It's Not a Bug, It's a Feature: Wait-free Asynchronous Cellular Genetic Algorithm

Frédéric Pinel[1], Bernabé Dorronsoro[2], Pascal Bouvry[1], and Samee U. Khan[3]

[1] FSTC/CSC/ILIAS, University of Luxembourg
frederic.pinel@uni.lu, pascal.bouvry@uni.lu,
[2] University of Lille, France
bernabe.dorronsoro_diaz@inria.fr,
[3] Department of Electrical and Computer Engineering, North Dakota State University, USA
samee.khan@ndsu.edu

Abstract. In this paper, we simplify a Parallel Asynchronous Cellular Genetic Algorithm, by removing thread locks for shared memory access. This deliberate error aims to accelerate the algorithm, while preserving its search capability. Experiments with three benchmark problems show an acceleration, and even a slight improvement in search capability, with statistical significance.

Keywords: Cellular Genetic Algorithm, Parallelism

1 Introduction

Evolutionary Algorithms (EAs) have been used for many years to solve hard combinatorial and continous optimization problems. These nature-inspired algorithms iteratively apply transformations to solutions, and converge to an optimal or near-optimal solution. However, EAs require many iterations to conduct their search. This motivates the design of concurrent versions of these algorithms, in order to exploit the parallelism available in current computers. Moreover, parallelism can also improve the search capability of the algorithms [3, 5].

In this paper, we propose a new Parallel Asynchronous Cellular Genetic Algorithm (PA-CGA). Our PA-CGA deliberately includes an error, that simplifies the design of the algorithm but also improves the speed of the algorithm. We compare this new PA-CGA with two known PA-CGAs, in terms of execution speed but also search capability.

Section 2 defines our parallelism objective. Section 3 presents the different models and how they are compared. Sections 4 describes the experiments.

2 Problem Description

The problem addressed in this paper is the parallelization of PA-CGA to improve its scalability: how does a PA-CGA *behave* as the number of threads increases.

The behavior of a PA-CGA should not be limited to runtime, but also include how well the algorithm searches solutions.

The next sections provide background information on parallel EAs and presents the PA-CGA.

2.1 Background

A survey of parallel genetic algorithm can be found in [9]. Concurrency in genetic algorithms is often introduced at the population level, because the evolutionary steps can be applied independently across a population of solutions. An evolutionary step is a sequence of operations, which generate new solutions (called children). The sequence is: parent selection (choosing individuals), crossover (generating a child from the parents), mutation (applying a small random change to the child) and replacement (criteria for the child to join the population). One evolution of all individuals in a population is called a generation.

The concurrency in the population generally occurs in three ways: master-slave, island and cellular. The master-slave model dispatches the operators' work to a number of slaves. In the island model, the population is partitioned into isolated evolutionary processes, which periodically exchange individuals. The cellular model, implemented in Cellular Genetic Algorithm (CGA) [1], is a fine-grain island model, where the periodical exchange of individuals is replaced with a more frequent update to a shared population. The CGA imposes a structure on the population of candidate solutions, usually a two-dimensional grid, and the parents for crossover are selected from the neighborhood of an individual. This increases the diversity in the population. The population structure in a CGA provides a fine-grain control over the evolution, which facilitates the exploration of different concurrency models [4].

Individuals evolving in parallel across the population usually evolve together, which requires synchronization. Asynchronous evolution relaxes this global time constraint [11, 12]: individuals evolve independently and the population is not of the same age (underwent the same number of evolutions). Asynchronous models are also known to improve search capability [2].

2.2 Parallel Asynchronous Cellular Genetic Algorithm

The PA-CGA we study was presented in [14, 15]. Parallelism in the PA-CGA is introduced at the population level. The population partition model of our PA-CGAs is inspired from [6, 7, 13]. As shown in Figure 1, the population is partitioned into a number of contiguous sub-populations, with a similar number of individuals. Each partition contains $pop\_size/\#threads$ individuals, where $\#threads$ represents the number of threads launched. The neighborhood of an individual may cross partition boundaries. The threads in a PA-CGA evolve their partition independently: they do not wait on the other threads in order to pursue their evolution. The combination of a concurrent execution model with overlapping neighborhoods leads to concurrent access to shared memory, and requires synchronization.

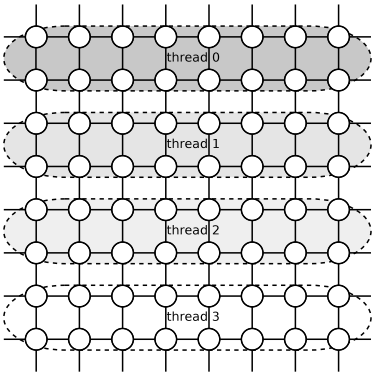


Fig. 1: Partition of an 8x8 population over 4 threads.

3 Approach

We present in Section 3.1, three parallel models for a PA-CGA: the *Island* [9], *Lock* [14] and *Free*, based on the principles of Section 2. The Free model is our contribution. It is an incorrect implementation of a PA-CGA: the thread locks protecting the shared population are removed, thus data consistency is not ensured. This is meant to improve the runtime and scalability of the PA-CGA. However, this change should impact the search capability of the PA-CGA. To investigate the behavior of the Free model, we compare the models across a selection of benchmark problems, presented in Section 3.2. The behavior of the models is observed across several metrics, presented in Section 3.3.

3.1 PA-CGA Models

Algorithm 1 Island model

```
while < max_gens do
    while < max_gens & not every 100 gens do        ▷ evolve local partition 100×
        for all individual in local partition do
            individual ← evolved(individual)        ▷ "←" follows replacement policy
        end for
    end while
    for all individual in global partition do       ▷ update the global partition
        rw_lock(global individual)
        global individual ← local individual
        rw_unlock(global individual)
    end for
end while
```

In the PA-CGA *Island* model, presented in Algorithm 1, each thread operates on two populations: (a) its local partition, and (b) the global population. Once 100 generations are completed, the thread-local partition (a) is copied to the global population (b). This copy is performed asynchronously (one individual at a time). The global population (b) is accessed by threads when they require individuals from another partition. This occurs at crossover, when a parent selected belongs to another partition. POSIX read-write locks [8] are used by the threads to read individuals from another partition, and to commit their partition to the global population. The Island model aims to reduce contention on the shared population by operating on a thread-local data as much as possible.

Algorithm 2 Lock model

```
for all gens do
    for all individual in global partition do
        child ← evolved(individual)
        rw_lock(global individual)
        global individual ← child                ▷ "←" follows replacement policy
        rw_unlock(global individual)
    end for
end for
```

The PA-CGA *Lock* model, presented in Algorithm 2, is the closest to the classic asynchronous CGA. The only difference is that each thread evolves the individuals of its partition only. Each individual is protected with a POSIX read-write lock. This allows for concurrent read access. When an individual can be replaced with a better child, the change occurs immediately (provided a thread lock is acquired), and is then visible to all other threads. The Lock model requires more communication across threads than the Island model, however, changes are reflected immediately.

Algorithm 3 Free model

```
for all gens do
    for all individual in global partition do
        child ← evolved(individual)
        global individual ← child                ▷ "←" follows replacement policy
    end for
end for
```

The PA-CGA *Free* model is the simplest of all models, as per Algorithm 3. A thread evolves its partition, and updates the global population immediately. However changes in the global population are made without thread locking. This is apparently an error, because a thread may read an individual that is currently being updated (dirty read). This is possible because of the representation of an

Table 1: Benchmark of combinatorial optimization problems

Problem	*Fitness function*	*n*	*Optimum*
MTTP	$f_{MTTP}(\boldsymbol{x}) = \sum_{i=1}^{n} x_i \cdot w_i$	200	-400.0
PPEAKS	$f_{PPEAKS}(\boldsymbol{x}) =$ $\frac{1}{N} \max_{1 \leq i \leq p}(N - HammingD(\boldsymbol{x}, Peak_i))$	100	100.0
MMDP	$f_{MMDP}(\boldsymbol{s}) = \sum_{i=1}^{k} fitness_{s_i}$ $fitness_{s_i} = 1.0$ if s_i has 0 or 6 ones $fitness_{s_i} = 0.0$ if s_i has 1 or 5 ones $fitness_{s_i} = 0360384$ if s_i has 2 or 4 ones $fitness_{s_i} = 0.640576$ if s_i has 3 ones	240	40.0

individual; usually a large array of word size elements. This model is considered wait-free, because a thread's progress is bounded by a number of steps it has to wait before progress resumes. Increasing the number of threads makes dirty reads more frequent, because it reduces the size of each partition, each thread evolves its partition faster, and more individuals lie on the border of a partition.

3.2 Benchmarks

The benchmark problems selected for our comparison are well-known combinatorial optimization problems, displaying different features like multi-modality, epistasis, large search space, etc. Due to the lack of space it is not possible to give details on these problems, but the reader is referred to [1]. They are summarized in Table 1 (name, fitness value, number of variables –n–, and optimum). They are the Massively Multi-modal Deceptive Problem (MMDP) –instance of 40 subproblems of 6 variables each–, the Minimum Tardiness Task Problem (MTTP) –instances of 200 tasks–, and the PPEAKS problem, with 100 peaks.

3.3 Metrics

The metrics for the comparison aim to capture the behavior of the three algorithms as the number of threads increases.

Our first metric is execution speed. It is the wall-clock runtime of the algorithms for the maximum number of generations. However, increased speed is useless if the search capability is degraded such that it requires more generations, therefore we add the following metrics:

- Success rate: the number of experiments when the optimum was found.
- Evaluation-efficiency: the number of evaluations required to find the optimum, when found. This is measured in evaluations (calculation of the fitness of an individual) instead of generations, because of the concurrent evolution in each partition.
- Time-efficiency: speed and evaluation-efficiency are combined by measuring the wall-clock time required to find the optimum (when found). This is useful from the perspective of a potential user of the algorithm.

Table 2: PA-CGA parameters

Parameter	*Value*
Population size	40×40
Asynchronous mode	fixed line sweep
Selection operator	L5, binary tournament
Crossover operator	two-point crossover
Crossover probability	1.0
Mutation operator	$\times 2$ flips
Mutation probability	1.0
Maximum generations	2500
Island synchronization period	100 generations
Runs	100

4 Experiments

This section defines the parameters, the environment and results for the experiments.

4.1 Experimental Setup

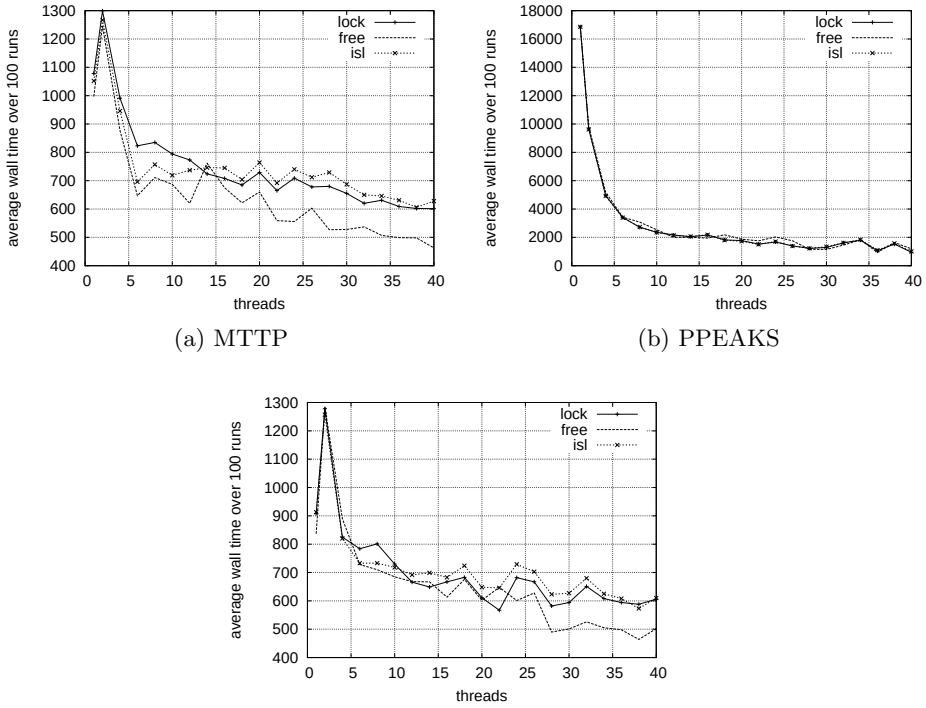
Table 2 summarizes the various parameters for the PA-CGA. The asynchronous mode sets the order in which the threads evolve the individuals in their partition. This is consistent with [2]. Mutation consists in randomly flipping two bits in the individual. The maximum number of generations is the stop condition per thread. The Island synchronization period specifies when the thread-local partition is committed to the global population (for other threads to access). For each benchmark, 100 searches or runs are performed. The individuals are randomly generated for each run.

The computer used for the experiments is a Bullx S6030, where one board holds four Intel Xeon E7-4850@2GHz processors of 10 cores each. We use one board for the experiments (up to 40 cores). The operating system is GNU/Linux 2.6.32-5-amd64 (Debian), GCC is version 4.4.5.

4.2 Experimental Results

In this section, we present the results from the benchmark problems, grouped by metric.

Runtime Figure 2 plots the average runtime (wall-clock) over the 100 runs in msec, as defined in Section 3.3. We can observe that all models reduce their runtime as the number of threads increases. The Free model is the fastest and scales the best, which is expected given the wait-free design, although not significantly for PPEAKS, Figure 2b. The small difference between models for PPEAKS is due to the fitness function of PPEAKS, which is more time consuming than MTTP and MMDP and therefore minors the synchronization delays. The speedup observed may seem low (especially for MTTP and MMDP), but the load is essentially due to synchronization.



(a) MTTP

(b) PPEAKS

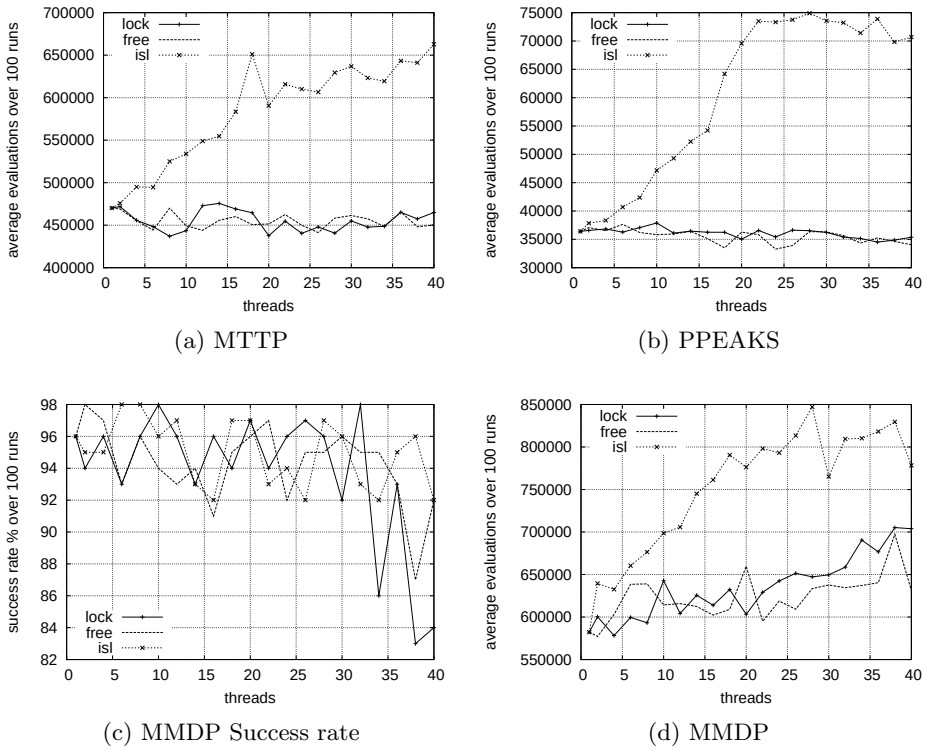
(c) MMDP

Fig. 2: Runtime

Evaluation Efficiency Figures 3 show the average number of evaluations needed to find the optimum, as defined in Section 3.3. Because this metric measures runs when the optimum is found, we first discuss the success rate.

The success rate for the different PA-CGA models for MTTP and PPEAKS is 100% across the runs (and is not plotted). For MMDP, Figure 3c, the rate is below 100%. All models display about the same success rate, which also decreases from 35 threads and up. At this point, the partitions become too small, the generations too fast, thus reducing diversity in the partitions, which hurts the search.

Regarding evaluation-efficiency, the Free model obtains similar or better results than Lock (Wilcoxon Signed-Rank test). On MTTP, PPEAKS and MMDP, Free is better in respectively 5, 10 and 20% of the cases. Also, the Lock and Free obtain constant results with the number of threads. The dirty reads in the Free model slightly help its evaluation-efficiency. The other observation is that the Island model does not scale well.



(a) MTTP (b) PPEAKS

(c) MMDP Success rate (d) MMDP

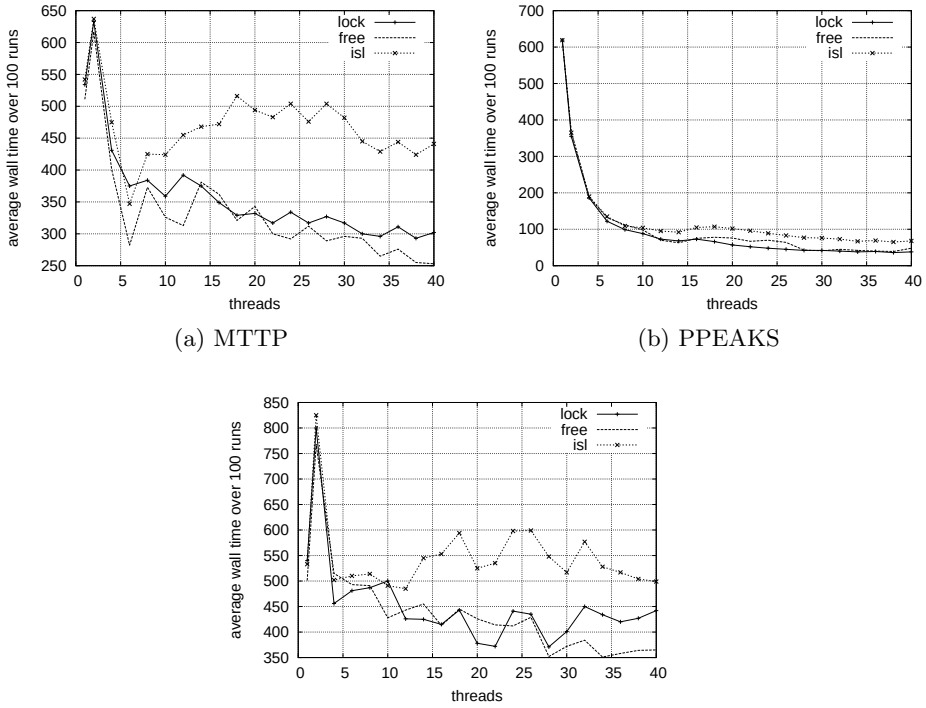
Fig. 3: Evaluations to optimum (when found)

Time Efficiency Figures 4 show the time elapsed to reach the optimum, when found, as defined in Section 3.3.

For the Island model, Figures 4a, 4c show that the gain in runtime is offset by the loss in evaluation-efficiency. For PPEAKS, the gain in runtime is so high, that time-efficiency manages to improve. The Lock and Free models do improve their time-efficiency with a greater number of threads, mainly because of the gain in speed. The Free model obtains the best results. This is due to the surprisingly good evaluation-efficiency, which means that the dirty reads do not harm the search, and may actually help.

5 Conclusions

We proposed a new PA-CGA parallel model, called Free. The Free model is based on a deliberate design error in the PA-CGA: all thread locks are removed, and access to the shared population leads to dirty reads. The absence of thread locks makes it wait-free. It is also the simplest PA-CGA design. This new model was compared to existing models: Island and Lock. The evaluation consisted in



(a) MTTP

(b) PPEAKS

(c) MMDP

Fig. 4: Time to optimum (when found)

solving three benchmark problems (MTTP, PPEAKS, MMDP) using 1 to 40 threads, on a 40-core machine. These benchmarks are not computationally intensive, therefore the differences between models is more apparent. Experiments show that the Free model scales the best, and provides better or equal search capability, compared to the previously published Island and Lock models.

Future work includes exploring other sources of randomness such as operating system thread scheduling, and removing partition borders.

Acknowledgment

This work is supported by the Fonds National de la Recherche Luxembourg: CORE Project Green-IT, INTER Project Green@cloud (i2r-dir-tfn-12grcl) and AFR contract no 4017742.

References

1. Alba, E., Dorronsoro, B.: Cellular Genetic Algorithms. Operations Research/Compuer Science Interfaces, Springer-Verlag Heidelberg (2008)
2. Alba, E., Giacobini, M., Tomassini, M., Romero, S.: Comparing synchronous and asynchronous cellular genetic algorithms. In: et al., J.M. (ed.) Proc. of the International Conference on Parallel Problem Solving from Nature VII (PPSN-VII). Lecture Notes in Computer Science (LNCS), vol. 2439, pp. 601–610. Springer-Verlag, Heidelberg, Granada, Spain (2002)
3. Alba, E., Tomassini, M.: Parallelism and evolutionary algorithms. IEEE Transactions on Evolutionary Computation 6(5), 443–462 (October 2002)
4. Alba, E., Blum, C., Asasi, P., Leon, C., Gomez, J.A.: Optimization techniques for solving complex problems, vol. 76. Wiley (2009)
5. Cantú-Paz, E.: Efficient and Accurate Parallel Genetic Algorithms, Book Series on Genetic Algorithms and Evolutionary Computation, vol. 1. Kluwer Academic Publishers, 2nd edn. (2000)
6. Folino, G., Pizzuti, C., Spezzano, G.: Parallel hybrid method for SAT that couples genetic algorithms and local search. IEEE Transactions on Evolutionary Computation 5(4), 323–334 (August 2001)
7. Folino, G., Pizzuti, C., Spezzano, G.: A scalable cellular implementation of parallel genetic programming. IEEE Transactions on Evolutionary Computation 7(1), 37–53 (February 2003)
8. IEEE and The Open Group: POSIX (ieee std 1003.1-2008, open group base specifications issue 7). http://www.unix.org (2008)
9. Luque, G., Alba, E., Dorronsoro, B.: Parallel Genetic Algorithms, chap. 5, Parallel Metaheuristics: A New Class of Algorithms, pp. 107–125. John Wiley & Sons (2005)
10. Manderick, B., Spiessens, P.: Fine-grained parallel genetic algorithm. In: Schaffer, J. (ed.) Proc. of the Third International Conference on Genetic Algorithms (ICGA). pp. 428–433. Morgan Kaufmann (1989)
11. Maruyama, T., Konagaya, A., Konishi, K.: An asynchronous fine-grained parallel genetic algorithm. In: Proc. of the International Conference on Parallel Problem Solving from Nature II (PPSN-II). pp. 563–572. Lecture Notes in Computer Science (LNCS), North-Holland (1992)
12. Muhlenbein, H.: Evolution in time and space - the parallel genetic algorithm. In: Foundations of Genetic Algorithms. pp. 316–337. Morgan Kaufmann (1991)
13. Nakashima, T., Ariyama, T., Ishibuchi, H.: Combining multiple cellular genetic algorithms for efficient search. In: Proc. of the Asia-Pacific Conference on Simulated Evolution and Learning (SEAL). pp. 712–716 (2002)
14. Pinel, F., Dorronsoro, B., Bouvry, P.: A new parallel asynchronous cellular genetic algorithm for de novo genomic sequencing. In: Proceedings of the 2009 IEEE International Conference of Soft Computing and Pattern Recognition. pp. 178–183. IEEE Press (2009)
15. Pinel, F., Dorronsoro, B., Bouvry, P.: A new parallel asynchronous cellular genetic algorithm for scheduling in grids. In: Nature Inspired Distributed Computing (NIDISC) sessions of the International Parallel and Distributed Processing Simposium (IPDPS) 2010 Workshop. p. 206b. IEEE Press (2010)