Scaling in Concurrent Evolutionary Algorithms

Juan J. Merelo[1], J. L. J. Laredo[2], Pedro A. Castillo[1], José-Mario García-Valdez[3], and Sergio Rojas-Galeano[4(✉)]

[1] Universidad de Granada/CITIC, Granada, Spain
{jmerelo,pacv}@ugr.es
[2] RI2C-LITIS, Université du Havre Normandie, Le Havre, France
juanlu.jimenez@univ-lehavre.fr
[3] Instituto Tecnológico de Tijuana, Calzada Tecnológico, s/n, Tijuana, Mexico
mario@tectijuana.edu.mx
[4] School of Engineering, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia
srojas@udistrital.edu.co

Abstract. The concept of *channel*, a computational mechanism used to convey state to different threads of process execution, is at the core of the design of multi-threaded concurrent algorithms. In the case of concurrent evolutionary algorithms, *channels* can be used to communicate messages between several threads performing different evolution tasks related to genetic operations or mixing of populations. In this paper we study to what extent the design of these messages in a communicating sequential process context may influence scaling and performance of concurrent evolutionary algorithms. For this aim, we designed a channel-based concurrent evolutionary algorithm that is able to effectively solve different benchmark binary problems (e.g. OneMax, LeadingOnes, RoyalRoad), showing that it provides a good basis to leverage the multi-threaded and multi-core capabilities of modern computers. Although our results indicate that concurrency is advantageous to scale-up the performance of evolutionary algorithms, they also highlight how the trade–off between concurrency, communication and evolutionary parameters affect the outcome of the evolved solutions, opening-up new opportunities for algorithm design.

Keywords: Concurrency · Concurrent evolutionary algorithms · Performance evaluation · Algorithm design · Distributed evolutionary algorithm

1 Introduction

Despite the emphasis on leveraging newer hardware features with best-suited software techniques, there are not many papers [20] dealing with the creation of concurrent evolutionary algorithms that work in a single computing node or that

J. C. Figueroa-García et al. (Eds.): WEA 2019, CCIS 1052, pp. 16–27, 2019.
https://doi.org/10.1007/978-3-030-31019-6_2

extend seamlessly from single to many computers. In that sense, concurrent programming seems to be the best option if we are dealing with a multi-core multi-threaded processor architecture where many processes and threads can coexists at the same time. The latter implies applications (and algorithms) should be able to leverage those processes to take full advantage of their capabilities.

The best way to do so is to match those capabilities at an abstract level by languages that build on them so that high-level algorithms can be implemented without worrying about the low-level mechanisms of creation or destruction of threads, or how data is shared or communicated among them. These languages are called concurrent, and the programming paradigm implemented in them concurrency-oriented programming or simply concurrent programming [2].

These languages, that include Perl 6, Go, Scala and Julia, usually support programming constructs that manage threads like first class objects, including operators for acting upon them and to using them as function's input or return parameters. The latter have implications in the coding of concurrent algorithms due to the direct mapping between patterns of communication and processes with language expressions: on the one hand it simplifies coding because higher-level abstractions for communication are available; on the other hand it changes the paradigm for implementing algorithms, since these new communication constructs and the overhead they bring to processing data need to be considered.

Moreover, concurrent programming adds a layer of abstraction over the parallel facilities of processors and operating systems, offering a high-level interface that allows the user to program modules of code to be executed in parallel threads [1].

Different languages offer different concurrency strategies depending on how they deal with shared state, that is, data structures that could be accessed from several processes or threads. In this regard, there are two major fields and other, less well known models using, for instance, tuple spaces [9]:

- Actor-based concurrency [24] totally eliminates shared state by introducing a series of data structures called *actors* that store state and can mutate it locally.
- Process calculi or process algebra is a framework to describe systems that work with independent processes interacting between them using channels. One of the best known is called the *communicating sequential processes* (CSP) methodology [12], which is effectively stateless, with different processes reacting to a channel input without changing state, and writing to these channels. Unlike actor based concurrency, which keeps state local, in this case per-process state is totally eliminated, with all computation state managed as messages in a channel.

Many modern languages, however, follow the CSP abstraction, and it has become popular since it fits well other programming paradigms, like reactive and functional programming, and allows for a more efficient implementation, with less overhead, and with well-defined primitives. This is why we will use it in this paper for creating *natively* concurrent evolutionary algorithms. We have chosen Perl 6, although other alternatives such as Go and Julia are feasible.

In previous papers [21,22] we designed an evolutionary algorithm that fits well this architecture and explored its possibilities. That initial exploration showed that a critical factor within this algorithmic model is the communication between threads; therefore designing efficient messages is high-priority to obtain good algorithmic performance and scaling. In this paper, we will test several communication strategies: a loss-less one that compresses the population, and a lossy one that sends a representation of gene-wise statistics of the population.

The rest of the paper is organized as follows: next we present the state of the art in concurrency in evolutionary algorithms, followed by Sect. 3 on the design of concurrent EAs in Perl 6. Experimental results are presented next in Sect. 4; finally, we discuss our conclusions in Sect. 5.

2 State of the Art

The parallelization of nature-inspired optimization algorithms has been an active field, allowing researchers to solve complex optimization problems having a high computational cost [17]. In the literature, most works are concerned with process-based concurrency, using, for instance, a Message Passing Interface (MPI) [10], or hybrids with fine-grain parallelization by using libraries such as OpenMP [6] that offer multi-threading capabilities [15]. Recent trends in software development have motivated the inclusion of new constructs to programming languages to simplify the development of multi-threaded programs. The theoretical support used by these implementations is based on CSP. Languages such as Go and Perl 6 implement this concurrency model as an abstraction for their multi-threading capabilities. (the latter including additional mechanisms such as *promises* or low-level access to the creation of threads). Even interpreted languages with a global interpreter lock, such as Python also have included *promises* and *futures* in their latest versions, to leverage intensive multi-threading IO capabilities.

The fact that messages have to be processed without secondary effects and that actors do not share state makes concurrent programming specially fit for languages with functional features; this has made this paradigm specially popular for late cloud computing implementations; however, its reception in the EA community has been scarce [11], although some efforts have lately revived the interest for this paradigm [26]. Several years ago it was used in Genetic Programming [4,13,29] and recently in neuroevolution [25] and program synthesis [28] using the functional programming features of the Erlang language for building an evolutionary multi-agent system [3].

Earlier efforts to study the issues of concurrency in EA are worth mentioning. For instance, the EvAg model [14] resorts to the underlying platform scheduler to manage the different threads of execution of the evolving agents; in this way the model scaled-up seamlessly to take full advantage of CPU cores. In the same avenue of measuring scalability, experiments were conducted in [18] comparing single and a dual-core processor concurrency achieving near linear speed-ups. The latter was further on extended in [19] by scaling up the experiment to up to 188 parallel machines, reporting speed-ups up to 960×, nearly four times

the expected linear growth in the number of machines (when local concurrency were not taken into account). Other authors have addressed explicitly multi-core architectures, such as Tagawa [27] which used shared memory and a clever mechanism to avoid deadlocks. Similarly, [16] used a message-based architecture developed in Erlang, separating GA populations as different processes, although all communication was taking place with a common central thread.

In previous papers [8,22], we presented a proof of concept of the implementation of a stateless evolutionary algorithms using Perl 6, based on a single channel model communicating threads for population evolving and mixing. In addition, we studied the effect of running parameters such as the *generation gap* (similar to the concept of *time to migration* in parallel evolutionary algorithms) and population size, realizing that the choice of parameters may have a strong influence at the algorithmic level, but also at the implementation level, in fact affecting the actual wallclock performance of the EA.

3 Design of a Concurrent Evolutionary Algorithm in Perl6

Perl 6 is a concurrent, functional language [5] which was conceived with the intention of providing a solid conceptual framework for multi-paradigm computing, including thread-based concurrency and asynchrony. It's got a heuristic layer that optimizes code during execution time. In the last few years, performance of programs written in Perl 6 has been sped-up by a 100× factor, approaching the same scale of other interpreted languages, although still with some room for improvement.

The `Algorithm::Evolutionary::Simple` Perl 6 module was published in the ecosystem a year ago and got recently into version 0.0.7. It is a straightforward implementation of a canonical evolutionary algorithm with binary representation and includes building blocks for a generational genetic algorithm, as well as some fitness functions used generally as benchmarks.

The baseline we are building upon, is similar to the one used in previous experiments [22]. Our intention was to create a system that was not functionally equivalent to a sequential evolutionary algorithms, that also follows the principle of CSP. We decided to allow the algorithm to implement several threads communicating state through channels. Every process itself will be stateless, reacting to the presence of messages in the channels it is listening to and sending result back to them, without changing state.

As in the previous papers, [21], we will use two groups of threads and two channels. The two groups of threads perform the following functions:

- The *evolutionary* threads will be the ones performing the operations of the evolutionary algorithm.
- The *mixing* thread will take existing populations, to create new ones as a mixture of them.

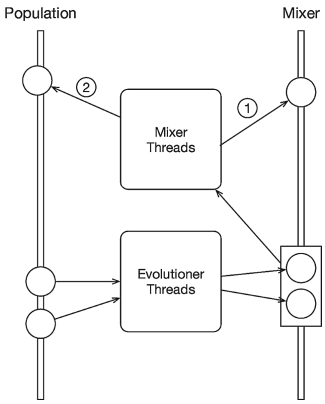

Fig. 1. General scheme of operation of channels and thread groups.

The main objective of using two channels is to avoid deadlocks; the fact that one population is written always back to the mixer channel avoids starvation of the channel. Figure 2 illustrates this operation, where the timeline of the interchange of messages between the evolver and mixer threads and evolver and mixer channels is clarified.

The state of the algorithm will be transmitted via messages that contain data about one population. Since using the whole population will incur in a lot of overhead, we use a strategy that is inspired in *EDA*, or Estimation of Distribution Algorithm: instead of transmitting the entire population, the message sent to the channel will consist of a prototype array containing the prob-

Besides, the two channels carry messages consisting of populations, but they do so in a different way:

- The *evolutionary* channel will be used for carrying non-evolved, or generated, populations.
- The *mixer* channel will carry, *in pairs*, evolved populations.

These will be connected as shown in Fig. 1. The evolutionary thread group will read only from the evolutionary channel, evolve for a number of generations, and send the result to the mixer channel; the mixer group of threads will read only from the mixer channel, in pairs. From every pair, a random element is put back into the mixer channel, and a new population is generated and sent back to the evolutionary channel.

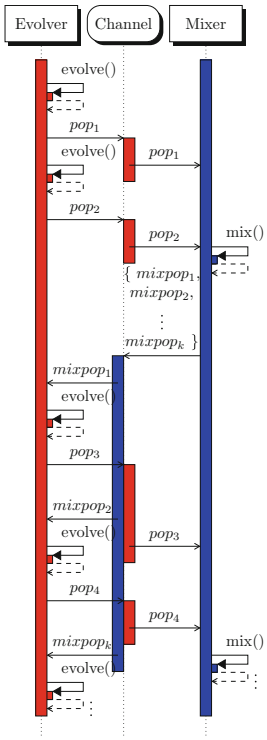

Fig. 2. Communication between threads and channels for concurrent EAs. The two central bars represent the channel, and color corresponds to their roles: blue for mixer, red for evolver. Notice how the evolver threads always read from the mixer channel, and always write to the evolver channel. (Color figure online)

ability distribution across each gene in the population. In this sense, this strategy is similar to the one presented by de la Ossa et al. in [7].

Nonetheless, our strategy differs from a pure EDA in that once the evolutionary thread have internally run a canonical genetic algorithm, it takes only the top quartile of best individuals to compute an array with the probability distribution of their genes (computed with frequentist rules) and then compose the message that is sent to the *mixer* threads.

A *mixer* thread, in turn, builds a new prototype array by choosing randomly at each gene location one probability parameter out of the two *populations* (actually, distributions), instead of working directly on individuals. While in the baseline strategy the selection took place in the mixer thread by eliminating half the population, in this new design the selection occurs in the evolutionary thread that selects the 25% best individuals to compose the probability distribution message. When the evolver thread reads the message back, it generates a new population using the mixed distribution obtained by the mixer.

4 Experimental Results

We focused on the scaling capabilities of the algorithm and implementation, so we tested several benchmark, binary functions: OneMax, Royal Road and Leading Ones, all of them with 64 bits (for function definitions see e.g. [23]).

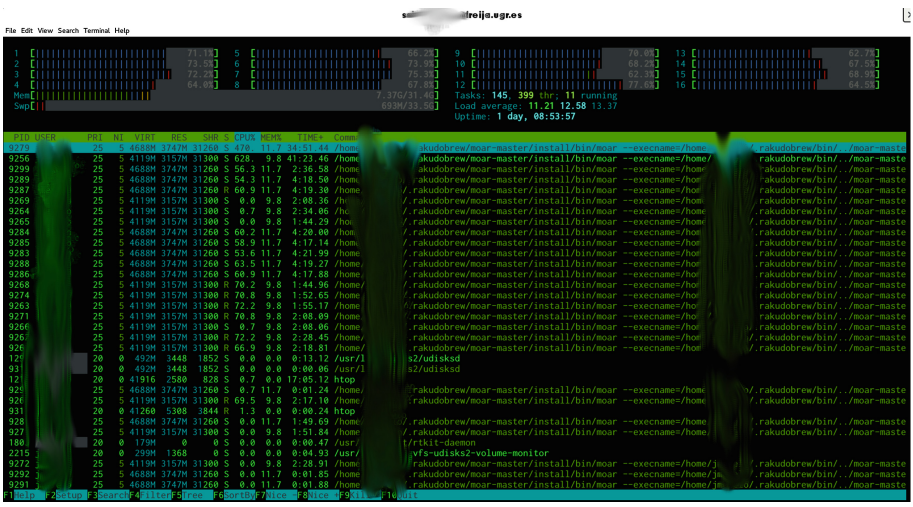

Fig. 3. An htop utility screenshot for the used machine running two experiments simultaneously. It can be seen all processors are kept busy, with a very high load average.

However, the intention of concurrent evolutionary algorithms is to leverage the power of all threads and processors in a computer so we must find out how it scales for different fitness functions. We are setting the number of initial populations to the number of threads plus one, as the minimum required to

avoid starvation, and we are using a single mixing thread. As reported in our previous papers [8,22], we are dividing the total population by the number of threads. The population size will be 1024 for OneMax, 8192 for Royal Road and 4096 Leading Ones. These quantities were found heuristically by applying the bisection method on a selector-recombinative algorithm, which doubles the population until one that is able to find the solution 95% of the time is found.

Experiments were run on a machine with the Ubuntu 18.04 OS and an AMD Ryzen 7 2700X Eight-Core Processor at 3.7GHz, which theoretically has $8 \times 16 = 128$ physical threads. Figure 3 shows the utility `htop` with an experiment running; the top of the screen shows the rate at which all cores are working, showing all of them occupied; of course, the program was not running exclusively, but the list of processes below show how the program is replicated in several processor, thus leveraging their full power.

Observe also the number of threads that are actually running at the same time, a few of which are being used by our application; these are not, however, physical but operating system threads; the OS is able to accommodate many more threads that are physically available if the code using them is idle.

We are firstly interested in the number of evaluations needed to find the solution (see Fig. 4), since as it was mentioned previously, a tradeoff should exist between the performance of the algorithm and the way it is deployed over different threads. In this case, population size does have an influence on the number of evaluations, with bigger populations tipping the balance in favor of exploration and thus making more evaluations to achieve the same result; the same happens with smaller populations, they tend to reach local minima and thus also increase exploration.

Figure 4 shows how the overall number of populations increases slightly and not significantly from 2 to 4 threads, but it does increase significantly for 6 and 8 threads, indicating that the algorithm's performance is worsen when we increase the number of threads, and consequently more evaluations are needed to achieve the same result. This is probably due to the fact that we are simultaneously decreasing the population size, yielding an earlier convergence for the number of generations (8) it is being used. This interplay between the degree of concurrency, the population size and the number of generations will have to be explored further.

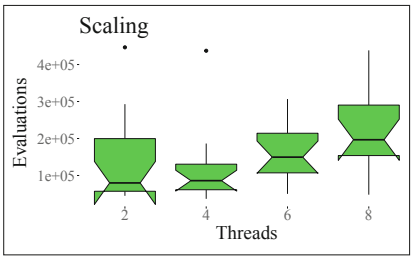

Fig. 4. OneMax: Number of evaluations vs. number of threads. Higher is better.

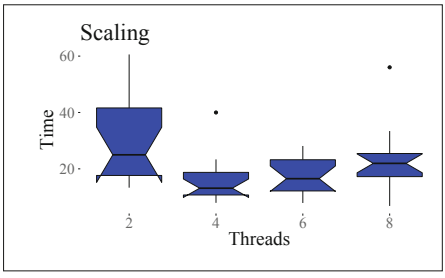

Fig. 5. OneMax: Total time vs. number of threads. Lower is better.

Besides, we also wanted to assess the actual wallclock time, plotted in Fig. 5. The picture shows that it decreases significantly when we go from 2 (the baseline) to 4 threads, since we are using more computing power for (roughly) the same number of evaluations. It then increases slightly when we increase the number of threads; as a matter of fact and as shown in Fig. 6, the number of evaluations per second increases steeply up to 6 threads, and slightly when we use 8 threads. However, the amount of evaluations spetn overcompensates this speed, yielding in a worse result. It confirms, nonetheless, that we are actually using all threads for evaluations, and if only we could find a strategy that didn't need more evaluations we should be able to get a big boost in computation time that scales gracefully to a high number of processors.

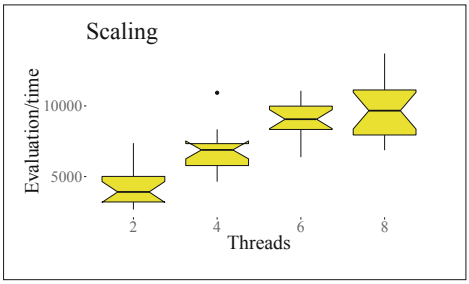

Fig. 6. OneMax: Evaluations per second vs. number of threads. Higher is better.

But we were interested also in checking whether the same kind of patterns are found for other fitness functions, with a different landscape, and also how much further scaling with the number of threads could go; that is why we set up another experiment with the Royal Road function. We run the experiment as above, but in this case there were some runs in which the solution was not found within the time limit of 800 s; in the first case, for two threads, this is indicated with a lighter shade corresponding to the number of instances where it did found the solution, 6 out of 15. The number of evaluations needed to find the solution is shown as a boxplot in Fig. 7; the total time in Fig. 8 and the number of evaluations per second in Fig. 9.

Since the Royal Road function is slightly heavier than Onemax, this number reaches a lower peak of approximately 25%. Comparing also Figs. 4 with 8 we see that that Royal Road needs one order of magnitude more evaluations than OneMax to find the solution. As we capped the running time at 800 s, this causes some the lack of success for the lowest number of threads.

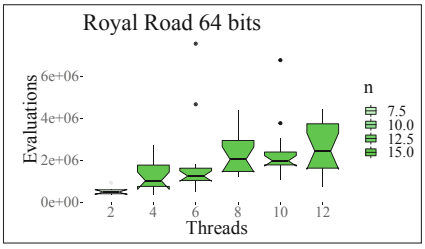

Fig. 7. Royal Road: Number of evaluations vs. num. of threads. Higher is better.

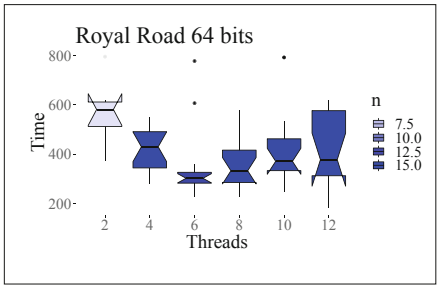

Fig. 8. Royal Road: Total time vs. number of threads. Lower is better.

The pattern these charts exhibit is very similar to those for OneMax. The performance scales up to a certain range, and then it plateaus, increasing only in speed, but not in wallclock time. From 2 to 6 threads speed increases 2×, due to an increase in speed with a slight decrease in the number of evaluations needed to find the solution; besides, success rate goes up from 2 to 4 threads, shifting from finding the solution 6 out of 15 times to finding it every time. This is due to the cap in the allowed runtime, which is 800 s; it is likely that if more time had been allowed, solution had been found.

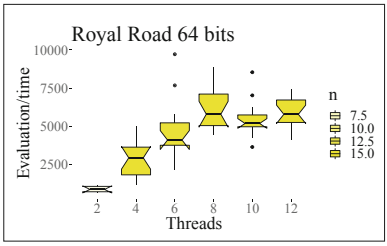

Fig. 9. Royal Road: Evaluations per second vs. number of threads. Higher is better.

However, we again find the same effect of increase in performance up to an optimal number of threads, 6 in this case. This is slightly better than it was for Onemax, which reached an optimal performance for 4 threads. The fact that the optimal population in Royal Road is 4× that needed for OneMax will probably have an influence. As a rule of thumb, performance will only increase to the point that population size will make the algorithm worsen in the opposite direction. Interestingly, it also proves that the peak performance is mainly due to algorithmic, not the physical number of threads (around 16 for this machine).

Besides, not all problems behave in the same way and scaling in performance and success rate is strongly problem-dependent. To illustrate that fact, we used another benchmark function, LeadingOnes, which counts the leading number of ones in a bitstring. Despite its superficial similarity to OneMax, it's in practice a more difficult problem, since it has got long plateaus with no increment in fitness. It is thus a more difficult problem which is why we had to increase the number of generations per thread to 16 and decrease the length of the chromosome to 48, as well as the time budget to 1200 s.

Even so, results were quite different, see Figs. 10, 11 and 12 (8 threads data not shown, because the solution was actually found with 2 and 4 threads only). The situation is inverted with respect to the other problems. Although, as shown in Fig. 12, the number of evaluations increases with the number of threads (and is actually higher than for Royal Road), the success rate *decreases* and the time to solution does the same.

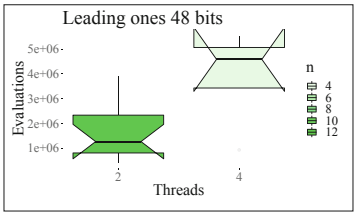

Fig. 10. Leading ones: Number of evaluations vs. num. of threads. Higher is better.

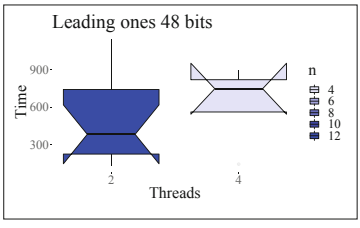

Fig. 11. Leading ones: Total time vs. number of threads. Lower is better.

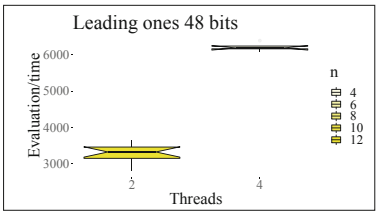

Fig. 12. Leading ones: Evaluations per second vs. number of threads. Higher is better.

This might be due, in part, to the intrinsic difficulty of the problem, with a flat fitness landscape that only changes if a single bit (among all of them) changes when it should; this might make the short periods of evolution before sending the message inadequate for this problem. But, additionally, the message only sends a statistical representation of the population. If there are not many representatives with the latest bits set, it can happen (and indeed it does) that the best solution in the previous evolution run is lost, and sometimes is not retrieved in the 16 generations it runs before communicating again. The latter gets worse with the increasing number of threads, since the population size decreases. As indicated at the beginning, balancing exploration and exploitation needs a population with the right size, and tipping the balance towards too much exploitation might be as negative (in terms of success rate) as too much exploration.

5 Conclusions

In this paper we studied to what extent scaling can be achieved on evolutionary concurrent algorithms on different types of problems. We observed that because OneMax simplicity requires a small population to find the solution, scaling is very limited and only means a real improvement if we use four threads instead of two. The Royal Road function with the same length is more challenging, showing an interesting behavior in the sense that it scaled in success rate and speed from 2 to 4 threads, and then again in speed through 6 threads. In this problem we measured up to 12 threads, and we noticed that the number of evaluations per second also reaches a plateau at around 6k evaluations. For 8 threads, our program uses actually 9 threads since there is another one for mixing. Since the computer only has 16 physical threads (2 threads x 8 cores), plus the load incurred by other system programs, this probably suggest a physical limit.

The experiments also indicate that creating a concurrent version of an evolutionary algorithm poses challenges such as designing the best communication strategy (including frequency and message formats) and distributing the algorithm workload, i.e. the population, among the different threads; anyway, physical features such as message size and number of threads must be considered.

As a consequence, several possible future lines of research arise. The first one is to try new algorithmic scaling strategies that consistently has a positive influence in the number of evaluations so that speedup is extended at least up to the physical number of threads. On the other hand, the messaging strategies proposed here are suitable for problem representation via a binary data structure. New mechanisms will have to be devised for floating-point representation, or other data structures. For that matter, general-purpose compressing techniques or EDAs extended to other abstract data types can be examined.

Finally, we are using the same kind of algorithm and parametrization in all threads. Nothing prevents us from using a different algorithm per thread, or using different parameters per thread. This opens a vast space of possibilities, but the payoff might be worth it.

Acknowledgements. This paper has been supported in part by projects DeepBio (TIN2017-85727-C4-2-P), TecNM Project 5654.19-P and CONACYT-PEI 220590.

References

1. Andrews, G.R.: Concurrent Programming: Principles and Practice. Benjamin/Cummings Publishing Company, San Francisco (1991)
2. Armstrong, J.: Concurrency Oriented Programming in Erlang (2003). http://ll2.ai.mit.edu/talks/armstrong.pdf
3. Barwell, A.D., Brown, C., Hammond, K., Turek, W., Byrski, A.: Using program shaping and algorithmic skeletons to parallelise an evolutionary multi-agent system in Erlang. Comput. Inform. **35**(4), 792–818 (2017)
4. Briggs, F., O'Neill, M.: Functional genetic programming and exhaustive program search with combinator expressions. Int. J. Know.-Based Intell. Eng. Syst. **12**(1), 47–68 (2008). http://dl.acm.org/citation.cfm?id=1375341.1375345
5. Castagna, G.: Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). CoRR abs/1809.01427 (2018). http://arxiv.org/abs/1809.01427
6. Dagum, L., Menon, R.: OpenMP: an industry-standard API for shared-memory programming. Comput. Sci. Eng. (1), 46–55 (1998)
7. delaOssa, L., Gámez, J.A., Puerta, J.M.: Migration of probability models instead of individuals: an alternative when applying the Island model to EDAs. In: Yao, X., et al. (eds.) PPSN 2004. LNCS, vol. 3242, pp. 242–252. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30217-9_25
8. García-Valdez, J.M., Merelo-Guervós, J.J.: A modern, event-based architecture for distributed evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO. ACM, New York (2018)
9. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. (TOPLAS) **7**(1), 80–112 (1985)
10. Gropp, W.D., Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface, vol. 1. MIT Press, Cambridge (1999)
11. Hawkins, J., Abdallah, A.: A generic functional genetic algorithm. In: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications. IEEE Computer Society, Washington (2001)
12. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978). https://doi.org/10.1145/359576.359585
13. Huelsbergen, L.: Toward simulated evolution of machine-language iteration. In: Proceedings of the First Annual Conference on Genetic Programming, GECCO 1996, pp. 315–320. MIT Press, Cambridge (1996)
14. Jiménez-Laredo, J.L., Eiben, A.E., van Steen, M., Merelo-Guervós, J.J.: EvAg: a scalable peer-to-peer evolutionary algorithm. Genet. Program. Evolvable Mach. **11**(2), 227–246 (2010)

15. Jin, H., Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using MPI and openmp on multi-core parallel systems. Parallel Comput. **37**(9), 562–575 (2011)
16. Kerdprasop, K., Kerdprasop, N.: Concurrent computation for genetic algorithms. In: 1st International Conference on Software Technology, pp. 79–84 (2012)
17. Lalwani, S., Sharma, H., Satapathy, S.C., Deep, K., Bansal, J.C.: A survey on parallel particle swarm optimization algorithms. Arab. J. Sci. Eng. **44**(4), 2899–2923 (2019)
18. Laredo, J.L.J., Castillo, P.A., Mora, A.M., Merelo, J.J.: Exploring population structures for locally concurrent and massively parallel evolutionary algorithms. In: 2008 IEEE Congress on Evolutionary Computation, pp. 2605–2612, June 2008
19. Laredo, J.L.J., Bouvry, P., Mostaghim, S., Merelo-Guervós, J.-J.: Validating a peer-to-peer evolutionary algorithm. In: Di Chio, C., et al. (eds.) EvoApplications 2012. LNCS, vol. 7248, pp. 436–445. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29178-4_44
20. Li, X., Liu, K., Ma, L., Li, H.: A concurrent-hybrid evolutionary algorithms with multi-child differential evolution and Guotao algorithm based on cultural algorithm framework. In: Cai, Z., Hu, C., Kang, Z., Liu, Y. (eds.) ISICA 2010. LNCS, vol. 6382, pp. 123–133. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16493-4_13
21. Merelo, J.J., García-Valdez, J.-M.: Going stateless in concurrent evolutionary algorithms. In: Figueroa-García, J.C., López-Santana, E.R., Rodriguez-Molano, J.I. (eds.) WEA 2018. CCIS, vol. 915, pp. 17–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00350-0_2
22. Merelo, J.J., García-Valdez, J.M.: Mapping evolutionary algorithms to a reactive, stateless architecture: using a modern concurrent language. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2018, pp. 1870–1877. ACM, New York (2018)
23. Rojas-Galeano, S., Rodriguez, N.: A memory efficient and continuous-valued compact EDA for large scale problems. In: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO 2018, pp. 281–288. ACM (2012)
24. Schippers, H., Van Cutsem, T., Marr, S., Haupt, M., Hirschfeld, R.: Towards an actor-based concurrent machine model. In: Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, pp. 4–9. ACM, New York (2009)
25. Sher, G.I.: Handbook of Neuroevolution Through Erlang. Springer, Heidelberg (2013). https://doi.org/10.1007/978-1-4614-4463-3
26. Swan, J., et al.: A research agenda for metaheuristic standardization. In: Proceedings of the XI Metaheuristics International Conference (2015)
27. Tagawa, K.: Concurrent differential evolution based on generational model for multi-core CPUs. In: Bui, L.T., Ong, Y.S., Hoai, N.X., Ishibuchi, H., Suganthan, P.N. (eds.) SEAL 2012. LNCS, vol. 7673, pp. 12–21. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34859-4_2
28. Valkov, L., Chaudhari, D., Srivastava, A., Sutton, C., Chaudhuri, S.: Synthesis of differentiable functional programs for lifelong learning. arXiv preprint arXiv:1804.00218 (2018)
29. Walsh, P.: A functional style and fitness evaluation scheme for inducting high level programs. In: Banzhaf, W., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1211–1216. Morgan Kaufmann (1999)