Stochastic Optimization Library Report

Domenico Santarsiero Alberto Guerrini Pierpaolo Marzo

June 2024

1 Introduction

In the realm of computational optimization, many real-world problems are characterized by complexity, non-linearity, and the presence of constraints, making traditional deterministic optimization methods less effective.
Stochastic optimization algorithms, which incorporate randomness into the search process, have emerged as powerful tools for tackling such challenging optimization problems. These algorithms are particularly adept at escaping local optima and exploring the solution space more broadly, which is essential for finding global optima in complex landscapes.

The Stochastic Optimization Library was conceived to address the growing need for versatile and robust optimization tools that can handle a wide variety of problem types, both constrained and unconstrained. By providing a unified interface across different algorithms, this library simplifies the implementation and usage of optimization techniques, making advanced optimization accessible to a broader audience.

This project has been realized in order to fill the gap between advanced optimization methods and their practical application. It abstracts the complexities of individual algorithm implementations, allowing users to focus on problem-solving rather than the intricacies of algorithmic design. Each algorithm in the library is equipped with detailed documentation, examples, and performance tests, ensuring that users can effectively leverage these tools to achieve optimal solutions.

Moreover, the inclusion of constraint-handling capabilities in the algorithms expands their applicability to real-world scenarios where constraints are often an integral part of the problem definition.

Available Algorithms

The library includes several algorithms, each with its own implementation, usage guidelines, and performance test results. Below is a detailed description of each one.

2 Algorithm

2.1 Artificial Bee Colony - ABC

The Artificial Bee Colony (ABC) algorithm is a metaheuristic optimization technique inspired by the foraging behavior of honey bees. It was introduced by Karaboga and Basturk and has proven effective for solving complex optimization problems, including constrained optimization problems (COPs). The version developed in this project refers to Karaboga's ABC implementation for solving constrained optimization problems.

The ABC algorithm simulates the intelligent foraging behavior of a honey bee swarm. The algorithm consists of three types of bees: employed bees, onlookers, and scouts, which work together to find the optimal solution.

Key Features

- **Exploration and Exploitation**: The ABC algorithm balances exploration (global search) and exploitation (local search) through the cooperative behavior of the bees.
- **Adaptivity**: It dynamically adapts the search process based on feedback from the environment, improving its ability to find optimal solutions.
- **Flexibility**: ABC can be applied to various types of optimization problems without significant modifications.

Algorithm Description

Algorithm 1 Pseudo-code of main body of ABC algorithm

```
Initialization
Evaluation
cycle = 1
repeat
    Employed Bees Phase
    Calculate Probabilities for Onlookers
    Onlooker Bees Phase
    Scout Bees Phase
    Memorize the best solution achieved so far
    cycle = cycle + 1
until cycle = Maximum Cycle Number
```

The ABC algorithm involves the following steps:

- **Initialization**: Generate an initial population of food sources (solutions) randomly.

- **Employed Bee Phase**: Each employed bee evaluates the fitness of its food source and explores neighboring solutions, updating its position with a probability related to the algorithm parameter MR.

$$x_{ij} = \begin{cases} x_{ij} + \beta_{ij}(x_{ij} - x_{kj}), & \text{if } R_j < MR \\ x_{ij}, & \text{otherwise.} \end{cases}$$

- **Onlooker Bee Phase**: Onlooker bees select new food sources to follow, based on their probability p_i, which is proportional to the fitness of the food source.

$$p_i = \begin{cases} 0.5 + \frac{fitness_i}{\sum_{j=1}^{n} fitness_j} \times 0.5 & \text{if feasible} \\ \left(1 - \frac{violation_i}{\sum_{j=1}^{n} (violation_j)}\right) \times 0.5 & \text{otherwise} \end{cases}$$

- **Scout Bee Phase**: If a food source cannot be improved further, it is abandoned, and a scout bee randomly searches for a new food source, meaning that the bee is reinitialized.

- **Termination**: The process repeats until a stopping criterion (maximum number of iterations or acceptable solution quality) is met.

2.2 Adaptive Standard PSO (SASPSO)

The SASPSO is an evolution of the general PSO algorithm that involves several improvements, listed below:

- **Adaptive search parameters**: In order to find optimal solutions efficiently, on one hand, in the early stage of the evolution, the exploration ability of PSO must be promoted so that particles can wander through the entire search space rather than clustering around the current population-best solution. On the other hand, in the later stage of the evolution, the exploitation ability of PSO needs to be strengthened so that particles can search carefully in a local region to find optimal solutions. This algorithm then adjusts the three PSO search parameters for any particle according to the iteration number and its distance from the actual global best. This also ensures the non-stagnation property of such algorithm.

- **Position update**: The position of any particle at the next iteration is obtained from three components: the current velocity, the personal best position, and the global best position. The isobaricenter of a particle is computed weighting the global and personal best with the cognitive and social search parameters. Then the next position is drawn randomly and uniformly in the hyperball centered in the isobaricenter with radius

equal to the distance between the isobaricenter and actual position of the particle. This ensures a stabler and more uniform behavior of the particle with respect to the standard update in the hypercube.

Our version can also solve constrained optimization problems (COPs) exploiting an adaptive relaxation method integrated with the feasibility-based rule. This technique is better than the common penalty rule since it does not require additional parameters preserving the self-adaptiveness property of the algorithm (more details on the implementation are found in the README file).

3 Results

In this section we provide our findings about the evaluation of our implementations.

For scalability and parallel speedup plots the programs have been executed on the MOX hpc cluster of the Politecnico di Milano, submitting them on the `giagatlong` queue. While for other tests that do not require such resources they have been run on our local machines.

3.1 Time Complexity and Problem Size

A first test on our local machine has been performed to assess the behaviour of our solvers with respect to solve time in function of the problem size. A good metric for the latter is the number of particles in the swarm.

The result for the SASPSO in Figure (1) shows a parallel speedup running locally on an Intel Core i7-13700H machine (20 logical threads, 8 performance + 4 power-efficient cores). The limited speedup is due to the non-trivial synchronization between OpenMP threads needed at each iteration. Of course, we must also consider the architecture of the processor, having non-homogeneous cores.

The result for the ABC in Figure (2) highlights shows a parallel speedup which increases with the dimension of the problem, reaching a maximum of around 10×, running on an Intel Core i7-13700H machine (20 logical threads, 8 performance + 4 power efficient cores). The serial execution time increases linearly with the number of particles, as expected for an algorithm with a computational complexity dependent on the number of particles. The speedup initially increases sharply as the number of particles increases, indicating a high efficiency of parallelization for smaller colony sizes. Beyond a certain point (around 500 particles), the speedup fluctuates but generally stays high, suggesting that the parallel implementation consistently performs well, though with some variations.

A better scalability study for both algorithms is provided below.

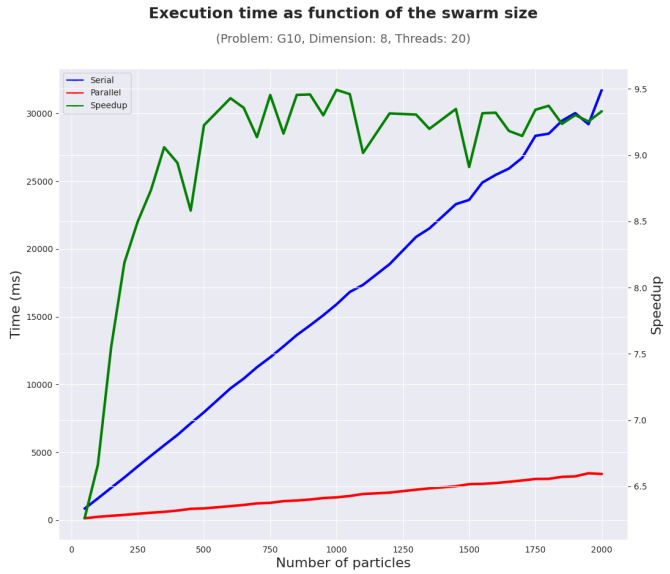

Figure 1: SASPO execution time as function of the problem size.

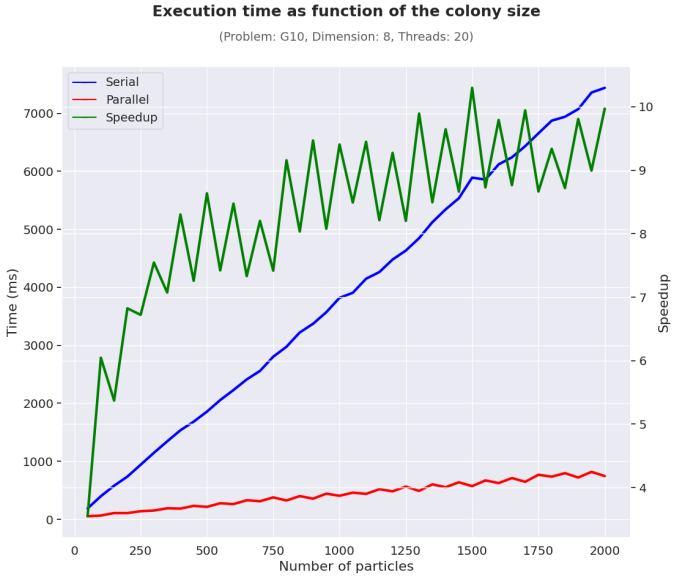

Figure 2: ABC execution time as function of the problem size.

3.2 Scalability

In order to compare how each of our algorithms scale, timing data has been collected varying the number of threads (and processes when possible) and the problem size. Scaling experiments have been performed solving the heavy 8-dimensional G10 problem in order to have quite reliable results. Each solver is analyzed on its own below.

Along the lines, the problem size is held constant as the number of threads is increasing. When doubling the number of cores, one expects a halving of the computational time, indicated by the black dashed ideal scaling lines. The above mentioned behaviour is referred to as strong scaling.

It is also possible to evaluate the weak scaling behaviour, just considering that each line differs by a factor of 2 in the number of particles, and that an ideal weak scaling is obtained when doubling both the number of particles and the number of processors gives the same total time.

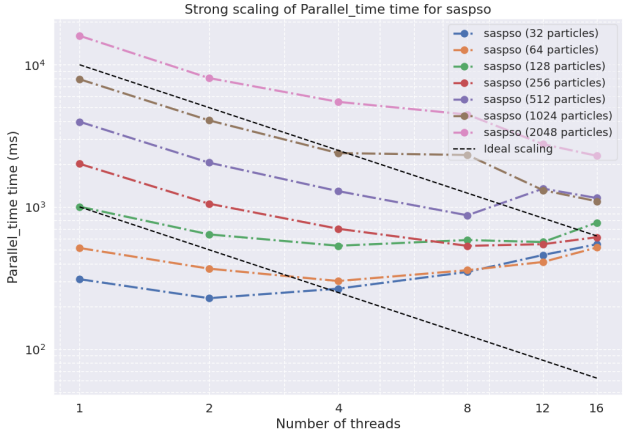

Figure 3: Execution time as function of the number of threads to evaluate the parallel speedup of SASPSO. Linear optimal speedup is represented by the dashed black line.

Figure (3) shows the results for the SASPSO solver. Here one can note a sub-optimal behaviour especially for smaller swarms and for higher number of threads (bottom and right area) as expected. The synchronization overhead is not negligible and has a stronger impact when execution time is quite low (i.e. less than 1 second). For bigger problems, among 1K and 2K particles, the solver shows an almost optimal behaviour up to 4 threads, then it experiences a deterioration, and finally from 8 to 16 threads it starts again to show almost optimal strong scalability. This behaviour can be caused by the different access time to shared memory from different groups of cores. Overall, for computations that last for less than 5 seconds it is possible to observe only small (and in some

cases even negative) improvements from multithreading. Given this definition, weak scaling is optimal for few processors (i.e. from 1 to 2) and quite bigger problem sizes. While for higher count of processors and problem sizes it becomes suboptimal, with higher degradation for smaller problems.

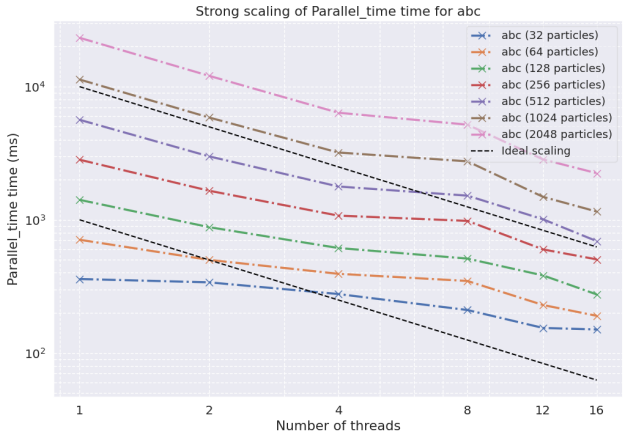

Figure 4: Execution time as function of the number of threads to evaluate the OMP parallel speedup of ABC. Linear optimal speedup is represented by the dashed black line.

Figure (4) refers to OMP parallelization of the ABC solver. It shows that, while the ABC algorithm benefits significantly from parallel execution, achieving substantial reductions in execution time, the efficiency of this scaling is influenced by problem size and overheads. In particular, we note an almost optimal behaviour with high dimension problems and a small number of threads (up to 4). A general small deterioration is experienced between 4 and 8 threads, and finally from 8 to 16 threads it starts again to show almost optimal strong scalability. Smaller problems exhibit less efficient scaling, emphasizing the need to match problem size with appropriate parallelization levels for optimal performance. Overall, also in this case, for computations that lasts for less than 5 seconds, it is possible to observe only small improvements from multithreading.

Figure (5), instead, refers to MPI parallelization of the ABC solver. It is evident that it shows a more consistent approach to ideal scaling. Unlike the OMP implementation, where smaller problem sizes showed significant deviations, MPI maintains closer adherence to the ideal scaling line across various problem sizes. This can be attributed to its distributed memory model and minimized communication overheads. As the problem size increases proportionally with the number of processes, the execution time per process remains relatively stable, indicating good weak scalability.

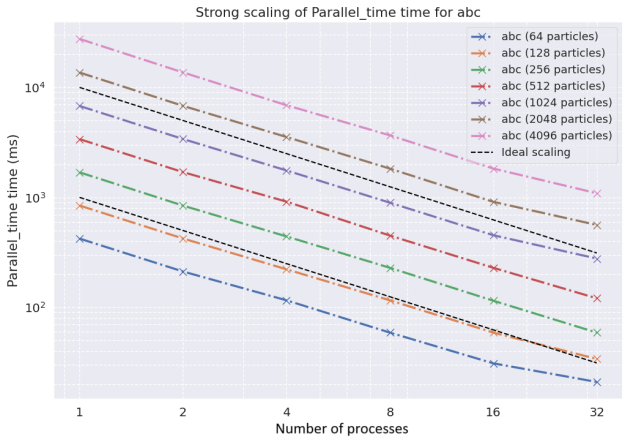

Figure 5: Execution time as function of the number of threads to evaluate the MPI parallel speedup of ABC. Linear optimal speedup is represented by the dashed black line.

3.3 Parallel Speedup

The actual parallel speedup of our algorithms can be evaluated plotting their solve time as a function of the number of used threads. Two problem sizes have been considered, i.e. swarms of 1K and 2K particles that are big enough to have quite stable and reliable results. Figure (6) shows that ABC appears to achieve better speedup than SASPSO for both particle sizes (1024 and 2048). For example, with 16 threads, SASPSO (2048 particles) achieves a speedup of around 13, whereas ABC (2048 particles) achieves a speedup of around 8. In general, though, both solvers have a similar behaviour, showing an almost linear speedup until 4 threads and then starting to suffer the synchronization overhead. Since 8 threads the solvers shows again an almost optimal speedup.

3.4 Error over iterations

Furthermore the error and constraint violation are represented as function of the iteration count to explain the optimization procedure and prove its correctness. In these results the G10 test problem in a 8D space has been optimized.

In Figure (7) the data for SASPSO solver has been represented. This plot highlights the policy to select the best among two particles that SASPSO 2011 utilizes:

- A feasible solution is preferred over an infeasible solution
- Among two feasible solutions, the one with better objective function value is preferred

- Among two infeasible solutions, the one with smaller total constraint violation is chosen

Starting from a high-violation good-fitness solution and a very relaxed violation threshold the algorithm decreases the threshold converging to feasible solutions. The first non-feasible solution will have a better fitness, then when the 0 violation threshold is reached the first feasible solutions will have a very bad fitness. From now the constraint violation will be kept to 0, and the algorithms finds feasible solutions that minimizes the fitness, converging to the optimum.

Figure (8) shows how ABC solver reflects exactly the same logic, although not having an adaptive constraint violation threshold.

3.5 Adaptive Search Parameters

In Figure (9) to evaluate the gain in using adaptive search parameters for the SASPSO algorithm, the same problem has been solved enabling and disabling the adaptiveness. Moreover, the optimization is performed both serially and in parallel to show that there are no performance losses using a parallel implementation.

In the results below the Gomez-Levy test problem in a 2D space is optimized. The swarm is composed of 5000 particles and 14k iterations are performed. The test has been run on an Intel Core i7-13700H machine.

Small differences can derive from the intrinsic randomness of this algorithm. Note also that the constraint violation converges in few iterations to zero since the search space has bigger feasibility areas than the one used before.

This plot shows that the parameter adaptivity feature of this implementation provides a much faster convergence with respect to static parameters (i.e. 200 vs 600 iterations). Both the serial and parallel version of the adaptive algorithm have comparable convergence rates. The adaptivity does not affect the convergence to the feasible search area as shown by the second plot. Moreover, the static version is more sensitive to the randomness of the algorithm since we note a quite differentiation between the two static solutions.

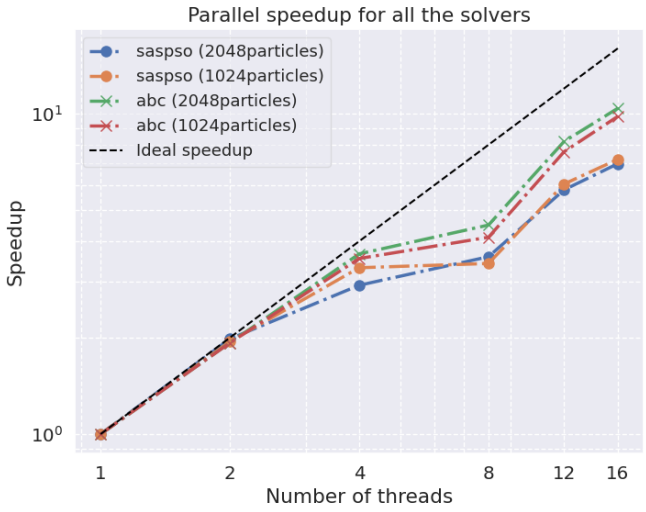

Figure 6: Speedup function of number of threads, according to different dimensions of the problem. It considers both ABC and SASPSO performances, both with OMP parallelization

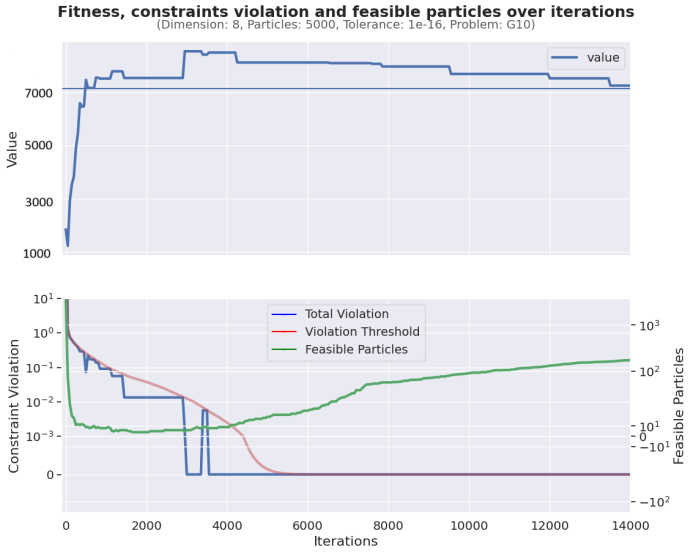

Figure 7: optimization of the G10 function with the SASPSO solver, showing distance from the optimal solution and constraint violation.

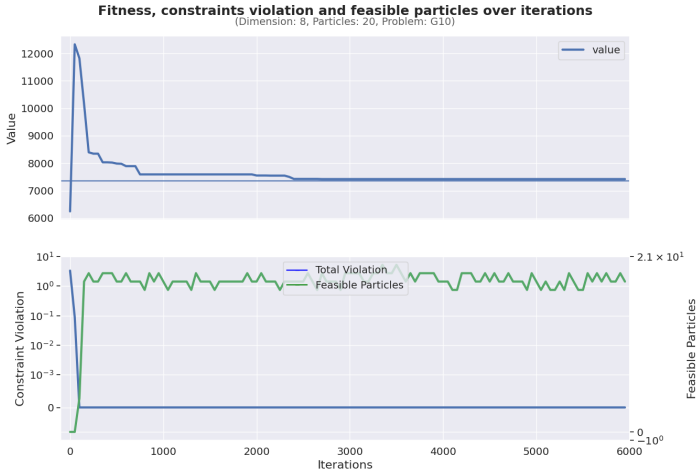

Figure 8: optimization of the G10 function with the ABC solver, showing distance from the optimal solution and constraint violation.

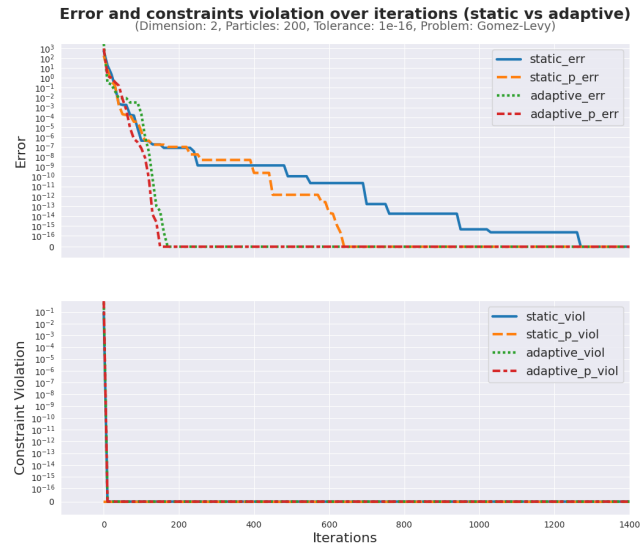

Figure 9: Error and constraint violation for the Gomez-Levy COP leveraging both adaptive and non-adaptive SASPSO, serially and in parallel.