Parallel Artificial Gorilla Troops Optimizer

High Performance Computing for Data Science Project 2023/2024

Erik Nielsen
mat.238755
University of Trento
38123 Povo TN, Italy
erik.nielsen@studenti.unitn.it

Pietro Ferrari
mat.239952
University of Trento
38123 Povo TN, Italy
pietro.ferrari@studenti.unitn.it

ABSTRACT

Optimization algorithms are able to find optimal solutions to complex problems in a feasible time. But in the case of cost functions with high dimensions size, most of these solutions lack of feasibility. One class of algorithms that are able to improve performances to solve the aforementioned problems are the *Evolutionary Algorithms*, which are based on evolving a population of agents towards an optimum. These algorithms, as they are based on different individuals executing independently different tasks, are ideal to be parallelized. In this paper are presented 2 approaches to parallelize the execution of the Artificial Gorilla Troops Optimization Algorithm, the first based on the *MPI* framework, the other with and hybrid approach with both *MPI* and *OpenMP*.

1 INTRODUCTION

Optimization algorithms are developed to find the best solution in a feasible time in a wide range of fields and on a high diversity of functions. These are divided into two main categories:

- the first which is based on calculus techniques and is deterministic, guarantees the exact results at the cost of highly demanding calculations, making it unfeasible most of the time [8];
- the second is stochastic, all the algorithms that through some randomization steps can make educated decisions, retrieving good outcomes in a feasible time [5].

In the second case, there is a field of metaheuristic algorithms that are based on the natural evolution of species and their social behavior within a group of individuals. These evolutionary-based algorithms [7] and swarm-based optimization solutions [9] are developed taking inspiration from how the offspring mutate, to achieve better fitness than previous individuals, and on how a group of particles explore a problem to retrieve the best solution.
In many cases these algorithms have many different possibilities to be parallelized [2], as they are based on 3 main steps: first, each individual explores the solution space, then each exploits the findings and concludes by gathering all together to return the best solution of the space explored by all the individuals. These structures guarantee the possibility of dividing and parallelizing the computational effort of each individual or different subgroup of individuals and then potentially speed up the required time to compute optimal solutions.
This paper explores and evaluates a possible parallelization implementation of the evolutionary algorithm Artificial Gorilla Troops Optimization Algorithm (GTO) [1].

2 RELATED WORK

The parallelization process is based on an already existing implementation of an optimization algorithm that has a serial execution. This section introduces the Artificial Gorilla Troops Optimization Algorithm [1], which will be recalled as *GTOs*, its serial workflow, and the needed computations and calculation required to accomplish the exploration and exploitation steps needed in the evolutionary algorithms.

2.1 Implementation

The GTOs algorithm is divided into three main steps to update the values of three variables that represent the gorillas in the troops, X which is the vector describing the position of each gorilla, GX as the candidate position of the gorilla in each phase of the algorithm and $X_{silverback}$ as the best solution of the iteration. Then the three steps can be individualized:

- the initialization step, where the hyperparameter β, W and p and the population size N are set;
- exploration step, in which each gorilla explores new solutions in the search space and updates the position values;
- exploitation step, where the candidates exploit the search space near the current $X_{Silverback}$ if a new best candidate is found, it will replace the previous one.

Each of the aforementioned steps has an evolutionary behavior to update the positions described in the following sections.

2.1.1 Exploration Phase. During the exploration phase equation 1 represents how the candidates search in the solution space and update its position.

$$GX(t+1) = \begin{cases} (UB - LB) \times \eta + LB & rand < p \\ (r_2 - C) \times X_r(t) + L \times H & rand \geq 0.5 \\ (X_i) - L \times (L \times (X(t) - GX_r(t)) + & \\ +r_3 \times (X(t) - (GX_r(t))) & rand < 0.5 \end{cases} \tag{1}$$

where the parameters are computed with the following equations.

$$C = F \times \left(1 - \frac{It}{MaxIt}\right) \tag{2}$$

$$F = \cos(2 \times r_4) + 1 \tag{3}$$

$$L = C \times l \tag{4}$$

$$H = Z \times X(t) \tag{5}$$

$$Z = [-C, C] \tag{6}$$

2.1.2 Exploitation Phase. In this phase instead, the candidates are set to decide which approach to use to "follow" the $X_{Silverback}$. In the first case, if $C \geq W$, GX is updated as represented in equation 7.

$$GX(t+1) = L \times M \times (X(t) - X_{silverback} + X(t) \tag{7}$$

where the parameters are computed with the following equations.

$$M_{original} = \left(\left| \frac{1}{N} \sum_{i=1}^{N} GX_i(t) \right|^g \right)^{\frac{1}{g}} \tag{8}$$

$$g = 2^L \tag{9}$$

In the implementation developed for the purpose of the project, M is computed without elevating and rooting to the power of g due to computational purpose. Therefore the baseline used in the section 4 use the equation 10

$$M_{baseline} = \left| \frac{1}{N} \sum_{i=1}^{N} GX_i(t) \right| \tag{10}$$

Furthermore, in the case of $C < M$, the GX is updated based on the equation 11.

$$GX(i) = X_{Silverback} - (X_{Silverback} \times Q - X(t) \times Q) \times A \tag{11}$$

where the parameters are computed with the following equations.

$$Q = 2 \times r_5 - 1 \tag{12}$$

$$A = \beta \times E \tag{13}$$

$$E = \begin{cases} N_1 & rand \geq 0.5 \\ N_2 & rand < 0.5 \end{cases} \tag{14}$$

2.1.3 Serial Algorithm. The aforementioned steps and equations are then set all together in the implementation of the serial algorithm described in 3. The algorithm requires two core methods which are the exploitation() and exploration() once. They are described separately from the serial algorithm because they are crucial and are reused in the parallel implementation.

Algorithm 1 Exploration methods

```
1: $C \leftarrow equation2$
2: $L \leftarrow equation4$
3: for $i \leftarrow 1$ to $N$ do                ▷ Exploration Phase
4:     $X_i \leftarrow equation1$
5: end for                                  ▷ End Exploration Phase
```

Algorithm 2 Exploitation method

```
1: for $i \leftarrow 1$ to $N$ do                ▷ Exploitation Phase
2:     if $|C| \geq 1$ then
3:         $GX_i \leftarrow equation7$
4:     else
5:         $GX_i \leftarrow equation11$
6:     end if
7: end for                                  ▷ End Exploitation Phase
```

Then combining the aforementioned equations to update the values and the method, in pseudocode 3 describes the serial execution of the GTOs.

Algorithm 3 Serial GTO

```
1: $X \leftarrow random(N)$                       ▷ Initialize gorilla population
2: $GX \leftarrow fitness$
3: while Stopping condition do                 ▷ Main Loop
4:     $exploration()$
5:     $GX \leftarrow fitness$
6:     if $GX > X$ then
7:         $X \leftarrow GX$
8:     end if
9:     $X_{Silverback} \leftarrow max(X)$
10:    $exploitation()$
11:    $GX \leftarrow fitness$                    ▷ Create New Group
12:    if New Fitness > Old Fitness then
13:        $X \leftarrow GX$
14:        $X_{Silverback} \leftarrow max(X)$
15:    end if
16: end while‘
17: $Outputs \leftarrow X_{Silverback}, max(fitness)$
```

It is important to note the separation between the exploration and exploitation phases inside the pseudocode, as they will be recalled during the parallelization discussion.

2.2 Complexity

The computational complexity (C) of GTOs [1] depends on the three main parts that compose the algorithms:

- N the initial population, leading to $C \leftarrow O(N)$
- Exploration phase and Exploitation phase, based on T number of maximum iterations and D problem dimensions, leading to $C \leftarrow O(T \times N) + O(T \times N \times D) \times 2$

The final complexity then $C \leftarrow O(N \times (1 + T + TD) \times 2)$ [1].

3 METHODOLOGY AND IMPLEMENTATION

Section 2 described the workflow of the serial implementation of the algorithm, which is the baseline to the development of two parallel approaches, the first using *MPI API* [4], and the second based on a hybrid workflow using also *OpenMP API* [3]. The presented methodologies are reported in the GitHub repository [6].

3.1 Data Dependencies

The meaning of analyzing the data dependencies in a parallel region is based on trying to avoid that different processes and threads do read and write operations on the same variables, creating *data races* issues. To perform this analysis, it took a nested for loop from the exploitation method. As the algorithm is based on updating the variable of the agents by iterating over its array structure, each step of the execution in terms of data dependency is similar to the snippet code that is provided.

```
void exploitation(double C, double L, double lb, double
    ub, double M[],
int gorilla_per_process, Gorilla GX[], Gorilla *
 silverback, Gorilla X[]) {
int j, k;
Gorilla old_silverback;
memcpy(&old_silverback, silverback, sizeof(Gorilla));
if (C >= W)
 for (j = 0; j < gorilla_per_process; j++) {
  for (k = 0; k < DIM; k++) {
   GX[j].coordinates[k] = L * fabs(M[k] / DIM) * (X[j
 ].coordinates[k] - old_silverback.coordinates[k]) +
 X[j].coordinates[k];
   boundaryCheck(ub, lb, &GX[j].coordinates[k]);
  }
 checkForUpdatePosition(&GX[j], silverback, &X[j]);
}
```

Listing 1: Expliotation loop if $C \geq W$

To analyze the data dependencies, it is useful to apply the 3 steps approach. In this particular case:

- Detection, within the nested loop, there are k, j variables that are used to write on the coordinates k of the gorilla j, creating a dependency condition.
- Classification, as the above detection is done in the very same iteration inside the loop, the dependency is considered *not loop carried*. Moreover, the dataflow relation, considering that the $j-th$ gorilla is read and the $k-th$ coordinate is written, is flow-dependence. These dependencies are summarized in table 1
- Removal, from the serial implementation, an approach to remove dependencies in case of usage of *OpenMP* is shown in 3.4.

Location	early/late	line	iteration	access	carried	type
GX	early	9	j	read	no	-
GX	late	12	j	write	no	out
X	early	9	j	read	no	-
X	late	12	j	write	no	flow
M	early	9	k	read	no	-
M	late	9	k	read	no	out
GX[j].xy	early	9	k	write	no	-
GX[j].xy	late	9	k	write	no	flow
X[j].xy	early	9	k	read	no	-
X[j].xy	late	9	k	read	no	flow
old_S.xy	early	9	k	read	no	-
old_S.xy	late	9	k	read	no	flow

Table 1: Data Dependencies

3.2 Common Methods

To exploit the parallel capabilities of GTOs, it is crucial to find the in the algorithm common structure used both in the serial execution and in the parallel one. Therefore, as the algorithm is already self-divided into three core parts, which are the initialization step, exploration, and exploitation phase of the evolutionary algorithm, these will be replicated in both approaches. The reason behind it is that the parallelism is done by creating N_{core} sub-group, in which each will replicate the serial algorithm locally on smaller gorilla troops. Therefore the $exploration()$ (pseudocode 1) and $exploitation()$ (pseudocode 2) methods follow the exact same steps described in 2.1.3.

Meanwhile, the parallel implementation requires a few crucial steps in which the outcomes computed from each parallel process are evaluated all together and part in which the global evaluations are resent to each process. To simplify these steps, pseudocode 4 and 5 are methods that find and set the new global best value (silverback) by evaluating the results of each sub-process.

Algorithm 4 findGlobalSilverback Function

```
1: $Gorilla * in\_data \leftarrow (Gorilla*)in$
2: $Gorilla * out\_data \leftarrow (Gorilla*)out$
3: **if** $in\_data \rightarrow fitness \leftarrow out\_data \rightarrow fitness$ **then**
4:     $memcpy(out\_data, in\_data, sizeof(Gorilla))$
5: **end if**
```

Algorithm 5 updateGlobalSilverback Function

```
1: $MPI\_Allreduce(silverback, \&globalSilverback)$
2: $memcpy(out\_data, in\_data, sizeof(Gorilla))$
```

These aforementioned functions are used commonly in both the parallelization approaches.

3.3 Parallelization with MPI

MPI is a way to program on distributed memory devices. The first parallel approach is entirely based on the *MPI API*. This API permits programming on distributed memory devices, simplifying the parallelization task as each parallel process executes on its own memory space, isolated by the others. These characteristics avoid issues in evaluating how to split memory between the processes, but the automation that provides the causes some inefficiency due to some latency that accure during the exchange of messages between the local memories and the shared one. Algorithm 6 describes the *MPI* parallel implementation

Algorithm 6 Parallel GTO with MPI

```
1: Initialize MPI
2: Set MPI Barrier
3: Create MPI Gorilla Struct
4: $\&myOp \leftarrow findGlobalSilverback()$
5: **for** $i \leftarrow 1$ to $T$ **do**
6:     **if** $rank \leftarrow 0$ **then**
7:         $C \leftarrow equation2$
8:         $L \leftarrow equation4$
9:     **end if**
10:    $MPI\_Broadcast(\&C)$
11:    $MPI\_Broadcast(\&L)$
12:    $exploration()$
13:    $updateGlobalSiverBack(myOp, \&silverback)$
14:    $MPI\_Allreduce(M, global_M)$
15:    $exploitation()$
16:    $updateGlobalSiverBack(myOp, \&silverback)$
17: **end for**
```

The main characteristics of the aforementioned approach can be explained in three parts:

- MPI variables and barrier initialization, where Gorilla data struct and $findGlobalSilverback()$ (pseudocode 4) are created in the MPI API, and a barrier is set to be sure that all the requested resources are allocated before starting the execution, then the main loop starts;
- inside the main loop, *MPI_Broadcast* to send all the hyperparameters to the sub-processes, followed by the exploration phase;
- then, *MPI_Allreduce* to update all the parallel processes, then the main loop ends with the exploitation phase and by updating the global silverback

3.4 Hybrid Parallelization, MPI and OpenMP

The MPI provides a useful API to retrieve simple parallel results. But, to enhance the parallelization efficiency, is possible to apply other frameworks on top of it to exploit the hardware in use. Then, another approach that this paper aims to propose is to apply OpenMP on top of the MPI approach. OpenMP is an API that allows parallelism inside the shared memory, meaning that on the memory of each core, parallelism occurs where every parallel thread has access to the allocated memory.
To apply hybrid parallelization over a serial algorithm, it must define parallel regions where the threads could enhance parallelism, then it is crucial to take into account the data dependencies shown in section 3.1. Therefore, the example of the exploitation snippet has been rewriting as follow:

```
1  void exploitation(double C, double L, double lb, double ub, double M[], int gorilla_per_process, Gorilla GX[], Gorilla *silverback, Gorilla X[], int n_threads) {
2      int j, k;
3      Gorilla old_silverback;
4      memcpy(&old_silverback, silverback, sizeof(Gorilla));
5      if (C >= W){
6       for (j = 0; j < gorilla_per_process; j++) {
7       // tmp variables ot avoid data race conditions
8       Gorilla tmp_GX = GX[j];
9       Gorilla tmp_X = X[j];
10      #pragma omp parallel default(none) shared(tmp_GX, tmp_X, j, C, L, old_silverback, ub,lb, M) private(k) num_threads(n_threads)
11      {
12       #pragma omp for
13       for (k = 0; k < DIM; k++) {
14        tmp_GX.coordinates[k] = L * fabs(M[k] / DIM) * (tmp_X.coordinates[k] - old_silverback.coordinates[k]) + tmp_X.coordinates[k];
15        boundaryCheck(ub, lb, &tmp_GX.coordinates[k]);
16       }
17      }
18      #pragma omp critical
19      GX[j] = tmp_GX;
20      checkForUpdatePosition(&GX[j], silverback, &X[j]);
21      }
22     }
```

Listing 2: Expliotation loop if $C \geq W$ - Hybrid Parallelization

In particular, there are three *pragmas* directives to execute the thread parallelization:

- Initialization of the parallel directives (line 10), in which are set the private and shared variables, and the number of threads retrieved at the beginning of the execution;
- Parallelization of the for loop (line 12);
- Apply the changes over the agents in a critical part of the parallelization (line 18), this part of the algorithm is delicate when parallelization is called, therefore only one thread must work on it.

Moreover, a crucial point is to create temporary variables within the outer loop. The reason to do so is when a shared variable is accessed simultaneously by j, k indexes there is a data race condition. Therefore in the outer and non-parallelize loop, there are instantiate the temporary variables.

3.5 PBS directives

To submit the parallel execution to the cluster (description in section 4.1), it is necessary to run a bash file with the proper PBS directives to allocate the right amount of resources requested by the parallel algorithm.

```
#!/bin/bash
#PBS -l select=1:ncpus=N-CORES:mem=2gb -l place=pack:excl
#set max execution time
#PBS -l walltime=0:20:00
#imposta la coda di esecuzione
#PBS -q short_cpuQ
module load mpich-3.2 # Load the required module
mpirun.actual -n N-CORES ./executables/GTO_parallel_DIM
```

Listing 3: PBS Job submission

In particular, each execution was set to have a wall time of 20 minutes, as the serial algorithm was set to have results within 20 minutes. Then it was decided to have in all tests 1 node exclusively allocated with N_{cores} available for the execution to retrieve the best performances for each run. It eas selected this configuration mainly due to that with 64 allocated cores the parallelism had difficulties in returning consistent results if each used CPU wasn't inside the same node, most probably due to some issue in coordinating latency between nodes.

3.6 Benchmark optimization functions

The aforementioned algorithms, both serial and parallelized approaches have been then tested on three different benchmark optimization functions. These functions have different characteristics, therefore they form an optimal test set to compare the speedup differences between the serial and parallel implementations. The chosen functions are the Hypersphere 1

$$f(x) = \sum_{i=0}^{N-1} x_i^2 \tag{15}$$

which has a single wide optimum in 3 dimensions, then Rastrigin 2

$$f(x) = 10N \sum_{i=0}^{N-1} (x_i^2 - 10\cos(2\pi x_i)) \tag{16}$$

plenty of minimums 3 dimensions to be evaluated by the algorithm, and last Styblinski Tang 3

$$f(x) = \frac{1}{2} \sum_{i=0}^{N-1} (x_i^4 - 16x_i^2 + x_i) \tag{17}$$

with different convex regions 3 dimensions in the search space. In the benchmark functions N represents the dimension-size of the problem and x the array of coordinates.

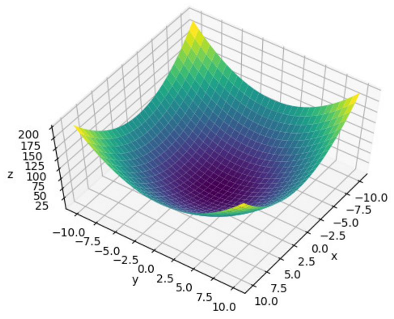

Figure 1: Benchmark 1: Hypersphere

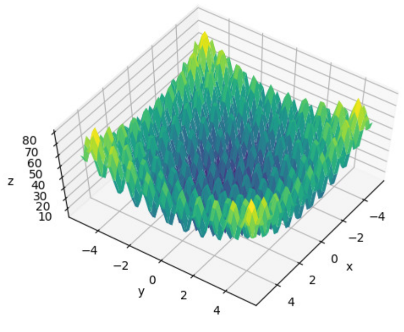

Figure 2: Benchmark 2: Rastrigin

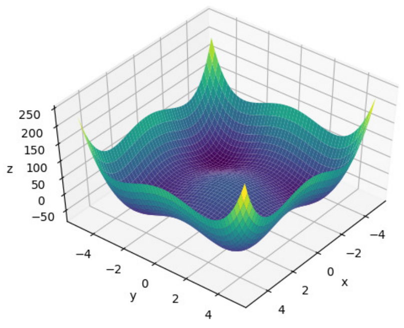

Figure 3: Benchmark 3: Styblinski Tang

4 EXPERIMENTAL EVALUATION

4.1 Hardware

To test the aforementioned algorithms, it was used the *High-Performance Computing* cluster (HPC) from the University of Trento. The main characteristic of the cluster are:

- OS of the nodes is Linux CenOS7;
- nodes: 126;
- CPU cores: 6092;
- CUDA cores: 37.376;
- RAM: 53 TB;
- All nodes are interconnected with 10Gb/s network, some have also Infiniband 40Gb/s connectivity and others have Omnipath 100 Gb/s connectivity.

4.2 Parallel processes and problem dimensions

The algorithms were tested on the benchmark functions defined in section3.6, with different properties in each execution. More in detail, both the serial and parallel implementation were executed with three different dimensions size, $DIM = \{256, 512, 2014\}$, then the parallel algorithm was applied with six different number of cores, $N_{cores} = \{2, 4, 8, 16, 32, 64\}$.

4.3 MPI parallelization results

In the following tables are listed the outcomes of the algorithms in each execution, it is important to note that the performances are merely and uniquely evaluated in terms of time of execution, to understand which are the true benefits of the parallelism. Each table 2, 3, 4 relates the number of parallel processes used with the execution time required per problem dimensions.

N_{cores}	**256**	**512**	**1024**
1	219.6	378.3	881.6
2	127.4	239.7	440.2
4	63.8	132.4	231.9
8	34.1	52.2	100.9
16	18.2	34.0	62.7
32	13.5	22.5	42.7
64	13.8	20.9	36.2

Table 2: Sphere time execution (s)

N_{cores}	**256**	**512**	**1024**
1	154.4	366.8	739.5
2	97.0	171.8	281.6
4	39.6	80.0	153.4
8	24.0	46.3	90.6
16	16.7	41.5	68.9
32	14.8	32.3	39.8
64	11.4	17.7	31.2

Table 3: Rastrigin time execution (s)

N_{cores}	**256**	**512**	**1024**
1	424.0	785.8	1703.9
2	239.8	474.0	709.5
4	108.9	246.7	455.5
8	58.3	94.7	190.7
16	29.6	56.4	109.1
32	19.9	35.7	66.0
64	17.3	29.0	53.3

Table 4: Styblinski Tang time execution (s)

Already by checking the execution time it is notable how the performances increase constantly with more parallel processes working on the task. This demonstrates the potentiality of parallelism in such algorithms. Nevertheless is important to have strong evaluation metrics to understand how actually efficiently the parallelism speeds up.

4.4 Algorithm Evaluations

To better visualize the performances of the algorithms, different evaluations have been applied to the results and then shown in the following graph, to better visualize the tendencies of the execution time and efficiency. In particular, the evaluation metrics are:

- Speedup, $S = \frac{ExecTime_{Serial}}{ExecTime_{Parallel}}$, ideally it should be linear;
- Efficiency, $E = \frac{S}{N_{cores}}$, in case of linear speedup, this should be constant;
- Strong Scalability, an algorithmis strongly scalable if by increasing the problem size, the efficiency remains constant;
- Weak Scalability, an algorithm is weakly scalable if by increasing the problem size and the number of processes by the same rate the efficiency remains constant.

It is important to note, as the number of cores is increased exponentially, as $N_{cores}(x) = 2^x$, the x ticks of the graphs have been rescaled with $x' = log_2(x)$ to better visualize the differences between each parallel execution.

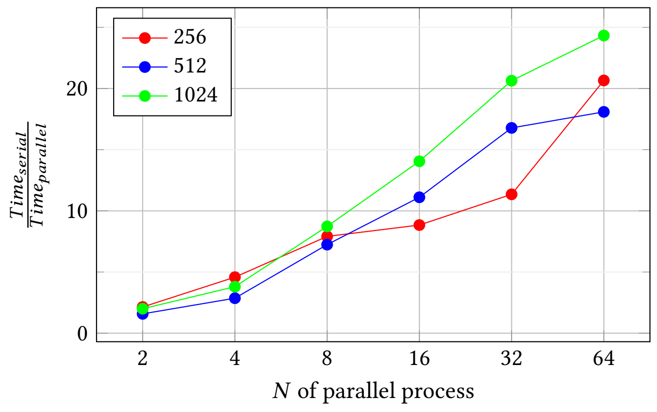

Figure 4: Hypersphere speedup performance of the parallel execution

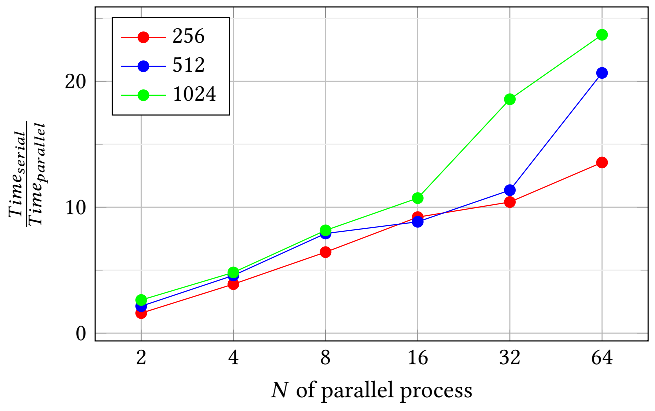

Figure 5: Rastrigin speedup performance of the parallel execution

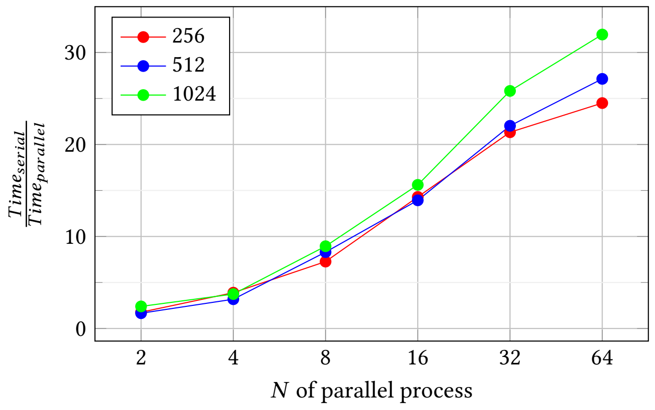

Figure 6: Styblinski Tang speedup performance of the parallel execution

4.4.1 *Speedup.*

The speedup graph 4, 5 and 6 demonstrate how by parallelizing the algorithm there is a significant speedup in execution time. Although, as the graphs are logarithmically rescaled, the actual tendency is not linear but logarithmic, meaning that the overhead time increases and affects the algorithm based on the number of cores. Another important note is that the speedup is similar between each benchmark till $N_{cores} = 8 - 16$, while with a higher number of cores, the three benchmarks diverge. The most probable reason is due to some queueing and allocating issues in the hardware, as the problem is manifesting while requesting a higher amount of resources. Nevertheless, it is a consistent speedup, meaning that even without having complete parallelism within the main loop of the algorithm, it is feasible to retrieve a boost in performance by elevating the number of cores.

4.4.2 *Efficiency.*

The graphs 7, 8 and 9 show the tendencies of the efficiency in the algorithm. If the parallelism is ideal it should be constant over the number of cores. Even though the speedup graph wasn't clear, now it is evident how the overhead and latency affect dramatically the efficiency of the parallelism, as with a lower number of cores the algorithm has relatable efficiencies, while for each execution with $N_{cores} \geq 8$ the efficiency drops, meaning that

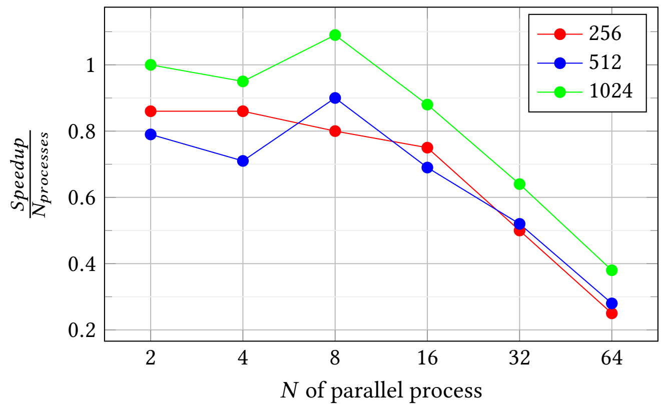

Figure 7: Hypersphere efficiency performance of the parallel execution

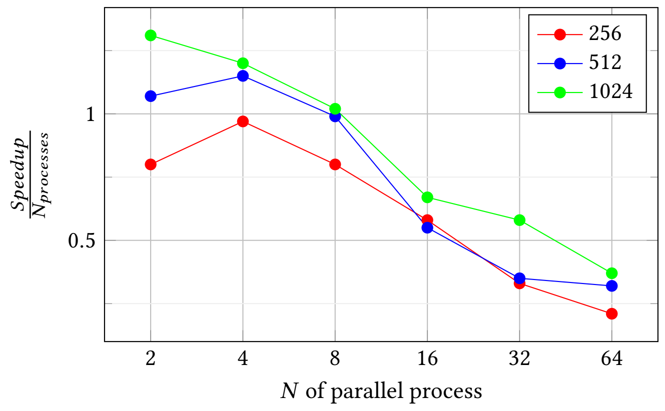

Figure 8: Rastrigin efficiency performance of the parallel execution

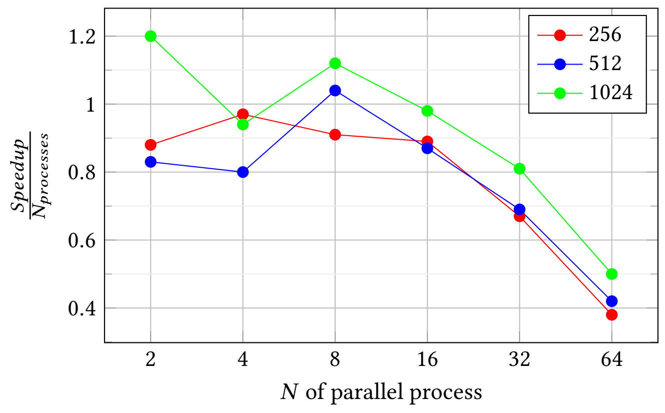

Figure 9: Styblinski Tang efficiency perfofmance of the parallel execution

the amount of resources requested to speedup the algorithm is higher compared to the returned performances.

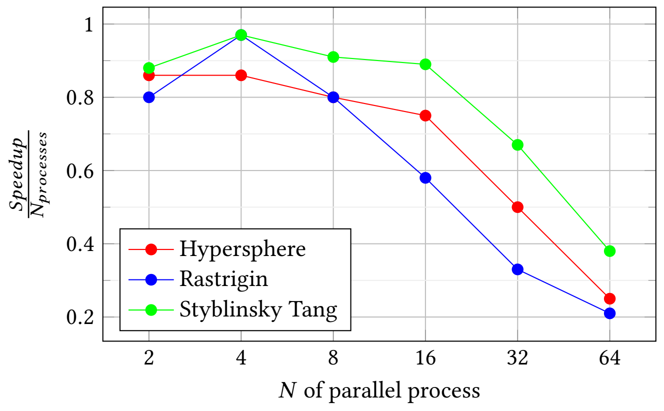

Figure 10: Strong scalability evaluation on dimension 256

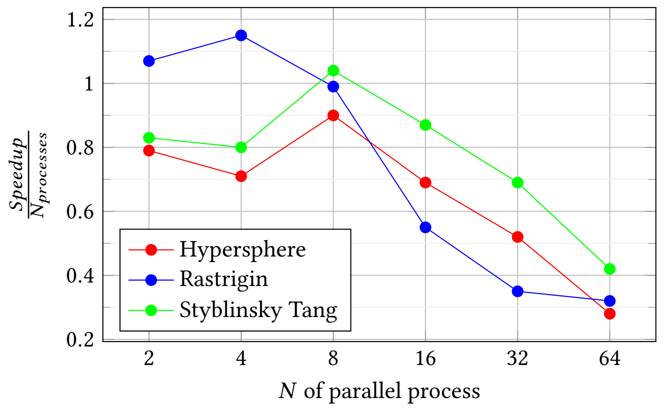

Figure 11: Strong scalability evaluation on dimension 512

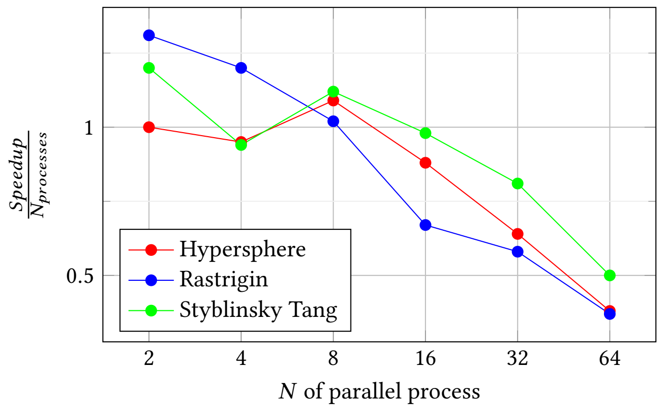

Figure 12: Strong scalability evaluation on dimension 1024

4.4.3 Strong Scalability. Even though the other metrics showed how the algorithm is able to enhance performance based on the number of available cores, even if it loses efficiency, the scalability metrics show the weak points of parallelization. The graphs 10, 11, 12 evidence how the efficiency is not constant with the same problem dimensions, moreover it drops dramatically by adding parallel events. This means that the used approach doesn't provide a strong scalability performance over $N_{cores} \geq 8$.

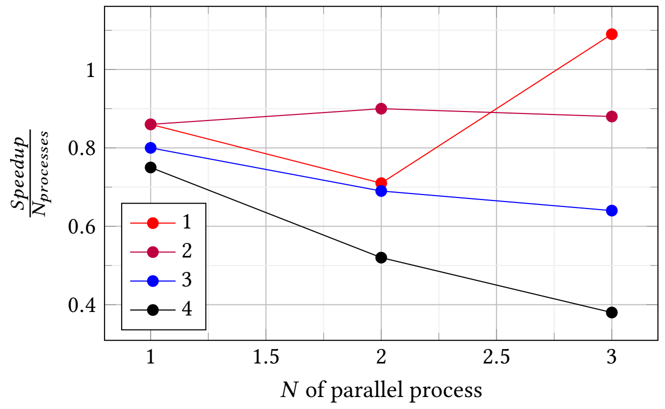

Figure 13: Weak scalability evaluation on Sphere

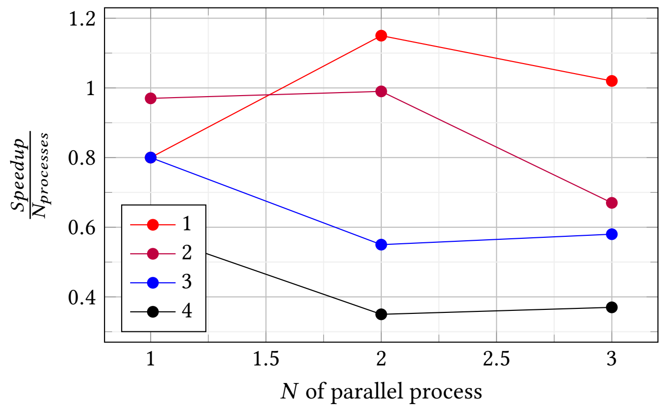

Figure 14: Weak scalability evaluation on Rastrigin

4.4.4 Weak Scalability. To have a more understandable graph, the legends in figures 13, 14 and 15 have been renamed with only indexes. The meaning of the index is summarized in the table 5. The weak evaluations were done considering only the diagonals that were complete, so with three values, to have a better visualization of the graphs. While it was reasoned that the approach is not strongly scalable, it is visible that growing both the N_{cores} and the problem dimensions remain similar. Due to the aforementioned issues, the similarity in efficiency is more evident in the case of the lower number of parallel events, but even with higher values, the efficiency doesn't show a critical drops, meaning that the algorithm performs better while considering the weak scalability.

4.5 Hybrid Parallelization Considerations

Including threads in a parallelization approach can bring benefits as it provides an even deeper parallel execution, but at the same time, it is required to manage in a precise manner the memory that

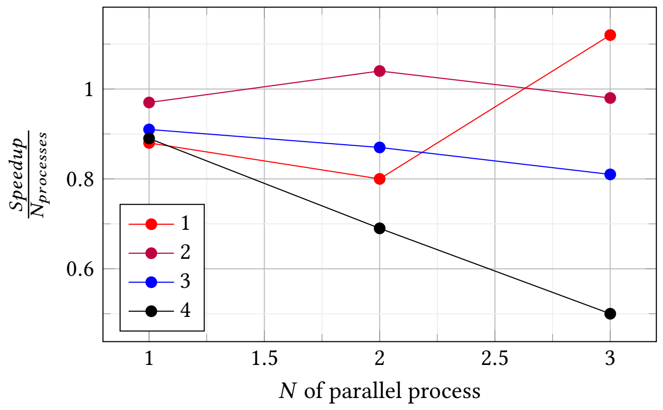

Figure 15: Weak scalability evaluation on Styblinsky Tang

index	N_1, DIM_1	N_2, DIM_2	N_3, DIM_3
1	(2,256)	(4,512)	(8,1024)
2	(4,256)	(8,512)	(16,1024)
3	(8,256)	(16,512)	(32,1024)
4	(16,256)	(32,512)	(64,1024)

Table 5: Index summary

is shared between each thread, to avoid data race as described in 3.1. This issue could bring an extreme amount of overhead during the execution that destroys the performance of the algorithm. In table 6 it is visible how the performance with this approach isn't even close to the only MPI execution. The execution time is referred to the three benchmarks with $DIM = 256$, and yet in one case, the execution of Styblinski Tang with $N_{cores} = 64$ is above ten minutes of walltime, meaning that the communication overhead that is created between the processes is affecting the algorithm grater than the parallelization benefits.

N_{cores}	**Sphere**	**Rastrigin**	**S-T**
1	154.4	219.6	424.0
2	257.0	214.5	460.2
4	255.4	155.7	349.8
8	193.0	155.8	350.7
16	203.3	247.9	350.7
32	213.4	168.6	473.2
64	265.3	217.7	-

Table 6: Execution of Hybird Parallelization over the 3 benchmarks with $DIM = 256$

5 CONCLUSION

In the paper was proposed a solution to enhance the performances of an evolutionary algorithm by applying parallelization methods. In the discussion of the evaluation, it is visible how parallelize give a significant speedup to all running time.
Moreover, it discussed two different approaches, the first using

only *MPI* framework while the second having a hybrid implementation with both *MPI* and *OpenMP*. It was discussed how the first methodology returns relevant speedup to the algorithm, demonstrating also promising weak scalability performances. On the other hand, the overhead encountered with the hybrid parallelization destroys completely the performances. To conclude, it presents an implementation that is able to return a consistent speedup over the serial execution, providing a potential solution for high dimensions problems.

5.1 Future works

In this project, it was developed and discussed a parallel approach to the Artificial Gorilla Troops algorithm. Even if the obtained outcomes overall are considerable and relevant, there are different evaluations and approaches applicable to the project. First, it should be interesting to evaluate the scalability with higher dimensions and number of parallel processes, to have an even deeper evaluation of the work done. Secondly, it was proposed an approach to parallelize a complex algorithm, therefore there are different methods to apply the parallelization. It would be interesting the understand if the whole main loop if parallelized could obtain better or similar performance. Then also applying a completely different approach could be a good starting point for future works.

REFERENCES

[1] Benyamin Abdollahzadeh, Farhad Soleimanian Gharehchopogh, and Seyedali Mirjalili. 2021. Artificial gorilla troops optimizer: A new nature-inspired metaheuristic algorithm for global optimization problems. *International Journal of Intelligent Systems* (2021). https://doi.org/10.1002/int.22535

[2] E. Alba and M. Tomassini. 2002. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 6, 5 (2002), 443–462. https://doi.org/10.1109/TEVC.2002.800880

[3] Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. 2001. *Parallel programming in OpenMP.* Morgan kaufmann.

[4] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA.

[5] Bing Li and Weisun Jiang. 2000. A novel stochastic optimization algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 30, 1 (2000), 193–198. https://doi.org/10.1109/3477.826960

[6] Erik Nielsen and Pietro Ferrari. 2023. *Parallel Artificial Gorilla Troops Optimizer.* https://github.com/NielsenErik/HPC4DS_Project

[7] S. Tamilselvi. 2022. Introduction to Evolutionary Algorithms. *Genetic Algorithms* (2022). https://doi.org/10.5772/intechopen.104198

[8] Nisheeth K. Vishnoi. 2021. Gradient Descent. *Algorithms for Convex Optimization* (2021). https://doi.org/10.1017/9781108699211.008

[9] Mohd Nadhir Ab Wahab, Samia Nefti-Meziani, and Adham Atyabi. 2015. A Comprehensive Review of Swarm Optimization Algorithms. *PLOS ONE* (2015). https://doi.org/10.1371/journal.pone.0122827