Parallel dragonfly algorithm

High Performance Computing for Data Science course 2024/2025

Mattia Santaniello[1] and Alessio Zeni[2]

[1]University of Trento

August 30, 2025

Abstract

The dragonfly algorithm is an approach used to solve optimization problems through the emulation of dragonfly movements, but it lacks scalability. In this paper, the authors propose an approach based on parallelization using dedicated libraries for high-performance multiprocess applications such as MPI, and the OpenMP API for multithread parallelization. A comparison of execution times is then conducted between presented implementation and the classic approach based on serial programming, to determine whether parallelization leads to performance improvements and enables possible future extensions.

1 Introduction

Swarm intelligence algorithms are inspired by the collective behavior of biological systems, where simple rules followed by individuals lead to the emergence of complex and adaptive group behavior. The concept was first introduced in Reynolds' Boids model (1987) [1], which simulated flocking behavior, and gained significant traction in Computer Science with the Ant Colony Optimization algorithm (1995) [2]. These algorithms are meta-heuristic in nature, leveraging decentralized and parallel exploration-exploitation strategies to solve optimization problems.

The dragonfly algorithm, first proposed in 2014 and published in 2015 [3], is a bio-inspired optimization algorithm that simulates the behavior of dragonfly swarms. By mimicking natural behaviors such as alignment, separation, cohesion, attraction to food sources, and distraction from enemies, the algorithm achieves a balance between exploration and exploitation. This makes it particularly effective at navigating complex search spaces while reducing the likelihood of being trapped in local optima. As of 2019 [4], the algorithm has been cited in over 300 scientific papers, highlighting its impact and popularity in the field.

This paper focuses on parallelizing the dragonfly algorithm to improve its scalability and performance. Using popular parallel APIs such as OpenMP for multithreading and MPI for distributed memory systems, we aim to demonstrate how parallelization can significantly reduce execution time while maintaining the algorithm's effectiveness. The results of this study could pave the way for future research and applications of the dragonfly algorithm in high-performance computing environments.

How the algorithm works

The execution of the dragonfly algorithm begins with the initialization of several arrays: the position vector P representing the positions of all dragonflies, the step vector ΔP representing their velocities, the position of food sources F, and the position of enemies E. The algorithm then enters an iterative loop where, for each dragonfly, coefficients are calculated to update its internal state. These coefficients are derived from the observed behaviors of dragonflies in nature, which include:

- **Alignment**: The tendency to match the velocity of neighboring dragonflies, calculated as:

  $$A_i = \frac{\sum_{j=1}^{M} V_j}{M},$$

  where V_j is the velocity of the j-th neighbor and M is the number of neighbors.

- **Separation**: The tendency to avoid collisions with other dragonflies, given by:

  $$S_i = -\sum_{j=1}^{M} (P - P_j),$$

  where P_j is the position of the j-th neighbor and P is the current dragonfly's position.

- **Cohesion**: The tendency to move toward the center of the group, expressed as:

  $$C_i = \frac{\sum_{j=1}^{M} P_j}{M} - P.$$

- **Attraction to food**: The tendency to move toward a food source, represented by:

  $$F_i = F^+ - P,$$

  where F^+ is the position of the food source.

- **Distraction from enemies**: The tendency to move away from enemies, modeled as:

$$E_i = E^- + P,$$

where E^- is the position of the enemy.

Once these coefficients are calculated, the step vector ΔP for each dragonfly is updated using the formula:

$$\Delta P_i^{t+1} = sS_i + aA_i + fF_i + cC_i + \omega \Delta P_i^t,$$

where s, a, f, c, and ω are weights for separation, alignment, attraction, cohesion, and inertia, respectively. These weights are randomly chosen within the interval $[0,1]$. The new position of each dragonfly is then computed as:

$$P_i^{t+1} = P_i^t + \Delta P_i^{t+1}.$$

If a dragonfly has no neighbors, it is assumed to move randomly. This is modeled using a random walk function, often implemented as a Lévy flight:

$$P_i^{t+1} = P_i^t + \text{Levy}(d),$$

where d is the dimension of the position vector. The Lévy flight function is defined as:

$$\text{Levy}(d) = 0.01 \times \frac{r_1 \times \sigma}{|r_2|^{1/\beta}},$$

where r_1 and r_2 are random numbers uniformly distributed in $[0,1]$, and σ is calculated as:

$$\sigma = \frac{\sin\left(\frac{\beta\pi}{2}\right)\Gamma(1+\beta)}{\Gamma\left(\frac{\beta+1}{2}\right)\beta 2^{(\beta-1)/2}},$$

with $\Gamma(x)$ being the gamma function and β a constant. Alternatively, Brownian motion can be used instead of Lévy flight, which has shown better performance in some cases [5].

2 General approach

With the overall procedure outlined, we can now present a naive implementation of the dragonfly algorithm. This initial implementation serves as a foundational baseline for further optimization and parallelization. The following code snippet illustrates the core structure of the algorithm:

```
void dragonfly_algorithm (Dragonfly *d,
                          float *average_speed,
                          float *cumulated_pos,
                          float *food,
                          float *enemy,
                          unsigned int N,
                          unsigned int random_seed) {

  unsigned int dimensions = d->dim;
  float S;
  float A;
  float C;
  float F;
  float E;
  float levy;

  float *cur_pos;
  float *cur_speed;

  for (unsigned int j = 0; j < d->N; j++) {
    unsigned random = random_seed + j;
    cur_pos = d->positions + j * dimensions;
    cur_speed = d->speeds + j * dimensions;

    // compute speed = sSi + aAi + cCi + fFi + eEi + w

    for (unsigned int i = 0; i < dimensions; i++) {
      S = ((cumulated_pos[i] - cur_pos[i]) / (float)N);
      A = average_speed[i];
      C = (cumulated_pos[i] / (float)N) - cur_pos[i];
      F = food[i] - cur_pos[i];
      E = enemy[i] + cur_pos[i];
      levy = RAND_FLOAT(1.0, &random);

      cur_speed[i] *= d->w.w;
      cur_speed[i] += d->w.s * S;
      cur_speed[i] += d->w.a * A;
      cur_speed[i] += d->w.c * C;
      cur_speed[i] += d->w.f * F;
      cur_speed[i] += d->w.e * E;
      cur_speed[i] += levy;

      cur_pos[i] += cur_speed[i];
    }
  }
}
```

Listing 1: first implementation of the dragonfly algorithm

After initializing the velocities and positions of agents, represented by two separate matrices with dimensions equal to the number of agents in the population times the problem's dimensionality, the algorithm begins its main execution loop. This loop runs for a fixed number of iterations, stored in the variable $d \to N$. During each iteration, the algorithm retrieves the position and velocity of the current dragonfly, represented by the pointers *cur_pos* and *cur_speed*, along with the cumulated coordinates of its neighbors. The execution proceeds with the following steps:

- Retrieve the current speed and position of the j-th dragonfly.
- Compute the separation, alignment, cohesion, food attraction, and enemy distraction vectors, storing them in local variables with their respective abbreviations.
- Calculate the product of the weights and the individual coordinates of the velocity array.
- Update the speed and position of the dragonfly, incorporating Brownian motion.

Notice that the computation of velocities and position updates is compacted into the same **for** loop. This optimization reduces execution time by performing all calculations locally within the loop, eliminating the need for auxiliary arrays to store intermediate coefficients. The rationale for dividing the separation coefficient by the number of neighbors will be explained later in this paper in section 3.

3 Critical analysis of the coefficients

During our implementation we discovered some ambiguities in the original paper:

- In the original paper [3], the term ”neighbors” is used frequently without providing a precise definition. While a general distance metric, such as Euclidean distance, could be assumed, the paper also claims that the algorithm runs in linear time with respect to the number of dragonflies and the iteration count. This claim would not hold if Euclidean distance were used, as calculating the distance between each pair of dragonflies would result in a time complexity of $O(N^2)$.

  A possible solution to achieve linear time complexity might involve using advanced geospatial algorithms, but we doubt this was the original intention of the authors. To address this ambiguity, we decided to divide the population into chunks and progressively merge these chunks until only one remains (as detailed in Section 3). This approach not only resolves the issue of defining neighbors but also mitigates the problem of premature convergence, which is common in swarm optimization algorithms.

- For the algorithm to be robust, it must ensure that no parameter values lead to divergence. As an initial solution, we introduced a `max_speed` parameter that changes over time to ensure stability. However, since this parameter was not described in the original paper, we omitted it in later implementations.

  This omission led to instabilities, particularly with the enemy coefficient (if it exceeded 0.5) and the separation coefficient. For example, the separation coefficient could cause exponential growth: if the distance between a dragonfly and the average position of the population is very small (e.g., 0.001) but there are many dragonflies in the neighborhood (e.g., 1000), the coefficient would become large (e.g., 1). In the next iteration, the dragonflies would move far from the center, causing the distance to grow exponentially.

  To address this, we modified the separation formula as follows:

$$S_i = -\frac{\sum_{j=1}^{M}(P - P_j)}{M},$$

  where M is the number of neighbors. This normalization ensures that the separation coefficient remains stable regardless of the number of neighbors.

- The enemy coefficient, as described in the original paper, does not behave as intended. Specifically, the sum of vectors does not necessarily point outward from the enemy (e.g., when position vectors are aligned). We assumed this was a typo, but even using the difference of vectors would not suffice. If the enemy is too close to the dragonfly, the difference vector would become smaller instead of larger, leading to incorrect behavior.

  A potential solution would be to normalize the difference vector by its length (i.e., use the inverse of the vector length). However, this formulation would deviate significantly from the original paper. As a result, we decided to retain the original formulation and evaluate its behavior empirically.

Chunk-based Optimization

As mentioned earlier, the dragonfly algorithm, like many other swarm optimization solutions, has a high chance of getting stuck in local minima, limiting its ability to find better candidates. One common approach to mitigate this issue in swarm algorithms is to avoid relying solely on Euclidean distance for neighborhood calculations. Instead, the population can be divided into random groups, which are progressively merged over time until only one group remains. This strategy allows for greater exploration in the early stages and focuses on exploitation as the algorithm converges.

In order to do that, and efficently, we divided the search space into chunks, with each thread processing an equal number of chunks. Every x iterations, these chunks are merged into larger logical chunks, each containing a power of 2 smaller chunks. These logical chunks act as neighborhoods for the dragonflies.

To share the contributions of each chunk, we used a modified butterfly algorithm where the contributions from all elements in a chunk is shared across all the relevant threads. As an optimization, if a chunk is entirely contained within a single thread, the contributions are shared locally without inter-thread communication.

Figure 1 illustrates the communication process. The circles represent different chunks, and the colors represent logical chunks. The blue arrows indicate communication within a process, while the red arrows represent communication across processes.

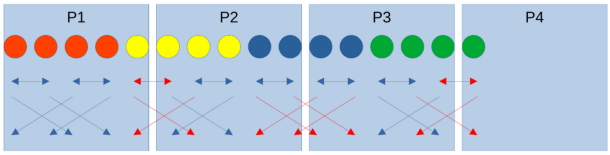

Figure 1: Illustration of communication between chunks.

Chunk-based Optimization Version 2

To improve the previous algorithm, we simplified the chunk structure by removing the first chunk block abstraction. This modification ensures that there is only one type of chunk, eliminating an unnecessary layer of abstraction, and decreasing the risk of off by one error. In Figure 1, the circles now represent butterflies, and the colors correspond to logical chunks.

Additionally, we modified the communication algorithm to address a limitation of the butterfly algorithm, which requires the number of participating threads to be a power of two. To overcome this constraint, we implemented a two-step approach: first, contributions are accumulated into a single element, and then the result is shared with all participating processes. This approach ensures

compatibility with any number of threads while maintaining efficient communication.

4 Parallelization with MPI

The Message Passing Interface (MPI) is a portable message-passing standard designed to function on parallel computing architectures. For our implementation, it is useful because it allows us to express multiprocess communication across multiple machines.

MPI on chunks

By far the biggest challenge in the parallelization with MPI was the communication of the chunks' partial computations (given that a chunk may or may not span multiple processes). MPI has an atomic function called *sendrecv*, that permits sending a message from a thread while receiving from (possibly) another thread. This is very useful, but it could have caused deadlocks if every process has to communicate with multiple chunks. So in order to use that, we needed to ensure that each process is responsible for the accumulation of one chunk. We did this by ensuring that if a chunk starts in another process and ends in this one, the process communicates its contribution to the left process. Then the accumulation is performed and the result is shared back to the current block.

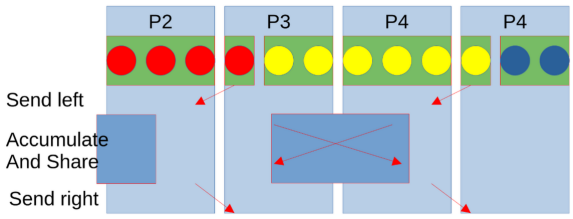

Figure 2: Illustration of communication between processes

The algorithm is divided in steps (isolated by an MPI barrier):

- **Send left**: First, the process that has the end of a block shares its contribution to the left.
- **Accumulate and share**: Then it is accumulated in the first process and broadcast back to all participating processes.
- **Send right**: Then the rightmost process in the block receives the accumulated value from the left.

In this way, the communication is efficient, and looking at the results, it scales very well. More on that in section 6.

MPI and training

Since the parallelization with MPI is highly efficient, it becomes feasible to refine the hyperparameters of small problem instances by scaling the computation across hundreds of cores. In the results section, we demonstrate how the hyperparameters for smaller instances of the dragonfly algorithm were trained on a shifted and rotated Rastrigin function, which is commonly used as a benchmark for optimization algorithms.

The learned hyperparameters were then fed back into the algorithm to enhance its performance. This iterative process was repeated twice, allowing the algorithm to progressively improve its ability to navigate complex search spaces. By leveraging the scalability of MPI, we were able to significantly reduce the time required for hyperparameter tuning while maintaining the accuracy of the results.

5 OpenMP optimization

OpenMP is a low-level API that allows us to execute a sequence of instructions on different threads belonging to the same CPU core, thus improving the performance of our implementation. Thanks to a compiler that supports this technology (in our case, *gcc*), we can include pragma directives in our code, which will wrap the set of instructions that we are interested in executing in parallel. This feature can be used to speed up the execution of some slow functions like the *computation-accumulate*.

computation_accumulate

This function, after defining the starting and ending points, computes the cumulated sums of dragonfly positions and speeds and saves the indexes of dragonfly positions with the best fitness, using them to set the next locations of food and enemies.

```
1  unsigned dim = d->dim;
2  float *cumulated_pos = current_chunk->comp.
       cumulated_pos;
3  float *cumulated_speed = current_chunk->comp.
       cumulated_speeds;
4  zeroed(cumulated_pos, dim);
5  zeroed(cumulated_speed, dim);
6
7  float next_enemy_fitness = FLT_MAX;
8  float next_food_fitness = FLT_MIN;
9  unsigned int indexes[2] = {0, 0};
10 end = end - d->start;
11
12 for (unsigned int k = start - d->start; k < end; k++) {
13    float * iter_pos = d->positions + dim * k;
14    float * iter_speed = d->speeds + dim * k;
15    sum_assign(cumulated_pos, iter_pos , dim);
16    sum_assign(cumulated_speed, iter_speed , dim);
17
18    float fitness = d->fitness( iter_pos ,
19                                seed, dim,
20                                fitness_data );
21    if ( fitness > next_food_fitness ) {
22       indexes[0] = k;
23       next_food_fitness = fitness ;
24    }
25    if ( fitness < next_enemy_fitness ) {
26       indexes[1] = k;
27       next_enemy_fitness = fitness ;
28    }
29 }
```

```
30
31  memcpy(current_chunk->comp.next_food,
32         d->positions + indexes[0] * dim,
33         sizeof(float) * dim);
34  memcpy(current_chunk->comp.next_enemy,
35         d->positions + indexes[1] * dim,
36         sizeof(float) * dim);
37
38  current_chunk->comp.next_enemy_fitness =
        next_enemy_fitness;
39  current_chunk->comp.next_food_fitness =
        next_food_fitness;
40  current_chunk->comp.n = end - (start - d->start);
```

Listing 2: the serial version of *computation_accumulate*

The implementation of a parallelized approach in this case is hindered by the fact that threads should read and write to the same memory areas like the *cumulated_pos* and *cumulated_speed*, which could be difficult to execute independently from each other due to race conditions. In order to avoid this problem, we split the current logical chunk into smaller ones, each one of them keeps track of: represents:

- the local cumulated positions
- the local cumulated speeds
- the best local enemy and food fitnesses
- the indexes of the positions of dragonflies with the best fitnesses

Parallel executions take care of a particular subchunk, also defining the initial and final positions by calculating the starting index as $thread\_ID \times elems\_per\_thread$ and the end one as $start\_index + ratio$; the final thread, however, could be responsible for more dragonflies than others since the number of threads does not always divide the number of elements perfectly. The individual threads then proceed to calculate the cumulated sums and the fitnesses of the current dragonfly, saving its position index if it maximizes the food fitness or minimizes the enemy fitness.

```
1   LogicalChunk *temp_chunks =
2       (LogicalChunk *)malloc(sizeof(LogicalChunk) *
        nr_threads);
3   unsigned int elems_per_thread = (end - start)/
        nr_threads;
4
5   #pragma omp parallel num_threads(nr_threads)
6     {
7       unsigned int thread_id = omp_get_thread_num();
8       void * tmp_fitness_data = NULL;
9       if (d->fitness_data_size != 0) {
10        tmp_fitness_data = malloc(d->fitness_data_size);
11        memcpy(tmp_fitness_data,
12               d->fitness_data,
13               d->fitness_data_size);
14      }
15
16      memcpy(&temp_chunks[thread_id], current_chunk,
        sizeof(LogicalChunk));
17      unsigned int thread_start = start + (thread_id *
        elems_per_thread);
18      unsigned int thread_end = thread_id == (nr_threads
        - 1) ? end: min(thread_start + elems_per_thread,end);
19      unsigned int local_seed = 0;
20
21      inner_new_computation_accumulate(
22        d,
23        &temp_chunks[thread_id],
24        &local_seed,
25        thread_start,
26        thread_end, tmp_fitness_data
27      );
28      if (d->fitness_data_size != 0) {
29        free(tmp_fitness_data);
30      }
31    }
32
33
```

Listing 3: the parallel version of *computation_accumulate*

After the join of all threads, the program performs a reduction of subchunks, calculating the optimal values. This methodology can be seen as a single critical section executed only once after all the parallel processing has ended; then as a final step, the positions and fitnesses of food and enemies are updated, without the need to define critical sections which would force the processor to update the position buffers sequentially.

```
1
2   memcpy(current_chunk,
3          &temp_chunks[0],
4          sizeof(LogicalChunk));
5
6   for (unsigned int i = 1; i < nr_threads; i++) {
7     for (unsigned int k = 0; k < dim; k++) {
8       current_chunk->comp.cumulated_pos[k] +=
9           temp_chunks[i].comp.cumulated_pos[k];
10      current_chunk->comp.cumulated_speeds[k] +=
11          temp_chunks[i].comp.cumulated_speeds[k];
12    }
13
14    if (current_chunk->comp.next_food_fitness <
15        temp_chunks[i].comp.next_food_fitness) {
16
17          current_chunk->comp.next_food_fitness =
18          temp_chunks[i].comp.next_food_fitness;
19
20          memcpy(current_chunk->comp.next_food,
21                 temp_chunks[i].comp.next_food,
22                 sizeof(float) * dim);
23    }
24
25    if (current_chunk->comp.next_enemy_fitness >
26        temp_chunks[i].comp.next_enemy_fitness) {
27
28        current_chunk->comp.next_enemy_fitness =
29          temp_chunks[i].comp.next_enemy_fitness;
30
31        memcpy(current_chunk->comp.next_enemy,
32               temp_chunks[i].comp.next_enemy,
33               sizeof(float) * dim);
34    }
35  }
36  free(temp_chunks);
37
```

Listing 4: the parallel version of *computation_accumulate*

6 Final execution and results

Now that we have described all the procedures, it is time to illustrate how the algorithm has been executed. This job has been performed by the High Performance Computing cluster at the University of Trento, which uses a particular technology called PBS to handle multiple tasks at the same time: this feature is useful in our case since, as we will see later, we execute multiple instances of the same problem.

Structure of the execution

As anticipated, we will execute the same problem using multiple instances with different amounts of resources in order to get a complete evaluation of performance. The overall execution uses from 1 to 8 threads and 4 to 2048 cores. Also, we limit the total number of threads to 2048 for cluster limitations, for a total of 34 runs, run twice.

performace theory

After the end of all tasks, it is possible to check how much gain in performance we have obtained. To do this, it is necessary to introduce some theoretical concepts that will provide us with the tools to give us a good metric system to evaluate our results. Speedup is defined as the ratio between the execution times of the serial and parallel implementations, so the formula will be:

$$Speedup = \frac{T_{serial}}{T_{parallel}}$$

From this definition, it is obvious that the greater the speedup, the better the parallel implementation performs. There is a particular case where the execution time of the denominator can be defined as $T_{parallel} = T_{serial}/p$, where p is the number of cores: this is called *linear speedup*, and in this situation it perfectly matches the classic speedup coefficient. Efficiency is another important parameter, and it is defined by the ratio of the speedup and the number of cores, giving the following formula:

$$E = \frac{Speedup}{p}$$

Also in this case, the efficiency decreases due to the increase in the number of cores. In the case of linear speedup, we get that the efficiency is equal to the inverse of p.

results

Now it is possible to show the results obtained from the cluster. Using the configuration with 4 cores and 1 thread will be considered equal to the serial time. The reason for this approach is that a single core computation would require 1024 times the workload of the 1024 process version, but we calibrated the 1024 core version to take 3 minutes; so it should take around $3 \times 1024/4 = 768$ min $\approx$ 13h. Computing with a single core would have taken 4 times that.

T/P	4	8	16	32	64	128	256	512	1024	2048
1	27879.62	14531.01	7700.45	3628.50	1873.90	1110.29	551.18	268.84	168.71	100.21
2	33202.40	14243.06	5811.25	3274.74	1946.32	903.55	425.83	249.58	129.12	–
4	13136.02	8843.29	4623.91	2419.19	1460.78	724.75	461.13	260.10	–	–
8	15110.34	11308.10	3850.66	2345.90	1281.55	737.66	541.53	–	–	–

Table 1: Time results

T/P	4	8	16	32	64	128	256	512	1024	2048
1	1	1.92	3.62	7.68	14.88	25.11	50.58	103.70	165.25	278.21
2	0.84	1.96	4.80	8.51	14.32	30.86	65.47	111.71	215.93	–
4	2.12	3.15	6.03	11.52	19.09	38.47	60.46	107.19	–	–
8	1.85	2.47	7.24	11.88	21.75	37.79	51.48	–	–	–

Table 2: Speedup results

Efficiency	4	8	16	32	64	128	256	512	1024	2048
1	1	0.96	0.91	0.96	0.93	0.78	0.79	0.81	0.65	0.54
2	0.42	0.49	0.60	0.53	0.45	0.48	0.51	0.44	0.42	–
4	0.53	0.39	0.38	0.36	0.30	0.30	0.24	0.21	–	–
8	0.23	0.15	0.23	0.19	0.17	0.15	0.10	–	–	–

Table 3: Efficiency results (normalized)

As we can see, the speedup increases almost linearly with the number of processes used. In the efficiency table, it is shown that the efficiency stays close to 1 until 512 processes, then it drops down. This probably happens because the distance between the nodes is increasing (maybe through a slower switch). Instead, increasing thread parallelism is not as effective, and it seems to give a 13% improvement when passing from 1 thread to 2 threads, 49% to 4 threads, and 53% to 8 threads. It seems that the thread-level parallelization can be improved; however, it seems that there is a hard limit, and we think it has to do with MPI core pinning. However, the MPI scaling can replace OpenMP even on the same machine, providing better parallelism.

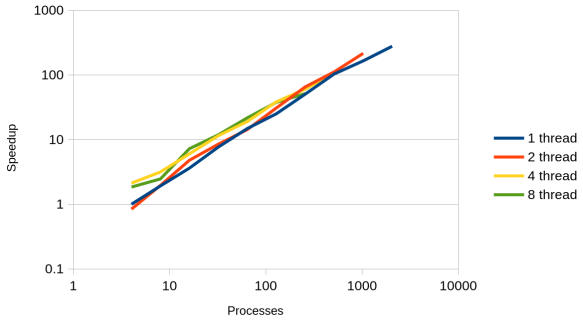

Figure 3: Speedup graphic

Efficiency tends to present unpredictable behavior. In fact, the coefficients grow for some particular setups, while for others, the numbers decrease. If we take a look at the plot below, it shows that there is high variance in the results.

T/p	4	8	16	32	64	128
1	1	1,02	0,97	0,99	0,92	0,97
2	1,04	1,03	0,99	1,06	1,03	0,87
4	1,01	1,01	1,05	1	1,02	1,1
8	1,02	1,01	1,01	1,08	1,15	1,01

Table 4: Efficiency results

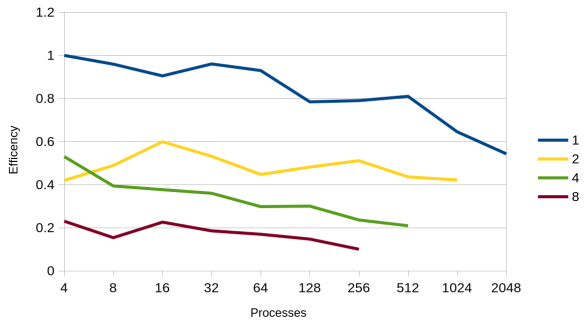

Figure 4: Efficency graphic

7 Final execution and results

To sum up, the presented code is highly flexible and scalable, and it could be used for small optimizations on low-end systems as well as training other methods in an HPC architecture. The code is available on GitHub, and it can be freely used in different applications.

References

[1] Craig Reynolds. Flocks, herds, and schools: A distributed behavioral model. *ACM SIGGRAPH Computer Graphics*, 21:25–34, 07 1987.

[2] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1:28–39, 12 2006.

[3] Seyedali Mirjalili. Dragonfly algorithm: A new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems. *Neural Computing and Applications*, 27, 05 2015.

[4] Yassine Meraihi, Amar Ramdane-Cherif, Dalila Acheli, and Mohammed Mahseur. Dragonfly algorithm: a comprehensive review and applications. *Neural Computing and Applications*, 32(21):16625–16646, Nov 2020.

[5] Hakan Gülcan Çiğdem İnan Acı. H. a modified dragonfly optimization algorithm for single- and multiobjective problems using brownian motion. *Computational Intelligence and Neuroscience*, 2019.