Testing the intermediate disturbance hypothesis in concurrent evolutionary algorithms

JJ Merelo[1], Mario García Valdez[2], and Sergio Rojas-Galeano[3]

[1] Universidad de Granada, Granada, Spain
jmerelo@ugr.es
[2] Instituto Tecnológico de Tijuana, Tijuana, Baja California, México
mario@tectijuana.edu.mx
[3] Universidad Distrital Francisco José de Caldas
Bogotá, Colombia
srojas@udistrital.edu.co

Abstract. Concurrency is a powerful abstraction that can be used to model and implement multi-deme evolutionary algorithms, opening up additional design questions such as what the different populations in various threads can do and how they interact with each other (via a combination of populations). One approach is synchrony: although threads can run asynchronously, they often perform the same amount of work, which brings a (rough) synchrony to appear within them. Our intention in this paper is to test if the intermediate disturbance hypothesis holds: this kind of synchrony is a small disturbance, which likewise big disturbances will not boost diversity; however, moderate disturbances will. We tested several ways of creating this intermediate disturbance by changing how different threads operate or modifying its working in alternative ways.

Keywords: Intermediate disturbance hypothesis · evolutionary algorithms · concurrency · distributed algorithms.

1 Introduction

The intermediate disturbance hypothesis was introduced by several researchers in the seventies [2] to explain the diversity found in different ecosystems; it states [18] that

> ... Moderate levels of disturbance foster greater species diversity than do low or high levels of disturbance

These levels refer to amplitude as well as frequency; a disturbance happening too often will wipe out all diversity, if it happens rarely the system reaches equilibrium bringing species that have some competitive advantage to dominate the ecosystem; in the same way, a high level of disturbance like sudden changes in temperature will not allow species to thrive except those that can stand them. In contrast, a moderate temperature range across day and night will allow a whole lot of adapted species to survive together, in the same way as it happens

in the rainforest. This disturbance may also refer to changes in the genetic pool brought in by the introduction of new populations.

Wider diversity is something that is much sought after in evolutionary algorithms, since diversity helps to avoid falling in local optima. This is also truth in multi-deme distributed or concurrent evolutionary algorithms, where *disturbances* will usually take the form of changes in the population, commonly selection or insertion policies. As a matter of fact, we investigated this kind of hypothesis in [14], finding that by evolving populations asynchronously (and thus, being in a different state of evolution), can result in an *intermediate* disturbance by interchanging members of the population; the latter in turn helps to boost the performance of the algorithm, obtaining better results than synchronously evolving populations or other scenarios with a lesser degree of disturbances.

Concurrent evolutionary algorithms are placed at a different level of abstraction than distributed algorithms; they can be run in parallel or not, distributed over many computers or simply many processors; nonetheless, they use high-level language constructs such as channels and messages that ensure that there are no deadlocks and every task runs smoothly until it's finished. Languages such as Raku (formerly known as Perl 6) [10], include channels as a built-in data structure, so by using them, it is feasible to implement concurrent evolutionary algorithms following Hoare's Communicating Sequential Processes model [8].

In this kind of evolutionary algorithms [12, 16, 13, 11] there is no *migration per se*, and although populations are not moved from one *island* to another, but *merged*, the intermediate disturbance principle may hold; therefore we should *let nature be our guide* [5] and apply it to validate if certain level of performance, scaling or both can be achieved.

However, there seems no way to introduce those intermediate disturbances from first principles, in such a way that it does not really change the nature of the algorithm in fundamental ways. For starters, in several occasions this intermediate disturbances happens inherently in asynchronous evolutionary algorithms [15], so adding further intermediate disturbance devices may probably amount to induce too much disturbance indeed, or maybe not. Thus, if we want to find out what actually works we should observe its effects and discard other possible causes; in a word, we will be trying to heuristically measure the effects of different types of disturbances on the population, which can only be made by changing the degree of evolution a population undergoes in every task. We will do that via alterations in the number of generations or in the population. In this way, we will pick the one that has the more positive outcome in algorithmic, performance terms, or both.

What we expect by altering the evolution degree of different populations is to create a disturbance, possibly of moderate size, when these populations mix. We will measure the effect and size of this disturbance indirectly by comparing the changes in the performance of the algorithm: higher diversity will mean better performance. Besides, we will check how this performance boost behaves when we scale from two to eight or ten simultaneous tasks.

The rest of the paper is organized as follows. Next, we will present the state of the art in the use or observation of the intermediate disturbance hypothesis in bioinspired algorithms; next we will describe the experimental setup for this paper. Section 4 will present the results, followed by a section with discussions and future lines of work that will close the paper.

2 State of the art

The intermediate disturbance hypothesis [2, 6] has not really had a long run in the area of bioinspired algorithms; instead, using *big* disturbances such as hypermutation, has been a preferred approach. This technique was popularized by another bioinspired methodology, immune systems, [9], which has been adopted under many different names such as the *plague* method that was proposed for genetic programming [3]. Even as their names suggest a degree of perturbation of the system that is beyond moderation, it is very likely that their real effect is that of a mild disturbance. The *plague*, for instance, just removes some individuals in the population; hypermutation does generate new (faulty) copies of some specific individuals, generating a moderate disturbance yielding the generation of diversity.

The same happens in the area of distributed evolutionary algorithms; these moderate disturbances are introduced, although not really by name. Pelta et al. [17] propose a "cooperative multi-thread strategy", where every thread runs a different optimization algorithm. The results of every thread will be a *moderate* disturbance to the solutions of the other threads, and this is achieved explicitly by using fuzzy rules to update thread method parameters as well as the starting point of the algorithm; however, while this strategy avoids low disturbance, it does not guarantee that catastrophic high disturbance happens, causing that a new solution much better than the existing ones, will *invade* the whole system.

This is why a more explicit approach to the observation or implementation of the intermediate disturbance hypothesis might help to better apply it to distributed evolutionary algorithms. As such, it has been used with a certain frequency by evolutionary algorithms, although mainly in the area of particle swarm optimization. Gao et al. [4] mention a *moderate disturbance strategy*, which adds exploration to the PSO algorithm; this *moderate* disturbance has the effect of increasing exploration and thus diversity, at least in a local scale; that is also the intention of the *intermediate disturbance* PSO proposed by He et al. [7], which perturbs the position of particles using the value initially proposed by Gao et al.

However, in the context of distributed evolutionary algorithms, it was introduced explicitly by us in the above-mentioned paper, [14], although we focused more on the algorithmic effects than in the real time-wise performance, since the implementation was not actually parallel. In that case, we compared populations operating synchronously with others that operated asynchronously by starting at different points in time; this simple procedure had the effect of a decrease in the number of evaluations needed.

A different implementation of the same concept was made in what we called a *multikulti* algorithm [1], which used an explicitly parallel population. This created the intermediate disturbance by using genotypic differences to select migrants from one island (in an island-based GA) to another; in that study, the moderate perturbation created by this inflow of population was explicitly proved to increase diversity, and consequently, performance.

In this paper we aim at adding to the state of the art by first, testing the intermediate hypothesis in a concurrent setting, trying to create different conditions that produce a moderate disturbance, and measuring its algorithmic as well as performance effects. We'll describe the setup next.

3 Experimental setup

The overall architecture of the algorithm used in this paper was presented in [16]. Essentially, it's a concurrent evolutionary algorithm written in Raku that uses channels for interchange of information among different tasks.

There are two kinds of tasks:

- The evolutionary task run a classical evolutionary algorithm for a number of generations with a fixed population. This is the task that we will scale.
- The mixer task will be used to interchange information among the evolutionary tasks; it picks up pairs of vectors representing probability distributions of populations, from the "mixer" channel, and then cross them over to create a new probability vector. The resulting vector along with a randomly chosen member of the original pair, will be sent back to the "evolutionary" channel.

The data flow is as follows.

- A set of probability vectors are initially generated. These will be the genewise Bernoulli probability distributions used to generate an initial binary population. We will generate 1 or 2 sets of probabilities more than the number of (evolutionary) threads; for instance, 12 vectors for 10 evolutionary threads.
- These sets are sent to the mixer thread, that mixes and sends them to the evolutionary channel.
- The evolutionary tasks pick up one probability vector, rebuild the population using it, and run an evolutionary algorithm for a fixed number of generations or until solution is found. If that's not the case, the top 25% fittest individuals of the population are used to estimate a new distribution probability vector, that is sent back to the mixer channel.
- Each task is event driven, so that they "react" when there's a vector available in the channel; it's implemented as a promise that will be fulfilled when the solution is found in that specific thread.

The timeline of this concurrent algorithm is shown in Figure 1; this monitor shows tasks as bars and specific events as triangles. The mixer task bar can be seen intermittent since every time it reacts it takes a very short time, but

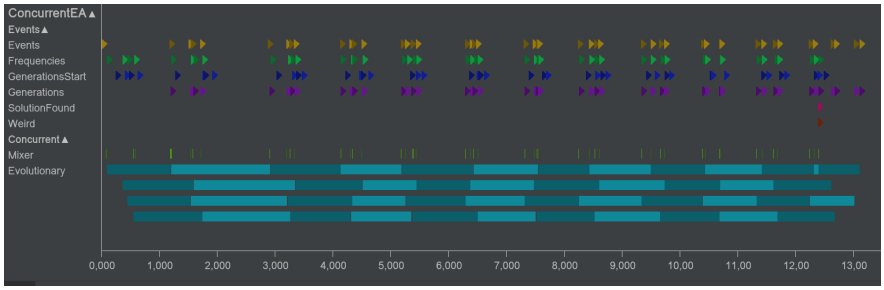

Fig. 1. Monitor showing the execution of the baseline concurrent evolutionary algorithm with 4 threads, as shown by the Comma IDE concurrency tool

nonetheless it keeps running in a single thread, and it starts before the others. Since generation (and mixer) start concurrently with evolution, new threads *wake up* as new information is available in the channel; they end at different points in the timeline, but their state of evolution should be approximately the same, since they have been arranged with the same number of evaluations and the population is the same. The monitor also shows that there are few gaps (except at the beginning) and that concurrency is use quite efficiently throughout all the run.

We will run every program 15 times with the same parameters, and will evaluate from 2 to 10 threads. For parametrizations where performance plateaus at 8, we will stop there. All experiments have been performed on a 8 CPU, 16 core system running Ubuntu 18.04 and Raku 2020.02.1. Experiment logs and resulting data are available in GitHub under a free license, as well as the scripts used to perform it. The version of Algorithm::Evolutionary::Simple used is 0.0.7. The benchmark problem used is Leading Ones with 60 bits; this problem is difficult enough to require a good-sized population to be solved. Results are probably extensible to other binary benchmark problems such as Royal Road or OneMax. In any case, since we are more interested in evaluating aspects of concurrent operation rather than the solving effectiveness of the algorithm, the specific problem is probably not as important. Population will be 1024, split among threads in the baseline configuration; for instance, there will be 512 individuals per thread for the 2-threads configuration.

In the next section, we will subject this baseline operation to different kinds of perturbations, looking for that Goldilocks zone where perturbations are moderate and wider diversity emerges.

4 Results

The number of generations was established in the initial papers as one with the best communications/computation tradeoff. However, here we decided to establish a baseline of non-perturbed values for comparison. A boxplot of the number

```
## Error in library(rstatix): there is no package called 'rstatix'
## Error in library(tidyverse): there is no package called 'tidyverse'
```

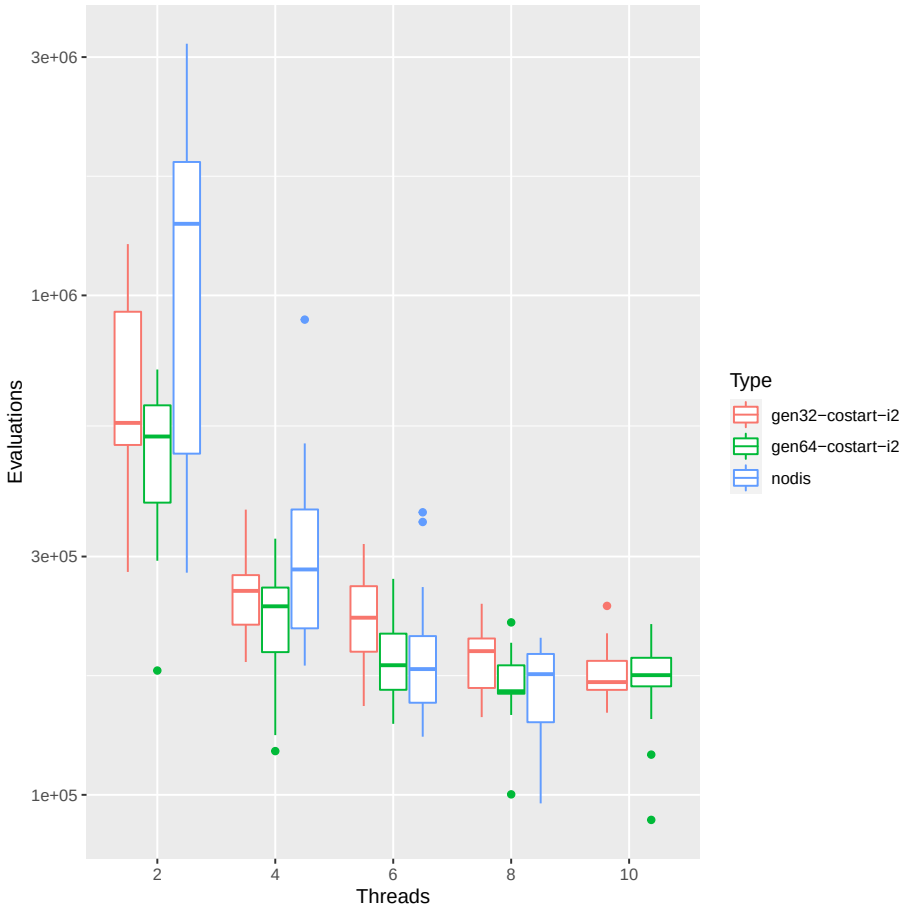

Fig. 2. Boxplot of the number of evaluations for 16 (labeled "nodis"), 32 and 64 generations.

of evaluations vs. threads is shown in Figure 2. First, we can check that the number of evaluations scales down with the number of threads, although it plateaus at 16 generations. Parallelism also evens out the differences between different parameters: while stopping for transmission at 16 generations is notably worse at 2 and 4 threads, the difference is not remarkable beyond that. Differences for 32 and 64 generations are never noticeable. We don't show figures for wall clock time taken, but it is notably worse; this is simply explained by the fact that longer stretches of running mean less communication between tasks.

From this baseline, subsequently we introduced a small perturbation: a single thread will use 9 or 25 generations; the rest of the threads will use 16. We will show results in Figure 3. Essentially, this change does not seem to affect the

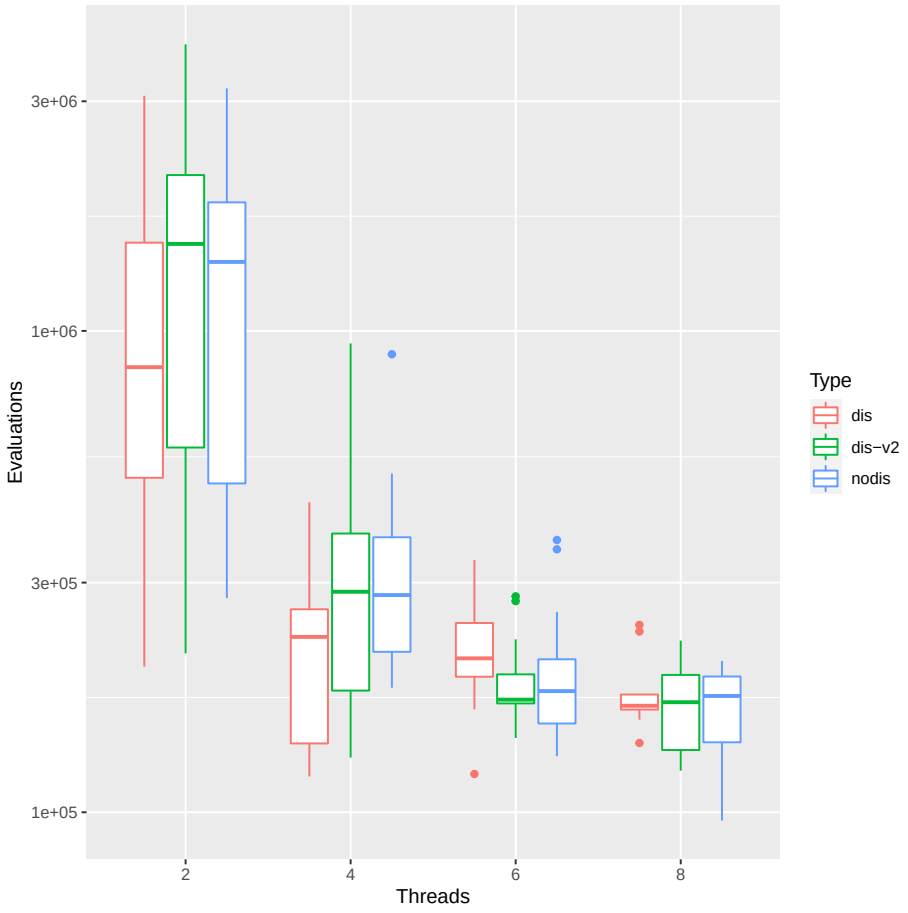

Fig. 3. Boxplot of the number of evaluations for 16 (labeled "nodis") and a "disturber" task that ends after 9 (`dis-v2`) or 25 (`dis`) generations.

average number of evaluations, and when it does, it is for the worse. The "late" thread, labeled `dis`, actually needs more evaluations to find the solution, but only for 4 threads. The rest of the results are basically the same. Thus we reason this qualifies as a small disturbance: it does not cause a noticeable change; this implies that the introduction of populations with differentiated phases in the evolution, as was done in [14], does not have any impact in this setting. It could be due to the fact that we are scaling up to 8 threads, but as a matter of fact,

it does not make any difference for 2 threads either. Notice that, as previously, there's no improvement from 6 to 8 threads, which is why we have not scaled up to 10 threads this time.

Let us try now another approach to creating disturbances. We will randomly vary the number of generations ranging randomly from 8 to 24, or the population, which will be sampled from the probability distribution vector to a number between half the baseline value (1024 / number of threads) to twice that value. A boxplot of the number of evaluations is shown in Figure 4 and the wall clock time is shown in Figure 5.

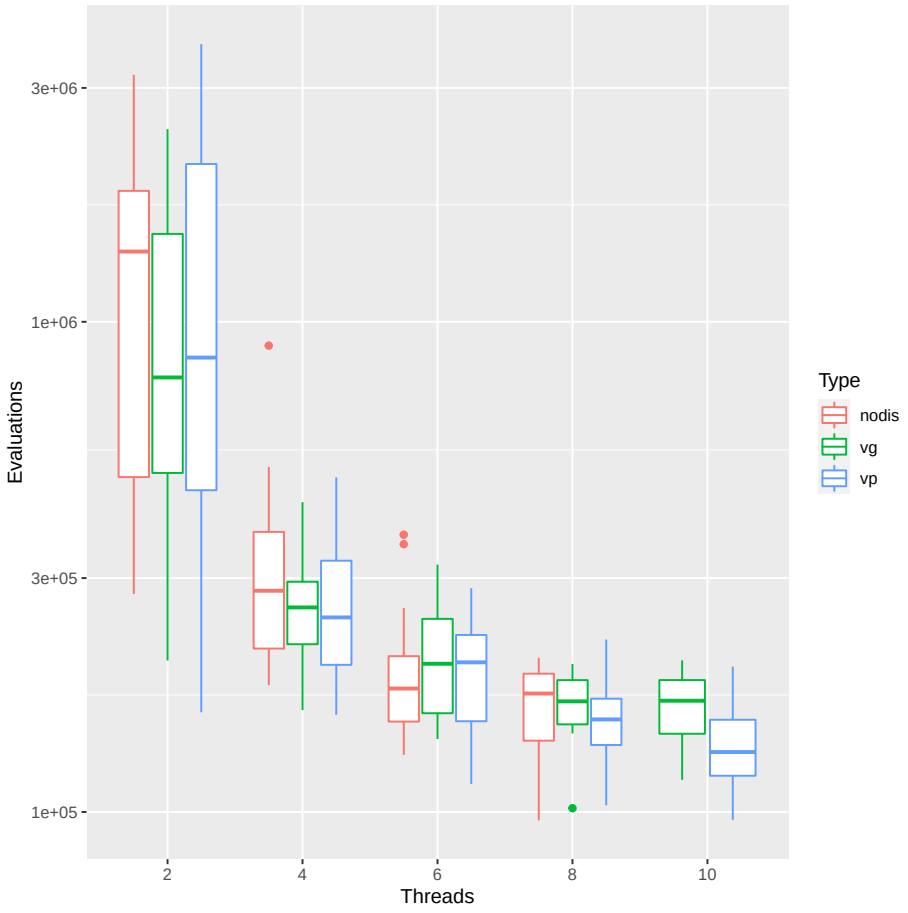

Fig. 4. Boxplot of the number of evaluations for 16 (labeled "nodis") and tasks with a variable population `vp` and variable number of generations `vg`

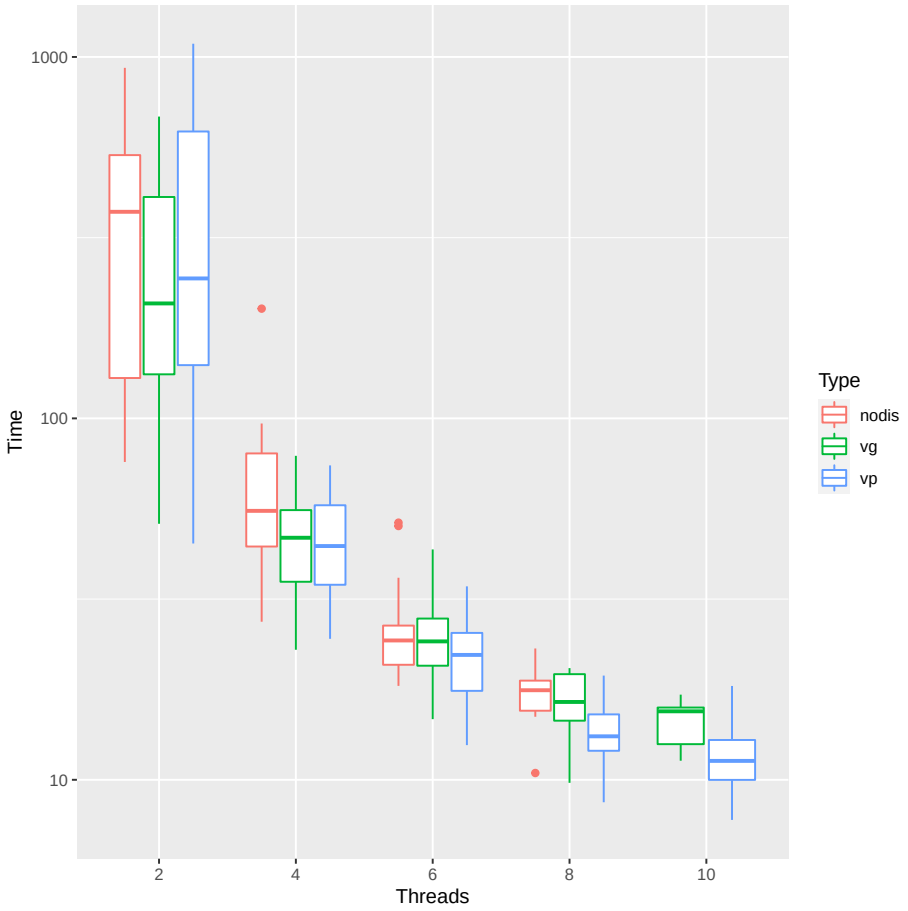

Fig. 5. Boxplot of the clocked time for experiments for 16 (labeled "nodis") and tasks with a variable population `vp` and variable number of generations `vg`. Please notice that the y axis is logarithmic.

Despite the median being slightly better when we use a variable population, the statistical variability of the results show no significant difference in the number of evaluations obtained for up to 8 threads between the baseline and the variable population version (there might be some improvement for 10 threads, but we didn't measure that); however, there is a significant difference in time (at the 10% level) for 4 and 8 threads; this can also be appreciated in the chart.

On the other hand, the difference in the number of evaluations is significant between variable population and variable number of generations for 10 threads, with the time needed showing a significant difference for 8 and 10 threads. Both effects are compounded in the number of evaluations per second, which is shown in Figure 6. In this case, the difference is significant for 2 to 8 threads; the median

number of evaluations per second for variable population tops up at 12254.234, while it's 9408.067 for the baseline without any kind of disturbance.

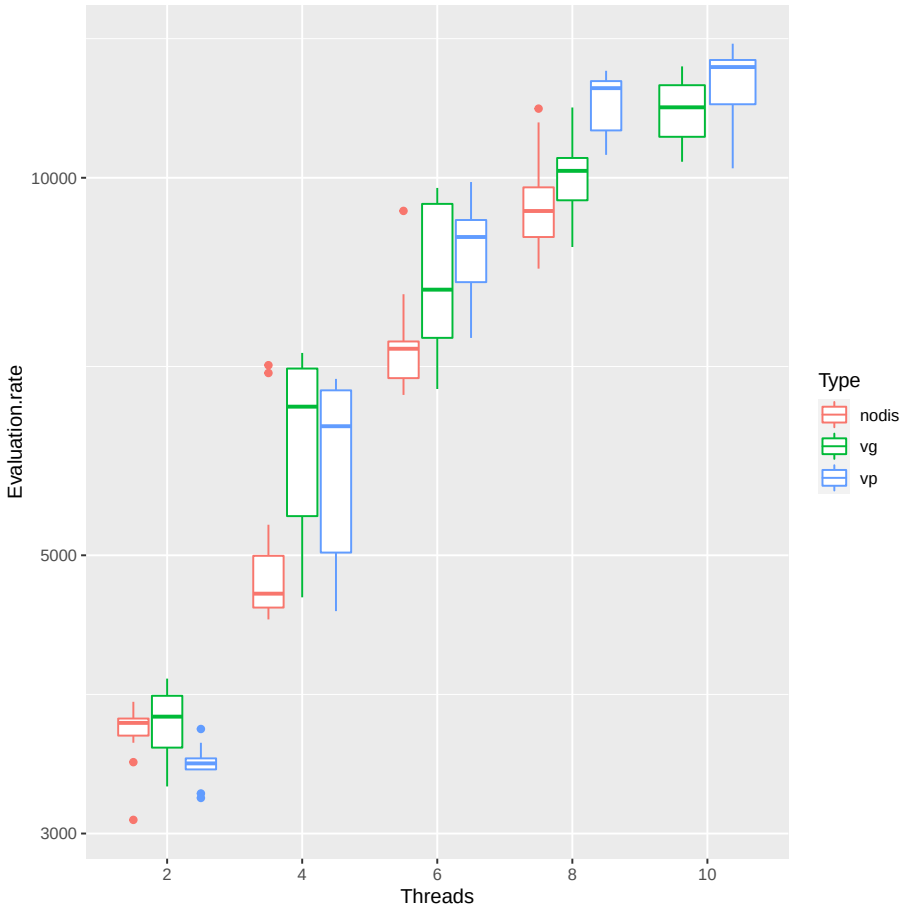

Fig. 6. Boxplot of the evaluation rate (evaluations per second) for experiments for 16 (labeled "nodis") and tasks with a variable population `vp` and variable number of generations `vg`, with a logarithmic y axis.

5 Discussion and conclusions

In this paper, we evaluated how creating a moderate amount of disturbance in a concurrent and multi-threaded evolutionary algorithm contributed to the diversity, and indirectly to the results of the algorithm across a wide scale; we have made several experiments changing the degree of evolution that every individual

population undergoes, with the intention of introducing disturbances by mixing populations that had been in different stages. In some cases, minimal differences were achieved with respect to the baseline system, and only for some parts of the scale-up to 10 threads.

However, using a strategy that generated a population of random size in a range that decreased with the number of threads proved to be computationally more efficient (finding the problem solution in fewer evaluations across a whole range of scales) but also faster (by boosting the number of evaluations per second), achieving a result that is overall much faster by an order of magnitude; while the median time for the baseline system and 2 threads was 372.94505s, the variable population with 10 threads managed to find the solution in a median time of 11.27906, 33 times faster (instead of just 5 times faster, which would be expected by the scale-up of the number of threads). This leads us to note that, at least this strategy, is efficient enough to be on a par with the state of the art.

Nonetheless, the interesting question is if it is a moderate disturbance or some other factor that achieved the result. Since we are using a random value in a range that goes from half the baseline population to twice the baseline population, on average, the population will have a bigger size; the ratio of evaluations to communications is then lower, and that might explain the small edge when the number of evaluations is not significantly better. However, there are cases when the number of evaluations is actually better, mainly at higher concurrency values. Arguably, this could constitute a moderate disturbance. As the population gets smaller, the range of variation with respect the total population also gets smaller; while for two threads, for instance, we could have one thread with 300 and the other with 900, and the difference between them is going to be 60% of the total population, for 8 threads the differences of a population between tasks is going to be a moderate size with respect to the total population. This would explain the good results happening at the bigger scales, and also the fact that there is an improvement for that strategy across all scales, from 2 to 10.

We conclude that intermediate disturbance in concurrent evolutionary algorithms introduces many different degrees of freedom in the system, and exploring them looking for good performance and scaling is still a challenge once we've established a good baseline performance measure. However, once we know the degree of disturbance that works the best, we could extend measurements to other benchmark functions, and also try to max out the number of threads available in a single machine. This is left as future work.

6 Acknowledgements

We are grateful to Jonathan Worthington and the rest of the Edument team for their disposition to help with implementation problems and provide suggestions to make this work correctly. This is extended to the rest of the Raku development team, which is an excellent and technically knowledgeable community committed to creating a great language. This paper has been supported in part by projects DeepBio (TIN2017-85727-C4-2-P).

References

1. Araujo, L., Guervós, J.J.M.: Diversity through multiculturality: Assessing migrant choice policies in an island model. IEEE Trans. Evolutionary Computation **15**(4), 456–469 (2011)
2. Connell, J.H.: Diversity in tropical rain forests and coral reefs. Science **199**(4335), 1302–1310 (1978)
3. Fernandez, F., Vanneschi, L., Tomassini, M.: The effect of plagues in genetic programming: A study of variable-size populations. In: European Conference on Genetic Programming. pp. 317–326. Springer (2003)
4. Gao, H., Zang, W., Cao, J.: A particle swarm optimization with moderate disturbance strategy. In: Proceedings of the 32nd Chinese Control Conference. pp. 7994–7999. IEEE (2013)
5. Goldberg, D.E.: Zen and the art of genetic algorithms. In: Schaffer, J.D. (ed.) ICGA. pp. 80–85. Morgan Kaufmann (1989)
6. Grime, J.P.: Competitive exclusion in herbaceous vegetation. Nature **242**(5396), 344–347 (1973)
7. He, Y., Wang, A., Su, H., Wang, M.: Particle swarm optimization using neighborhood-based mutation operator and intermediate disturbance strategy for outbound container storage location assignment problem. Mathematical Problems in Engineering **2019** (2019)
8. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (Aug 1978). https://doi.org/10.1145/359576.359585
9. Jansen, T., Zarges, C.: Analyzing different variants of immune inspired somatic contiguous hypermutations. Theoretical Computer Science **412**(6), 517–533 (2011)
10. Lenz, M.: Perl 6 Fundamentals. Springer (2017)
11. 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.) Applied Computer Sciences in Engineering. pp. 17–29. Springer International Publishing, Cham (2018)
12. 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. pp. 1870–1877. GECCO '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3205651.3208317, http://doi.acm.org/10.1145/3205651.3208317
13. Merelo, J.J., Laredo, J.L.J., Castillo, P.A., García-Valdez, J.M., Rojas-Galeano, S.: Scaling in concurrent evolutionary algorithms. In: Workshop on Engineering Applications. pp. 16–27. Springer (2019)
14. Merelo, J.J., Mora, A.M., Castillo, P.A., Laredo, J.L.J., Araujo, L., Sharman, K.C., Esparcia-Alcázar, A.I., Alfaro-Cid, E., Cotta, C.: Testing the intermediate disturbance hypothesis: Effect of asynchronous population incorporation on multi-deme evolutionary algorithms. In: Rudolph, G., Jansen, T., Lucas, S., Poloni, C., Beume, N. (eds.) Parallel Problem Solving from Nature - PPSN X. LNCS, vol. 5199, pp. 266–275. Springer, Dortmund (13-17 Sep 2008). https://doi.org/10.1007/978-3-540-87700-4_27
15. Merelo-Guervós, J.J., García-Sánchez, P.: Modeling browser-based distributed evolutionary computation systems. CoRR **abs/1503.06424** (2015), http://arxiv.org/abs/1503.06424

16. Merelo Guervós, J.J., Laredo, J.L.J., Castillo, P.A., Valdez, J.M.G., Rojas-Galeano, S.: Exploring concurrent and stateless evolutionary algorithms. In: Kaufmann, P., Castillo, P.A. (eds.) Applications of Evolutionary Computation - 22nd International Conference, EvoApplications 2019, Held as Part of EvoStar 2019, Leipzig, Germany, April 24-26, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11454, pp. 405–412. Springer (2019). https://doi.org/10.1007/978-3-030-16692-2_27, https://doi.org/10.1007/978-3-030-16692-2_27
17. Pelta, D., Sancho-Royo, A., Cruz, C., Verdegay, J.L.: Using memory and fuzzy rules in a co-operative multi-thread strategy for optimization. Information Sciences **176**(13), 1849–1868 (2006)
18. Reece, J.B., Urry, L.A., Cain, M.L., Wasserman, S.A., Minorsky, P.V., Jackson, R.B., et al.: Campbell biology. No. s 1309, Pearson Boston, MA (2014)