Evolutionary Computation with Islands: Extending EvoLP.jl for Parallel Computing

Xavier F. C. Sánchez-Díaz[0000−0003−2271−439X] and Ole Jakob Mengshoel[0000−0003−2666−5310]

Norwegian University of Science and Technology
{xavier.sanchezdz, ole.j.mengshoel}@ntnu.no
https://saxarona.github.io
https://www.ntnu.edu/employees/ole.j.mengshoel

Abstract. The use of evolutionary computation for optimisation is a relevant area of research in many fields of science and the industry, where complex problems are frequently encountered. As an effort to support the research in this niche, we present an extension for EvoLP.jl: the evolutionary computation playground in Julia, that includes three new operators for implementing island models for genetic algorithms. The extension enables the framework to run using the Message Passing Interface protocol, an international standard for communication in parallel architectures that is available in most high performance computing clusters today. We study the advantages of the implementation by performing a series of tests on well-known numerical optimisation benchmarks of various difficulties and on several dimensions. Both the code and the data are available in a GitHub repository. This work enables researchers to implement powerful parallel evolutionary algorithms without moving away from the high level of abstraction that the framework provides.

Keywords: Evolutionary computation · Genetic algorithms · Optimisation · Parallel computing.

1 Introduction

Context. Evolutionary Computation (EC) algorithms are heuristic techniques for optimisation with high relevance in many engineering and statistics applications where non-convex fitness landscapes arise: structural design [34], layout optimisation [22], resource allocation [24], as well as optimising AI models through feature selection [17], neuroevolution [31] and hyperparameter optimisation [6]. EC mimics how individuals and species evolve and adapt to their environments, where those who adapt better will survive and reproduce, and conversely, *unfit* individuals perish. The EvoLP.jl [32] software for EC, implemented in the Julia programming language, provides a friendly platform for designing and experimenting with Evolutionary Algorithms (EA). EvoLP.jl is free, open source, and hosted on the Norwegian Open AI Lab's GitHub repository.[1] EvoLP.jl is de-

[1] See https://github.com/ntnu-ai-lab/EvoLP.jl

signed as an experimentation *playground*, with several "swappable" modules (or *building blocks*) to create custom algorithms in one's own workstation quickly.

Challenges and opportunities. Many industrial problems pose challenging optimisation scenarios that can be solved using EC. The challenges include (i) expensive fitness function evaluations and (ii) multimodal fitness landscapes [2,13,34]. In the multimodal setting, diversity preservation is important, due to the challenges of genetic drift and premature convergence [7]. Parallel computing can potentially tackle both challenges (i) and (ii). First, parallel computing power can handle the expense of fitness function evaluation. Second, certain EC algorithms, including the island model [12,7], map naturally to parallel computers. The island model evolves multiple populations in parallel, typically with one island per computer, and thereby encourages solution diversity.

Contributions. The contributions of this paper are twofold. First, we study uni- and multimodal problems, with an emphasis on the latter. Different EA diversity preservation mechanisms for multimodal problems have been studied in the literature, including crowding [14,16], fitness sharing [8], and the island model [12,7,27]. We study the island model in this paper, and in particular consider its integration into EvoLP.jl. Our second contribution is based on the fact that EAs are inherently parallel. Thus, we are investigating the implementation of the island model on parallel computers. Specifically, we extend EvoLP.jl [32] to work with high-performance computing (HPC) resources. We discuss the extensions and modifications needed in EvoLP.jl to run EAs in parallel architectures using the Message Passing Interface (MPI) protocol. In an empirical study with EvoLP.jl, we compare the parallel island model with a serial variant, with a focus on their different exploration and diversity preservation capabilities.

1.1 An Introduction to the EC Terminology

In this paper, we consider a Genetic Algorithm (GA) in which a *population* of *individuals* (candidate solutions) are evaluated using an objective (or *fitness*) function to be optimised [7,8,29]. Each individual is a *vector* of features. We assume the population to be a *bag of real-valued vectors.* The evolution process is simulated in an iterative manner via four basic operations: *selection*, recombination or *crossover*, *mutation* and *survival.* Selection is a *policy* (also called an operator) that dictates how to select the subset of individuals that will reproduce, based on the fitness of each individual. Crossover is another policy—it generates new vectors by combining two selected individuals. The mutation operator is (usually) a stochastic component that slightly modifies a solution for exploring the fitness landscape. Finally, survival is another selection policy that determines the subset of surviving individuals for the next iteration. For the purpose of this paper, we use a *generational* GA in which the survival policy is total replacement: the old population dies and is entirely replaced by its offspring.

1.2 Mathematical Notation

We use italic bold symbols for individuals ($\boldsymbol{x}$), with their i-th index (or feature) shown as x_i. For sets of individuals, we use blackboard bold font like $\mathbb{P}$ and $\mathbb{M}$, and their cardinality (or their size) is represented with two vertical bars, as $|\mathbb{P}|$. For policies, topologies, algorithms and objective functions we use maths script font like $\mathscr{S}$ and $\mathscr{R}$. We refer to random variables as uppercase letters in serif font, and the distribution they are sampled from is typeset using calligraphic maths font: $X \sim \mathcal{U}(\mathbb{P})$ means that X is a random variable sampled from a uniformly random distribution over the set of individuals $\mathbb{P}$. Finally, we use brackets to delimit a finite range of numbers: define the set of integers $[n] := \{1, \ldots, n\}$ for vector indices. For vector values, we use $[l_b, u_b]$ to denote a closed range between two real numbers l_b and u_b, the lower and upper bounds respectively.

2 Background

2.1 The Island Model for Diversity Preservation

Among different EA diversity preservation mechanisms [8,14,16], we focus on the island model [12,7,27]. In the island model of the GA [7], multiple populations run in parallel and communicate. After a fixed number of generations (known as *migration rate*), selected individuals *migrate* from one population to another. The model resembles species migration from continental areas to remote islands, and how some populations specialise in their own environmental niches (e.g. the Galápagos finches).

In short, with multiple populations evolving at the same time but with some interaction (the migration), there is a *passive* diversity preservation mechanism. To describe how several island populations evolve in parallel and occasionally exchange individuals, we adopt the definitions by Izzo et al. [12]:

Definition 1. *Let an* ***archipelago*** *be a 2-tuple* $\mathbb{A} = (\mathbb{I}, \mathscr{T})$*, where* $\mathbb{I}$ *is a set of* n *islands* I*, and* $\mathscr{T}$ *is the migration topology represented as a directed graph with* $\mathbb{I}$ *as its vertices.*

Definition 2. *Every* ***island*** $I_i \in \mathbb{I}$ *with* $i \in [n]$ *is a 4-tuple* $I_i = (\mathscr{A}_i, P_i, \mathscr{S}_i, \mathscr{R}_i)$ *with the following components:*

- *An algorithm* $\mathscr{A}$ *with an epoch (or migration rate)* μ
- *A population description* $P = (\mathscr{P}, \mathbb{P})$ *where the population* $\mathbb{P}$ *is tied to its objective function* $\mathscr{P}$
- *A migratory selection policy* $\mathscr{S}$ *that determines the subpopulation (known as* ***deme****)* $\mathbb{M} \subseteq \mathbb{P}$ *to be sent to neighbouring islands (wrt* $\mathscr{T}$*)*
- *A migratory replacement policy* $\mathscr{R}$ *that specifies how a received deme* $\mathbb{M}$ *should be inserted into a population* P

The pseudocode describing the communication scheme for each island I_i is presented in Algorithm 1. Considering this formulation, we use the MPI communication protocol for parallel computing architectures [18]. MPI is an international standard provided by many parallel computers including HPC clusters.

Algorithm 1 Pseudocode for each island $I_i \in \mathbb{I}$

```
1: Initialise $P$
2: while termination criterion is not met do
3:     $P' \leftarrow \mathscr{A}(P, \mu_i)$
4:     $\mathbb{M} \leftarrow \mathscr{S}_i(P')$
5:     Send $\mathbb{M}$ to adjacent islands (wrt topology $\mathscr{T}$)
6:     Receive $\mathbb{M}'$ solutions from adjacent islands
7:     $P'' \leftarrow \mathscr{R}(P', \mathbb{M}')$
8:     $P \leftarrow P''$
```

2.2 Parallelism for Evolutionary Computation

Many parallel and distributed models for EC have been thoroughly studied before [11,20,21,28]. Some of these approaches focus on distributing the workload for improved performance using map/reduce calls [19,23]. Other approaches modify the underlying EA instead, in order to make it asynchronous. However, asynchronous EAs suffer from limitations such as being dependent of initialisation [25] or experience performance drops in linkage problem settings [10]. A third approach—most notable due to its ease of modelling—is the island model discussed in Section 2.1.

Early theoretical work estimated the optimal number of populations and processors to use in island models [4]. More recently, it has been shown that with enough processors, archipelagos with homogeneous islands perform similarly to archipelagos with different island populations [9]. An in-depth mathematical description of a *generalised* island model was developed by Izzo et al. [12]. This generalisation is used as the basis for the additional operators described in this work, as discussed in Section 3.

Some EC software packages already include some form of parallelism. For example, for Python, LEAP [5] allows distributed evaluation in both sync- (map/reduce) and asynchronous mode. Pagmo (C++) [1] includes support for distributed evaluation and island models. For Java, ECJ [26] includes support for both sync/async communication, and island models as well. Similarly, an extension to the GA package in R has been developed [27] that includes island model support. In Julia, Evolutionary.jl and Metaheuristics.jl [15] allow paallel evaluation using threads. Our work represents the first effort in parallel modelling for a registered Julia package, with EvoLP.jl [32] being the first package for EC to include support for island models.[2]

3 Implementing Parallel Islands in EvoLP.jl

We now elaborate on how to implement island models using EvoLP.jl [32]. We follow the same paradigm we used for its design: implementing at a high level

[2] EvoLP.jl is registered as a package in the General Julia Registry: it can be installed via the package manager in the Julia REPL.

of abstraction where each step of the evolutionary process is performed by one operator. In EvoLP.jl, this is achieved by first using a `type` to set the desired policy, and then performing the transformations considering such policy by using a function. The available building blocks in EvoLP.jl are:

- **Generator**. A function for randomly initialising the population.
- **Selector**. A policy for selecting parents for recombination using the `select` function.
- **Recombinator**. A policy for performing crossover (between two parents) using the `cross` function.
- **Mutator**. A policy for performing mutation (on an individual) using the `mutate` function.

Using the definitions in Section 2.1, we now describe the necessary operations for performing the communication in an island model using a similar level of abstraction: selecting a `type` policy first and then performing a transformation using a function. For this, we assume that each island is a process, and that we are in control of communication. This is different from other EC libraries. For example, in LEAP [5], islands are simulated. In pagmo [1], the communication is transparent to the user as the archipelago is coded as a single object. In EvoLP.jl we use communication operators as building blocks, more in line with the playground metaphor.

3.1 The Drift, Strand and Reinsert Operators

Using the EvoLP.jl blocks, we can code a GA in a standard way [32], making it run on a single core. To make it an island model, adding the following Julia code at the end of the main loop suffices:

```julia
for i in 1:max_it  # inside main loop
  ...
  if i % mu == 0  # migration time
      # 1. Select and send deme: drift
      _, s_req = drift(MigrateSelect, population, fits, dest)
      # 2. Receive deme: strand
      M, r_req = strand(MigrateSelect, dims, src)
      # 3. Add new deme to population: reinsert!
      worst_idx = reinsert!(population, fits, MigrateReplace, M)
      # 4. Delete old deme
      deleteat!(population, worst_idx)
      deleteat!(fits, worst_idx)
      # 5. Evaluate new deme using function f
      append!(fits, f.(M))
      # 6. Wait
      MPI.Barrier(comm)
  end
end
```

The full example can be consulted in this work's GitHub repository.[3] We now define the communication operators formally, using the previous definitions and thinking of individuals moving between islands in an archipelago.

Drift: an operator representing both the selection and the **send** steps of the communication (lines 4 and 5 in Algorithm 1).

Definition 3. *Given a defined migration selection policy $\mathscr{S}$, `select` and encode the deme $\mathbb{M}$ that will **drift** away to the destination island using the MPI `Send` procedure.*

Strand: an operator representing the **receive** step of the communication (line 6 in Algorithm 1).

Definition 4. *Given a sent deme $\mathbb{M}'$ that **strands** at the destination island, receive the deme $\mathbb{M}'$ using the MPI `Recv` procedure and then decode it.*

To prevent any errors due to the nondeterministic nature of communication in a parallel setting, manipulation of the population (insertion and replacement of the individuals) is considered separately as follows.

Reinsert: an operator that acts into play once the communication has finished. The receiving island's population P is modified by appending the *stranded* individuals.

Definition 5. *Given a received deme $\mathbb{M}'$, reintroduce all individuals $x \in \mathbb{M}'$ by following its replacement policy $\mathscr{R}$.*

After reinsertion, the algorithm designer must make sure to perform the replacement. For this, the `reinsert` operator returns the indices of the individuals that should be removed. Due to the non-deterministic nature of parallel communication, NaN values may appear during computation. This can be avoided by using the MPI *blocking* communication procedures (`Send` and `Recv`). Additionally, an MPI `Barrier` can be used at the end of the main loop which will allow all islands to synchronise. This is what we have done in the Julia code above.

4 Testing the Island Model

To test the new operators, we perform a comparison of the parallel island model against the serial counterpart with no communication, i.e, using the same algorithm but without the communication code shown in Section 3.1. To do the parallel tests, we use Julia 1.7.2 on Idun, NTNU's HPC solution [30]. A single script is submitted through the Slurm Workload Manager, which initialises MPI and then runs the code for one island in a distributed workload with as many

[3] See https://github.com/saxarona/idun-islands

CPUs as needed. For the parallel island model, we used 64 CPUs for a total of 64 solvers—one for each island. All MPI calls are provided by the `JuliaMPI.jl` package [3] which is based on OpenMPI. OpenMPI is available on most of the HPC clusters. The EC solver and its componentes were implemented using the provided blocks in EvoLP.jl v1.2 [32].

4.1 Tests Setup

Parallel Island Model Parameters

- The island model is an *archipelago* $\mathbb{A}$ with $|\mathbb{I}| = 64$ and a 1-way ring topology $\mathscr{T}$. For simplicity, all islands $I_i \in \mathbb{I}$ are identical.
- Each island I_i uses a generational GA as its solver $\mathscr{A}$.
- In our archipelago, migration occurs every 10 iterations, i.e., $\mu = 10$. In some tests, we change the migration rate. When that is the case, we describe it accordingly in its own subsection.
- To select the migrating individuals, we set the migratory selection policy $\mathscr{S}$ as random uniform, i.e., every $\boldsymbol{m}$ individual in the deme $\mathbb{M}$ is obtained as $\boldsymbol{m} \in M \sim \mathcal{U}(\mathbb{P})$.
- To select the set of individuals that will be replaced $\mathbb{P}_r$, we set the migratory replacement policy $\mathscr{R}$ as $\textsc{Worst}(k) = \mathbb{P}_r = \text{argmax}_{\boldsymbol{x}_i} f(\boldsymbol{x})$ for all $i \in [k]$.

Solver Parameters We use a generational GA solver $\mathscr{A}_i$ that uses the following operators and parameters:

- **Generator**: Continuous uniform initialisation, i.e., with an initial population of d-dimensional vectors $\boldsymbol{x} \in P$ such that $P \sim \mathcal{U}([l_b, u_b])$ for different lower and upper bounds l_b, u_b.
- **Selector**: Rank-based selection.
- **Recombinator**: Uniform crossover.
- **Mutator**: Gaussian mutation with standard deviation $\sigma = 0.1$
- **Algorithm parameters**: population size $|\mathbb{P}| = 30$ with a fixed stopping criterion of 100 generations. Both the crossover and mutation probabilities are set to 100%, and mutation affects all the offspring, similar to how evolution strategies work [7].

These solver parameters are the same for the serial approach and for each island I_i in the parallel island model. For an in-depth description of operators above, we invite the reader to refer to the EvoLP.jl v1.2 documentation.[4]

4.2 Test 1: Consistency on Easy Functions

Goal. Our goal is to perform a comparison of the two approaches as studied empirically with three fitness functions. Regarding the overall fitness of the

[4] See https://ntnu-ai-lab.github.io/EvoLP.jl/stable/index.html

solutions, are there any noticeable benefits of using the parallel island model over a serial approach? This would reflect in improved exploration and diversity preservation.

Method. Using EvoLP, we evolve 64 populations in two settings with equal parameters: using the parallel island model (with communication) and using a serial approach (without communication). The parallel island implementation uses the communication operators between 64 islands: the drift, strand and reinsert methods. The serial consists of evolving a single island (no communication) 64 times, with a full reinitialisation after every run. Both approaches use the same algorithm implementation with the parameters described in Section 4.1, and the parameters are kept the same through all the runs.

We optimise three continuous minimisation benchmarks, well-known in the optimisation community: Ackley (1), Rosenbrock (2) and Michalewicz (3) test functions. The *Ackley* function (1) is a multimodal function with multiple local minima far away from a central, global minimum $f(\boldsymbol{x}^*) = 0$ achieved with the optimiser at $\boldsymbol{x} = [0, .., 0]$, for all $x_i, i \in [d]$:

$$f(\boldsymbol{x}) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1} d\cos(cx_i)\right) + a + \exp(1). \quad (1)$$

We set $a = 20, b = 0.2$ and $c = 2\pi$, which are the typically used values. The island migration rate μ was set to 10 generations, for all dimensions $d \in \{2, 5, 10\}$. For the initial population, the values for $x_i \in \boldsymbol{x}$ are $x_i \in [-32.768, 32.768]$.

The *Rosenbrock* function (2) is a unimodal function with optimum $f(\boldsymbol{x}^*) = 0$ and optimiser $\boldsymbol{x}^* = [1, .., 1]$ for all $x_i, i \in [d]$:

$$f(\boldsymbol{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2. \quad (2)$$

The parameters a and b were kept at their default values as in EvoL.jl, the typically used $a = 1$ and $b = 5$. The migration rate μ was set to 10 generations throughout the tests for all number of dimensions. For the initial population, the values for $x_i \in \boldsymbol{x}$ are $x_i \in [-2.048, 2.048]$.

The *Michalewicz* function (3) is a function with multiple valleys and different optima depending on the number of dimensions [33]:

$$f(\boldsymbol{x}) = -\sum_{i=1}^{d} \sin(x_i) \sin^{2m}\left(\frac{ix_i^2}{\pi}\right). \quad (3)$$

The optional parameter m was kept to its typical value $m = 10$. The island migration rate μ was set to 5 generations for all problem dimensions, as the optimum in this function is more difficult to converge to. For all dimensions, the initial values for $x_i \in \boldsymbol{x}$ are $x_i \in [0, \pi]$.

Each function is tested over three different number of dimensions d: 2, 5 and 10. These optimisation functions vary in difficulty, and difficulty increases with dimensionality. In the case of Michalewicz, it is a multimodal function which requires the solver to have some kind of diversity preservation mechanism.[5]

[5] See Surjanovic and Bingham's Simulation Library: https://www.sfu.ca/~ssurjano/

Results and discussion. Figure 1 shows the results for this test. As depicted for the Ackley function, the parallel approach (in blue) was closer to the known optimum (shown as a dashed line) more consistently than the serial approach (in orange). In contrast, the Michalewicz function (that features different optima per number of dimensions) seemed to be more difficult to traverse as both approaches get similar distributions. These results suggests that using the parallel island approach might bring convergence benefits and higher fitness on some functions.

4.3 Test 2: High Multimodality

Goal. The goal of this test is to highlight the impact of the exploration mechanism of island models regarding the fitness of solutions when tested on highly multimodal functions.
Method. For this test, we use the same solver implementation but now on more difficult, highly multimodal and non-linear functions: Eggholder (4) and Rana (5). The *Eggholder* function (4) is a difficult function that features a rugged landscape. Optima can be consulted in [33]. The values $x_i \in \boldsymbol{x}$ for the initial population are $x_i \in [-512, 512]$.

$$f(\boldsymbol{x}) = -\sum_{i=1}^{n-1} \Bigg[(x_{i+1} + 47) \sin\left(\sqrt{\left|x_{i+1} + \frac{x_1}{2} + 47\right|}\right) \\ -x_i \sin\left(\sqrt{|x_1 - (x_{i+1} + 47)|}\right) \Bigg]. \tag{4}$$

The *Rana* function (5) is a similar function to the Eggholder, however it is far more rugged and it is symmetrical. Global minima can be consulted in [33]. The values $x_i \in \boldsymbol{x}$ for the initial population are the usual, $x_i \in [-512, 512]$.

$$f(\boldsymbol{x}) = \sum_{i=1}^{n-1} \Bigg(x_i \cos\sqrt{|x_{i+1} + x_i + 1|} \sin\sqrt{|x_{i+1} - x_i + 1|} \\ +(1 + x_{i+1}) \sin\sqrt{|x_{i+1} + x_i + 1|} \cos\sqrt{|x_{i+1} - x_i + 1|} \Bigg). \tag{5}$$

As opposed to the functions in Test 1, these functions have non-trivial global minima, and their minima do not have identical components [33]. Dimensionality $d \in \{2, 3, 5\}$ and we fixed the migratory rate at $\mu = 10$. All other parameters were kept the same as in Test 1.
Results. The results for this test can be consulted in Figure 2. As with the previous test, the island approach gets results closer to the optima more often. This is in line with the expectation as migration preserves population diversity by working as an exploration mechanism: The replacement policy refines the population by removing the worst candidates. In contrast, the lack of communication makes diversity preservation and convergence to the best solution difficult for the serial GA approach, especially in higher dimension problems.

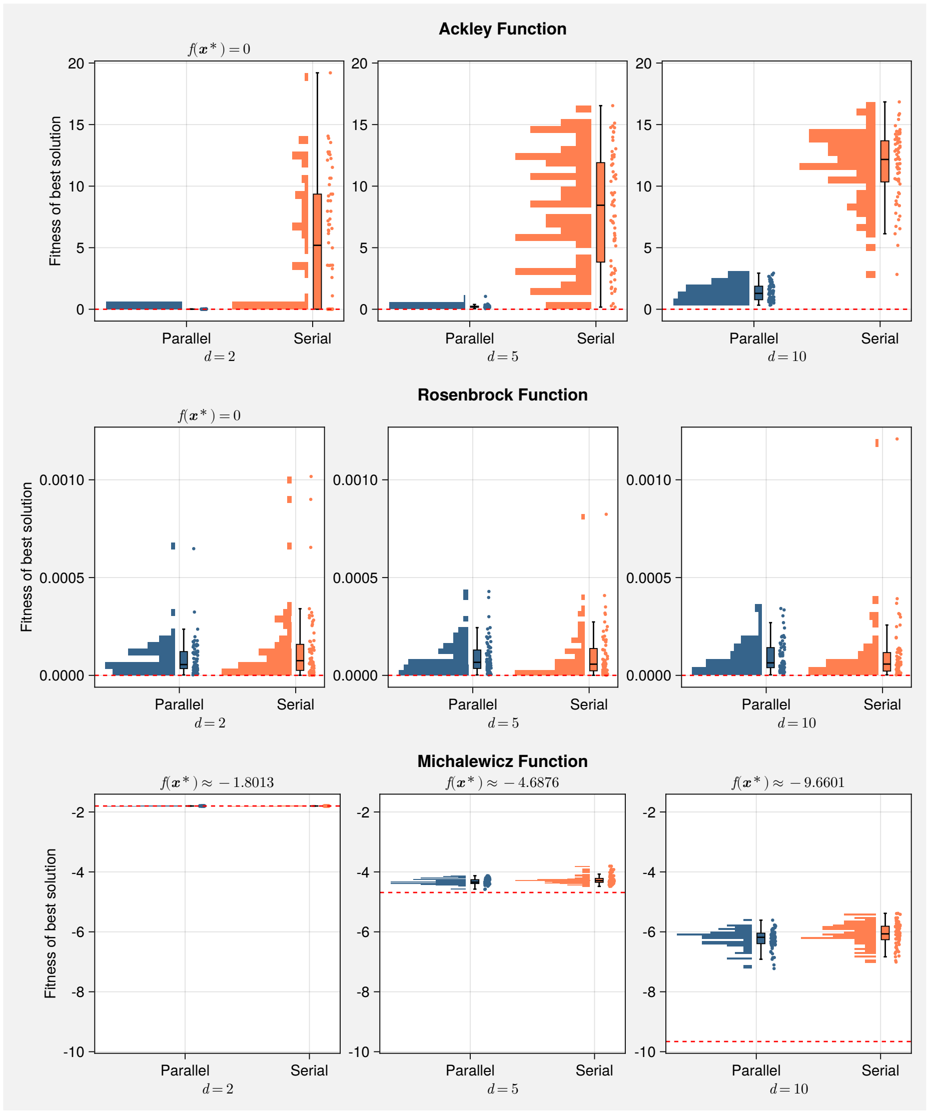

Fig. 1. Results overview for Test 1, with different functions per row and different number of dimensions per column. In each plot, the global minimum, $f(\boldsymbol{x}^*)$, is marked with a red dashed line. Each plot includes a histogram, a box plot and a scatter plot of the distributions of the best solution found in each of the 64 populations. Results of the 64-island parallel archipelago (featuring the drift, strand and reinsert methods) are presented in blue. For the serial results (shown in orange), a single population was evolved 64 times; with a full restart of the algorithm after meeting the termination criterion.

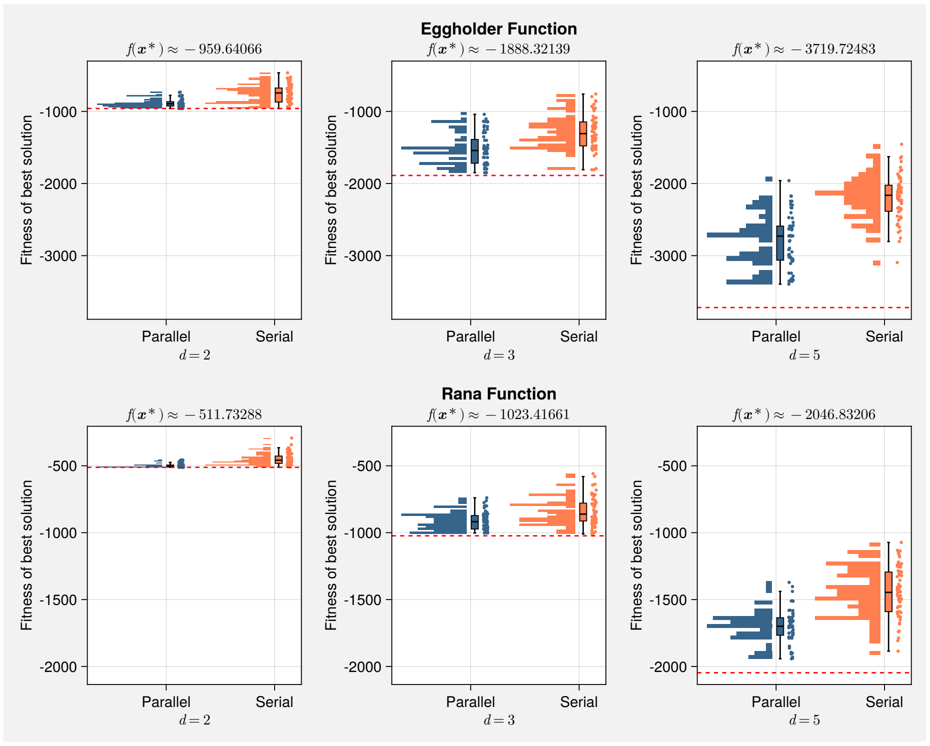

Fig. 2. Results overview for Test 2, with different functions per row and different number of dimensions per column. As with Test 1, in each plot we include a histogram, a box plot and a scatter plot of the distributions of the best solution found for each of the 64 populations. The known minimum, $f(\boldsymbol{x}^*)$ is marked in each plot as a red dashed line. The results of the parallel approach (in blue) show how the communication mechanisms help to find better solutions more consistently, while the serial approach (in orange) has difficulty optimising these highly rugged functions.

5 Conclusion and Future Work

In this paper we described three new operators for EvoLP.jl, namely: DRIFT, STRAND and REINSERT. These operators represent a meaningful extension to the framework as they allow the implementation of island models for GAs in parallel and distributed architectures without sacrificing the abstraction that the framework already provides. We defined the operators on a theoretical foundation, and then tested them in an HPC cluster using the MPI communication protocol. The tests highlighted that the communication in the parallel island model is beneficial for the exploration in the algorithm, which represents an advantage in a complex multimodal landscape. In addition, the tests showed how the parallel implementation gets better results than the serial implementation more consistently, over different functions with various complexities and number of dimensions.To ensure reproducibility, we provided an extensive description of the tests and their setup. Moreover, both the code and the data are available in the project repository.

For future work, we plan to ship this functionality as an extension in EvoLP.jl v2.0, which is planned for Q1 2024. In this way, researchers can use these operators directly from the official Julia package, regardless if they have access to HPC or want to use multiprocessing on their own workstations. Other potential areas of future work include expanding the analysis of diversity preservation in multimodal landscapes and the development of additional migratory selection and replacement policies. Exploring different topologies and an analysis of the MPI communication overheads is also a possibility.

Acknowledgements

We would like to acknowledge the financial support for EvoLP.jl, partly funded by Project no. 311284 of the Research Council of Norway, as well as access to computing resources from the Department of Computer Science of the Norwegian University of Science and Technology. We also thank the Norwegian Open Artificial Intelligence Lab for hosting EvoLP.jl in its GitHub repository.

References

1. Biscani, F., Izzo, D.: A parallel global multiobjective framework for optimization: Pagmo. Journal of Open Source Software **5**(53), 2338 (Sep 2020). https://doi.org/10.21105/joss.02338
2. Burak, J., Mengshoel, O.J.: A multi-objective genetic algorithm for jacket optimization. In: Proceedings of the GECCO Companion. pp. 1549–1556. GECCO '21, ACM, New York, NY, USA (Jul 2021). https://doi.org/10.1145/3449726.3463150
3. Byrne, S., Wilcox, L.C., Churavy, V.: MPI.jl: Julia bindings for the Message Passing Interface. Proceedings of the JuliaCon Conferences **1**(1), 68 (Jul 2021). https://doi.org/10.21105/jcon.00068

4. Cantú-Paz, E., Goldberg, D.E.: On the scalability of parallel genetic algorithms. Evolutionary Computation **7**(4), 429–449 (Dec 1999). https://doi.org/10.1162/evco.1999.7.4.429
5. Coletti, M.A., Scott, E.O., Bassett, J.K.: Library for evolutionary algorithms in Python (LEAP). In: Proceedings of the GECCO Companion. pp. 1571–1579. GECCO '20, ACM, New York, NY, USA (Jul 2020). https://doi.org/10.1145/3377929.3398147
6. Del Ser, J., Osaba, E., Molina, D., Yang, X.S., Salcedo-Sanz, S., Camacho, D., Das, S., Suganthan, P.N., Coello Coello, C.A., Herrera, F.: Bio-inspired computation: Where we stand and what's next. Swarm and Evolutionary Computation **48**, 220–250 (Aug 2019). https://doi.org/10.1016/j.swevo.2019.04.008
7. Eiben, A., Smith, J.: Introduction to Evolutionary Computing. Natural Computing Series, Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-44874-8
8. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edn. (1989)
9. Gong, Y., Fukunaga, A.: Distributed island-model genetic algorithms using heterogeneous parameter settings. In: 2011 IEEE Congress of Evolutionary Computation (CEC). pp. 820–827 (Jun 2011). https://doi.org/10.1109/CEC.2011.5949703
10. Guijt, A., Thierens, D., Alderliesten, T., Bosman, P.A.: The Impact of Asynchrony on Parallel Model-Based EAs. In: Proceedings of the GECCO. pp. 910–918. GECCO '23, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583131.3590406
11. Harada, T., Alba, E.: Parallel Genetic Algorithms: A Useful Survey. ACM Computing Surveys **53**(4), 86:1–86:39 (Aug 2020). https://doi.org/10.1145/3400031
12. Izzo, D., Ruciński, M., Biscani, F.: The Generalized Island Model. In: Fernández de Vega, F., Hidalgo Pérez, J.I., Lanchares, J. (eds.) Parallel Architectures and Bioinspired Algorithms, pp. 151–169. Studies in Computational Intelligence, Springer (2012). https://doi.org/10.1007/978-3-642-28789-3_7
13. Lu, Z., Whalen, I., Boddeti, V., Dhebar, Y., Deb, K., Goodman, E., Banzhaf, W.: NSGA-Net: Neural architecture search using multi-objective genetic algorithm. In: Proceedings of the GECCO. pp. 419–427. GECCO '19, ACM, New York, NY, USA (Jul 2019). https://doi.org/10.1145/3321707.3321729
14. Mahfoud, S.W., et al.: Crowding and preselection revisited. In: PPSN. vol. 2, pp. 27–36 (1992)
15. Mejía-de-Dios, J.A., Mezura-Montes, E.: Metaheuristics: A Julia Package for Single- and Multi-Objective Optimization. Journal of Open Source Software **7**(78), 4723 (Oct 2022). https://doi.org/10.21105/joss.04723
16. Mengshoel, O.J., Galán, S.F., de Dios, A.: Adaptive generalized crowding for genetic algorithms. Information Sciences **258**, 140–159 (2014). https://doi.org/https://doi.org/10.1016/j.ins.2013.08.056
17. Mengshoel, O.J., Foss, F., Sánchez-Díaz, X.F.C.: Controlling Hybrid Evolutionary Algorithms in Subset Selection for Multimodal Optimization. In: Proceedings of the GECCO Companion. pp. 507–510. GECCO '23 Companion, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583133.3590545
18. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0 (Jun 2021)
19. Muniasamy, R.P., Singh, S., Nasre, R., Narayanaswamy, N.: Effective Parallelization of the Vehicle Routing Problem. In: Proceedings of the GECCO. pp. 1036–1044. GECCO '23, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583131.3590458

20. Nowostawski, M., Poli, R.: Parallel genetic algorithm taxonomy. In: KES99. pp. 88–92 (1999). https://doi.org/10.1109/KES.1999.820127, https://ieeexplore.ieee.org/document/820127
21. Rudolph, G.: Global optimization by means of distributed evolution strategies. In: Schwefel, H.P., Männer, R. (eds.) Parallel Problem Solving from Nature. pp. 209–213. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1991). https://doi.org/10.1007/BFb0029754
22. Salcedo-Sanz, S., Gallo-Marazuela, D., Pastor-Sánchez, A., Carro-Calvo, L., Portilla-Figueras, A., Prieto, L.: Evolutionary computation approaches for real offshore wind farm layout: A case study in northern Europe. Expert Systems with Applications **40**(16), 6292–6297 (Nov 2013). https://doi.org/10.1016/j.eswa.2013.05.054
23. Salza, P., Ferrucci, F.: Speed up genetic algorithms in the cloud using software containers. Future Generation Computer Systems **92**, 276–289 (Mar 2019). https://doi.org/10.1016/j.future.2018.09.066
24. Schjølberg, M.E., Bekkevold, N.P., Sánchez-Díaz, X.F.C., Mengshoel, O.J.: Comparing Metaheuristic Optimization Algorithms for Ambulance Allocation: An Experimental Simulation Study. In: Proceedings of the GECCO. pp. 1454–1463. GECCO '23, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583131.3590345
25. Scott, E., De Jong, K.: Initialization Matters for Asynchronous Steady-State Evolutionary Algorithms. In: Proceedings of the GECCO Companion. pp. 1570–1578. GECCO '23 Companion, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583133.3596404
26. Scott, E.O., Luke, S.: ECJ at 20: Toward a general metaheuristics toolkit. In: Proceedings of the GECCO Companion. pp. 1391–1398. GECCO '19, ACM, New York, NY, USA (Jul 2019). https://doi.org/10.1145/3319619.3326865
27. Scrucca, L.: On Some Extensions to GA Package: Hybrid Optimisation, Parallelisation and Islands EvolutionOn some extensions to GA package: Hybrid optimisation, parallelisation and islands evolution. The R Journal **9**(1), 187–206 (2017)
28. Shahrzad, H., Miikkulainen, R.P.: Accelerating Evolution Through Gene Masking and Distributed Search. In: Proceedings of the GECCO. pp. 972–980. GECCO '23, ACM, New York, NY, USA (Jul 2023). https://doi.org/10.1145/3583131.3590508
29. Simon, D.: Evolutionary Optimization Algorithms. John Wiley & Sons (Jun 2013)
30. Själander, M., Jahre, M., Tufte, G., Reissmann, N.: EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure (Feb 2022). https://doi.org/10.48550/arXiv.1912.05848
31. Stanley, K.O., Clune, J., Lehman, J., Miikkulainen, R.: Designing neural networks through neuroevolution. Nature Machine Intelligence **1**(1), 24–35 (Jan 2019). https://doi.org/10.1038/s42256-018-0006-z
32. Sánchez-Díaz, X.F.C., Mengshoel, O.J.: EvoLP.jl: a playground for evolutionary computation in Julia. In: NAIS'23: Symposium of the Norwegian AI Society, in press. Bergen, Norway (June 2023)
33. Vanaret, C., Gotteland, J.B., Durand, N., Alliot, J.M.: Certified Global Minima for a Benchmark of Difficult Optimization Problems (Mar 2020). https://doi.org/10.48550/arXiv.2003.09867
34. Zavala, G.R., Nebro, A.J., Luna, F., Coello Coello, C.A.: A survey of multi-objective metaheuristics applied to structural optimization. Structural and Multidisciplinary Optimization **49**(4), 537–558 (Apr 2014). https://doi.org/10.1007/s00158-013-0996-4