Evolutionary Islands: A Distributed System for Evolutionary Algorithms

David Ding
fding@college.harvard.edu

Lily Tsai
lilliantsai@college.harvard.edu

Abstract— **This paper introduces Evolutionary Islands, a failure-tolerant, self-organizing distributed system for running evolutionary algorithms with island subpopulations and migrations between islands. To reach distributed consensus between islands, we require a variant of the Paxos algorithm, and to ensure robustness in the face of failure, we implement data replication among the islands. To test our system, we simulate random failures and slow network connections and measure its performance. The code for our system can be found online[1].**

I. Introduction

Our project aims to develop a distributed system framework for evolutionary island algorithms. Previous research has shown that evolutionary algorithms can achieve super-linear speedups (depending on the problem being solved) by separating out the population into smaller subpopulations to evolve [2]. Other research has shown that island algorithms improve on the solutions that were achieved using a regular evolutionary island algorithm [7]. In our project, we propose a different distributed system algorithm that previous papers have proposed, with the goals of making agent distribution completely self-organized by the islands in the cluster (with any island capable of being a leader), and robust to the failure of any one island.

This paper is organized as follows. In Section II, we give a broad overview of evolutionary algorithms and the traditional version of the Paxos distributed consensus algorithm. In Section III, we provide details about our system model and the rules of the migration algorithm that we implement. We describe how migrations distribute agents, and the different states an island transitions throughout the evolutionary algorithm. In Section IV, we present the distributed system challenges inherent in our system, and discuss our solution implementations. In Section V, we describe in detail our system infrastructure, present our measurements of our system performance, and discuss our system's limitations.

II. Background and Related Work

A. Evolutionary Algorithms

An evolutionary algorithm aims to create an optimal solution by generating a "population" of solutions or *agents*, evaluating these solutions, and then mutating or combining most successful of these solutions to produce solutions for the next generation. These algorithms draw inspiration from biological evolution, and the idea of "survival of the fittest."

An evolutionary algorithm that uses evolutionary islands first separates the population of agents into separate subpopulations. Each subpopulation undergoes evolutionary pressure for E number of epochs. After each individual epoch, the subpopulation's agents evolve into a new generation of agents derived from the most successful agents in that subpopulation. However, an agent on one island in one subpopulation cannot interact with agents on other islands during these E epochs. After E epochs, the agents in each island are "mixed," i.e. agents on one island are traded with agents from other islands.

The main goal of an evolutionary algorithm with islands is to allow each island to develop separate (and hopefully different) strategies for success in isolation, and then to mix agents across islands to allow for these separate strategies to combine and produce a more generally successful agent. It also expands the range of strategies that one subpopulation of agents may encounter; one worry in a large/infinite population evolutionary algorithm is that the agents may all settle into some suboptimal solution, since an agent's fitness is relative to the fitness of the agents in its population. In addition, smaller populations may be able to reach specific solutions in a search space more quickly than a large population due to the faster convergence of a smaller population. Finally, separating evolution into islands allow us to distribute the work across multiple islands and handle larger workloads.

Previous research on island evolutionary algorithms has shown that the isolation achieved by creating islands of agent subpopulations afforded greater diversity than evolutionary algorithms run on the population in its entirety [7]. In this paper, the migration topology is a ring; during migration number m, each island sends the best 0.5% of agents to the island m moves to its left and deletes its worst 0.5% agents. The global population therefore has duplicates of the top agents from each island. The paper performs the island evolutionary algorithm to find solutions to problems with multiple subproblems, i.e. problems in which increasing the population size does not help to solve the problem such as the deceptive problem [3]. In its evaluation, it demonstrates that using migration with evolutionary islands does not perform any worse than using a single large population.

Other work in this field have tested island evolutionary algorithms for automatic robot programming [6] and VLSI circuit design [4]. However, most work on distributed island evolutionary algorithms have not addressed the problems of island failures and/or agent loss, focusing instead on the performance of the algorithm itself.

B. Distributed Consensus and Failure Tolerance

A distributed system faces many problems that a single-island system does not, including those of consensus (reach-

[1]https://github.com/tslilyai/evolutionary_islands

ing agreement among the many islands among which work is divided) and failure tolerance (not losing the work or having inconsistencies between islands in the case that one island crashes or experiences another type of failure). In our design, we focus on reaching a consensus between islands in the face of non-Byzantine failures [5].

Previous research in the area of distributed consensus centers around the research done by Lamport: the Paxos algorithm [1]. In the basic Paxos algorithm, nodes reach consensus on a particular value as follows:

1) *Prepare*: machine $M1$ proposes ballot to start voting by sending a Prepare message to a quorum (often just a simple majority). This ballot has a unique proposal number $b1$.
2) *Promise*: If $b1$ is greater than any other previous proposal number received, any recipients of the Prepare send back a Promise to ignore future proposals having a number less than $b1$. The recipient then sends back the latest proposal number/value it accepted in the past to $M1$. If no Promise is sent, the Prepare request is simply ignored.
3) *Accept*: Once $M1$ has heard enough Promises, $M1$ sets the value of the proposal to the value associated with the highest proposal number sent back during the Promise phase, and sends an Accept to the machines which had promised with this value.
4) *Accepted*: The Promiser accepts the accept request sent only if it has not promised to any proposal with a number greater than $b1$.

In our implementation of Paxos, we modify the Promise and Prepare phases to reach consensus on how and when to run migration.

III. System Model and Design

The system model consists of a distributed cluster of centralized server clusters. There are three levels of machine hierarchy we consider:

1) Cluster of island machines; each machine represents an island of evolution
2) Island machines that act as centralized servers for evolving their agent population
3) Agent simulation servers that respond to and talk with the island machines

For our project, we assume that we can consider each island of evolution as an integrated unit; in particular, we do not consider communications within an island. We therefore assume that each island is a single machine.

A. Migration Model

The island machines must self-organize an agent exchange between themselves. This raises several interesting questions which must be answered in order to determine which challenges we face when designing the system:

1) Immigration Policy: What should the immigration policy between agents be? (e.g. should only the best agents migrate between islands, or should agents be chosen at random?)
2) Migration Frequency: How frequently should agents migrate? (e.g. should agents migrate every epoch, or only once the population seems to have reached a local optimum?)
3) Migration Density: How many migrants should there be per migration?
4) Migrant Topology: how should migrants be distributed during a migration? (e.g. islands send to each other in a ring)

We propose a migration design as follows:

1) Immigration Policy: Agents are chosen at random because the evolutionary algorithm per island has already eliminated the worst agents in each subpopulation. An alternative would be to only exchange the top agents in each population; however, we assume that the evolution per island has already done an adequate job at terminating unfit agents, and we also do not want to eliminate the possibility that an agent that is not at the top of one subpopulation may actually perform better in the global population.
2) Migration Frequency: Agents migrate every E epochs, where each epoch consists of an evolution of the subpopulations with in each island. This allows evolution to occur within a subpopulation before agents are mixed, but also ensures that all agents get to encounter agents of different islands at reasonable frequencies so that no population gets stuck in a suboptimal solution. If our goal were to search for subproblem solutions or different optimal solutions, we might instead only migrate once no progress was being made in a particular subpopulation. E defaults to 100, but can be altered by the user for their specific purpose.
3) Migration Density: Each migration, every agent in each subpopulation is sent to an island at random (which could be its current home island). We choose to migrate all agents, rather than simply a subset of agents, to optimize the mixing of populations. With a different goal (i.e. attempting to achieve different local optimal solutions), we may choose to migrate only a select few agents each migration so that the gene pool of a particular island remains mostly constant.
4) Migrant Topology: Migrants are distributed as follows:
    - Each island is associated with a unique island ID (*mid*).
    - During a migration, each island sends to every other island its ID and a list of its agents shuffled in a random order.
    - Once all the islands receive all lists, they select the nth subset of agents from each list to be the new agents on their island, where n is the position of the island in a list of sorted mids.

    This distribution policy ensures that agents move at most one hop per migration, and that, given the number of agents is divisible by the number of islands, each agent has an equal probability of ending up in any island. The exact way we deal with consensus and failure complications with this design are addressed in IV.

B. Island State Machine

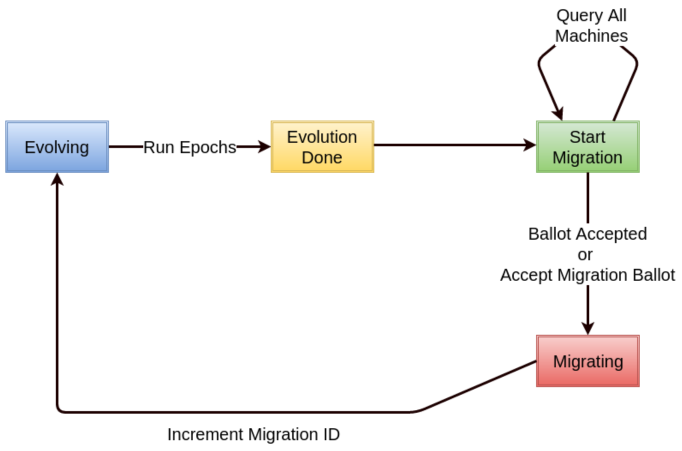

Fig. 1. Island States Machine

An island transitions through the following states within its lifetime:

- Evolving: The island is currently running its E epochs of evolution. During this time, the island ignores any migration proposals and does not interact with any other islands.
- Evolution Done: The island has finished evolving its agents and is ready to migrate. During this time, the island queries all other island for their status to get an initial list of islands to include in its migration proposal. Once these queries have either received responses or have timed out, the island transitions to the "Start Migration" phase. Should the island receive any queries from other islands, the island will send out a list of its agents in a randomly shuffled order along with its status.
- Start Migration: The island has proposed a migration ballot, and is also available to accept other migration ballots if allowed to do so under our modified Paxos protocol (see Section IV-B). If it has not accepted a migration proposal or its proposal has not been accepted after a time limit (currently set to 4 seconds), the island will re-query all islands again to generate a new migration ballot proposal. Should the island receive any queries from other islands, the island will send out a list of its agents in a randomly shuffled order along with its status.
- Migrating: During this phase, an island uses the shuffled list of agents from each island it has received from the status queries of the other islands to select its agents for the next round. This step is done locally and requires no interaction with any other islands. When in this state, the island ignores any migration proposals and does not send anything other than its status in response to a status request. Once migration has finished, the island increments a Migration ID and returns to the evolution phase.

IV. System Challenges

1) Fault Tolerance:
   - Exchange Failure Tolerance: the number of agents before and after an exchange must be preserved. If two servers exchange agents, either both servers receive new agents, or the exchange never happened. We also should not see the appearance of any duplicate agents, since that would defeat the purpose of an evolutionary algorithm.
   - Machine Failure Tolerance: failures/exits of islands should not lose the agents that are on those islands. If there are K islands, and S_k is the set of agents playing on island k, the failure of k cannot result in the loss of S_k. We assume a model in which an island can handle at most only the number of agents it is currently evolving. Thus, when an island fails, we must ensure that other islands do not become overloaded with the failed island's agents.
2) Migration Consensus: all islands should agree on what islands are participating in any one migration and on the starting time of the migration. This prevents islands from erroneously taking agents or losing agents. After a migration has started, the participating islands and agents should not change.
3) Sufficient Shuffling: an island should exchange with as many servers as possible. In other words, there should be no cluster of agents that compete only between themselves.

A. Fault Tolerance: Agent Redundancy

In our model, each island could crash at any time, and any network message could potentially be dropped. To avoid losing our best agents due to crashes, we use the following protocol.

At any point in time, each island stores all the agents across all the islands for redundancy in case the island owning the agents fails. This allows for $n-1$ tolerance for a system of n islands, since all agent data is kept unless all n islands fail. This list is updated every time an island responds to a status request once the island is done evolving its batch of agents, as these responses will include a list of the respondent island's ready-to-migrate agents.

If the number of islands decreases in a migration compared to the previous, the load capacity of the entire system goes down by the number of agents belonging to the failed island (we assume a model in which islands are always operating at full capacity). To every island not participating in the new migration (and therefore likely to have failed at some point), we still keep its agent representations stored on the live islands in case we ever wish to retrieve them. However, these agents no longer are exchanged in migration or are evolved. An alternative would be to start up a new island and assign it the agents of the failed island; however, we do not make assumptions that there will be the resources to do such a resurrection.

B. Migration Consensus

Each island undergoes E epochs of evolution before it can participate in a migration. To reach consensus on whether or not to begin a migration and which islands will be participating in the migration, we use a variant of Paxos [1].

Each island is obligated to respond, at any given point in time, to status query requests asking if it is alive, with a message indicating its status (if it is currently evolving, or if it is done with its batch).

Each island that is ready to migrate, i.e. has finished evolving, periodically queries all remaining islands to see if they are still alive. After waiting T seconds (our specified timeout period), if any responding island said it is not done with its E epochs yet, we sleep for a short period of time before querying again. Otherwise, the request has either timed out, in which case we assume the recipient island experienced a network failure and/or crashed, or we have received a status response back from the recipient island along with the island's list of agents in a shuffled order.

Once an island has requested statuses from every other island, it sends out a ballot to start a migration to all islands. Unlike traditional Paxos, this ballot also includes the value that will be accepted by all islands should the ballot be passed. This value consists of a list of `mids` of the islands that are participating in the migration (which are the islands that the proposing island has successfully heard back from with a list of their agents). A quorum in our system is the maximum of the number of participating islands in the migration proposal and a simple majority number of islands.

An island I is allowed to vote on a ballot if and only if it is allowed to do so under Paxos, along with a few additional requirements:

1) All islands in the proposed ballot value, i.e. all islands who are proposed to participate in the migration, must have responded with their shuffled list of agents to I's status requests. This ensures that I will have the necessary agent information to participate in the migration.
2) I itself must be in the proposed ballot migration participants.

Due to this scheme, a migration ballot can only pass if there are a majority of islands participating in the migration.

The guarantees of Paxos will ensure that after a ballot is passed, all the islands have agreed on a subset of islands that will participate in this migration.

Once the migration ballot is accepted and passes, each participating island selects agents to evolve as discussed in Section III. This ensures that all agents exchanged during this migration will be equally distributed at random over all participating islands, and therefore ensure adequate shuffling among the participants. Because upon each status request, an island will send out its agents in the same shuffled order, we are ensured that no agent will be duplicated.

Each island has a migration id that increments per migration. All messages sent are tagged with the island ID and the migration ID of the sender. Islands ignore messages associated with an earlier migration ID than their own; therefore, an island that is not included in a migration cannot participate in any future migration. Note that this only occurs should the island fail to respond to at minimum a quorum of islands; thus, an island can only be considered failed if it becomes disconnected from the majority of the network, or if it in fact crashes. While we could easily allow an island to rejoin even after missing a migration, we decided against it to keep our model of migrations cleaner and not introduce "stale" agents into the population, i.e. agents which have missed several rounds of evolution.

N	Migration Round	Paxos Ballots	Time spent on migration
6	0	9	10.0157 s
6	1	9	11.5164 s
8	0	9	10.0167 s
8	1	9	11.5141 s
16	0	21	11.5303 s
16	1	18	11.5154 s
32	0	37	11.5589 s
32	1	34	10.0824 s
48	0	50	11.5403 s
48	1	49	10.8644 s

TABLE I
MIGRATION PERFORMANCE VS N, THE NUMBER OF ISLANDS.

V. EVALUATION

We implemented a model of our system using Python. Each island is modeled as a separate process, and islands communicate via sockets. The islands are multi-thread processes that act as servers for requests from other islands and as clients to request another island to perform an action. To model random failures, each island possesses a special "die" thread, which has a chance of calling `_exit` every T seconds with probability p. Thus, the time each island runs before it fails is drawn from an exponential distribution.

Each island creates a server socket for other islands to connect to. In addition, each island maintains a mapping from machine ids to a persistent client socket that connects the island to another island's server. In the initialization phase, the island forks a server thread that listens for incoming requests such as `GETSTATUS` and Paxos ballots. Messages are sent and received using the `send` and `recv` system calls, and they are sent as JSON strings that are preceded by a 4 byte integer indicating the length of the message. To simulate network delay, occasionally the islands would sleep for some time before sending the message.

The rest of the implementation follows the design described previously. We implemented simply placeholders for the evolution functions in order to test our system.

When we ran our model, we verified that migrations continue to happen even when islands die, and that when islands die, other islands eventually notice, thus allowing migration to proceed.

Table I shows the performance overhead of migration. We measured the time spent on migration, defined to be the time elapsed between the last machine that finished the evolution phase, and the last machine to finish migration and start the new evolution phase. We also counted the number of Paxos ballots that are proposed before a migration starts. We did this for various numbers of machines: 6, 8, 16, 32, and 48 (due to limitations with our own computers, we were not able to fork 64 processes and maintain $64^2 = 4096$ pairwise sockets between the processes). We found that the number of Paxos ballots that are sent out is roughly equal to the number of machines, but the time spent in agreeing on and running a migration round does not depend on the number of machines. This is probably because the running time is dominated by the `sleep` calls we make in the code to avoid inundating the network with useless messages (such as get status messages when we know that the other machine is still evolving, or excessive numbers of Paxos ballots).

A. Limitations

Our system will always guarantee a consistent set of agents to evolve, and it will also guarantee that the agent data will not get lost over time, even should an island fail. However, it could lose evolutionary progress, as the consensus algorithm does not guarantee choosing the optimal sets of islands to participate in the next migration. For example, if island 1 loses connection to $n/2 - 1$ of the islands, and all other islands are fully connected, Paxos still allows a ballot from island 1 to be accepted since its ballots could still reach a quorum. This will kill those $n/2 - 1$ islands. A better subset for migration would be the remaining $n - 1$ islands, killing island 1 instead. Moreover, possibly a more sophisticated algorithm could allow information to be relayed across a connected component of the connection graph.

Because we assume a model in which all islands are always operating at full capacity, we may lose potential top agents in the global population when an island fails and we stop evolving its agents. An alternative would be to take the agents of a particular island when it dies, and determine some type of global ranking. From this ranking, we could select the top agents and redistribute among the remaining islands. This would be inefficient ($O(n)$ where n is the number of agents in the global population) and require reaching a consensus around the islands which are still alive. This also introduces extra complications should an island die during the redistribution. We conjecture that since the islands already are selecting for the top agents via evolution, we do not lose too much progress should an island die. This hypothesis is difficult to prove due to the random nature of an evolutionary algorithm.

We currently do not support resurrecting or adding islands; however, one can imagine that a new island could be added by simply initializing a random set of agents on the island and adding the island to the cluster, or by initializing the island with the agents of a failed island. Because we never add islands back into the system, if more than half islands fail, the system can no longer make progress; our quorum size stays fixed at a minimum of $\lceil M/2 \rceil$ where M is the initial number of islands in the system.

We also do not treat Byzantine failures in our system, which would make failure tolerance much more difficult. We can, however, imagine that our system could have some way to verify the results of a migration by checking for duplicates or keeping count of the agents distributed in the system; in addition, Byzantine failures during the evolution itself is less harmful than expected because the evolution includes several aspects of randomness.

VI. Conclusion

In this project, we describe Evolutionary Islands, a distributed system for evolutionary island algorithms. Unlike previous evolutionary island algorithms run on several machines, our system reaches consensus on when and how to run migrations without some master control and deals gracefully with the failure of other machines in the system. We also propose a variant of Paxos that allows a machine to propose both a ballot and the ballot's value, more strictly restricts voting on a ballot, and dynamically changes the quorum size of a ballot.

We tested our system with random machine failures and slow network connections to determine how performance is affected and how migration consensus handles these failures. In doing so, we discovered that the time spent in reaching consensus was independent of the number of machines in the system, and that most of the overhead came from scheduled blocking to avoid inundating the network.

VII. Acknowledgements

We, David Ding and Lily Tsai, affirm our awareness of the standards of the Harvard College Honor Code. We would also like to acknowledge Professor Jim Waldo, our mentor and teacher of CS262, Introduction to Distributed Systems, for his guidance during this semester and during this project.

References

[1] Lamport, L., et al. Paxos made simple. *ACM Sigact News 32*, 4 (2001), 18–25.

[2] Lässig, J., and Sudholt, D. Parallel problem solving from nature, ppsn xi: 11th international conference, kraków, poland, september 11-15, 2010, proceedings, part i. R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph, Eds., Springer Berlin Heidelberg, pp. 234–243.

[3] Li, M., and Kou, J. The schema deceptiveness and deceptive problems of genetic algorithms. *Science in China Series: Information Sciences 44*, 5 (2001), 342–350.

[4] Martin, W. N., Lienig, J., and Cohoon, J. P. Island (migration) models: evolutionary algorithms based on punctuated equilibria. *Handbook of evolutionary computation 6*, 3 (1997).

[5] Schneider, F. B. Replication management using the statemachine approach. *Mullender [7]* (2005).

[6] Tongchim, S., and Chongstitvatana, P. Speedup improvement on automatic robot programming by parallel genetic programming. In *Proceedings of IEEE International Symposium On Intelligent Signal Processing and Communication Systems* (1999), pp. 77–80.

[7] Whitley, D., Rana, S., and Heckendorn, R. B. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology 7* (1998), 33–47.