An Actor Model Implementation of Distributed Factored Evolutionary Algorithms

Stephyn G. W. Butcher
Johns Hopkins University
Baltimore, MD
steve.butcher@jhu.edu

John W. Sheppard
Montana State University
Bozeman, MT
john.sheppard@montana.edu

ABSTRACT

With the rise of networked multi-core machines, we have seen an increased emphasis on parallel and distributed programming. In this paper we describe an implementation of Factored Evolutionary Algorithms (FEA) and Distributed Factored Evolutionary Algorithms (DFEA) using the Actor model. FEA and DFEA are multi-population algorithms, which make them good candidates for distributed implementation. The Actor model is a robust architecture for implementing distributed, reactive programs. After walking through the translation of the serial pseudocode into an Actor implementation, we run validation experiments against an FEA baseline. The evidence supports the claim that the Actor versions preserve the semantics and operational performance of the FEA baseline. We also discuss some of the nuances of translating serial pseudocode into an actual distributed implementation.

KEYWORDS

Actor Model, Concurrency, Distributed Computing, Parallel Computing, Factored Evolutionary Algorithms

ACM Reference Format:
Stephyn G. W. Butcher and John W. Sheppard. 2018. An Actor Model Implementation of Distributed Factored Evolutionary Algorithms. In *GECCO '18 Companion: Genetic and Evolutionary Computation Conference Companion, July 15–19, 2018, Kyoto, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3205651.3208261

1 INTRODUCTION

Evolutionary algorithms of all stripes can be computationally intensive and expensive. This computational cost can come from either the actual evolutionary algorithm or fitness/objective function evaluations. However, because these operations are all CPU-bound, they are not likely to get much help from mere concurrency. And with the apparent demise of Moore's Law, we find ourselves in the same position as everyone else in software engineering: how do we take advantage of more cores (either on a single machine or across multiple machines). One solution to this problem is distributed parallelism, but there is more than one way to implement distributed parallelism.

There is also the question of which evolutionary algorithm to implement. If we try to implement, say, particle swarm optimization (PSO) in a distributed parallel fashion, do we assign a single particle to each core? This seems impractical if solving the problem requires 100 particles and thus 100 cores. If we only have 10 cores, we could batch our work and parcel it out to all the cores, but this adds incidental complexity we would like to avoid. One alternative would be to use a multi-population evolutionary algorithm as such algorithms would have the "batches" designed into them at the start.

In this paper we describe an Actor model implementation of Factored Evolutionary Algorithms (FEA) and Distributed Factored Evolutionary Algorithms (DFEA). We start by describing the Actor model in Section 2. We then describe FEA and DFEA and the translation of the their serial versions into the Actor model (Section 3). We then subject our implementations to a validation test against 19 benchmark optimization functions, the results of which are presented in Section 4. In the final section, we describe key insights based on our implementation process and thoughts for future work.

2 ACTOR MODEL

The Actor model was originally proposed by Hewitt *et al.* as a modular computational architecture for artificial intelligence [10]. The architecture was developed further by Agha into a metalinguistic model of the *potentially* concurrent execution of processes [1]. Later, Ericsson built the the Actor model into the OTP (originally "Open Telecom Platform"), part of the runtime of the Erlang programming language. Leveraging Erland and the OTP, the AXD301 project was able to achieve "nine nine's" (i.e., 99.9999999%) reliability [2]. As a metalinguistic construct, there are Actor libraries available for many programming languages such as Akka [3] (for JVM languages such as Java and Scala) and Thespian [16] for Python. In the following, we will focus mostly on the Akka/Thespian-style implementations of the Actor model.

Although implementations vary [5], the key properties of an actor are:

(1) they may only communicate via asynchronous messages;

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan

ACM ISBN 978-1-4503-5764-7/18/07…$15.00
https://doi.org/10.1145/3205651.3208261

(2) messages may be received at anytime and are queued in an inbox; and
(3) upon reading a message, the actor may perform a computation.

These properties have interesting implications. First, there is no way to access the state of an actor without sending it an asychronous message. This differs from the Object model as we have generally come to know it, where "messages" are now synchronous method calls. It has been reported apocryphally that Alan Kay—who coined "object oriented"— stated the Actor model is the closest to what he originally meant by the term. While the statement cannot be verified, Kay has repeated that message passing was the key idea of object oriented programming, and not inheritance or types [13]:

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

which, although not an explicit endorsement of the Actor model, is a fairly good description of it.

Second, although actors are not operating system or green threads, they do represent a mechanism of concurrent programming. The actual details are handled by the *Actor System*, which is responsible for spawning actors, maintaining their addresses, monitoring inboxes and delivering messages, restarting failed actors, and making sure every actor gets a chance to execute. The last point is handled by a thread pooling mechanism and load balancing. The default implementation is something like a "round robin" approach where every actor has a chance to act on a single message in its inbox. We must note, however, that an Actor implementation can still experience deadlock if messages and state transitions are not properly designed.

Third, an actor can be run anywhere. An actor has local state and an interface defined by the messages it understands. When an actor sends a message to another actor, it sends it to the actor's *address* maintained by the Actor System. The receiving actor may be on the same core, a different core, or even a different machine.

In object oriented languages like Java and Python, actors are generally implemented as a subclass of some Actor base class. The instance fields of the Actor become its state and the subclass overrides something like a *receive* method with formal parameters *message* and *sender*. When it is the actor's turn to execute, the Actor *System* will call the actor's *receive* method if there is a message on in the actor's inbox. The actor may also implement instance methods for code organization but clients may *only* interact with an actor instance via messages through the actor's *address*. Algorithm 1 shows a simple example of such a *receive* method.

Actor A accepts two messages: *IncrementCount* and *Retrieve–Count*. If an instance of Actor A receives an *IncrementCount* message (Line 1), the *count* is incremented by the value indicated (Line 2). If, instead, the instance of Actor A receives a *RetrieveCount* message (Line 3), a new message *CurrentCount* is sent back to the sender containing the current count, *count*

Algorithm 1 Actor A - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf IncrementCount then
2:     count ← count + message.increment
3: else if message instanceOf RetrieveCount then
4:     tell(sender, CurrentCount(count))
5: end if
```

(Line 4). This is a common pattern for implementing an Actor's *receive* method and in many respects acts like a finite state machine. For example, an actor may receive a message and update its state, and then optionally send a message.

Most actors are instantiated by other actors and thus all messages between them are asynchronous. However, at the *serial* boundary, we require an ability to "inject" messages to get this mostly reactive system rolling and facilities for this are usually provided by the Actor System.

3 IMPLEMENTATION

In this section, we discuss the translation of FEA and DFEA from a serial version to an Actor model version. We will start with FEA and give an overview of the main components of the algorithm. Following that discussion, we will explain a corresponding Actor model implementation. We will then do the same thing for DFEA.

3.1 FEA and Actor FEA

Factored Evolutionary Algorithms [17] are a family of algorithms shown to be effective for many optimization problems, including weights of neural networks [9, 15] and abductive inference in Bayesian networks [7, 8]. FEA decomposes the variables of an optimization problem into possibly overlapping subsets, called "factors," to each of which an evolutionary algorithm is assigned. Through a process of updating, competing, and sharing, the partial solutions discovered in the subpopulations are combined into a full solution that is then shared with the subpopulations. Any evolutionary algorithm can be used as an optimizer in FEA, including algorithms such as Particle Swarm Optimization (PSO) [14] or Genetic Algorithms (GA) [11]. Research into FEA has demonstrated that they perform better than their single population counterparts; although, factor architecture—problem decomposition—has been shown to affect performance [17].

The serial implementation of FEA is described in Algorithms 2–4. The main algorithm is composed of three steps: Update, Compete and Share. During the Update Step (Algorithm 2, Lines 6–10), each individual factor (S) updates until the stopping criteria are met. In general, stopping criteria could be quite sophisticated. For example, we might want to stop if five iterations have passed or the average fitness of the population has stopped changing by some ϵ. In all the validation experiments, we have chosen to stop after a fixed number of iterations.

Algorithm 2 Factored Evolutionary Algorithms

Input: Function f, Evolutionary Algorithm *ea*
Output: Context $\mathbf{c}$ as candidate solution $\mathbf{x}$

```
1: X ← factorize(X)
2: S ← ea.initialize(f, X)
3: c ← initialize-context(S)
4: O ← identify-optimizers(X)
5: repeat
6:     repeat
7:         for S in S do
8:             S ← ea.update(S)
9:         end for
10:    until stopping criteria
11:    c ← compete(f, S, O, c)
12:    share(f, S, ea, c)
13: until stopping criteria
14: return c
```

Algorithm 3 FEA Compete

Input: Objective function f, Subpopulations $\boldsymbol{S}$, Optimizers $\boldsymbol{O}$, Global context $\mathbf{c}$
Output: Global context $\mathbf{c}$

```
1: for j = 1 to d do
2:     fitness ← f(c)
3:     value ← c[j]
4:     for i in O_j do
5:         candidate ← S[i].best
6:         c[i] ← candidate.x[i]
7:         if f(c) ≤ fitness then
8:             value ← candidate.x[i]
9:             fitness ← f(c)
10:        end if
11:    end for
12:    c[i] ← value
13: end for
14: return c
```

After the Update Step is done, the Compete Step resolves differences between the existing context, $\mathbf{c}$ or candidate solution, and the newly discovered constituents in each subpopulation, S. The Compete Step is described in Algorithm 3. The goal of the Compete Step is to pick the best value for each X_j, whether it might be the current value, $\mathbf{c_j}$, or one of the values, x_j, in one of the optimizers, $\mathbf{O} \in \boldsymbol{O}$, of X_j. This is accomplished by an outer loop that iterates through the variables (Lines 1–13) and an inner loop that iterates over the optimizers of X_j (Lines 4–11).

After the context has been updated, the Share Step (Algorithm 4) begins. The Share Step is a bookkeeping step where the newly updated context is applied to each swarm. Each subpopulation only optimizes the variables of its factor, X, which is a subset of the full set of variables being optimized, $\mathbf{X}$. In order to evaluate X, the subpopulation needs to know the values of $\mathbf{c}$ that are *not* for X. Their variables are

Algorithm 4 FEA Share

Input: Objective function f, Subpopulations $\mathbf{S}$, Evolutionary Algorithm *ea*, Context $\mathbf{c}$
Output: Subpopulations $\mathbf{S}$

```
1: for S in S do
2:     r ← c \ S.X
3:     f_r ← partial(f, r)
4:     p ← ae.worse(S)
5:     p.x ← c \ r
6:     S.f ← f_r
7:     ae.reevaluate(S)
8: end for
```

called the *residuals*. Additionally, we apply a form of *elitism* by replacing the worst individual in the subpopulation by the appropriate values of X from $\mathbf{c}$. Finally, individuals in the subpopulation must be re-evaluated based on the new residuals.

We now turn to the Actor model implementation of FEA. A common pattern in Actor model implementations is a manager/workers pattern where a job is divided into units of work, and each unit is given to a worker to complete. The results are then aggregated back together and returned to the original client. We can use a similar pattern for the Actor implementation of the FEA. The main challenge here is that the "disperse and collect" cycle is repeated until some stopping criterion is met. We will break the algorithm into two actors: the FEA Actor (Algorithm 5, and the FEA Factor Actor (Algorithm 6).

The FEA actor responds to two messages: *InitFEA* (Line 1) and *NewValue* (Line 14). Additionally, it will send *InitFactor* (Line 10), *Update* (Lines 12 and 21), *NewSolution* (Line 20), and *CandidateSolution* (Line 23) messages. Whereas an object oriented solution might have a synchronous method call *solve* as an interface, the FEA actor's interface is the asynchronous messages *InitFEA* and *CandidateSolution*.

Upon receipt of the *InitFEA* message, the FEA actor proceeds almost identically to the first part of Algorithm 2. The main difference is in Lines 2–3 and Lines 8–13. In Line 2 we save the client who sent the FEA actor the *InitFEA* message so that we can respond later. We also save the *problem* record, which encapsulates information both about the problem and parameters for the algorithm. In Lines 8–13 we create the workers, FEA Factor actors, sending them both an *InitFactor* message and a *Update* message, after saving the reference to each worker's address. Before continuing with the FEA actor and the *NewValue* message, we describe the FEA Factor actor (Algorithm 6).

The FEA Factor actor responds to three messages: *InitFactor* (Line 1), *Update* (Line 4), and *NewSolution* (Line 9). When the FEA Factor actor receives the *InitFactor* message it saves the *problem* from the message and then initializes the subpopulation based on the particular evolutionary algorithm, optimization problem, and factor. When it receives the *Update* method, the actor updates the subpopulation for i

Algorithm 5 FEA Actor - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf InitFEA then
2:     client ← sender
3:     problem ← message.problem
4:     𝒳 ← factorize(X)
5:     S ← ea.initialize(f, 𝒳)
6:     c ← initialize-context(S)
7:     O ← identify-optimizers(𝒳)
8:     for X in 𝒳 do
9:         worker ← actorOf(FEAFactor())
10:        worker.send(InitFactor(problem, X))
11:        workers[X] ← worker
12:        worker.send(Update())
13:    end for
14: else if message instanceOf NewValue then
15:    cache[message.x_i] ← message.value
16:    if new values received from all actors then
17:        if FEA iterations not complete then
18:            compete()
19:            clearCache()
20:            broadcast(workers, NewSolution())
21:            broadcast(workers, Update())
22:        else
23:            client.send(CandidateSolution(c))
24:        end if
25:    end if
26: end if
```

Algorithm 6 FEA Factor Actor - receive

Input: message *message*, sender *sender*
Output: None

```
1: if message instanceOf InitFactor then
2:     problem ← message.problem
3:     S ← ae.initialize(f, X)
4: else if message instanceOf Update then
5:     for i times do
6:         S ← ae.update(S)
7:     end for
8:     sender.send(NewValue(X, S.best)
9: else if message instanceOf NewSolution then
10:    applySolution(S, message.c)
11: end if
```

iterations. This is exactly the same as the corresponding lines in Algorithm 2. In order to make the loop in Algorithm 2 run concurrently, we have turned the iteration loop (Algorithm 2, Line 6) into a message sending loop (Algorithm 5, Line 12 and Line 21). The inner loop from Algorithm 2 then runs on the individual actors. When the FEA Factor actor's part in this distributed Update Step is done, it sends a *NewValue* message back to the FEA actor.

Returning to Algorithm 5, the FEA actor responds to the *NewValue* message by caching the value (Line 15). It then tests to see if it has received all the expected new values. This cache-and-test pattern implements the bookkeeping required to coordinate the workers. If all the expected new values have been received—and the desired number of FEA iterations have been completed—the Compete Step is executed. This Compete Step is otherwise identical to Algorithm 3 except that, instead of extracting the values from the subpopulations, the values have already been extracted and saved to the cache. After this new Compete Step is finished, the cache is cleared for the next round, and a *NewSolution* message is sent to all the workers.

In the context of this paper, *broadcast* is just a helper function that loops over the actor references, sending each the same message. It is not a "fire-and-forget" broadcast or any other type of pub/sub message passing. Actors always send messages to specific actors. This is followed by an *Update* message to everyone. If the FEA iterations have completed (or, more generally, the stopping criteria have been met), then the FEA actor sends a *CandidateSolution* message to the original client.

3.2 DFEA and Actor DFEA

Recognizing the importance of distributed algorithms in computationally intensive optimization problems, a distributed version of FEA was developed, the Distributed FEA [4]. In this section we discuss the conversion of DFEA (Algorithms 7 and 8) to the Actor model.

Despite DFEA having been designed with distributed state in mind (each subpopulation maintains its own context), the translation of the algorithm actually takes more work. The main reason for this is that, although the *state* is distributed in DFEA, the *manipulation* of the state is not. In the FEA actor, the context (candidate solution) was maintained by the supervising actor, and all the FEA Factor actors (workers) communicated only with the supervisor. As we shall see, in the *D*FEA implementation, each DFEA Factor actor has its own context (candidate solution), and although they are spawned by the DFEA actor as a supervisor, the supervisory role ends there. Throughout the computation, the DFEA Factors communicate and coordinate with each other. This makes the implementation somewhat more complicated.

The serial implementation of DFEA is shown in Algorithm 7. In most respects, it is similar to the FEA code in Algorithm 2, with one exception. In FEA, the Compete Step acts as a single arbiter of possibly better values of **c** obtained from the optimizers of each variable. Because the context is distributed among the factors in DFEA, we must pick some factor to be the arbiter of each variable. Not every factor can arbitrate X_3, for example. Thus we must assign an arbiter role for each variable to some factor (Line 5).

The Reconcile Step (Algorithm 8) is similar to FEA's Compete Step (Algorithm 3). The FEA Compete Step compares the context, **c**, to the new values discovered by the subpopulations during the previous Update Step. The DFEA Reconcile

Algorithm 7 Distributed Factored Evolutionary Algorithm

Input: Function f, Evolutionary Algorithm *ae*
Output: Best context $\mathbf{c}$ as candidate solution $\mathbf{x}$

1: $\boldsymbol{\mathcal{X}} \leftarrow factorize(\mathbf{X})$
2: $\mathbf{S} \leftarrow ae.initialize(f, \boldsymbol{\mathcal{X}})$
3: $\boldsymbol{C} \leftarrow$ *initialize-contexts*$(\mathbf{S})$
4: $\boldsymbol{O} \leftarrow$ *identify-optimizers*$(\boldsymbol{\mathcal{X}})$
5: $\boldsymbol{\mathcal{A}} \leftarrow$ *identify-arbiters*$(\boldsymbol{\mathcal{X}})$
6: **repeat**
7: **repeat**
8: **for** S in $\mathbf{S}$ **do**
9: $S \leftarrow ae.update(S)$
10: **end for**
11: **until** stopping criteria
12: $\boldsymbol{C} \leftarrow reconcile(f, \mathbf{S}, \boldsymbol{O}, \boldsymbol{\mathcal{A}}, \boldsymbol{C})$
13: $share(f, \mathbf{S}, ea, \boldsymbol{C})$
14: **until** stopping criteria
15: $\mathbf{c} \leftarrow$ *select-best-context*$(f, \boldsymbol{C})$
16: **return** $\mathbf{c}$

Algorithm 8 DFEA Reconcile

Input: Function f, Subpopulations $\mathbf{S}$, optimizers $\boldsymbol{O}$, arbiters $\boldsymbol{\mathcal{A}}$, Local contexts $\boldsymbol{C}$
Output: Local contexts $\boldsymbol{C}$

1: **for** $j = 1$ **to** d **do**
2: $\mathbf{c} \leftarrow \boldsymbol{C}[\boldsymbol{\mathcal{A}}(x_j)]$
3: $fitness \leftarrow f(\mathbf{c})$
4: $value \leftarrow \mathbf{c}[j]$
5: **for** k in $\boldsymbol{O}_j$ **do**
6: $candidate \leftarrow \boldsymbol{\mathcal{S}}[k].best$
7: $\mathbf{c}[j] \leftarrow candidate.\mathbf{x}[j]$
8: **if** $f(\mathbf{c}) \leq fitness$ **then**
9: $value \leftarrow candidate.\mathbf{x}[j]$
10: $fitness \leftarrow f(\mathbf{c})$
11: **end if**
12: **end for**
13: $\mathbf{c}[j] \leftarrow value$
14: **for** $k = 1$ **to** $|\mathbf{S}|$ **do**
15: $\boldsymbol{C}[k].\mathbf{c}[j] \leftarrow \mathbf{c}[j]$
16: **end for**
17: **end for**
18: **return** $\boldsymbol{C}$

Step does the same thing, except that it looks up the context belonging to the arbiter of the current variable (Line 2). Additionally, the DFEA Reconcile Step must perform a more extensive update across contexts in order to keep the information flow semantically equivalent to FEA (Lines 14–16). The Share Step is equivalent to FEA's Share Step and not repeated here. Finally, DFEA returns the best of the contexts as the candidate solution (Algorithm 7, Line 15).

In the FEA Actor model implementation, the FEA actor was more complex than the FEA Factor actor. In the DFEA Actor model implementation, this is reversed; the DFEA Factor actor is more complicated. This follows directly from the nature of the algorithm, which distributes a local context to each individual subpopulation where it is then maintained and coordinated. However, the algorithm does not specify a corresponding means of manipulating and coordinating those local contexts. This is to be expected since there is no single way to specify pseudocode appropriate for all possible concurrency implementations and picking one could make the translation to another equally complicated. In this case, we at least have a clear idea of the intended semantics.

Algorithm 9 DFEA Actor - receive

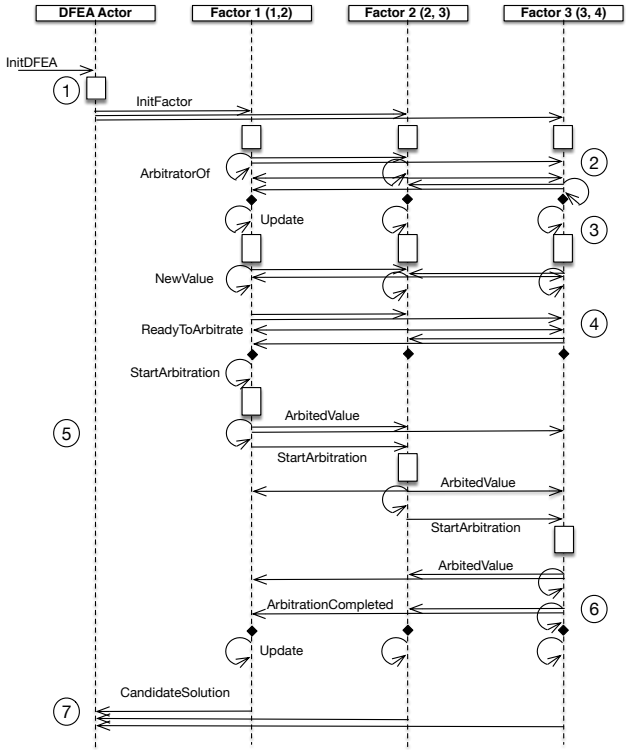
Input: message *message*, sender *sender*
Output: None

1: **if** *message* instanceOf *InitDFEA* **then**
2: *client* $\leftarrow$ *sender*
3: *problem* $\leftarrow$ *message.problem*
4: $\boldsymbol{\mathcal{X}} \leftarrow factorize(\mathbf{X})$
5: $\mathbf{S} \leftarrow ae.initialize(f, \boldsymbol{\mathcal{X}})$
6: $\mathbf{c} \leftarrow$ *initialize-context*()
7: $\boldsymbol{O} \leftarrow$ *dfeaa:identify-optimizers*$(\boldsymbol{\mathcal{X}})$
8: $\boldsymbol{\mathcal{A}} \leftarrow$ *dfeaa:identify-arbiters*$(\boldsymbol{\mathcal{X}})$
9: **for** $\mathbf{X}$ in $\boldsymbol{\mathcal{X}}$ **do**
10: **workers**[$\mathbf{X}$] $\leftarrow actorOf(DFEAFactor())$
11: **end for**
12: *broadcast*(**workers**, *InitFactor*(*problem*, $\mathbf{c}$, $\mathbf{X}$, $\boldsymbol{\mathcal{A}}$))
13: *broadcast*(**workers**, *Update*())
14: **else if** *message* instanceOf *CandidateSolution* **then**
15: **if** all candidate solutions received **then**
16: $\mathbf{c} \leftarrow$ *select-best-context*$(f, \boldsymbol{C})$
17: *client.send*(*CandidateSolution*($\mathbf{c}$))
18: **end if**
19: **end if**

Our particular implementation uses 11 messages: *InitDFEA*, *InitFactor*, *ArbiterOf*, *Update*, *NewValue*, *ReadyToArbitrate*, *StartArbitration*, *ArbitedValue*, *ArbitrationComplete*, and *CandidateSolution*. Rather than list pseudocode for two very long receive methods or 11 individual handlers for each message, we will describe the DFEA actor and Factor actors in terms of a specific example using the sequence diagram in Figure 1. Assuming a problem of $4d$, there will be four variables: X_1, X_2, X_3, and X_4. Factor 1 is optimizing (X_1, X_2) and arbitrating X_1. Factor 2 is optimizing (X_2, X_3) and arbitrating X_2. Factor 3 is optimizing (X_3, X_4) and arbitrating X_3 and X_4.

We start in an Initialization phase. At (1) in Figure 1, an asynchronous *InitDFEA* message is sent to the DFEA actor's mailbox (All the messages are sent asynchronously in this example; denoted by open arrowheads). The message contains information about the problem being solved and the parameters for DFEA itself such as stopping criteria. Just as with the serial version (Algorithm 7, Lines 4, Lines6–5), the Actor model version begins by initializing factors, optimizers, and assigning arbiters to each factor. This is followed by the creating a Factor actor for each factor, in this case: Factor 1, Factor 2 and Factor 3. After these actors are created,

Figure 1: DFEA and DFEA Factor Actors Sequence Diagram



they each receive an *InitFactor* message on their individual mailboxes.

The *InitFactor* message signals each actor to take the information on the problem, factors, optimizers and arbitrators contained in the message and initialize their subpopulation using the indicated evolutionary algorithm. This corresponds to Algorithm 7, Line 2 in the serial version of the algorithm. After each Factor actor has initialized, (2) they send an *ArbiterOf* message to all their peers indicating the variables for which they are the arbiter. There is a corresponding coordination point (black diamond) where a Factor actor must wait until it has heard from all the arbiters for the variables it is optimizing. Using Factor 1 as an example, after it has been initialized, it sends *ArbiterOf* message to all its peers indicating that it is the arbiter for X_1. Because Factor 1 is optimizing X_1 and X_2, it waits to hear from the Factors optimizing X_1 and X_2 before proceeding. This is a pattern that we saw before in the FEA Actor, where a cache is used to coordinate multiple actors and then tested to change to the next state. We indicate this *cache-and-test* pattern with the black diamond in the diagram. The difference in this case is that when Factor 1 sent the *ArbiterOf* messages, it sent one to itself. We do this to avoid special case code in the actors which would require the actor to know who it is; as we will see later, we were not entirely successful.

After a Factor has discovered which actors will arbitrate each of the variables it optimizes, (3), it sends itself an *Update* message, which starts the Update phase. At this point, the Factor will run its evolutionary algorithm on its subpopulation until the stopping criteria are met. This corresponds to Algorithm 7, Lines 7–11. Upon completion, the factor will send a *NewValue* message for each of the variables it optimizes to the arbiter of that variable. In the case of Factor 1, it will send a *NewValue* message for X_1 and one for X_2. Again we see the pattern of avoiding special code: Factor 1 sends the message about X_1 to itself. It does not send a new value message to itself for X_2 because it is not an arbiter of X_2.

With (4) we enter another coordination point that uses the cache-and-test pattern. A Factor is ready to arbitrate X_1 if it has heard from all the optimizers of X_1. When it has, it sends a *ReadyToArbitrate* message to all of the actors. Factor 3, because it is the arbiter of X_3 and X_4 will await messages for both variables and send a message for each variable.

Once all of the Factor actors have received all of the *ReadyToArbitrate* messages, the Arbitration phase begins, (5). We have assumed that the variables will be arbitrated in order, X_1, X_2, X_3 then X_4, for simplicity. As a result, when Factor 1 knows that all the factors are ready to arbitrate, it sends itself a *StartArbitration* message. This message begins the reconciliation process described in Algorithm 8 with a few key differences. First, instead of iterating from the "outside" over all the variables, in the Actor model implementation, the actor is working from the "inside" with a specific variable to arbitrate. Second, the Factor does not have global access to the subpopulation information so it cannot reach into the subpopulations of optimizers of X_i and find x_i. Instead, these were the values communicated via *NewValue* messages before arbitration commenced.

When the current factor (Factor 1) is finished, it sends *ArbitedValue* messages to all of its peers. This will enable factors further down the arbitration order to use those values during reconciliation just as with the serial version. After sending those messages, it sends a single *StartArbitration* message to the next factor in the arbitration order. We can easily calculate this from the current information in Factor 1. If it arbitrates X_1 then the next arbiter must be X_2. If a different or changing arbitration order were desired, this would need to be communicated and coordinated as well.

When the final factor finishes arbitration, (6), it sends an *ArbitrationCompleted* message to all of its peers. Factor 3 is able to discover that it is the final factor when it finishes arbitrating X_4 and discovers there is no "next" variable to arbitrate. At this point, we have another cache-and-test pattern as all the factors await the message indicating that the Arbitration phase is over. If the stopping criteria for the DFEA is not met, all the factors will send themselves *Update* messages and the Update phase begins anew (this case is shown). If the stopping criteria has been met, each Factor sends a *CandidateSolution* message to the DFEA actor, (7).

Because each individual actor knows what the stopping criteria are, this message does not have to be coordinated on the Factor side.

We have not shown how the DFEA actor handles the *CandidateSolution* messages. For testing, we used the cache-and-test pattern to await all of the messages. When they were all received, the best solution was chosen and sent back to the driver program. However, there is no reason that the DFEA actor cannot maintain a single best solution and revise it as *CandidateSolution* messages come in. The DFEA actor could then be queried via a message at any time for what it thinks is the best solution so far.

4 VALIDATION

DFEA was meant to preserve the semantics of FEA in the presence of distributed state in the form of local contexts. Because of the similarities in the algorithms and the fact that neither the Compete/Reconcile or Share Steps consume random numbers, the output of each algorithm when initialized with the same random seed is identical. We can thus see that DFEA preserved the semantics of FEA.

The above Actor model implementations of both FEA and DFEA are meant to preserve the semantics of the original algorithms. However, because these are distributed algorithms running on multiple threads, we are unable to verify the implementations against the baseline the same way that we did for DFEA. Instead we have turned to experimental means.

4.1 Design

In order to test the hypothesis that the Actor implementations preserved the semantics of the original algorithm, we executed FEA (baseline), FEA Actor, and DFEA Actor implementations against 19 benchmark optimization functions. We picked benchmark optimization functions that were scalable to multiple dimensions from [12] and [18]. These are all minimization problems, and most of them have a minima at $\mathbf{x}^* = [0, 0, ..., 0]$ and $f(\mathbf{x}^*) = 0$. The notable exceptions are the Exponential, Eggholder and Michalewicz functions. For these experiments we used $32d$ versions of the functions.

For our evolutionary algorithm we chose PSO. The PSO parameters were $\omega = 0.729$ and $\phi_1 = \phi_2 = 1.49618$. In all cases, there were 20 FEA iterations with 5 update iterations per iteration. The factor architecture was the same for all algorithms and functions: we used a "Simple Centered" factor architecture of (x_i, x_{i+1}), thus, the first factor was (x_1, x_2), the second was (x_2, x_3), etc. Each subpopulation (swarm) had 10 particles.

4.2 Results

Each function was optimized by each algorithm 50 times, and the mean minimum value found was recorded for each run. The results were then bootstrapped 500 times to estimate 95% confidence intervals/credible intervals [6]. Selected results for the mean minima and confidence intervals are shown in Table 1.

In every case, the FEA Actor implementation performed as well as the FEA baseline (serial) implementation. This was also true for the DFEA Actor implementation. Additionally, in almost every case except one (18 out of 19), the FEA Actor and DFEA Actor implementations performed equally as well. The odd function out was the Zakharov function where the DFEA Actor implementation appears to have performed slightly better than the FEA Actor implementation (last row of Table 1). Given the consistent performance of the algorithms, however, this is likely to have been a statistical fluke. We believe the evidence supports the hypothesis that the Actor implementations preserved the semantics of the baseline algorithms.

5 DISCUSSION

Surprisingly, although DFEA was designed to represent distributed state, the translation of FEA into the Actor model was easier than DFEA's translation. This is largely because although the Actor model is effective for concurrency, it lacks primitives for coordinated, distributed state. It is thus easier for the FEA Actor to launch as many Factor Actors as needed and take care of the coordination required for the Compete and Share Steps than for the DFEA Actor to do the same. In the case of the DFEA Actor implementation, the DFEA Actor delegates any coordinating role to its Factor Actors who then coordinate among themselves. Another way to think of this is that DFEA Factor actors are cooperative peers whereas the FEA Factor actors are solitary workers.

When thinking about implementing a peer pattern using the Actor model, the implementation often becomes confusing because we have to think of the actor not only as the sender of the message but also the receiver of the message. There are a few places where this breaks down. For example, in order to start the Arbitration phase, we have to test the current actor to see if they are supposed to go first. Similarly, when the Arbitration phase is over, we have to check to see if the current actor is the last arbiter to go and then it sends out a different message.

The reactive nature of the Actor model has interesting implications that are not fully utilized in these experiments. In some instances, we may have a very difficult optimization problem for which we would want a provisional answer and then updates to that answer. The Actor model, by virtue of its reactive nature, would support this use case directly. Either implementation could be reconfigured to run indefinitely rather than for some fixed number of FEA iterations. An API service could be launched in an Actor System that could talk to the Actor System this running (D)FEA Actor instance. The API service could then send a *RequestSolution* message to the (D)FEA Actor instance and wait for the reply returning it to the user. The API could be used by program requiring the solution.

6 CONCLUSIONS

In this paper we presented an Actor model implementation of Factored Evolutionary Algorithms and Distributed Factored

Table 1: Results for FEA Baseline and FEA and DFEA Actor Implementations

Function	FEA Baseline	Actor FEA	Actor DFEA
Dixon-price	2.75e+01 (2.01e+01, 3.52e+01)	3.20e+01 (2.35e+01, 4.05e+01)	3.22e+01 (2.25e+01, 4.17e+01)
Eggholder	-2.13e+04 (-2.16e+04, -2.10e+04)	-2.14e+04 (-2.17e+04, -2.11e+04)	-2.13e+04 (-2.16e+04, -2.10e+04)
Michalewicz	-3.07e+01 (-3.08e+01, -3.06e+01)	-3.08e+01 (-3.10e+01, -3.07e+01)	-3.08e+01 (-3.10e+01, -3.07e+01)
Sargan	3.43e+03 (1.71e+03, 5.28e+03)	1.71e+03 (6.06e+02, 3.03e+03)	2.09e+03 (9.22e+02, 3.69e+03)
Zakharov	8.09e+02 (7.82e+02, 8.34e+02)	8.21e+02 (7.92e+02, 8.51e+02)	7.56e+02 (7.29e+02, 7.83e+02)

Evolutionary Algorithms. The Actor implementation of FEA involved a fairly straight-forward translation of the serial pseudocode to a parallel implementation. This involved a common pattern in Actor-based implementations where a supervisor breaks a task into pieces and then spins up a worker Actor for each piece. This pattern matched FEA exactly.

Although DFEA has the same general steps as FEA, the semantic intent is closer to that of peers rather than workers. This made the translation of the serial pseudocode into a parallel implementation a bit more challenging, even though the basics had been worked out. The Actor-based implementation involved using a peer pattern, which required us to think of each Actor as not only the sender of the message but the receiver of the message. In some cases, this required code to handle special cases as in the start of the Arbitration phase.

The evidence presented by our validation experiments strongly indicate that our implementations faithfully reproduce the semantic intent of the original algorithms. Using PSO as the evolutionary algorithm, we ran experiments for three implementations: FEA baseline, FEA Actor, and DFEA Actor. Using 19 benchmark optimization functions, we showed that both the FEA Actor and the DFEA Actor performed comparably to the FEA baseline. There was one strange case where the DFEA Actor implementation performed better than the FEA baseline.

There are at least two areas of future work we would like to investigate. First, one of the hallmarks of the Actor model is resilience, and Erlang is famous for the aphorism, "let it crash." In the Actor model, exceptions are not caught. Instead, the actor is crashed and its parent brings up a fresh version. This would present challenges for our current DFEA Actor implementation. Second, research on DFEA has suggested that for many problems, coordination between the peers (consensus) can be relaxed. We would like to enhance our current DFEA Actor implementation to address both of these areas.

REFERENCES

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[2] Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors.* Ph.D. Dissertation. Royal Institute of Technology, Stockholm, Sweden.

[3] Jonas Boner. 2009. Akka. https://akka.io/. (2009). Accessed: 2018-03-19.

[4] Stephyn Butcher, Shane Strasser, Jenna Hoole, Benjamin Demeo, and John Sheppard. 2016. Relaxing Consensus in Distributed Factored Evolutionary Algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*. ACM, 5–12.

[5] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 31–40.

[6] B. Efron. 1979. Bootstrap Methods: Another Look at the Jackknife. *Ann. Statist.* 7, 1 (01 1979), 1–26. https://doi.org/10.1214/aos/1176344552

[7] Nathan Fortier, John Sheppard, and Karthik Ganesan Pillai. 2013. Bayesian abductive inference using overlapping swarm intelligence. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*. 263–270.

[8] Nathan Fortier, John Sheppard, and Shane Strasser. 2014. Abductive inference in Bayesian networks using distributed overlapping swarm intelligence. *Soft Computing* 19, 4 (2014), 981–1001.

[9] Nathan Fortier, John W Sheppard, and KG Pillai. 2012. DOSI: training artificial neural networks using overlapping swarm intelligence with local credit assignment. In *Joint 6th International Conference on Soft Computing and Intelligent Systems (SCIS) and 13th International Symposium on Advanced Intelligent Systems (ISIS)*. 1420–1425.

[10] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *IJCAI*, Vol. 3. Stanford Research Institute, 235–245.

[11] John H Holland. 1975. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

[12] Momin Jamil and Xin-She Yang. 2013. A literature survey of benchmark functions for global optimisation problems. *International Journal of Mathematical Modelling and Numerical Optimisation* 4, 2 (2013), 150–194.

[13] Alan Kay. 2003. Clarification of "object-oriented". http://www.purl.org/stefan_ram/pub/doc_kay_oop_en. (2003). Accessed: 2018-03-19.

[14] James Kennedy and Russell Eberhart. 1995. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*. 1942–1948.

[15] Karthik Ganesan Pillai and JW Sheppard. 2011. Overlapping swarm intelligence for training artificial neural networks. In *Proceedings of the IEEE Swarm Intelligence Symposium (SIS)*. 1–8.

[16] Kevin Quick. 2015. Thespian: Python Actor Model Library. http://thespianpy.com/. (2015). Accessed: 2018-03-19.

[17] Shane Strasser, Nathan Fortier, John Sheppard, and Rollie Goodman. 2017. Factored Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* 21, 3 (2017), 281–293.

[18] Frans Van den Bergh and Andries Petrus Engelbrecht. 2004. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation* 8, 3 (2004), 225–239.