A NOVEL PARALLEL IMPLEMENTATION OF PARTICLE SWARM OPTIMIZATION

Samuel Bartlett, Kristof Czirjak, Julius Gruber, Florian Pauschitz

Department of Mathematics
ETH Zurich
Zurich, Switzerland

ABSTRACT

Particle Swarm Optimisation is a heuristic differential-free algorithm originating in bioinformatics. A common problem in academia is to efficiently find global optima of functions. A novel parallel implementation of the algorithm is proposed in this report. We executed the code [1] on the Euler VI supercomputing cluster of ETH Zurich. The main finding is that a sufficient speed-up is achieved compared to the sequential version. A notable scaling for computationally more expensive functions was attained.

1. INTRODUCTION

Finding the global optimum of a function, be it the maximum or the minimum, is of considerable relevance in nowadays problem-solving. The necessity to find optima is present in numerous fields including finance (maximising profits), healthcare (detecting anomalies) and engineering (optimising workflow) to mention a few. These fields rely heavily on the speed and precision of the algorithms used.

Although deterministic algorithms have been thoroughly studied, at times their performance is lacking. In fact, finding a numerically exact solution may be too time-consuming or, at worst, it can not be found at all by the program. Especially for these cases, heuristic algorithms are used to obtain admissible solutions for a given problem with lower computational effort and shorter run-time.

The algorithm considered in this paper is the Particle Swarm Optimisation (PSO). Here, the concept of swarm intelligence is utilised, which is inspired by animal swarms. However, due to the novelty of this algorithm (first paper published on PSO was in 1995 [2]), the differences of performance, between various implementations, have not yet been explored fully. In addition to this, the growth of a single processor's computational power has slowed down. Hence, it has become increasingly relevant to exploit the rapid growth of multi-core machines. It is therefore crucial to explore the potential to parallelise PSO on computer clusters.

Thereupon we present an efficient parallel implementation of the Particle Swarm Optimisation that was optimised for clusters. In addition, we will try to understand which implementation of PSO, parallel synchronous (PSPSO) or parallel asynchronous (PAPSO), performs better.

Related work. Here we briefly mention a few papers that had an impact on our work. A PSPSO has been developed by the authors of [3]. They tested their work on problems ranging from two to 50 dimensions. However, their experiments were only executed on up to eight processors. This is common in this field and highlights the need for us to experiment on a cluster.

Another idea was to implement a PAPSO, which has been done by Koh *et al.* [4] and by Venter *et al.* [5]. The former came to the conclusion that "*PAPSO exhibits excellent parallel performance when a large number of processors (more than 15) are utilised*"[4]. Additionally, their results indicate that PAPSO has a shorter execution time than PSPSO on clusters. This is an interesting statement that caught our attention.

A few hurdles were encountered when trying to find an algorithm – let it be PSO or a different nature-based algorithm – to compare our implementation against. Firstly, in most papers like [3], implementations were only tested on specific functions and not on common benchmark functions. Secondly, most papers did not include their code. Hence, comparison was not possible. Thirdly, we were unable to execute most codes that we found on the cluster, due to various errors [4][6]. There was one exception, this being a student project code [7]. Unfortunately, this implementation does not have a corresponding scientific paper.

2. BACKGROUND: GLOBAL OPTIMISATION ALGORITHMS

We briefly discuss mathematical optimisation and introduce the Particle Swarm Optimisation as a part of the family of Global Optimisation algorithms.

Mathematical optimisation. The main goal of mathematical optimisation is to find the optimum of a function f over a domain $\Omega \subseteq \mathbb{R}^n \to \mathbb{R}$.

Particle Swarm Optimisation. PSO is a population-based heuristic optimisation technique first proposed by R.

Eberhart and J.Kennedy in 1995 [2]. It is inspired by the behaviour of large swarms of particles where all particles share some common information. As illustrated in Algorithm 1, the simulation starts with a population of particles on a d-dimensional grid, randomly distributed with initial positions and velocities. Every particle carries information of its current position x_i, its velocity and the best position it has ever achieved p_i. The common data among all entities is the position of the best solution found by the swarm: the global attractor. At every iteration step, the program updates each particle's position and velocity and evaluates the chosen fitness function at these new positions. If the resulting value is smaller than the attractor's value then the particle's position becomes the new global attractor. Over several iterations, a global optimum may be found in this way. More details can be read in the pseudo-code in **Algorithm 1**. The lower bound of the run-time is the product of the number of iterations, number of particles, number of dimensions and the evaluation time of the fitness function $f(\vec{x})$.

```
input  : An initialised population of particles with
         randomly distributed initial positions
output: An approximate optimum
1  while number of iterations not attained do
2      for each particle i do
3          for each dimension d do
4              Generate random numbers:
                 $r_p, r_g \sim U(0,1)$
5              Update particle velocity:
                 $v_{i,d} \rightarrow \omega \cdot v_{i,d} + c_p \cdot r_p(p_{i,d} - x_{i,d}) + c_g \cdot r_g(g_d - x_{i,d})$
6              Update particle position:
                 $x_i \rightarrow x_i + \Delta t \cdot v_i$
7              if $f(x_i) < f(p_i)$ then
8                  Update best known position:
                     $p_i \rightarrow x_i$
9                  if $f(x_i) < f(g)$ then
10                     Update swarm´s best known
                         position:$g \rightarrow x_i$
11                 end
12             end
13         end
14     end
15 end
```

Algorithm 1: Pseudo-code of PSO: *During initialisation we set w (weight), c_p and c_g (constants) and Δt (usually set to 1) as well as the function f.*

3. PARALLEL PSO

In the following, we shall introduce two parallel versions of PSO as proposed in [4]. We wrote three codes based on [8] using C++ and MPI.

Parallel Synchronous Particle Swarm Optimisation. The underlying principle of PSPSO is that, in every time step, every processor gets a fixed number of particles to evaluate.

We decided to implement two versions of PSPSO. Both are a slight variation to the PSPSO described in [4], which suggests evaluating the fitness of the particles in parallel in every time-step. The update of the positions, velocities, and local and global optima follows in sequence.

The first variation of our code initializes a smaller population on each thread separately. Initially, we evaluate all the particles at their primal position and store the position and fitness value of the best position found in an array. Then we gather these arrays on the base thread with MPI_Gather. Here, all the best fitness values are compared in order to find the global optimum and the corresponding particle. This particle is stored in the base population and broadcast to all threads together with its fitness value using MPI_Broadcast. Next, each thread updates the velocities, positions and the local optima of their own population with the normal update function described for PSO. The best values are then saved into the array to be gathered at the beginning of each time-step again.

The second variation initializes one big population in the base thread. We then send an average number of particles to each thread. This is achieved by using MPI_Scatter and a custom MPI datatype created with MPI_Type_create_struct. This MPI datatype represents a particle and holds this particle´s position, velocity, local and global best fitness, general fitness and the parameters required for the update step.

We then evaluate every particles' fitness. Using this the local fitness values, velocities and positions were updated accordingly. Then, these fitness values were stored in an array. These are gathered into another array in the base thread, where all the values are compared to the global best value. If any value is better, the global optimum is updated and broadcast back to all threads. This cycle continues in each time-step until the required amount of iterations have completed.

Optimisations to PSPSO. In this section, we present the optimisations applied to the parallel synchronous PSO described first.

While the base processor completes the minimum search loop for the best value, all others are waiting. This loop is critical to ensure a synchronised global best value, but it does not necessarily need to be done for every iteration. Of course, this will alter the result, but the gain in runtime hopefully counterbalances this slight loss of precision.

Therefore, the global optimum is only synchronised every 4 iterations, thus introducing a coarse-grained, fine-grained approach.

Initially, the global best values were gathered from all processors and then the base processor executed a minimum search and update loop. It was soon realised that an MPI_Reduce could substitute the MPI_Gather as well as the loop. MPI_Reduce is a higher-level operation and should offer more opportunities to optimise and distribute the computation over more resources compared to the loop. Henceforth, a new MPI_Datatype was introduced along with a new minimum search operator specific to the new datatype. The datatype fully describes a single particle and the new operator executes a minimum search on the fitness values for a vector of the new particle datatype.

After the completion of the MPI_Reduce implementation, the MPI_Allreduce routine was considered. This could replace the MPI_Reduce - MPI_Bcast pair. As MPI routines are highly optimised it is expected to be more efficient than our implementation.

Up to now, every processor sends and then receives packages of data that have the size of two doubles for the position as well as one double for the fitness value. This equals to $3 \times 8 = 24$ bytes. We experimented with a new struct with two floats and a double for which an MPI datatype was implemented . This structure has the size of 16 bytes, $2 \times 4 = 8$ bytes for the floats and 8 bytes for the double. As it is aligned in memory no padding is needed. However, this might result in a loss of accuracy as floats are less precise.

Parallel Asynchronous Particle Swarm Optimisation. The basic idea of the PAPSO is that there are two types of processors. There is one master processor and multiple worker processors with distinctive jobs.

The worker processors have a single task. After receiving the position of a particle, they complete the function evaluation and then send the new value back to the master processor. MPI_Recv and MPI_Send are used for communication. The index of the current particle is transmitted via the MPI_Tag. The value of the MPI_Tag is always the index of the current particle plus one. The reason for this is made clear later in the discussion of the termination of the worker processors.

The master thread initializes the swarm, including the vectors storing the positions, velocities and best values. It also launches the worker processors. Once the master processor receives a fitness value from a worker, it sends a new particle back. After that, the position and velocity for the received particle are updated, bounds are checked and, if a better value is found, the global optimum is updated. Once this is finished a new iteration is started. To assert that the loop has completed a given number of iterations, a variable was introduced. It counts the total number of particle-evaluations divided by the number of particles simulated.

As the number of iterations is not clearly defined, the worker processors have no way of knowing when the master processor's main iteration loop is finished. This means MPI can not be finalised as the worker processors are in an infinite loop of receive-evaluate-send. To combat this, a terminator loop was introduced at the end of the main process in which a termination signal is sent to the worker processors. This is achieved through an MPI_Tag of value zero. As the tag is the index of the current particle plus one, the value zero will be unique and the worker processors will terminate, followed by the master.

Optimisations to PAPSO. In this part, we present the optimisations applied to the parallel synchronous PSO.

The first optimisation was to make use of the fact that at the start of the computation every worker thread is waiting. Initially "number-of-processors" many MPI_Sends were used to send the first particles to the worker processors. Replacing the sends with a single MPI_Scatter should lead to a speed-up as the MPI_Scatter routine is highly optimised.

As the optimality of PAPSO heavily depends on the required computational effort within the worker processes, an idea of precomputing the velocities was tested. In the original PAPSO the master processor updates the velocities of all particles. Yet, in this step, parts of the velocity equation only depend on variables independent of other particles. Henceforth, part of the velocity computation was transferred to worker processors to increase their workload.

The next idea is to change as many blocking routines as possible into non-blocking ones. All MPI_Recv routines must stay blocking, as this is the only way processors may continue to work with relevant data. In the same way, the worker processors also need to wait until the master is ready to receive their results and therefore must use blocking MPI_Send. When the master wants to communicate with a worker the code guarantees that this specific core is either waiting for this message or has no other MPI-routine scheduled before this one. Therefore, whenever the master sends a position to a worker, a non-blocking send can be used. However, this only has an impact if the worker has not yet reached its MPI_Recv. As there are no costly computations between these two routines, we do not predict a significant gain in performance.

Another experiment is to send two particles to worker processors at the same time. This acts similar to loop unrolling. The goal here is to assist the master processor pipeline the computations that happen between every receive and send routine.

Compiler flag Optimisations. The last but highly important optimisation to evaluate comes from the compiler flags. Compiler flag optimisations were applied to both PAPSO and PSPSO. We experimented on the GCC compiler with eight different pairs of flags that were the most promising to attain speed-up:

$$\{O1, O2, O3, \emptyset\} \times \{\emptyset, march = native\}.$$

The benefits and drawbacks of the -O# flags are known. Each tries to reduce code size and execution time by turning on specific flags. We do not go into further detail.
The -march=native flag generates instructions for the specific machine used for compilation, which in our case is the Euler cluster.

4. EXPERIMENTAL RESULTS

In this section, the reader will be led through the results obtained by the optimisations conducted.Finally, the best implementation will be compared to the above mentioned parallel PSO code by A. Venkatesh [7].

Experimental Setup. The code was compiled with the GCC compiler on version 4.9.2 and with OpenMPI on version 1.6.5. The software mainly employs functions and classes from the standard C and C++ libraries. Especially in the evaluations of the benchmark functions, we made use of the mathematical tools provided by the cmath header. In addition, the compiler options march=native and O3 were activated. The only exception here being, when we ran experiments to deduce how much different compiler options improve the code's performance.

In addition to this, random numbers were generated using the standard C++ uniform distribution. Ranges were explicitly set for each benchmark function and the current time was used as a seed to ensure randomness even within runs on multiple calls to the random generator.

Moreover, the implementations were run on the Euler VI cluster of the Swiss Federal Institute of Technology Zurich (ETHZ). It is composed of 216 compute nodes, each equipped with two 64-core AMD EPYC 7742 processors (2.25 GHz nominal, 3.4 GHz peak) and 512 GB of DDR4 memory clocked at 3200 MHz. A dedicated 100 Gb/s InfiniBand HDR network connects them [9].

Finally, the experiments were run in a 2-dimensional space with 50 samples, 100 particles and 10.000 iterations. Note, while the PSPSO can run with only one processor, the PAPSO needs at least two: one as a master and the other one as a worker processor.

Benchmark functions. To assess the efficiency and correctness of an optimisation algorithm, test functions are needed. They are designed to thoroughly test the finest details of the algorithm with a multitude of local minimums and other pitfalls to assert the algorithm runs correctly. The benchmark functions implemented were gathered from [10] [11]. For ease of understanding the results, only a few representative functions were chosen, which are listed in the table below.

Function Name	Function
Griewank	$1 + \frac{1}{4000}(x^2 + y^2) - cos(x)cos(\frac{y}{\sqrt{2}})$
Ackley	$20 - 20exp(-\frac{1}{5}\sqrt{\frac{1}{2}(x^2 + y^2)})$ $-\exp(\frac{1}{2}cos(2\pi x)cos(2\pi y))$
Holdertable	$\lvert sin(x)cos(y)exp(\lvert 1 - \frac{\sqrt{(x^2+y^2)}}{\pi}\rvert)\rvert$
Rastrigin	$20 + (x - 10cos(2\pi x) + (y - 10cos(2\pi y)$
Long	Griewank_f & wait(1 microsecond)

Table 1. Our benchmark objective functions

Precision of the solutions. First of all, one has to make sure that the precision of the solutions is preserved over the different benchmark functions and implementations. From the table below we see that the non-deterministic PAPSO is slightly less stable in producing a good fitness value compared to PSPSO in the same number of iterations. All precision values for the PSPSO codes are fairly accurate.

Optimization Name	Griewank	Ackley	Holdertable	Rastrigin	Long
PAPSO Naive	0.177	0.351	0.595	0.414	0.340
PAPSO PRE	0.108	0.150	0.162	0.162	0.090
PSPSO Naive	0.010	0.090	0.030	0.008	0.243
PSPSO Smaller	0.010	0.008	0.015	0.007	0.257
PSPSO All Reduce	0.009	0.007	0.029	0.008	0.234

Table 2. Mean deviations of experimental results

Compiler flags. Concerning compiler optimisations, no single combination of the above-mentioned options was superior in speed-up compared to the others. Thus, as mentioned above, we decided to use the -march=native and -O3 compiler flag combination. As this should result in the most optimisations and was on average around 0.01 seconds faster than having no compiler flags.

PAPSO optimisations. Before the specific optimisations are considered, PAPSO in general has to be inspected.

While PAPSO seems promising at first, the following results may surprise the reader. In most of the cases it is the slowest implementation, mostly slower than the sequential version (as observable in figure 1 for the run-time and figure 3 for the speed-up compared to the sequential PSO). This is because only the function evaluations are done in parallel. If the current (benchmark) function is not complex enough and therefore is completed swiftly, two problems arise. Firstly, the worker processors will catch up to one another and thus, will spend more time waiting for a new job then doing actual computations. This derives from the fact that the position and velocity updates, as well as the bound checking are done in the master processor, with this process still taking a fixed amount of time. This time roughly equals the run time of the sequential version plus the time it takes to complete all MPI routines. The second problem is that in PAPSO individual MPI_Send-MPI_Recv pairs are implemented. These are not highly optimised collective routines like MPI_Gather or MPI_Reduce, as used in PSPSO.

When the fitness function evaluation exceeds a certain time threshold, the PAPSO code will however present a de-

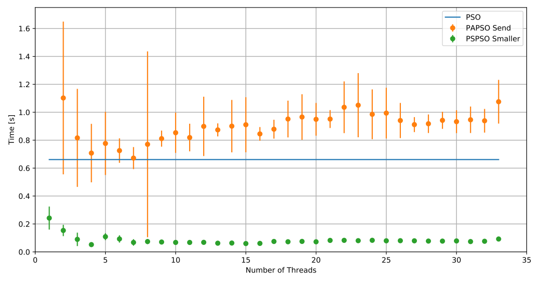

Fig. 1. Run-time with deviation for the *Holdertable* function.

cisive speed-up compared to the sequential version as can be seen in figure 2. This threshold is reached when the time spent on evaluation is equal to the time the master process takes to prepare a new job. The speed-up converges to a constant with an increasing number of cores. The threshold in the plot mentioned above was reached artificially by adding a 1 millisecond wait inside the function evaluation.

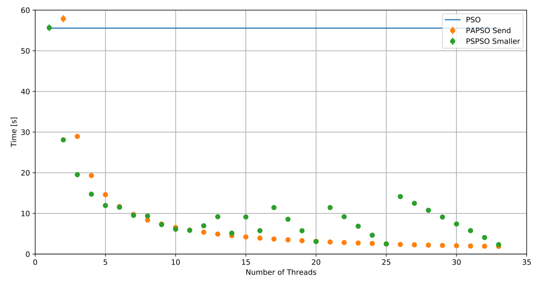

Fig. 2. Run-time with deviation for the *Long* function.

Now we will look at some optimisations to the naive PAPSO. First, one can see that precomputing parts of the velocity leads to worse performance, as precomputing requires larger buffers to be sent. These larger buffer sizes have a large latency, which leads to this supposed optimisation to run slower than the naive PAPSO (PAPSO Send), as shown in figure 3.

Furthermore, both the non-blocking and particle unroll versions did not provide us with a clear speed-up compared to the naive PAPSO implementation (see figure 3). On one hand, this was foreseen for the non-blocking optimisation, as the master process generally does not have to wait on the worker processor. On the other hand, we assume that the expected pipe-lining did not occur, due to the high complexity of the variable types inside the impacted operations.

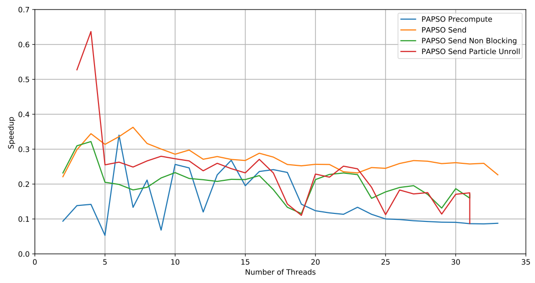

Fig. 3. Relative speed-up of different PAPSO versions with the *Holdertable* function.

PSPSO optimisations. We now present the more promising results from the PSPSO implementation. These results correspond to the first version, where each thread initializes their own population of particles. The second version was also tested but was found to be less promising. It did not produce sound results. A reason being that large packages needed to be sent frequently. However, here could also have been a conceptual error, and as the other version was very promising, we decided not to further pursue the second version.

Our first PSPSO implementation shows considerable speed-up compared to PSO for slow evaluation functions like *Holdertable* or *Long*. But even for fast functions, a decent speed-up is attained.

The reader might be surprised to see in figure 2 that our best parallel code executed on one processor is almost as fast as the sequential code. This can be explained as the sole processor does not have to send messages and never waits for another processor to give it data.

Another conclusion from figure 1 is that indeed PSPSO is much faster than PAPSO. This stems from the fact that the latter has to call MPI-routines more often as discussed above. In function Long (see figure 2) we observe that for more than 10 cores the asynchronous code's performance is more stable than the synchronous codes. This is specific to this function only.

We now compare our PSPSO implementations against each other. From figure 4 one can see that using All Reduce with smaller packages produces a clear speed-up of up to 13 times the speed of the sequential PSO. The reader will also note that the normal All Reduce optimisation slightly improves upon the naive PSPSO implementation, but both still achieve a maximum speed-up of 6 times the speed of the sequential PSO. Similar results were achieved for the *Holdertable* function. Surprisingly, the coarse-grained, fine-grained implementation does not yield a considerable speed-up. Similar results are found for the MPI_Reduce - MPI_Bcast implementation. It seems, that in contrast to our expectations,

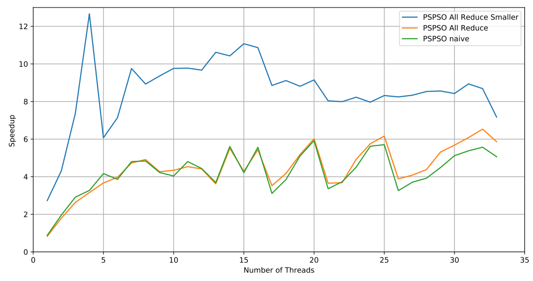

Fig. 4. Relative speed-up of different PSPSO versions with the *Holdertable* function.

a minimum search loop was more efficient than a minimum search operator. In summary, the highly tuned All Reduce MPI routine assisted through the Smaller Packets optimisation was the fastest.

Comparison to other Parallel PSO. To grasp the validity of our implementation, we compared our best PSPSO to a similar parallel synchronous PSO implementation by [7]. Note, that this code was implemented by a student. Superior ones may exist. However, as these either did not work on the Euler cluster or the paper they were presented in did not provide results for benchmark functions on networks with more than 8 cores, we were left with no choice but to compare our code to the aforementioned implementation.

As shown in figure 5, our implementation is slightly faster for most functions. A similar convergence of fitness values is also achieved.

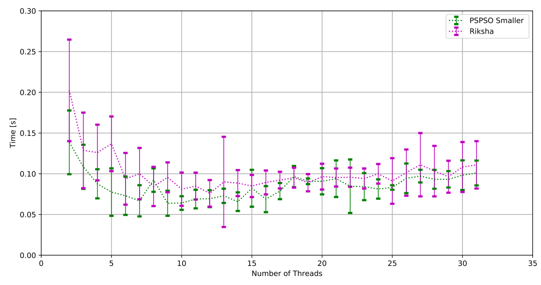

Fig. 5. Run-time with deviation for the *Holdertable* function.

5. CONCLUSIONS

After implementing multiple parallel versions of PSO we came to the following conclusions. Although the paper by Koh *et al.* [4] stated that PAPSO is faster than PSPSO for multiple cores, the opposite was found. We concluded that PSPSO is the preferable approach to solve common benchmark functions, as the workload is distributed more efficiently. Our implementations turned out to be significantly faster than the sequential version, especially for difficult functions. This indicates that for the code to scale well on clusters the fitness function has to have a high computational cost.

Through the comparison of our parallel synchronous implementation to [7] which yielded similar speed-up, we suggest the following: We believe there is a possible speed-up limit that we have reached or gotten close to. Additionally, we suppose that the use of Parallel PSO on higher dimensional optimisation problems will result in more worthwhile improvements, as we have consistently seen better performance for more complicated functions.

6. REFERENCES

[1] https://gitlab.ethz.ch/jgruber/dphpc.

[2] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, 1995, vol. 4, pp. 1942–1948 vol.4.

[3] Vijay Kalivarapu, Jung-Leng Foo, and Eliot Winer, "Synchronous parallelization of particle swarm optimization with digital pheromones," *Advances in Engineering Software*, vol. 40, no. 10, pp. 975 – 985, 2009.

[4] Haftka R. Fregley B.J. B. Koh, George A., "Parallel asynchronus particle swarm optimization," *International Journal For Numericla Methods in Engineering*, 2006.

[5] G. Venter, "A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations," *6th World Congresses of Structural and Multidisciplinary Optimization*, 2005.

[6] J.-B. Mouret and S. Doncieux, "SFERESv2: Evolvin' in the multi-core world," in *Proc. of Congress on Evolutionary Computation (CEC)*, 2010, pp. 4079–4086.

[7] https://github.com/abhilashvenkatesh/Parallelization-of-PSO.

[8] https://github.com/tonykero/Moe.

[9] https://scicomp.ethz.ch/wiki/Euler.

[10] Panigrahi *et al.*, *Handbook of Swarm Intelligence: Concepts, Principles and Applications*, Adaptation, Learning, and Optimization. Springer Berlin, 2011.

[11] Kashif Hussain *et al.*, "Common benchmark functions for metaheuristic evaluation: A review," 2017.