case, for example, of the island model [21]. In a parallel implementation of the island model, each node of a cluster of connected machines executes the evolution of a specific subpopulation, and migrations of good individuals can occasionally occur by serializing them and sending them in messages between islands.

Thus, most software packages implementing EAs, like ECJ [16] or DEAP [7], support distributed computing either natively or based on external, general-purpose distributed computing environments, such as SPACE, that has been developed and tested on an evolutionary robotics application [11].
In this paper we will focus on ActoDeS [2, 3, 14], a powerful and efficient Java library for actor-based and event-driven software development. Based on ActoDeS, we have recently developed ActoDatA [13], an actor-based Java library for data analysis and machine learning. In this work, we evaluate ActoDatA by using such a library to implement SmcGP-Islands, an application where a distributed, island model-based EA running Sub-machine code genetic programming [18] evolves low-resolution binary images classifiers as described in [4]. It should be noted that the main contribution of this paper is not describing a new island model; our aim is to compare

an implementation of a classical island model using ActoDatA to an equivalent reference implementation based the *IslandExchange* utility from ECJ.
ActoDatA is briefly overviewed in section 2. SmcGP-Islands is described in section 3 and compared against an equivalent implementation created with ECJ in section 4. Finally, the results are summarized in section 5.

2 ACTODATA

The actor model, firstly introduced in [8], is a formalism for concurrency and event-driven programming. The use of actors as a primitive allows the designer of a concurrent system to isolate software entities from one another, avoiding several issues like race conditions and deadlocks. In such systems, the computation unfolds as series of messages exchanged between actors, and reactions of actors to events related to incoming messages. This conceptualization of software components as separated and independent processes makes it possible to distribute systems on several machines. Over the years, several flavours of the actor model were introduced and extensively used. Among the most frequently adopted variants, the one proposed by Agha [1] is considered one of the most successful, probably thanks of its adoption by the popular *Akka* [12] platform.

The model proposed by Agha is also implemented in ActoDeS [2, 3, 14]. In ActoDeS, an actor that receives a message can react in three ways: (*i.*) change its internal state (and consequently potentially influence its following reactions); (*ii.*) create other actors; (*iii.*) sending messages to other actors. This approach allows the system to be modeled as a set of passive lightweight processes, optimized to use shared system resources and processor time only when needed, i.e., when the actor needs to react to a message event. Doing so allows very large numbers of concurrent processes to run at the same time, enabling an easy scaling up. In combination with the asynchronous message-passing model, easy to implement on networks of connected machines, it is also very easy to scale out an actor-based system on multiple machines. ActoDeS achieves this by distributing the actors on multiple interconnected *actor spaces*, which are containers for many actors on the same Java Virtual Machine (JVM) instance. Actors can trasparently communicate with one another by using their respective *Reference*s, which are abstractions that represent potential destinations of messages.

ActoDatA is a novel software library designed to support the development of data analysis (DA) and machine learning (ML) applications with the help of the actor computation model. ActoDatA is built on top of ActoDeS and presents itself as a Java and Kotlin library. It is an attempt to (*i.*) promote actor-based development for DA and ML applications with high parallelism and horizontal scaling potential; (*ii.*) define a software design paradigm to guide the programmer in developing complex DA and ML architectures; (*iii.*) reduce boilerplate code by providing an implementation of the most common features of DA and ML software.

ActoDatA enriches the tools provided by ActoDeS with a set of specialized actor types, namely:
(*i.*) *acquirers* - which acquire data instances from an external source;
(*ii.*) *preprocessors* - which perform some kind of transformation, filtering, or aggregation on the data;
(*iii.*) *reporters* - which submit data instances to the environment in which the application is running;
(*iv.*) *engines* - which perform, possibly in parallel, resource-intensive computation on the data;
(*v.*) *controllers* - which coordinate the activity of many engines;
(*vi.*) *dataset managers* - which manage collections of data instances;
(*vii.*) *master* - a single actor which creates all the actors of the initial structure, starts the application, and manages structural aspects of the application at run-time.
The user is required to create specializations of these actor types (by extending the corrisponding Java classes) in order to customize the behavior of each component of the application. Some common specializations are built in the library, e.g., the *mapper*, *filter*, *reducer* and other preprocessor specializations, allowing users to build useful generic operations on data, most of which are inspired by common concepts of functional-style programming. Fig. 1 shows an example of how these preprocessors act as functors that operate on data messages and can be composed to easily define a computation distributed on ActoDatA actors.

Some actors, namely acquirers, preprocessors, controllers and reporters, can use express protocols to build *data pipelines*, which constitute the main mechanism used by the library to carry data between chains of communicating actors, where each actor performs a specific computation on the data themselves. ActoDatA actors and data pipelines can be used to model applications that can acquire and process high volumes of data with efficient use of the resources provided by computer clusters. Such applications can be characterized by high structural-level complexity and, for this reason, ActoDatA provides an express application programming interface (API) to ease the definition of an application's structure. This API allows the programmer to declaratively define which actors compose the structure, how the actors are connected by means of data pipelines and other communication protocols, and how the actors are distributed on the actor spaces of the application. An example of usage of this API can be found in the listing in Fig. 2. This mechanism facilitates the creation of structures for DA and ML applications. Fig. 3 schematizes a simple example of a typical distributed DA application. However, the paradigm made of ActoDatA actors and pipelines is general enough to support the development of other types of distributed applications, like the one described in the next sections.

ActoDatA provides some other programming facilities to support the developer in the creation of actor-based DA and ML applications. One example is the *promise construct*, that makes it possible to define complex asynchronous algorithms by means of the interleaving of procedural code blocks and suspension points. In such points, the control flow is suspended while the actor awaits for the occurrence of a required external event (for example, a response from another actor). However, note that in such points the actor is free to operate and react to events concerning other control flows and interactions with other actors. In this approach, the user is not required to use any classical multi-threaded synchronization primitive. Conversely, the user builds chains and branches of closures sequentially composed by means of the methods provided by the `Promise` class.

ActoDatA is a modular library, currently composed of three modules. The first is ActoDatA-core, which contains all the main

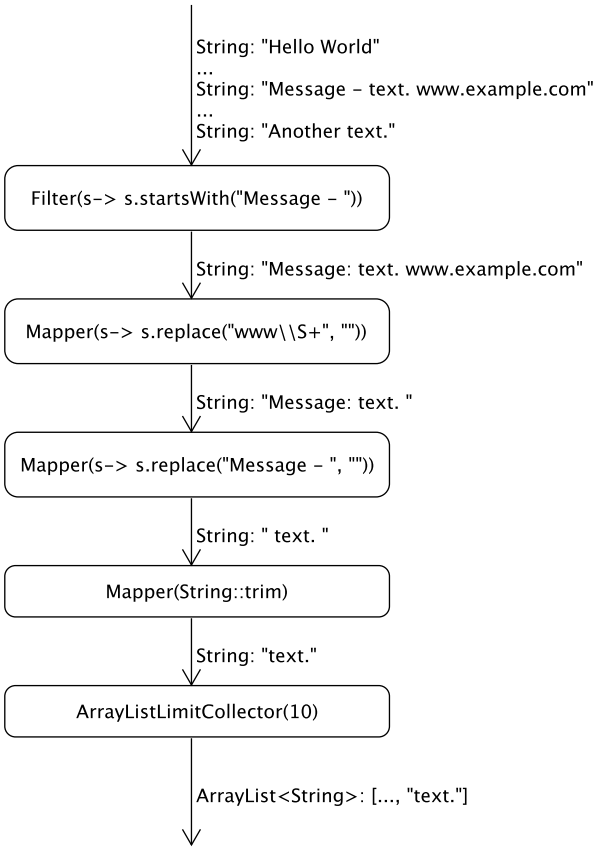

Figure 1: A chain of preprocessors that filters, transforms and collects text data instances into `ArrayLists`. Each rounded rectangle in the diagram represents a different actor, while the arrows between them represent the established data links of the pipeline.

functionalities required for the creation of any ActoDatA application. Another module is ActoDatA-kotlin, which is built on top of the core module to provide additional facilities to take advantage of the many advanced features of the Kotlin programming language. Kotlin was adopted since it is very recently proving to be a valid programming language to support DA and to develop ML applications [19]. Many libraries and tools to perform DA were recently proposed by the community, such as interactive notebooking facilities [10], deep learning [9], numerical analysis libraries, data manipulation and visualization tools.

Finally, an ActoDatA-weka module is under development, and it is meant to allow easy interoperation of ActoDatA with Weka by means of prebuilt specialized actors that integrate Weka classifiers to perform ML tasks.

```
// Creating a new structure.
ActoDataStructure s = new ActoDataStructure();

// Acquirer on the first remote actor space.
var acquirer = s.acquirerNode(() ->
    new TwitterAcquirer(
        ckey, csecret, token, kword
    )
).on((otherSpaces, here)-> otherSpaces.get(0));

// Preprocessor on second remote actor space.
var preproc = s.preprocessorNode(() ->
    new Mapper<>((Status x) -> x.getText()
        .toLowerCase())
).on((otherSpaces, here)-> otherSpaces.get(1));

// Filter on third remote actor space.
var filter = s.preprocessorNode(() ->
    new Filter<String>() {
      @Override
      public boolean test(String x) {
        return !x.isBlank();
      }
    }
).on((otherSpaces, here)-> otherSpaces.get(2));

// Reporter on the master's (here) actor space.
var reporter = s.reporterNode(() ->
    new Reporter<String>() {
      @Override
      public void reportData(String x) {
        System.out.println(x);
      }
    }
).on((otherSpaces, here) -> here);

// Link all the actors in a single pipeline.
acquirer.link(preproc)
    .link(filter)
    .link(reporter);


```

Figure 2: A simple example usage of the ActoDatA Structure API. This listing shows code that creates an acquirer, a preprocessor, a filter preprocessor and a reporter on four different connected actor spaces, and then links them in a pipeline.

3 SMCGP-ISLANDS

ActoDatA can be used as candidate support library to develop many ML applications. To assess its performance in developing distributed evolutionary algorithms, in this work we describe SmcGP-Islands, an application based on ActoDatA in Java. SmcGP-Islands is an implementation of an island model [6] running Sub-machine code Genetic Programming (SmcGP).

SmcGP [18] is a particular genetic programming (GP) paradigm suitable for binary pattern recognition applications. Algorithms that follow this paradigm efficiently execute compute-bound operations

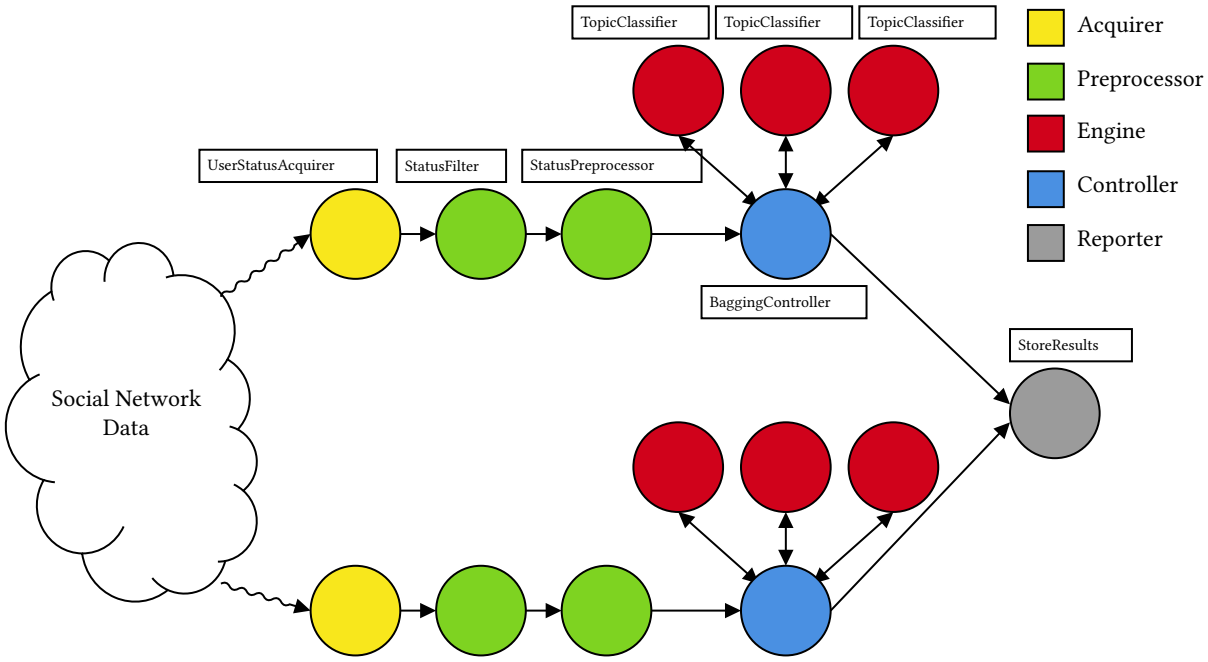

Figure 3: An example of an ActoDatA DA application made up of several data pipelines. In each pipeline, acquirer actors extract data from an external source (e.g., social network public data) and send them to preprocessors, which perform several simple transformations on the data instances. Then, ensembles of controllers and coordinated engines perform some complex processing on the data. The results are then collected by a reporter which produces the final output of the application.

on binary data by parallelizing several single-bit operations into a single CPU instruction. The idea is to take advantage of the single-instruction-multiple-data (SIMD) parallelism available in all modern CPUs. As a matter of fact, using GP to evolve trees using an instruction set including only logic bitwise-operations operating on N-bit words produces N-bit words as outputs. Each of the latter can represent the output of a set of N binary classifiers that could be trained on data instances represented by terminals encoded as N-bit words. Such data could be, for example, stock market information about buy and sell opportunities [20], binary patterns and images, etc. Including shift intructions in the GP instruction set allows GP trees to perform convolution-like operations on neighboring pixels, otherwise the value of each output bit would only depend on the values of the corresponding bits in the input words.

In SmcGP-Island, we replicated the application described in [4] using an island model. In such an application, low-resolution (13x8 pixels) binary patterns, each portraying a numerical digit (from `0` to `9`, see Fig. 4), are packed row-wise into N-bit words that are given as input to GP trees encoding complex logic functions as described above. The dataset have been extracted from rear Italian license-plate images and includes 11034 patterns, divided into a training set of 6024 patterns, almost uniformly distributed among the ten digits, and a 5010-pattern test set including exactly 501 patterns per digit.

The result of each classifier/tree is an N-bit binary value, where each bit can be independently evaluated as a binary classifier for a specific class (e.g., whether the input image contains the digit

Figure 4: Examples of patterns in the data set used to train the binary classifiers, represented in grayscale (top) and as the binary patterns (bottom) used as input to the GP programs.

8 or not). The fittest output bit determines the overall fitness of an individual. To evolve such programs with GP, a particular set of functions and terminals is defined. The function set used in SmcGP-Islands is listed in Table 1.

We implemented two variants of SmcGP-Island. The first was developed with ECJ [16] and its bundled IslandExchange [15] facility. When executed, the IslandExchange's main method launches a server that waits for all the islands to connect to it before starting the execution. Communication occurs on TCP/IP sockets, with the server that acts as communication broker and coordinator. Each island in ECJ is a separate Java process, running in its own JVM instance, and each island is configured to send its migrants to another island, following a static closed-ring topology. The ECJ application is set to run in synchronous mode, in order to comply with

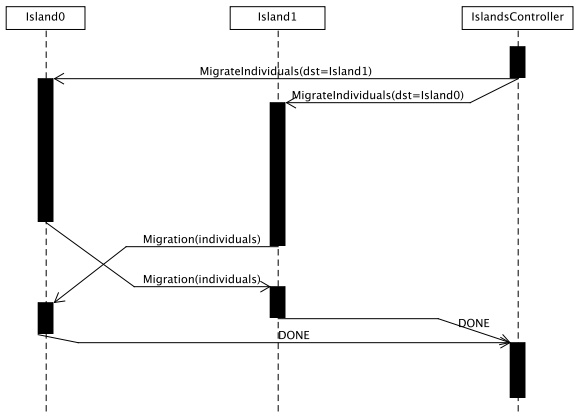

Figure 5: Example of an execution of the migration protocol with a minimal ring of two islands.

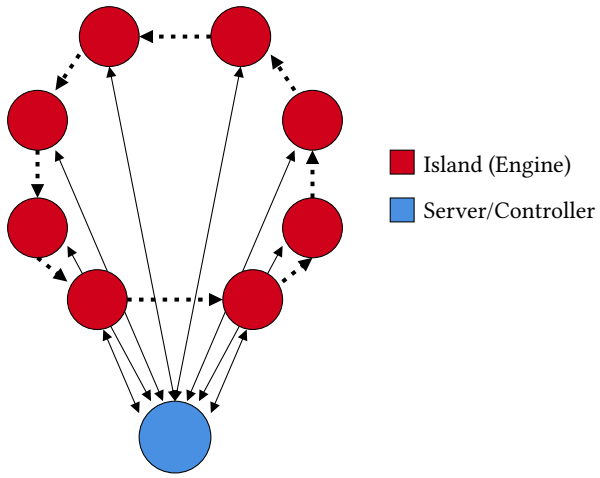

Figure 6: The architecture of SmcGP-Islands, configured as a ring of eight islands. Dotted arrows indicate the direction of migrations. The controller coordinates all the islands by communicating with each of them (double-headed arrows).

the *bulk-synchronous parallel* model. The adoption of this parallel computation model has the disadvantage of creating a significant overhead, with idle islands waiting at each synchronization barrier. However, it was chosen for several reasons:
(*i.*) it ensures that migrations start and end at the same generations in each source and destination island;
(*ii.*) it ensures that the island ring stays the same at each migration;
(*iii.*) the communication channels are stressed by gathering communication events in coincidental time intervals, because migration and synchronization messages are exchanged at the same time between all islands; this makes the application a good benchmark to assess performances under high usage of network resources. Fig. 6 illustrates the architecture of SmcGP-Islands.

Function(s)	Arity	Description
`AND`	2	Bitwise-and (`a & b`)
`OR`	2	Bitwise-or (`a | b`)
`XOR`	2	Bitwise-exclusive-or (`a ^ b`)
`NAND`	2	Bitwise-negated-and (`~(a & b)`)
`NOR`	2	Bitwise-negated-or (`~(a | b)`)
`NOT`	1	Bitwise-not (`~a`)
`SHR1 SHR2 SHR4`	1	Circular right shift by 1, 2, 4
`SHL1 SHL2 SHL4`	1	Circular left shift by 1, 2, 4
`Zero`	0	Terminal constant: `0`
`One`	0	Terminal constant: `1`
`PAT1 PAT2 PAT3 PAT4`	0	Terminal used as input value, containing a portion of the image.
`ERC`	0	*Ephemeral Random Constant*, initialized by picking a random integer value on the set of all possible integers representable by 32 bits.
`ERC16`	0	*Ephemeral Random Constant*, initialized by picking a random integer value between 0 and 15.

Table 1: The function and terminal set used in SmcGP-Islands (using 32-bit words). Terminals have arity equal to zero.

In the second implementation, the IslandExchange facility is replaced by an ActoDatA set of actors. In particular, each island is defined as an ActoDatA engine actor that manages the evolution of a subpopulation, and the activity of the engines is coordinated by an ActoDatA controller actor. The controller commands each island to start each iteration, then waits for each island to complete the iteration by signaling it with a `DONE` message. Every m_{rate} iterations, the controller creates and sends a `MigrateIndividuals` message request to each island, specifying how many individuals that island must send and the islands to which migrants should move, identified by their ActoDeS reference. Islands process in parallel their `MigrateIndividuals` message by selecting a set of migrants of the specified size and encapsulating them into a `Migration` message, which is sent to the destination island. When an island receives a `Migration` message, it adds the individuals in the message to its population and sends a `Done.DONE` message to the controller. As soon as the controller receives a `Done.DONE` message from each island, it goes back to coordinating the usual EA iterations. This migration protocol provides the flexibility required to implement more complex and dynamic topologies and is summarized by the example in the diagram in Fig. 5.

An ActoDatA master actor is used to set up the application. When the application is launched in distributed mode with a number $n \geq 2$ of connected actor spaces, the master creates the controller on a node and distributes all the island engines evenly among the other nodes. Note that every island is an actor, not an independent JVM process. Multiple ActoDatA engine islands can be, of course, launched on the same actor space, running on the same JVM instance. This is one of the core architectural differences between the ActoDatA implementation and the pure ECJ implementation of

the application, which instead executes each island as its own JVM process.

In the ActoDatA implementation, the evolutionary aspects of each single island are managed by ECJ with the same configuration and parameters as in the pure ECJ variant. This means that all the details strictly related to population initialization, fitness evaluation and breeding of each island are the same as in the pure ECJ implementation, while the set up, location, synchronization, and communication details are handled by ActoDatA and its actors.

In both implementations, each island is set to be executed sequentially and not in multithreaded mode, so parallelism can only be exploited by multiple islands running at the same time. This was done to ensure architectural consistency between the two systems, in order to highlight the differences of using ActoDatA instead of IslandExchange. The application is configured to initialize each sub-population of 1000 individuals using ramped half-and-half initialization, with a grow-probability of 0.5, and program depth ranging from 5 to 7. To evolve the population, two ECJ breeding pipelines are used for each island. The crossover probability is set to 0.8. The offspring resulting from crossover cannot exceed a maximum depth of 15. The reproduction pipeline has a likelihood of 0.2 ad follows a tournament selection with a tournament size of 7. Moreover, randomly selected solutions can mutate with a likelihood of 0.03, and generated subtrees can have a minimum depth of 2 and a maximum depth of 5. Finally, migrations occur at the third generation and every three generations thereafter, and each migration trasfers 10 migrants from one island to the other.

To compare the two implementations, the applications were executed on a cluster composed of four nodes connected by Gigabit Ethernet. Each node was equipped with 16 GB of RAM and two quad-core Intel Xeon E5504 CPUs with 2GHz clock frequency. This means that the software was executed in parallel by 8 cores when launched on a single machine, and up to 32 cores when the software was distributed over all the machines. Each machine runs Linux Ubuntu version 20.04 and uses Java 14. Ten runs were launched for each *mode* M, with $M \in N \times L$, where N is the set of possible number of nodes used to run the software in parallel ($N = \{1, 2, 3, 4\}$) and L is the set of implementations whose statistics has been sampled ($L = \{PureECJ, ECJ\&ActoDatA\}$).

For each run, 32 islands were spawned, equally distributed[1] on the $n \in N$ nodes. Because of its minimal computational requirements, the Controller/Server is always executed on one of the nodes also used by the islands, and never on a dedicated node. The *ECJ&ActoDatA* implementation always uses a JVM for the controller and the master, and another JVM per node to host the islands, so there are always $n + 1$ JVM processes running on the cluster. The *Pure ECJ* implementation, instead, requires a distinct process for each of the islands and one for the server, so there are always 33 JVM processes running.

4 CONVERGENCE TIME

Time measurements refer to the convergence speed of the algorithm, i.e., the actual time that the run took to produce at least an individual $I^\star$ with fitness $F_{ind}(I^\star) \leq 0.017$ on any island. The choice of this fitness thershold derives from the results obtained in [4] that indicate that such a fitness is almost always reachable and corresponds to a good accuracy on the test set. It should be noticed that the experiments we performed were not finalized at maximizing the classification results of [4] as we chose such an application just as a benchmark to assess the efficiency of its distributed implementation in the *Pure ECJ* and *ECJ&Actodata* versions. Time measurements are denoted with T_n when the system is executed on n nodes of the cluster.

Average, minimum, maximum and standard deviation values of the execution times, obtained over 10 runs by all configurations, are listed in Table 2.

The results indicated a tendency of the *ECJ&Actodata* implementation to be faster than the *Pure ECJ* implementation, as shown in the graph in Fig. 7. To test the statistical significance of the results, a set of Wilcoxon rank-sum tests have been carried out, with a threshold on the p-value of 0.05 (Table 3). The results of the Wilcoxon tests confirm that when the algorithm is distributed over more than one node, there is a significant difference between the convergence times of the ActoDatA implementation and the pure ECJ implementation.

A possible reason for this might be related to one of the intrinsic advantages of the actor model in ActoDeS and ActoDatA, and the consequent optimization of message dispatch among islands. As a matter of fact, in the ActoDatA-powered version, islands are modeled as actors, and multiple islands in the same actor space coexist on the same machine[2]. When multiple actors communicate within the same actor space, the messages exchanged among them are not serialized. Instead, the references to the Java object representing each message are simply added to the inbox of the recipient actor,

[1]For the $n = 3$ nodes case, two nodes host 11 islands each, and the third one hosts 10 nodes and the Controller/Server.

[2]To take full advantage of the internal parallelism provided by the multicore CPUs in each machine, since each island executes its iterations sequentially.

Implementation	n	Avg. T_n (SD)	Min	Max
Pure ECJ	1	419s (110s)	274s	560s
Pure ECJ	2	261s (46s)	181s	347s
Pure ECJ	3	225s (76s)	74s	361s
Pure ECJ	4	204s (76s)	139s	420s
ECJ & ActoDatA	1	325s (99s)	186s	451s
ECJ & ActoDatA	2	203s (44s)	140s	296s
ECJ & ActoDatA	3	143s (28s)	83s	179s
ECJ & ActoDatA	4	124s (35s)	56s	171s

Table 2: Mean, Standard Deviation, Minimum and Maximum values of execution time (T_n), for each implementation and number of nodes (n) used.

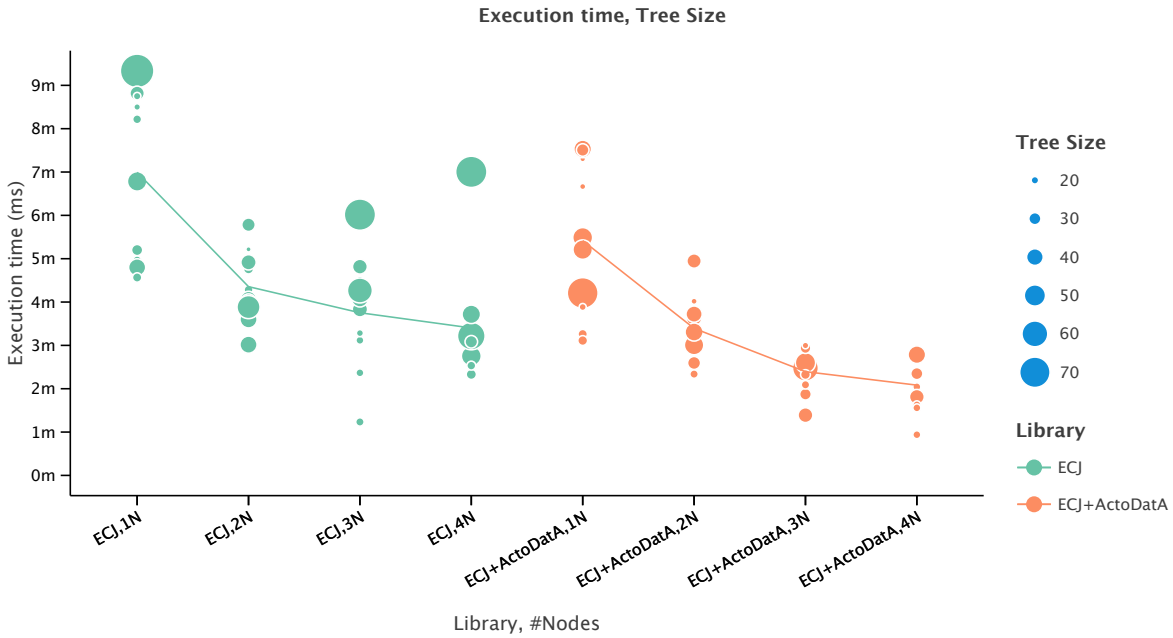

Figure 7: Scatter plot that displays the execution times and the tree size of the best individual of each run. Runs are grouped by implementation type and number of nodes used. The line in each implementation group links the mean time values.

exploiting the advantage given from sharing the same memory heap space. When a migration occurs, only migration messages from islands that do not coexist on the same actor space need to be serialized. This contrasts with the ECJ approach, where each island has to be executed in its own JVM process, and all migration messages have to be serialized to be sent via TCP/IP sockets, and deserialized on the destination JVM process, probably causing more overhead and congestion on the network links. Further investigations are needed to confirm the above hypotheses. The results on the convergence speed measurements also confirm that, with an increasing number n of nodes, the time required to find a solution decreases for both implementations. Table 4 shows the average speedup (speedup is $S_n = T_1/T_n$ where T_i is the execution time with i nodes) and efficiency (efficiency is $E_n = S_n/n$) values for all implementations, with 2, 3 and 4 nodes.

Convergence Time - ECJ & ActoDatA vs ECJ.

- H_0: *With* $n = 1...4$, ECJ & ActoDatA *finds a solution as fast as* ECJ.
- H_1: *With* $n = 1...4$, ECJ & ActoDatA *finds a solution faster than* ECJ.

n	p-value	H_0 rejected?
1	0.232	NO
2	0.037	YES
3	0.027	YES
4	0.014	YES

Table 3: Results of Wilcoxon rank-sum tests, for each number n of nodes considered, in comparing the convergence times of the two implementations.

Implementation	n	Avg S_n	Avg E_n
Pure ECJ	2	1.60	80.2%
Pure ECJ	3	1.86	62.0%
Pure ECJ	4	2.05	51.3%
ECJ & ActoDatA	2	1.60	80.0%
ECJ & ActoDatA	3	2.27	75.8%
ECJ & ActoDatA	4	2.62	65.6%

Table 4: Average speedup and efficiency for parallel executions on $n = 2, 3, 4$ nodes.

5 CONCLUSIONS

We have shown that ActoDatA, described in section 2, even if in its present preliminary version, is a promising software library, enabling the creation of applications for big data analysis and machine learning that leverage the flexibility of the actor model. With ActoDatA, the user can create arbitrarily complex topologies of specialized actors and define the behavior of the application with minimal amount of boilerplate code, both in Java and Kotlin.

We have used ActoDatA to develop the SmcGP-Islands application, which distributes over several nodes the evolution of several sub-machine code genetic programming populations for the classification of low-resolution binary images of numerical digits. Two different implementations of this application were compared. The first implementation uses ActoDatA to handle the distribution and

communication aspects, and the other one uses the specific facilities of the ECJ library, a popular research system for evolutionary computation in Java. Both implementations use internally the same ECJ code to implement the evolutionary algorithm code of each single island, and run until they reach a pre-set fitness threshold. The results showed a significant advantage for ActoDatA in terms of convergence time, speedup and efficiency on the number of nodes used.

Future work will focus more on the quality of results and includes implementing island models with different topologies, as easily permitted by ActoDatA, evaluating different migration and data distribution schemes, and solving other challenging problems which can benefit from a distributed implementation.

REFERENCES

[1] Gul Agha. 1990. Concurrent object-oriented programming. *Commun. ACM* 33, 9 (1990). https://doi.org/10.1145/83880.84528

[2] Giulio Angiani, Paolo Fornacciari, Gianfranco Lombardo, Agostino Poggi, and Michele Tomaiuolo. 2018. Actors based agent modelling and simulation. In *Communications in Computer and Information Science*, Vol. 887. https://doi.org/10.1007/978-3-319-94779-2_38

[3] Federico Bergenti, Eleonora Iotti, Agostino Poggi, and Michele Tomaiuolo. 2016. Concurrent and Distributed Applications with ActoDeS. In *MATEC Web of Conferences*, Vol. 76. https://doi.org/10.1051/matecconf/20167604043

[4] Stefano Cagnoni, Federico Bergenti, Monica Mordonini, and Giovanni Adorni. 2005. Evolving binary classifiers through parallel computation of multiple fitness cases. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 35, 3 (2005), 548–555. https://doi.org/10.1109/TSMCB.2005.846671

[5] Vinícius Veloso de Melo, Álvaro Luiz Fazenda, Léo Françoso Dal Piccol Sotto, and Giovanni Iacca. 2020. A MIMD Interpreter for Genetic Programming. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*. Springer, 645–658.

[6] A. E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing* (2nd ed.). Springer Publishing Company, Incorporated.

[7] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research* 13, 1 (2012), 2171–2175.

[8] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. *Advance Papers of the Conference* (1973).

[9] JetBrains. [n.d.]. *KotlinDL: High-level Deep Learning API in Kotlin.* https://github.com/jetbrains/kotlindl retrieved on Jan 30th, 2021.

[10] JetBrains and KotlinCommunity. [n.d.]. *Kotlin-jupiter: a Kotlin kernel for Jupiter.* https://github.com/Kotlin/kotlin-jupyter retrieved on Jan 30th, 2021.

[11] Guillaume Leclerc, Joshua E. Auerbach, Giovanni Iacca, and Dario Floreano. 2016. The Seamless Peer and Cloud Evolution Framework. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (Denver, Colorado, USA) *(GECCO '16)*. Association for Computing Machinery, New York, NY, USA, 821–828. https://doi.org/10.1145/2908812.2908886

[12] Lightbend. [n.d.]. *Akka Platform by Lightbend.* https://www.lightbend.com/akka-platform

[13] Gianfranco Lombardo, Paolo Fornacciari, Monica Mordonini, Michele Tomaiuolo, and Agostino Poggi. 2019. A multi-agent architecture for data analysis. *Future Internet* 11, 2 (2019). https://doi.org/10.3390/fi11020049

[14] Gianfranco Lombardo and Agostino Poggi. 2020. A preliminary experimentation for large scale epidemic forecasting simulations. In *CEUR Workshop Proceedings*, Vol. 2706.

[15] Sean Luke. 2010. *The ECJ Owner's Manual.*

[16] Sean Luke, Eric O. Scott, Liviu Pnait, Gabriel Balan, et al. [n.d.]. *ECJ 27: A Java-based Evolutionary Computation Research System.* https://cs.gmu.edu/~eclab/projects/ecj/

[17] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. 2011. Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. *Information Sciences* 181, 20 (2011), 4642–4657. https://doi.org/10.1016/j.ins.2010.08.045

[18] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. 2008. *A field guide to genetic programming.* Lulu. com.

[19] Ievgen Sidenko, Galyna Kondratenko, and Valentyn Petrovych. 2020. Neural network technologies for diagnosing heart disease. In *CEUR Workshop Proceedings*, Vol. 2762.

[20] Nils Svangard, Peter Nordin, and Stefan Lloyd. 2002. Parallel Evolution of Trading Strategies Based on Binary Classification Using Sub-Machine-Code Genetic Programming. In *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution And Learning (SEAL'02)*.

[21] Darrell Whitley, Soraya Rana, and Robert B. Heckendorn. 1999. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology* 7, 1 (1999).