J Supercomput (2017) 73:2214–2228
DOI 10.1007/s11227-016-1923-4

Multithreaded computing in evolutionary design and in artificial life simulations

Maciej Komosinski[1] · **Szymon Ulatowski**[1]

Published online: 25 November 2016

Abstract This article investigates low-level and high-level multithreaded performance of evolutionary processes that are typically employed in evolutionary design and artificial life. Computations performed in these areas are specific because evaluation of each genotype usually involves time-consuming simulation of virtual environments and physics. Computational experiments have been conducted using the Framsticks simulator running a multithreaded version of a standard evolutionary experiment. Tests carried out on five diverse machines and two operating systems demonstrated how low-level performance depends on the number of physical and logical CPU cores and on the number of threads. Two string implementations have been compared, and their raw performance turned out to fundamentally differ in a multithreading setup. To improve high-level performance of parallel evolutionary algorithms, i.e. the quality of optimized solutions, a new distribution scheme that is especially useful and efficient for complex representations of solutions—the *convection distribution*—has been introduced. This new distribution scheme has been compared against a random distribution of genotypes among threads that carry out evolutionary processes.

Keywords Concurrency · Multithreading · Simulation · Evolution · Optimization · Performance

✉ Maciej Komosinski
maciej.komosinski@cs.put.poznan.pl

[1] Institute of Computing Science, Poznan University of Technology, Piotrowo 2, 60-965 Poznan, Poland

1 Introduction

In the areas of evolutionary design and artificial life, evolutionary processes [5,7] are used to optimize designs (structures, constructs) or to mimic the biological world. In both cases, computer simulation plays a key role. However, simulating physics requires intensive computation, and the more the detail is expected, the more the computation is necessary. Fortunately, it is often possible to divide the simulated system into independent parts so that computations can be performed in parallel. This opens up a way to speed up the process of simulation, but still requires a lot of computing power and some method of distribution among multiple processors.

The most trivial way to distribute computation in evolutionary processes with a single gene pool is the master–slave architecture [1,4,17,32,37] where slaves perform the time-consuming evaluation of genotypes, and the master performs selection, crossing over, and mutation (Fig. 1). This can be generalized into a coarse-grained architecture [1,4] where slaves perform separate and independent evolutionary processes. In the approach described here, slaves occasionally send their results to the master process that performs migration (receives genotypes and redistributes them back to slaves), but there is no direct communication between slaves.

For tests and computational experiments, the Framsticks simulator [25] is employed here. Since its initial releases in 1996, this simulator has been used as a computing engine in a number of diverse applications, including comparison of genetic encodings in artificial life and evolutionary design [23], estimating symmetry of evolved and designed agents [14], employing similarity measure to organize evolved constructs [18,20], bio-inspired visual-motor coordination [15] and real-time coordination, modeling robots [31] and optimizing fuzzy controllers, user-driven (interactive, aesthetic) evolution, synthetic neuroethology [27,28], analyses of brain activity evoked by perception of biological motion [34,35], modeling perception of time in humans [21], modeling foraminiferal genetics, morphology, simulation, and evolution [22], and modeling communication, predator–prey coevolution, speciation, and other biological phenomena [6]. These applications and the fact that Framsticks core is implemented in a low-level language (C++) for high efficiency but also features a higher-level scripting language, make it a representative example of software that is used for modeling and simulation of life, and in particular, evolutionary processes [19].

Many of the applications enumerated above require considerable amounts of computing power, and in most cases the more the computing resources are available,

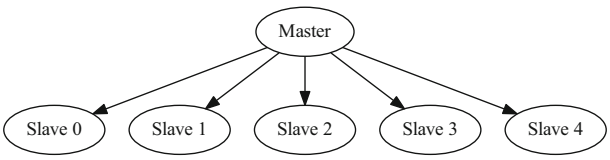

Fig. 1 A basic parallel architecture typically used for multithreaded evolutionary experiments. In the most trivial scenario, slaves evaluate genotypes, and the master process performs mutation, crossover, and selection. In a more complex setup used in this work, slaves are independent evolutionary processes, and the master thread migrates genotypes between slaves

the more meaningful the experiments and their results are. With modern computers equipped with many processors and cores and a clear direction of hardware development in the near future, using multithreading allows to exploit more computing power on a single machine in a single experiment.

The Framsticks environment allows for a flexible configuration of the way computing is parallelized, distributed, and organized. Multi-level and hybrid architectures are possible both in centralized and distributed scenarios [1,17,37], because every Framsticks server can perform multithreaded computation, genotype transfer, or both [24]. This paper focuses on a basic one-machine architecture shown in Fig. 1: the master thread can create, delete, and control slave threads. The master thread does not perform any continuous work and only redistributes genotypes among slaves during migrations. Unlike the traditional master–slave evolutionary algorithm where the master process runs the optimization algorithm and slaves evaluate individual solutions, here slaves perform independent evolutionary processes. For distributed configurations, this architecture is typically employed in machines that are end nodes. This is where the biggest gain can be achieved from parallel optimization, and this is currently the most popular architecture among researchers that do not use specialized high-end configurations.

The goal of this work is to investigate and improve both low-level and high-level performance of parallel evolutionary algorithms in optimization of solutions that need much computational power to evaluate. Optimizing three-dimensional structures in evolutionary design and in artificial life is an example of such applications. Section 2 focuses on technical aspects of raw, low-level performance of multithreaded evolutionary optimization and investigates how the number of threads, gene pool capacity, CPU architecture, and string implementation influence the number of evaluated genotypes. Section 3 focuses on high-level performance of parallel evolutionary algorithms—performance is no longer measured as the number of genotype evaluations; it is rather the actual fitness that is achieved by evolutionary optimization. To improve the quality of optimized solutions, a new distribution scheme of genotypes among slave processes is introduced and tested. Section 4 summarizes this work.

2 Multithreading performance

All tests reported in this work were performed using the `standard-mt` experiment definition—a multithreaded version of the most common and versatile Framsticks evolutionary optimization experiment [24]. This experiment script performs physical simulation of creatures built from genotypes that are mutated and crossed over in the course of a steady-state (i.e., non-generational) evolution [16,26,38]. The most computationally expensive part of the optimization process is the evaluation of fitness of each genotype. This evaluation is based on the creature's performance in the simulated physical world. The number of evaluations performed in the fixed amount of time (500 seconds) is our measure of performance.

Each test has been run for varying thread count and varying values of *capacity*. *Capacity* is the size of the slave gene pool (the number of genotypes) and as such, it influences the dynamics of the evolutionary optimization process.

In the multithreaded implementation, *capacity* and *mix_period* parameters determine the migration frequency. A migration occurs after reaching the desired number of genotype evaluations, expressed as the percentage of the gene pool *capacity*: there are $capacity \times mix\_period/100$ evaluations between migrations. For example, for the default value of $mix\_period = 1000$ that is used in all the experiments in this section, the number of evaluations performed by each slave between migrations is $10 \times capacity$ of the gene pool.

In order to precisely compare raw performance across multiple runs, the actual genetic optimization has been disabled by removing the sources of genotype variability—only the evaluation (simulation) and selection is performed, continuously operating on identical genotypes of medium complexity. The amount of memory required by all threads combined was at least one order of magnitude smaller than the amount of available RAM, and slaves did not perform any file or network operations.

The experiments were run on five machines—a choice of laptop, desktop, and server computers running Linux and Windows:

- **4/4-L.** Desktop 4-core Intel Core2 Quad processor Q6600 [9] running Linux Debian x86_64 3.14-2-amd64 and having its CPU clock forced to a constant frequency of 1.6 GHz for better reproducibility.
- **4/4-W8.** Desktop 4-core Intel Core i5-2500 [10] running 64-bit Windows 8.1 Pro in safe boot mode which forced the CPU clock to a constant frequency of 3.3 GHz.
- **4/8-W7.** Desktop 4-core 8-thread Intel Core i7-4790 processor [13] running 64-bit Windows 7 Pro in safe boot mode which forced the CPU clock to a constant frequency of 3.6 GHz.
- **4/8-W8.** Laptop 4-core 8-thread Intel Core i7-4700MQ processor [12] running 64-bit Windows 8.1 Pro in safe boot mode which forced the CPU clock to a constant frequency of 2.4 GHz.
- **16/32-L.** Server 8-core 16-thread ($\times 2$ CPU) Intel Xeon E5-2660 processor (max turbo frequency 3 GHz) [11] running Linux Ubuntu x86_64 3.2.0-52-generic.

This choice of machines corresponds to configurations that are currently most often used by researchers for evolutionary experiments, either as standalone computers, or as end-nodes in a hybrid distributed evolutionary architecture. This also allows to study in detail the most interesting range, from 4-core to 16-core machines, with and without hyper-threading technology [8,29], and investigate relationships between the number of cores, threads, and the resulting performance.

In all 3D charts presented in this section, red, green, and blue surfaces demonstrate 40%, 70%, and 100% progress of the experiment, respectively. The blue surface presents the final results, while red and green surfaces are approximate and only shown to illustrate progress at intermediate stages of the experiment.

2.1 String implementation: reference counting vs. copying

In most computer programs, manipulating strings (sequences of characters) is a frequent and ubiquitous operation. The two basic approaches to managing string contents are characterized in Table 1. Usually, the physical simulation and the simulation of neural networks do not use strings at all. On the other hand, a typical evolutionary

Table 1 Two basic approaches to managing string contents that were compared in computational experiments

	COW (copy-on-write)—uses reference counting	PU (private unprotected)—allocates private character buffers for individual string objects
Synchronization in a multi-threaded environment	Reference counter is protected by a mutex	No synchronization necessary
String copy operation (constructor, assignment)	Only a reference is copied—faster for long strings	String contents must be copied—faster for short strings
Memory	Efficient: usually stores only one copy of each unique string contents	Inefficient: stores each string content separately

optimization algorithm processes genotypes that are strings, and even if they remain unchanged, they are copied. The scripting subsystem of each application likely relies heavily on processing of strings. It is impossible to predict which of the two string implementations is more efficient and how big is the difference on various machines; hence we compared the performance of both in practical, multithreaded evolutionary experiments using Framsticks.

The experiments showed that the COW string implementation was seriously limiting multithreaded performance [36] because synchronization was based on a single *pthreads* mutex [3] shared between all strings. When increasing the number of CPU cores, the PU string implementation enabled a nearly linear parallelization speedup of the evolutionary experiment and did not introduce any significant memory footprint. The PU approach was up to 1.35x faster than COW for the 4/4-L machine and up to 10.8x faster for the 16/32-L machine. The difference in performance of both string implementations on the 16/32-L machine is illustrated in the two bottom rows in Fig. 2.

This speedup was possible because the synchronization of string access was no longer necessary, and physical simulations and evolutionary algorithms can perform independent computations in each thread, with only a small minority of operations involving inter-thread communication and synchronization. Therefore, the PU string implementation was used in all the experiments in the following sections. The need for locking (which causes delays) is the price paid for the ability to access shared memory space, which is not the only possible implementation. Another possibility would be to use operating system processes instead of threads—this would provide a complete separation between processes, but at the same time would make it more difficult to access shared data during master-slave interactions.

2.2 Simulation and evolution

Since evolutionary processes and simulation performed by individual threads are highly independent and there is nearly no additional locking and synchronization

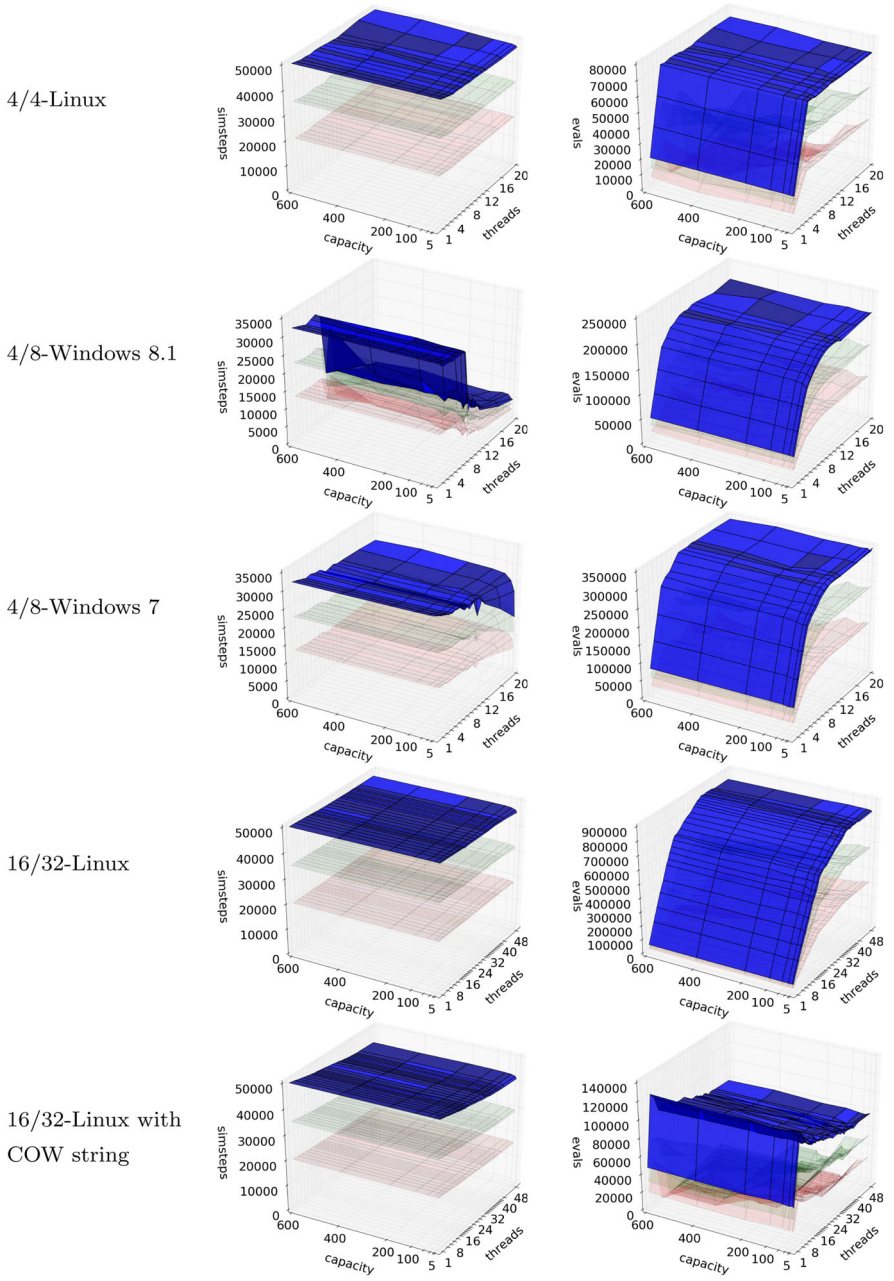

Fig. 2 The number of master simulation steps (*left column*) and total slave genotype evaluations (*right column*) for four machines (the characteristics for the 4/4-W8 machine were similar to the 4/8-W8 machine). Slave gene pool capacities are 5, 10, 25, 50, 100, 200, 400, 600. The number of threads varies from 1 to 20, and for the 16/32-L machine, from 1 to 48. We show results for thread count larger than the number of cores to enable direct comparisons between machines

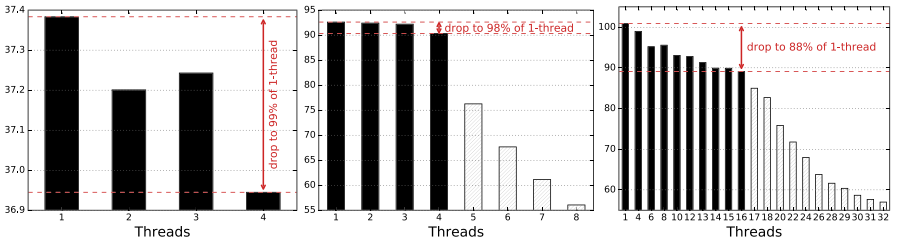

Fig. 3 Genotype evaluation count per second per thread (excluding migration time) for configurations 4/4-L, 4/8-W8, and 16/32-L. Physical cores are shown in black

apart from migrations, an almost linear speedup can be achieved when the number of threads is increased. This can indeed be observed in Fig. 2 except for the COW string implementation discussed in the previous section.

Our performance measurements were able to reveal key properties of CPU architectures and their quantitative influence on evolutionary performance. For configuration 4/4-L that has 4 physical cores, there are two nearly linear segments in the performance shown in the right column in Fig. 2: a linear slope for 1–4 threads and a nearly constant performance for 4+ threads. Configurations 4/8-W7, 4/8-W8, and 16/32-L feature hyper-threading technology (Intel's simultaneous multithreading implementation, SMT [8,29]) and can execute two simultaneous threads on each core, with slightly less performance compared to one thread per core, as additional threads share the same CPU hardware resources. This technology yields two nearly linear slopes: one for increasing the number of threads up to the number of physical cores, and another, less steep slope, up to twice the number of cores. From this perspective, the CPU behaves as if it contained additional, less capable, "logical" cores.

This influence of the hyper-threading technology is also demonstrated in Fig. 3, which compares raw per-thread performance for varying thread count. This analysis excludes the migration slowdown caused by the short periods when all slave threads are stopped—migrations are an important part of the experiment, but they do not contribute to our performance measure which is the genotype evaluation count. The parallelization speedup is not exactly linear, but quite close, especially for the quad-core configuration 4/4-L where the combined 4 threads' performance was just 1% lower than 4× single thread. Parallelization speedup is similar in configuration 16/32-L where 4 simultaneous threads also reached 99% of the theoretical ideal, but the speedup dropped to 88% for 16 threads.

When tested across different values of the gene pool capacity, smaller gene pools experience more frequent migrations (Fig. 4), as the number of genotypes created and evaluated between migrations is proportional to the gene pool capacity. For a given capacity, the number of migrations decreases with a decreasing processing power of a single slave thread (i.e., with an increasing number of threads), but it is also influenced by the master thread being delayed because of slave threads consuming more processing power. Any such delay increases the time interval between migrations decreasing the number of migrations, which is especially visible for small capacities where the migration period is short. While the number of performed migrations varies

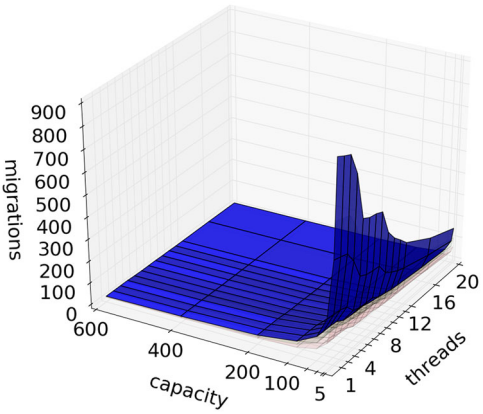

Fig. 4 Migration count for configuration 4/8-W8. This relationship looked similar for all configurations

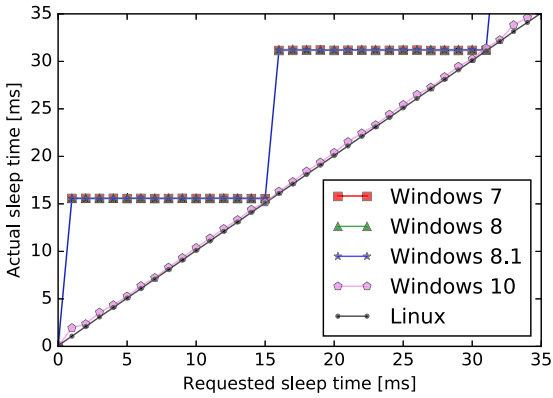

Fig. 5 The interval the thread waits to be resumed after a "sleep" function is called. Windows 7, 8 and 8.1 behave in exactly the same way. Linux provides a practically perfect, linear characteristic. Windows 10 (not employed in performance tests) makes a thread wait for slightly longer than requested. Lines between points are only guides for the eye

highly as Fig. 4 shows, the influence on the number of evaluated genotypes is so minimal that it cannot be noticed in the right column in Fig. 2.

The left column in Fig. 2 shows the number of simulation steps performed by the master thread. For most of the time, the master thread's job in this evolutionary experiment is just waiting for slave events, which means the number of steps is not correlated to the actual amount of work performed in the experiment. It does, however, show the different handling of the master thread depending on the CPU load and the operating system scheduling policy (cf. [33]).

To avoid unnecessary use of the CPU by the master thread, this thread was asked to sleep for 10 milliseconds in each simulation step. Given the test duration of 500 seconds and the 10-ms delay, the expected number of steps in ideal conditions is $500/0.01 = 50{,}000$. The actual number of steps varies; it is close to 50,000 on Linux machines 4/4-L and 16/32-L, and it does not exceed 35,000 on Windows machines 4/8-W7 and 4/8-W8, which on average sleep for 5.6 milliseconds more than requested [30] as illustrated in Fig. 5.

In configurations 4/4-L, 4/8-W7, and 16/32-L (but not in 4/8-W8), the master thread performance barely decreases under increased slave thread load. This suggests that

the operating system measures the actual CPU time used by each individual thread and schedules accordingly. The master thread, waiting most of the time, uses less CPU than its fair share and, therefore, it is not limited by the CPU shortage when the share decreases with an increasing thread count. The positive side effect of such behavior is that the master thread latency during the slave event handling is minimized. This does not seriously influence the experiment (except for, perhaps, slightly disrupting the migration count by delaying migrations), because the amount of the useful work depends almost entirely on the performance of slave threads. Configuration 4/8-W8 was the only laptop machine; as such, its mobile processor did not have an integrated heat spreader like desktop processors had, and this might influence the way the operating system scheduler allocated a busy CPU to a mostly idle thread. These differences between platforms in the way the master thread is managed do not affect the number of evaluated genotypes and the performance of the evolutionary process; if a particular scheduling behavior were required by an application, it can be enforced by setting priorities of threads and processes accordingly.

3 Convection distribution scheme

Since 1980s, a number of parallel evolutionary architectures have been proposed and implemented, differing in the way the population is decentralized, the topology of connections between nodes, their roles, and the way migration of genotypes is performed [1,4,17,32,37]. The most trivial approach to distributing genotypes to subpopulations (slaves) in centralized (master-slave) and coarse-grained architectures is to send to each slave the entire gene pool, or a random sample of the entire gene pool. This approach leverages the raw power of parallel evolutionary processes, but does not take advantage of any specific logic like migrating best genotypes [1,17,37].

In this section a new distribution scheme is introduced, namely the *Convection distribution* scheme. This way of distributing genotypes is especially valuable for applications in evolutionary design and artificial life, where solutions are extremely complex due to sophisticated genotype-to-phenotype mapping, and it is much easier to migrate genotypes based on known fitness values or measured performance than on compound phenotypic (e.g., morphological) characteristics. The proposed approach differs from the random distribution of genotypes in that it controls the selective pressure in each slave—each slave receives genotypes that share similar fitness, and this approach still does not require any direct communication between slaves. This distribution scheme is called *convection* because it facilitates continuous evolutionary progress just like a convection current or a conveyor belt: each slave always tries to independently improve genotypes of a specific fitness range which overall ensures more fitness diversity and avoids the domination of (and the convergence towards) the current globally best genotypes [5]. These occasional, short ascending trends (convections) are visible in the entire range of fitness values in Fig. 9. As an effect of slave–master–slave migrations, genotypes move from one slave to another and they can follow different paths in the fitness landscape on their way towards improvements.

In the convection distribution scheme, genotypes in the master's gene pool are sorted according to fitness. Then each slave receives a subset of genotypes that fall

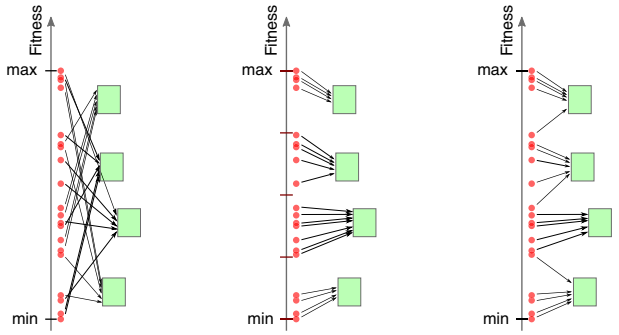

Fig. 6 An illustration of three distribution schemes—from *left* to *right*: random, convection intervals (equal width) and convection intervals (equal number of genotypes). The fitness of 20 genotypes is shown as circles, and 4 subpopulations (slaves) are depicted as boxes

within a range of fitness values. In the computational experiment, two methods of determining fitness ranges have been compared. In the first method, the entire fitness range has been divided into equal intervals (as many as there are slaves); if there are no genotypes in some fitness range, the corresponding slave receives genotypes from the nearest lower non-empty fitness interval. In the second method, the genotypes in master have been sorted according to fitness and then divided into as many sets as there are slaves so that each slave receives the same number of genotypes. This idea is illustrated in Fig. 6.

The experiment concerned evolution of simulated 3D structures that maximized vertical position of the center of mass using the *f1* genetic encoding. This encoding is a direct mapping between letters and parts of a 3D structure: 'X' represents a rod (a stick), parentheses encode branches in the structure, and additional symbols influence properties like length or rotation. The encoding is able to represent arbitrary tree-like 3D structures. Mutations modify individual aspects of the structure by adding or removing parentheses in random places in the genotype, or by adding and removing random symbols. Two-point crossover is used, and additional repair mechanisms validate the genotype by fixing parentheses if needed. Details of this genetic encoding are provided in [23].

There were 30 slaves (threads), each running a steady-state evolutionary algorithm with a gene pool capacity of 100. In evolutionary algorithms, positive and negative selection schemes are used to decide which solutions should be reproduced and which ones should be removed from the population. One of the most popular methods of selection is tournament selection [2], where a "tournament" is held between k randomly chosen individuals to select the single individual with the best fitness. In our experiments, the negative selection was random, and the positive selection was tournament selection of size $k = 2$ (lower selective pressure) or $k = 5$ (higher selective pressure). The master thread managed migrations of 30×100 genotypes. Two migration frequencies were compared: less frequent (evolution stops after 100 migrations performed every 10,000 genotype evaluations per slave) and more frequent (evolution stops after 1000 migrations performed every 1000 evaluations per slave). Since there were $10\times$ more migrations when they occurred $10\times$ more often, the total number of

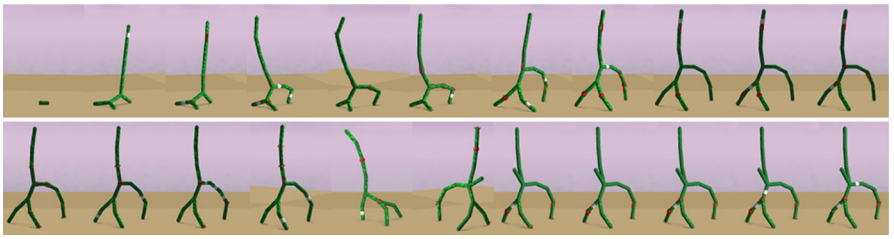

Fig. 7 A sample sequence (*top* to *bottom*, *left* to *right*) of the best individuals found in the master population after subsequent migrations. Their behavior in simulation is shown at https://www.youtube.com/watch?v=ZRIeOYpTS04

evaluated genotypes in all experiments was the same. Altogether, in this computational experiment, there were 20 independent evolutionary runs for each tournament size (2 and 5), for each migration frequency, and for each distribution scheme (random, convection based on the equal width of fitness intervals, convection based on the equal number of sorted genotypes)—a total of $20 \times 2 \times 2 \times 3 = 240$ evolutionary runs. The best individual in the last master population is considered the final result of each experiment.

In this experiment we focus on the analysis of performance (the quality of obtained results) of parallel evolutionary algorithms with different distribution schemes. Discussing the specifics of evolved 3D structures, albeit interesting, is outside of the scope of this paper. To illustrate outcomes of the evolutionary process and sample structures that were evolved, a sequence of best individuals found after subsequent migrations is shown in Fig. 7.

Figure 8 summarizes the results of this experiment for tournament size of 2 (left column, low selection pressure) and 5 (right column, high selection pressure), and different migration frequencies (top and bottom rows). Despite the difficulty of this optimization task and numerous local optima causing high variance of the best achieved fitness, both convection distribution schemes performed similarly well. In all experiments both schemes proved to be significantly better than the random distribution; p values are shown for a two-tailed t test. The improvement provided by the convection distribution schemes is more pronounced when the selective pressure is higher (the tournament selection of size 5).

The convection distribution scheme can be very efficiently implemented because it only concerns fitness values and does not involve operations on phenotypes or computing pairwise statistics like estimating diversity or dissimilarities between individuals [5,18,20]. This is especially important in research on artificial life and evolutionary design, where estimating similarity of solutions is problematic and time-consuming. In such applications, the convection distribution scheme proves to be a simple and fast method that significantly improves the dynamics of the distributed evolutionary process. It can also be implemented as a selection scheme in a non-distributed, single population architecture. By promoting diversity of fitness values, convection distribution schemes encourage diversity of solutions and, therefore, counteract population stability and premature convergence, as demonstrated in Fig. 9.

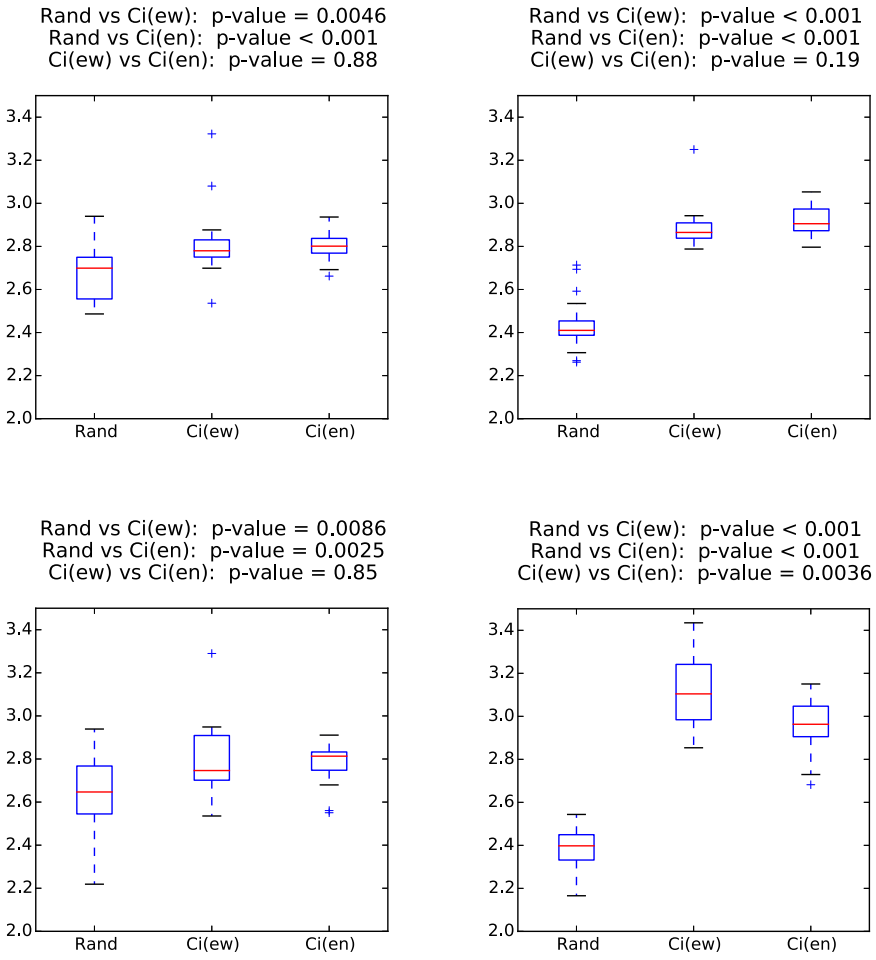

Fig. 8 Comparison of three genotype distribution schemes: Random, Convection intervals (equal width) and Convection intervals (equal number of genotypes). In all boxplots, *vertical axis* is fitness—the elevation of the center of mass of the best found 3D structure. *Columns* tournament selection of size 2 (*left*) and 5 (*right*). *Rows* 100 migrations every 10,000 genotype evaluations (*top*) and 1000 migrations every 1000 evaluations (*bottom*)

4 Summary and further work

This article discussed multithreaded performance of evolutionary processes that are typically employed in evolutionary design and artificial life. On a technical note, the experiments revealed that the string implementation that used reference counting and a mutex was much less efficient in a multithreading setup than a private unprotected string implementation. The negative impact of the mutex on overall performance increased as the number of threads increased. For more than 4 threads and the copy-on-write string, the overall performance started decreasing even if there were more than 4

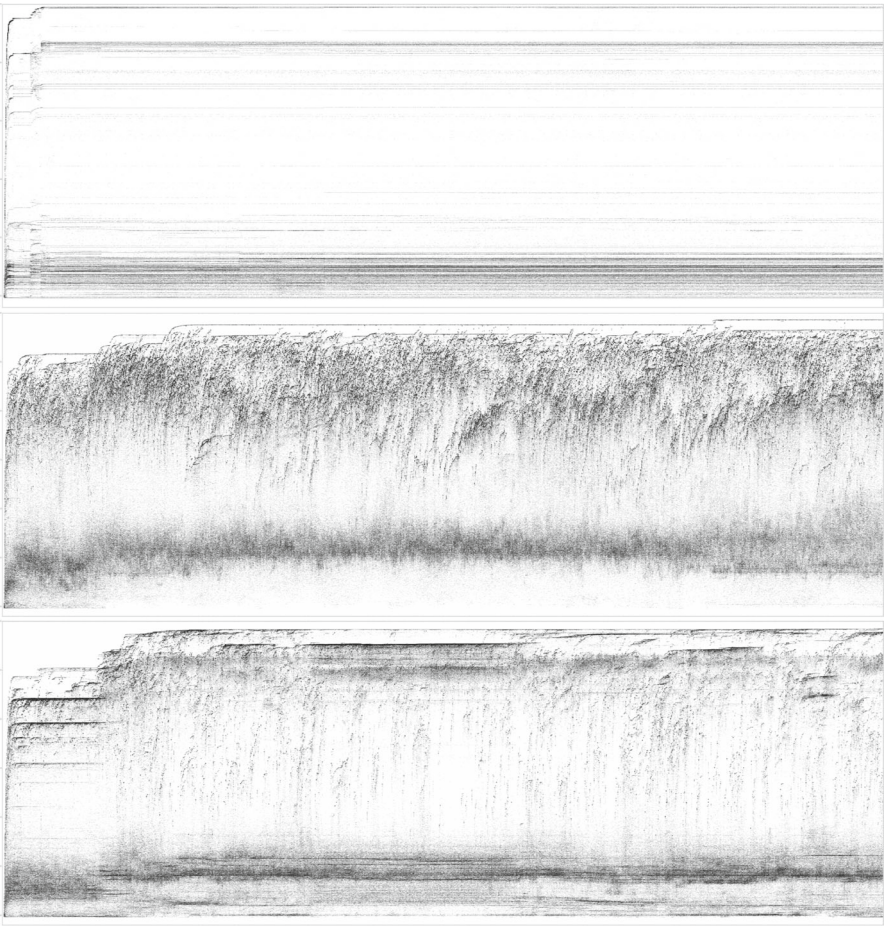

Fig. 9 Representative examples of population dynamics in three genotype distribution schemes—from *top* to *bottom*: random, convection intervals (equal width) and convection intervals (equal number of genotypes). The vertical axis is fitness, and the horizontal axis corresponds to migrations. There are 1000 migrations shown in each chart, and for each migration, fitness of each of the 3000 individuals in the master population is depicted as a dark dot. In all three experiments visualized above, tournament selection of size 5 was used

CPU cores available. This illustrates the rationale behind using string implementations that do not require synchronization in multithreaded applications.

Further performance analyses confirmed that the evolutionary algorithm that requires the simulation of physics and control systems to evaluate genotypes can be efficiently parallelized. This is because in evolutionary design and artificial life experiments, evaluation of genotypes can usually be implemented as highly independent (an exception would be the environment where most individuals interact frequently). The CPU architecture (the number of physical and logical CPU cores) determines the speedup that can be achieved given a specific number of independent subpopulations (threads). For a maximal performance in the evolutionary architecture considered here,

the number of subpopulations should be equal to the number of "logical" CPU cores, as the master thread only performs migrations.

The convection distribution scheme that was introduced in this paper proved to be significantly better than the random (uniform) distribution of genotypes among slave subpopulations. One of the reasons for this efficiency may be the fact that genotypes with similar fitness values are usually similar, and crossing over of similar parent genotypes is less likely to degrade the quality of their children. The performance of the convection distribution scheme can likely be further improved by employing more sophisticated ways of determining fitness intervals. The influence of the frequency of migrations on the performance of the evolutionary algorithm should be investigated as well. The promising performance of this distribution scheme should be tested on optimization benchmark functions, and these tests should include one-threaded, one-population architecture, where convection distribution turns into convection selection.

Acknowledgements The research presented in the paper received support from Polish National Science Center (DEC-2013/09/B/ST10/01734).

References

1. Alba E, Tomassini M (2002) Parallelism and evolutionary algorithms. IEEE Trans Evol Comput 6(5):443–462
2. Blickle T, Thiele L (1996) A comparison of selection schemes used in evolutionary algorithms. Evol Comput 4(4):361–394
3. Butenhof DR (1997) Programming with POSIX threads. Addison-Wesley Professional, Boston
4. Cantú-Paz E (1997) A survey of parallel genetic algorithms. Tech. Rep. 97003, University of Illinois at Urbana-Champaign
5. Črepinšek M, Liu SH, Mernik M (2013) Exploration and exploitation in evolutionary algorithms: a survey. ACM Comput Surv (CSUR) 45(3):35
6. de Back W, Wiering M, de Jong E (2006) Red Queen dynamics in a predator–prey ecosystem. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, pp 381–382. http://igitur-archive.library.uu.nl/vet/2007-0302-210407/wiering_06_red.pdf
7. Dasgupta D, Michalewicz Z (2013) Evolutionary algorithms in engineering applications. Springer, New York
8. González A, Latorre F, Magklis G (2011) Processor microarchitecture: an implementation perspective. Synthesis Lectures on Computer Architecture. Morgan & Claypool, San Rafael
9. Intel Corporation: Intel Core2 Quad Processor Q6600 (8M Cache, 2.40 GHz, 1066 MHz FSB. http://ark.intel.com/products/29765 (2007)
10. Intel Corporation: Intel Core i5-2500 Processor (6M Cache, up to 3.70 GHz. http://ark.intel.com/products/52209 (2011)
11. Intel Corporation: Intel Xeon Processor E5-2660 (20M Cache, 2.20 GHz, 8.00 GT/s Intel QPI). http://ark.intel.com/products/64584 (2012)
12. Intel Corporation: Intel Core i7-4700MQ Processor (6M Cache, up to 3.40 GHz). http://ark.intel.com/products/75117 (2013)
13. Intel Corporation: Intel Core i7-4790 Processor (8M Cache, up to 4.00 GHz). http://ark.intel.com/products/80806 (2014)
14. Jaskowski W, Komosinski M (2008) The numerical measure of symmetry for 3D stick creatures. Artif Life J 14(4):425–443. doi:10.1162/artl.2008.14.4.14402

15. Jelonek J, Komosinski M (2006) Biologically-inspired visual-motor coordination model in a navigation problem. In: Gabrys B, Howlett R, Jain L (eds) Knowledge-based intelligent information and engineering systems, Lecture notes in computer science, vol 4253. Springer, Berlin, pp 341–348. doi:10.1007/11893011_44. http://www.framsticks.com/files/common/BiologicallyInspiredVisualMotorCoordinationModel.pdf
16. Jones J, Soule T (2006) Comparing genetic robustness in generational vs. steady state evolutionary algorithms. In: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation. ACM, La Jolla, pp 143–150
17. Knysh DS, Kureichik VM (2010) Parallel genetic algorithms: a survey and problem state of the art. J Comput Syst Sci Int 49(4):579–589. doi:10.1134/S1064230710040088
18. Komosinski M (2016) Applications of a similarity measure in the analysis of populations of 3D agents. J Comput Sci. doi:10.1016/j.jocs.2016.10.004
19. Komosinski M, Adamatzky A (eds) Artificial life models in software, 2nd edn. Springer, London (2009). doi:10.1007/978-1-84882-285-6. http://www.springer.com/978-1-84882-284-9
20. Komosinski M, Kubiak M (2011) Quantitative measure of structural and geometric similarity of 3D morphologies. Complexity 16(6):40–52. doi:10.1002/cplx.20367
21. Komosinski M, Kups A (2015) Time-order error and scalar variance in a computational model of human timing: simulations and predictions. Comput Cognit Sci 1(1):1–24. doi:10.1186/s40469-015-0002-0
22. Komosinski M, Mensfelt A, Tyszka J, Goleń J (2016) Multi-agent simulation of benthic foraminifera response to annual variability of feeding fluxes. J Comput Sci. doi:10.1016/j.jocs.2016.09.009
23. Komosinski M, Rotaru-Varga A (2001) Comparison of different genotype encodings for simulated 3D agents. Artif Life J 7(4):395–418. doi:10.1162/106454601317297022
24. Komosinski M, Ulatowski S (2013) Parallel computing in Framsticks. Research report RA–18/2013, Poznan University of Technology, Institute of Computing Science. http://www.framsticks.com/files/common/ParallelComputingFramsticks.pdf
25. Komosinski M, Ulatowski S (2016) Framsticks web site. http://www.framsticks.com
26. Lozano M, Herrera F, Cano JR (2008) Replacement strategies to preserve useful diversity in steady-state genetic algorithms. Inf Sci 178(23):4421–4433
27. Mandik P (2002) Synthetic neuroethology. Metaphilosophy 33(1&2):11–29. http://www.petemandik.com/philosophy/papers/synthneur.pdf
28. Mandik P (2003) Varieties of representation in evolved and embodied neural networks. Biol Philos 18(1):95–130. http://www.framsticks.com/files/common/Mandik_RepresentationsInNeuralNetworks.pdf
29. Marr DT, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, Upton M (2002) Hyper-threading technology architecture and microarchitecture. Intel Technol J 6(1):4–15
30. Microsoft: process and thread functions: Sleep (2016). https://msdn.microsoft.com/en-us/library/windows/desktop/ms686298(v=vs.85).aspx
31. Mohamed R, Raviraj P (2011) Biologically inspired design framework for robot in dynamic environments using Framsticks. Int J Bioinform Biosci 1(1):27–35
32. Nesmachnow S, Cancela H, Alba E (2012) A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. Appl Soft Comput 12(2): 626–639. doi:10.1016/j.asoc.2011.09.022. http://www.sciencedirect.com/science/article/pii/S1568494611004248
33. Nikseresht MR, Somayaji A, Maheshwari A (2010) Customer appeasement scheduling. Tech. Rep. TR-10-18, School of Computer Science, Carleton University arxiv:1012.3452
34. Pyles J, Garcia J, Hoffman D, Grossman E (2007) Visual perception and neural correlates of novel 'biological motion'. Vis Res 47(21):2786–2797. doi:10.1016/j.visres.2007.07.017
35. Pyles J, Grossman E (2009) Neural adaptation for novel objects during dynamic articulation. Neuropsychologia 47(5):1261–1268
36. Sutter H (2005) Exceptional C++ style: 40 new engineering puzzles, programming problems, and solutions. The C++ in-depth series. Addison-Wesley, Boston
37. Tomassini M (1999) Parallel and distributed evolutionary algorithms: a review. In: Neittaanmki P, Miettinen K, Mkel M, Periaux J (eds) Evolutionary algorithms in engineering and computer science. Wiley, New York
38. Vavak F, Fogarty TC (1996) A comparative study of steady state and generational genetic algorithms for use in nonstationary environments. In: Evolutionary computing. Springer, Berlin, pp 297–304