Parallel dragonfly algorithm

High Performance Computing for Data Science course 2024/2025

Mattia Santaniello[1] and Alessio Zeni[2]

[1]University of Trento

August 24, 2025

Abstract

Dragonfly algorithm is an approach used to solve optimization problems through the emulation of dragonflies movements, but it lacks scalability. In this paper the authors propose an approach based on parallelization using dedicated libraries for high performance multiprocess applications like MPI, and 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) [**?**], which simulated flocking behavior, and gained significant traction in Computer Science with the Ant Colony Optimization algorithm (1995) [**?**]. 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 [1], 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, 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 [2].

2 General approach

Now that we have seen the overall procedure it is possible to write a naive implementation of the algorithm in order to have a base for further developments. An example of the implementation can be observed in the following code snippet:

```
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 the initialization of velocities and positions of agents, which will be represented by two separated matricies with the same dimension equal to the number of members of our population times the dimension of our problem, the algorithm starts to execute a for loop with a fixed amount of iteration stored in the variable $d \to N$, retrieve the position and the velocity of the current dragonfly, represented by pointers *cur_pos* and *cur_speed*, and the cumulated coordinates of its neighbours. The execution then proceed with the following steps:

- retrieve current speed and position of the j-th dragonfly
- compute separation, alignment, cohesion, food attraction and enemy distraction vectors, stored into local variables named with their abrreviations
- calculate product between weigths and and the single coordinates of the velocity array
- update speed and position the dragonfly considering also brownian motion

Notice how the computation of velocities and speed coordinates is compacted into the same **for** loop; this allows to reduce time execution, because the overall calculation is made locally all in the same place, leaving out the necessity of using auxiliary arrays for the coefficients. The reason why we divide the separation coefficient for the number of neighbours will be explained later in this paper

chunks

As anticipated earlier dragonfly algorithm, as many other swarm optimization solutions, has a good chance to be stuck into a local minima, without the possibility to find better candidates. We can avoid this and improve this approach introducing groupments of dragonflies into chunks, calculating local minima of different areas and compare them in order to get as close as possible to the global solution.

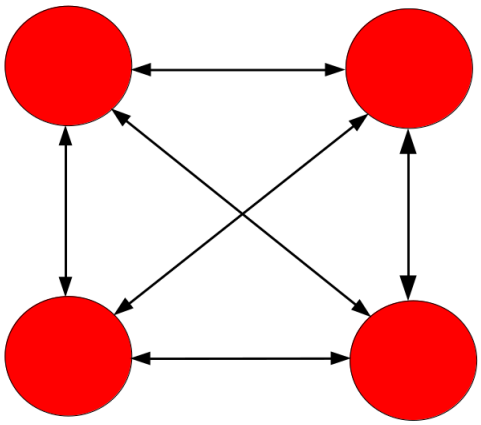

Figure 1: illustration of communication between chunks

The implementation of this process is quite simple: each chunk will contain a subset of components of the global population that will specialize to compute the point with the best local fitness for both enemies and food of that specific region, then the local status will be communicated to all other chunks, updating in this way the global status of the program thanks to the update of positions, and the local configuration of each region; the algorithm ends displaying the best fitness found during the execution.

3 Critical analysis of the coefficients

From this implementation we discovered some critical points that should be improved:

- In the original paper [3] and in the algorithm often uses the therm neighbours without giving a precise definition. Assuming a general distance metric (eg Euclidean Distance), could be an option, but later in the paper it describes how the alghorithm runs in linear time with respect to the number of dragonflies and the iteration count. This would be false in case of euclidean distance, because in order to define for each butterfly the neighbours, we should calculate the distance between each pair of dragonflies, which would lead to a time complexity of $O(N^2)$. Maybe using some complex geospatial algorithms it could be possible to archive that, but we doubt this was the original intention of the authors. Later in the paper also another crithical problem in respect to other algorithm was the premature convergence. So we decided to divide the population in chunks, and progressively merge chunks until we will remain with only one. So in this way we addressed both problems.

- In order to be a solid algorithm it must be garanteed that does not exist some values that would make the solution diverge. As a first approach we introduced a max_speed parameter that would change over time and would assure stability. But given that in the original paper it was not described we omitted from later implementations. This lead to instabilities from the enemy coefficient (if bigger than 0.5), but mainly from the separation coefficient. This is a problem, because we could be see the coefficient as n times the distance between the dragonfly and the average position of the population. For example if the distance is 0.001, but there are 1000 dragonflies in the neighbourhood, then the coefficient would be 1, at the very next iteration the dragonflies would have a distance of 1 from the center and so on, causing an exponential growth. In order to fix this we modify the separation formula as: $S_i = -\frac{\sum_{j=1}^{M} P - P_j}{M}$.

- The enemy coefficient does not represent what the author describes. The sum of vectors does not necessarily point outwards in respect to the enemy (eg with aligned pos vectors). We assumed a typo, but even the difference of the vectors would not be enough, because if the enemy is too close to the dragonfly, the difference would get smaller and not bigger. A possible solution would be to normalize the difference vector with the inverse of the vector length, but the formulation would have been to different from the original one. So we decided to keep the original formulation and see if it actualy would do something.

4 Parallelization with MPI

OpenMPI is a library that enables code execution on multiple cores, wheter they belong to the same CPU or not; it can be useful in case of performace improvement of those algorihms that require to solve complex tasks, just like the dragonfly algorithm: in fact this library can be easily adapted to the illustrated implementation in order to boost performance.

MPI on chunks

As we said earlier chunks are independent from each otherm but usually they need to comunicate with their neighbours in order to keep updated the global status of the program execution. It is possible to run chunks groups simoultaneously, using MPI builtin function to connect the processes. The instructions computed by each process are available in a custom function called *dragonfly-compute*; after calculating the ID of the initial and final chunks and initializing message structures using the **offsetof** macro the function will execute the update loop that calculates the new velocities and positions of the dragonfly for each chunk associated. To do this we first need to update the current location of enemies and food present in the chunk using the function **computation_status_merge**, using

the position of food and enemies of other chunks retrieved by the function **message_broadcast**; finally the process will proceed to update current position and velocities of dragonfiles using the function **dragonfly_compute_step** illustrated earlier

MPI and parameters

Since MPI has been useful in the parallelization of the chunks then it can also become useful in the training of parameters.

5 OpenMP implementation

OpenMP is a low level API that allows us to execute a sequence of instructions on different threads belonging to the same CPU core, improving in this way the performances of our implementation. This feature can be used to speed up the execution of some slow functions, in our case these will be dragonfly-compute-step and message-accumulate

General use

First of all it is necessary to understand how OpenMP should be inserted in the code: first of all we have to use a compiler that supports this technology, in our case we have gcc, which allows OpenMP execution thanks to the flag -fopenmp. Second we need to include a pragma directive in our code, which will wrap the set of instructions that we are interested to execute in parallel. Since the implemented results will perform loops mainly then it is fundamental to calculate the work balance between the number of iterations, if the main body is a loop, and the number of threads available. Each thread has to define the starting and the end point of the loop that has to execute, calculating the starting index as $rank * ratio$ and the end one as $ratio * (rank + 1)$, where *rank* is the ID of the thread

dragonfly-compute-step

One of the two functions that we are interested to execute using OpenMP is *dragonfly-compute-step*: if we take a look on the serial version we notice that each computation step is independent from other iterations. This means that the execution flow can be split in many subtasks assigned to threads. The following code snippet illustrate how all the procedure has been implemented

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

  unsigned int base_random = rand_r(&d->seed);
  unsigned int rest = d->N % NR_THREADS;
  unsigned int ratio = d->N / NR_THREADS;

  ...

  #pragma omp parallel num_threads(NR_THREADS)
  {

    unsigned int rank = omp_get_thread_num();
    // printf("rank=%d\n", rank);
    unsigned int base = ratio * rank;
    unsigned int limit = ratio * (rank + 1);
     inner_dragonfly_step(d, average_speed,
                          cumulated_pos, food,
                          enemy, N,
                          base, limit, base_random);
  }

 ...

 if (rest != 0) {
    unsigned int r_base = ratio * NR_THREADS;
    unsigned int r_end = d->N;

     inner_dragonfly_step(d, average_speed,
                          cumulated_pos,
                          food, enemy, N,
                          r_base, r_end, base_random);
  }
 }

 weights_step(&d->w);

 }
```

Listing 2: parallelized dragonfly-compute-step

After we got all basic informations the *inner_dragonfly_step* function can be executed, wrapping it into the **pragma** directive that we have seen earlier: in this way each thread will perform the necessary operations only on a specific portion of the overall positions and speeds arrays. If the number of threads does not divide perfectly the size of the arrays then we compute the rest returned from the ratio calculation and the remaining cells will be processed using the serial approach.

computation_accumulate

The *computation_accumulate* must compute the cumulated positions and speeds of the chunk and find the best food and enemy positions. The implementation of a parallelized approach in this case is more complicated to achieve, because threads should read and write on the same memeory areas which could be difficult to execute independently from each other. This time it is necessary to define local buffers for each thread, so they can store temporary results without the risk of using overlapping memeory locations.

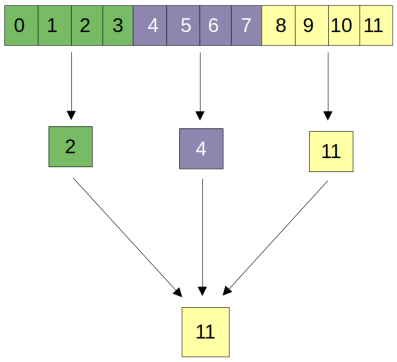

Figure 2: Illustration of the reduce process performed by the computation_accumulate function

The parallel section provides that each thread must compute the local cumulated sums of velocities and positions, then they must select the position with the best fitness value in order to find the best local position of food and enemies. Once the local execution is completed it will update the global status of the chunk using a critical section, with one thread that can execute that specific portion of the program

6 Final execution and results

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

structure of the execution

As anticipated we will execute the same problem using multiple instances with different amount of resources in order to get a complete evaluation of performance. The overall execution use from 1 to 8 threads, and 4 to 1024 cores

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 give us the tools that will give us a good metric system to evaluate our results.Speedup is defined as the ratio between the the execution times of the serial and parallel implementations, so the formula will be equal to:

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

from this definition it is obvious that the more the speedup is the more the parallel implementation performs better. There is a particular case where the time execution 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 match with 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 decrease due to the increase of the number of cores. In case of linear speedup we get that the efficiency is equal to the inverse of p

results

Now it is possible to show results obtained from the cluster, using the configuration with 4 cores and 1 thread will be considered equal as the serial time

T/p	4	8	16	32	64	128
1	1	2,03	3,87	7,94	14,76	31,19
2	1,04	2,05	3,97	8,50	16,43	27,90
4	1,01	2,03	4,21	8,04	16,37	35,26
8	1,02	2,02	4,02	8,62	18,32	32,26

As we can see the speedup tends to increase when the number of cores increase, however when we use more threads it happens that the speedup start to decrease at a certain point for some core configuration. As we can see from the plots it seems there are threshold at some specific points, depending on the number of threads used; notice how the inclinations of the lines in the plot tends to be less and less evident.

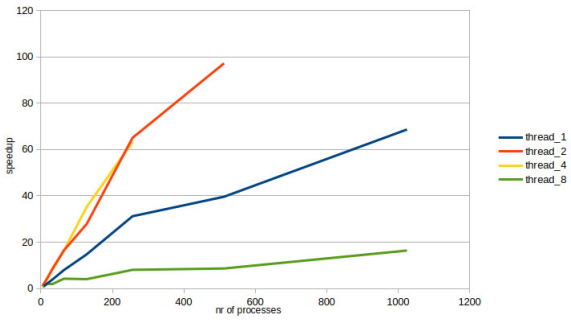

Figure 3: speedup graphic

Efficiency tends to present an unpredictable behaviour, in fact the coefficients grows for some particular setups, for others instead numbers decrease. If we take a look on the plot below it shows that there is a high variance in the datas, with the lines that tends to be more stretched when the number of threads gets higher

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

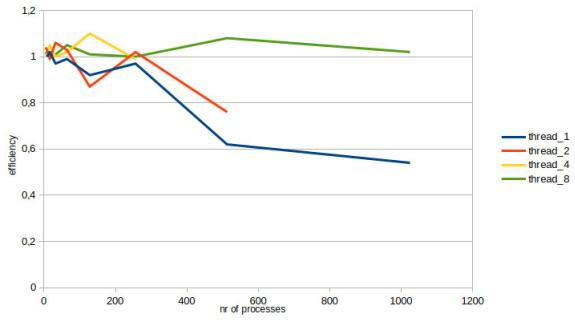

Figure 4: efficiency graphic

References

[1] Yassine Meraihi Amar Ramdane-Cherif Dalila Acheli Mohammed Mahseur. Dragonfly algorithm: a comprehensive review and applications. *Neural Computing and Applications*, 2020.

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

[3] Seyedali Mirjalili. Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problem. *Neural Computing and Applications*, 2016.