Parallel dragonfly algorithm

High Performance Computing for Data Science course 2024/2025

Mattia Santaniello[1] and Alessio Zeni[2]

[1]University of Trento

February 14, 2025

Abstract

Dragonfly algorithm is a new kind of approach used to solve optimization problems through the emulation of dragonflies movements, but it lacks scalability. In this paper the authors try to set up an approach based on parallelization thanks to the use of API for high performance computing like MPI and OpenMP, then there will be a comparison of execution times between presented implementation and the classic approach based on serial programming, in order to see if parallelization could lead to a performance improvement and possible future extensions.

1 Introduction

Recent years have seen the raise and evolution of swarm intelligent algorithms, based on meta-heuristic approach where single individuals of a complex population follow an exploration-exploitation methodology to find the best solution in their local search space; each member of the group work independently from the others, leading to a subdivision of the labour which decrease the overall amount of execution time, working as decentralized and parallel units; but also, due to the fact that they mimic the behaviour of biological life forms, it is possible then that they could generate random patterns through the interaction with each other. The dragonfly algorithm has been proposed for the first time in 2016, and its purpose is to simulate the behaviour of a dragonfly population according to the principles of swarm intelligent algorihms seen previously. It has gained more and more popularity: it is in estimated that during 2019 more than 300 scientific papers cited the algorithm [?], due to the fact that it has demonstrated some characteristics similar to other algoritms of the same kind, however in contrast with other swarm intelligent algorithms however it present less probability to be trapped into a local optima [?]. This paper aims to define a new approach based on popular parallel APIs, and also trying to understand if parallelization could lead to a significant speed up in performances that can be considered by future researches on this algorithm

How the algorithm works

The execution begins with the initialization of some arrays which are the position P of all dragonfiles and the 'step' vector ΔP representing the velocities of each single individual,the position of source foods F and the position of enemies E; then the algorithm executes an iteration loop where, for each dragonfly, we calculate some coefficient that will be used to update the internal state of the program. This procedure is simulated through the observation of repetitive patterns observed in the behaviour of dragonflies, which are:

- Alignment of their velocity with other members of the same group, given by the formula $A_i = \frac{\sum_{j=1}^{M} V_j}{M}$, where V_j is the velocity of the j-th neighbour
- Separation of single members in order to avoid collision, and it is retrieved by the equation $S_i = -\sum_{j=1}^{M} P - P_j$, where P_j is the position of the j-th neighbour and P the current individual position.
- Cohesion towards the center of the group, given by $C_i = \frac{\sum_{j=1}^{M} P_j}{M} - P$
- Attraction to a common source food, expressed by the formula $F_i = F^+ - P$, where F^+ is the position of the current prey and P is the position of the current dragonfly
- Distraction of enemies, represented by the equation $E_i = E^- + P$ where E^- is the position of the current enemy

Once all the parameters have been calculated we can now update the current step vector of each dragonfly using the following equation:

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

where s, a, f, c are respectevly the separation weight, the alignment weight, the attraction weight, the cohesion weight and ω is the inertia weight, all of them are randomly chosen in the interval $[0, 1]$. When all step vectors have been updated we can now set the new position of the dragonflies using this equation, which takes the new step vector as an input:

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

. However if there is a dragonfly which does not have any neighbour then we assume that the overall population moves in a random way: the most common method to do that is to use a random walk function. The previous formula then must be modified into this one here:

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

where d is the dimension of the position vector, and $Levy(d)$ is the Levy flight function, which is calculated in this way:

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

where $r1$ and $r2$ are two random numbers uniformally distributed in the interval $[0,1]$, and σ is a vector calculated using the following formula:

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

where $\Gamma(x)$ is the gamma function [?], and β is a constant.Another valid approach is to apply, instead of Levy flight, the brownian motion model, used especially in gas and particle analysis: this seems to return better performances [?].

2 first 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 dim = d->dim;
    // for each dragonfly
  for (unsigned int j = 0; j < d->N; j++) {
    float *cur_pos = d->positions + dim * j;
    float *cur_speed = d->speeds + dim * j;

    // compute separation : Si = -sumall(X-Xi)
    memcpy(d->S, cumulated_pos, sizeof( float ) * dim);
     scalar_prod_assign (d->S, 1.0/( float )N, dim);
    sub_assign (d->S, cur_pos, dim);
     scalar_prod_assign (d->S, d->w.s, dim);

    // compute alignament: Ai = avarage(Vi)
    memcpy(d->A, average_speed, sizeof( float ) * dim);
     scalar_prod_assign (d->A, d->w.a, dim);

    // compute cohesion: Ci = avarage(Xi)-X
    memcpy(d->C, cumulated_pos, sizeof( float ) * dim);
     scalar_prod_assign (d->C, 1.0 / ( float )N, dim);
    sub_assign (d->C, cur_pos, dim);
     scalar_prod_assign (d->C, d->w.c, dim);

    // food attraction : Fi=X_food - X
    memcpy(d->F, food, sizeof( float ) * dim);
    sub_assign (d->F, cur_pos, dim);
     scalar_prod_assign (d->F, d->w.f, dim);

    // enemy repulsion: E=X_enemy+X
    memcpy(d->E, enemy, sizeof(float) * dim);
    sum_assign(d->E, cur_pos, dim);
     scalar_prod_assign (d->E, d->w.e, dim);

    brownian_motion(d->levy, dim, &d->seed);


    // compute speed = sSi + aAi + cCi + fFi + eEi + w
     scalar_prod_assign (cur_speed, d->w.w, dim);
    sum_assign(cur_speed, d->E, dim);
    sum_assign(cur_speed, d->F, dim);
    sum_assign(cur_speed, d->C, dim);
    sum_assign(cur_speed, d->A, dim);
    sum_assign(cur_speed, d->S, dim);
    sum_assign(cur_speed, d->levy, dim);

    // update current pos
    sum_assign(cur_pos, cur_speed, dim);
  }

  // update weights
  weights_step (&d->w);
}
```

Listing 1: first implementation of the dragonfly algorithm

As you can see this implementation use heavily the sum_assign and similar functions, which we define to ease the computation and keep them similar to the mathematical notation.

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 $d \rightarrow N$, and will repeatetly compute 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 arrays placed in the struct Dragonfly
- calculate product between weigths and arrays
- update speed and position the dragonfly considering also brownian motion

3 Critical analysis of the coefficients

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

- Chunks: In the paper 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.

- Stability: 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 $\max_{speedparameterthatwouldchangeovertimeandwouldassurestability.Butgiventhatintheoriginalpaperitwasnotdescribedweomittedfromla}$ $-\frac{\sum_{j=1}^{M} P-P_j}{N}$.

- 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.

chunks

We can improve this approach introducing groupments of dragonflies into chunks, allowing us to calculate local minima of different areas and compare them in order to get as close as possible to the global minima. which will be considered autonomous agents that will calculate local minima of our fitness function, updating then the global status of by sending messages to all other dragonfly chunks.

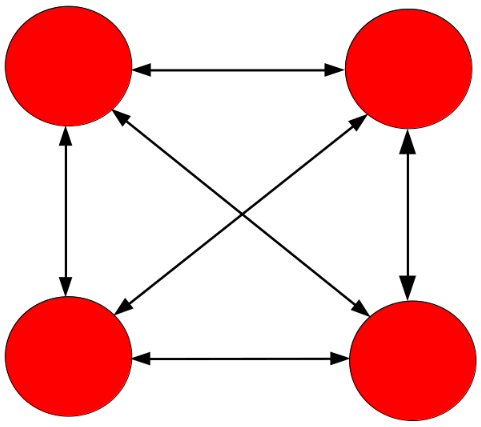

Figure 1: illustration of communication between chunks

A

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. 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

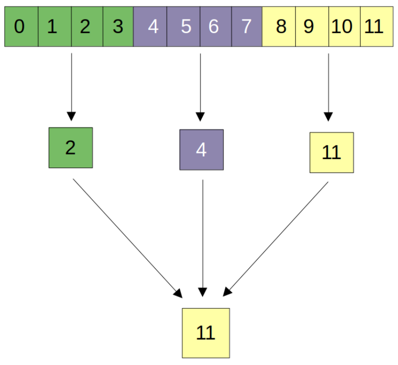

Figure 2: Illustration of the reduce process performed by the computation_accumulate function

References

[1] Chnoor M. Rahman Tarik A. Rashid Abeer Alsadoon Nebojsa Bacanin Polla Fattah Seyedali Mirjalil. A survey on dragonfly algorithm and its applications in engineering. *Evolutionary Intelligence*, 2021.

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

[3] `https://en.wikipedia.org/wiki/Gamma_function`.

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