Parallel Grey Wolf Optimizer

High Performance Computing for Data Science Project 2024/2025

Mattia Rampazzo
mat.247954
University of Trento
38123 Povo TN, Italy
mattia.rampazzo@studenti.unitn.it

Martin Reinhard
mat.242616
University of Trento
38123 Povo TN, Italy
martin.reinhard@studenti.unitn.it

***Abstract*— Metaheuristic algorithms are widely employed in optimization problems, particularly when the function's gradient is unavailable, the problem is highly complex, or the search space is large. Grey Wolf Optimization (GWO) is a relatively recent population-based metaheuristic inspired by the hunting and leadership hierarchy of grey wolves. It has gained attention due to its strong exploration-exploitation balance, convergence efficiency, and robustness in solving complex optimization problems. In this paper, we present a parallel implementation of GWO using the MPI framework to enhance its scalability and computational efficiency. By distributing the search process across multiple processors, our approach aims to accelerate convergence and improve performance in handling large-scale optimization tasks. We conduct a thorough evaluation of the parallelized GWO, analyzing its speedup, efficiency, and effectiveness in comparison to the sequential implementation. The results demonstrate the potential of parallel GWO in solving high-dimensional and computationally intensive optimization problems, making it a viable approach for large-scale applications.**

I. INTRODUCTION

Optimization problems are widespread across scientific and engineering disciplines, often requiring the discovery of optimal or near-optimal solutions within a vast search space. Traditional optimization techniques, such as gradient-based methods, are highly effective when the objective function is smooth and differentiable. However, in real-world scenarios, many optimization problems involve highly complex, non-differentiable, multimodal, or high-dimensional functions, making traditional approaches impractical. This is where metaheuristic algorithms play a crucial role.

Metaheuristics are high-level problem-independent strategies designed to explore and exploit a solution space efficiently. Unlike deterministic algorithms, metaheuristics incorporate stochastic elements to escape local optima, making them particularly useful for solving complex optimization problems. These techniques can be broadly categorized into:

Single-solution-based methods, including Simulated Annealing (SA) [1], Tabu Search (TS) [2], and Variable Neighborhood Search (VNS) [3], operate on a single candidate solution, iteratively refining it through local search mechanisms. These approaches are particularly effective for fine-tuning and escaping local optima.

Population-based methods, such as Genetic Algorithms (GA) [4], Particle Swarm Optimization (PSO) [5], and Differential Evolution (DE) [6], maintain a diverse set of solutions to balance exploration and exploitation effectively. Among these, nature-inspired algorithms have gained prominence for their ability to mimic biological, ecological, or physical processes to efficiently navigate complex search spaces.

Grey Wolf Optimization (GWO) is one such swarm intelligence algorithm that has demonstrated superior performance across various optimization tasks. GWO is a relatively recent population-based metaheuristic introduced by Mirjalili et al. in 2014 [7]. It is inspired by the leadership hierarchy and hunting behavior of grey wolves in nature. The algorithm models four roles within a wolf pack (alpha, beta, delta, and omega) where the pack collectively searches for the optimal solution through processes such as encircling, hunting, and attacking prey. GWO is particularly known for its strong balance between exploration and exploitation, minimal parameter tuning requirements, and competitive performance across a wide range of applications, including feature selection, engineering design, and machine learning.

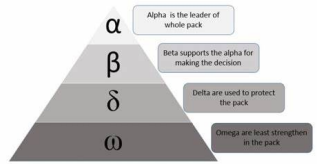

Fig. 1. Wolf Hierarchy

Despite its advantages, GWO can be computationally expensive for high-dimensional and large-scale problems. To address this limitation, this paper explores the potential for parallelizing GWO using the Message Passing Interface (MPI) framework. By distributing the computational workload across multiple processors, we aim to improve its scalability, enhance its convergence speed, and enable its application to large-scale optimization problems.

The remainder of this paper is structured as follows: Section 2 provides an overview of the GWO mathematical framework and algortithm. Section 3 details the proposed parallel GWO approach, including the parallelization strategy and MPI implementation. Section 4 presents experimental results, comparing the performance of sequential and parallel

implementations. Finally, Section 5 concludes the paper with insights into future research directions.

II. RELATED WORK

This section provides an overview of the Grey Wolf Optimization (GWO) algorithm. We explore the core mechanics of the GWO algorithm, detailing the roles of the alpha (α), beta (β), delta (δ), and omega (ω) wolves and how they cooperate to find optimal solutions. In particular, we delve into both its mathematical formulation and serial implementation. We also discuss the computational complexity of GWO and its suitability for solving large scale optimization problems.

The Grey Wolf Optimization algorithm is a population-based metaheuristic inspired by the hierarchical structure and cooperative hunting behavior of grey wolves. It effectively balances exploration and exploitation, making it a powerful approach for solving complex optimization problems. The algorithm relies on a population of N candidate solutions (wolves), each represented as a real-valued vector in a d-dimensional search space. Within the wolf pack, there are four different roles: α, β, δ, and ω, representing the current best solution, the second-best solution, the third-best solution, and other candidate solutions, respectively. The hunting process of grey wolves is simulated through the following three phases: encircling prey, hunting prey, and attacking prey.

a) 1. Encircling Prey: In the encircling phase, wolves move towards their prey using the following equations:

$$D = |C \cdot X_p - X| \tag{1}$$

$$X(t+1) = X_p - A \cdot D \tag{2}$$

where X_p is the position of the prey, X represents the wolf's position, D is the absolute difference between them, and $X(t+1)$ is the updated position of the grey wolf at time $t+1$. The coefficient vectors A and C introduce dynamic weightings that control the wolves' movement. These are defined as:

$$A = 2a \cdot r_1 - a, \quad C = 2 \cdot r_2, \tag{3}$$

where a decreases linearly from 2 to 0 over the course of iterations, ensuring a gradual transition from exploration to exploitation, and $r_1, r_2 \in [0, 1]$ are random numbers that introduce stochasticity.

b) 2. Hunting Prey: The hunting phase is guided by the α, β, and δ wolves. Instead of moving towards a single best solution, each wolf adjusts its position based on the three leaders using the following equations:

$$D_\alpha = |C_1 \cdot X_\alpha - X| \tag{4}$$

$$D_\beta = |C_2 \cdot X_\beta - X| \tag{5}$$

$$D_\delta = |C_3 \cdot X_\delta - X| \tag{6}$$

$$X_1 = X_\alpha - A_1 \cdot D_\alpha, \quad X_2 = X_\beta - A_2 \cdot D_\beta, \quad X_3 = X_\delta - A_3 \cdot D_\delta \tag{7}$$

The final updated position of each wolf is the average of these three adjusted positions:

$$X(t+1) = \frac{X_1 + X_2 + X_3}{3}. \tag{8}$$

This mechanism ensures that wolves move closer to the most promising regions of the search space while maintaining diversity in their movements.

To illustrate the GWO position update process more clearly, we include Figure 2, which visually depicts the movement of a wolf (green) towards the estimated prey position (yellow), influenced by the α, β, and δ wolves. The figure clearly shows how distances D_α, D_β, and D_δ determine the new position of the wolf.

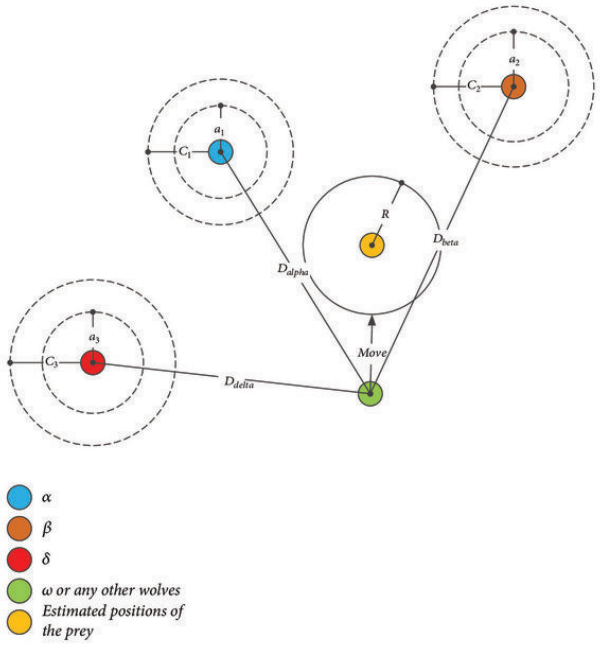

Fig. 2. Position update in Grey Wolf Optimization (GWO). The omega wolf (green) moves based on the influence of the alpha (blue), beta (brown), and delta (red) wolves, adjusting its trajectory toward the estimated prey position (yellow).

c) 3. Attacking Prey: As iterations progress, the algorithm enters the attacking phase, where the wolves close in on the prey. This is controlled by the coefficient vector A, which gradually reduces in magnitude, causing wolves to take smaller steps toward the best solutions. When $|A|$ is small, exploitation is enhanced and the algorithm refines solutions in a local search manner. Conversely, when $|A| > 1$, wolves explore the search space more broadly, preventing premature convergence to local optima.

1) Serial Implementation: The above equations define the core mechanics of the GWO algorithm. In practice, these equations are implemented iteratively to update the candidate

solutions' positions. The following pseudo-code illustrates how the theoretical model is realized in a serial implementation. The process starts with creating a random population of grey wolves (candidate solutions). Over the course of iterations, the α, β, and δ wolves estimate the probable position of the prey. Each candidate solution updates its distance from the prey. The parameter a is decreased from 2 to 0 in order to emphasize exploration and exploitation, respectively. Candidate solutions tend to diverge from the prey when $|A| > 1$ and converge towards the prey when $|A| < 1$. Finally, the GWO algorithm terminates upon meeting a predefined stopping criterion.

Algorithm 1 Grey Wolf Optimizer Algorithm

```
0: Initialize population P with random candidate solutions
0: Evaluate fitness for all candidates in P
0: while stopping criteria not met do
0:    Identify alpha, beta, and delta wolves from P
0:    for each wolf do
0:       Update position using equations (4)-(8)
0:       Evaluate fitness value for the new position
0:    end for
0: end while=0
```

The computational complexity of the serial implementation of GWO is approximately $O(N \times T \times d)$, where N is the population size, T is the number of iterations, and d is the problem dimension. Since every wolf updates its position at each iteration, the execution time scales linearly with the number of function evaluations. While GWO is efficient for moderate-scale problems, its serial implementation becomes computationally expensive for high-dimensional optimization tasks.

Over the years, several extensions have been proposed to enhance GWO's convergence behavior and solution quality. These improvements include adaptive parameter tuning, hybridization with other metaheuristics, and novel strategies for balancing exploration and exploitation. Despite these advancements, the inherent sequential nature of the serial GWO still limits its scalability.

This performance bottleneck motivates the development of a parallelized version of GWO. By distributing the computational workload across multiple processors, a parallel implementation can significantly reduce execution time while maintaining, or even improving, the quality of solutions. This makes parallel GWO particularly well-suited for large-scale optimization problems where computational efficiency is crucial.

III. METHODOLOGY

In this section, we describe our proposed solution for the parallel implementation of the Grey Wolf Optimization (GWO) algorithm. We begin by reviewing original algorithm core componentism of the sequential implementation and then explain how we extend this approach into a parallel framework using MPI. The parallelization is developed in two stages: an initial master–slave model and a later evolution toward a fully distributed system using an all-gather communication strategy.

A. Parallel design analysis

The execution of the GWO can be broadly divided into four key stages: population initialization, fitness evaluation, leader(alpha/beta/delta) selection, and position updating. Each of these stages presents significant opportunities for parallelization, which can dramatically improve computational efficiency without altering the fundamental logic of the algorithm.

Population initialization involves randomly generating a set of candidate solutions (wolves) within the search space. Since each wolf's starting position is independent, this generation process is inherently parallel. Assigning each wolf to a separate process enables concurrent generation of the entire population, dramatically reducing initialization time. Fitness evaluation, a crucial step assessing each wolf's quality based on a problem-specific objective function, is similarly parallelizable. Distributing these evaluations across processing units reduces computation time, particularly for computationally expensive objective functions.

Once fitness values are computed, the algorithm selects the top three wolves, designated as alpha, beta, and delta, who will guide the rest of the population. This selection process is fundamentally a reduction operation, requiring the identification of the best fitness values. While some level of synchronization is necessary to ensure the correct identification of these top candidates, efficient parallel reduction techniques can speed up this step, minimizing bottlenecks.

The final and most computationally intensive phase of GWO is position updating. Here, each wolf adjusts its position based on the locations of the alpha, beta, and delta wolves using the above mentioned update equations. Since each wolf's movement is independent of others, this stage could be fully parallelizable, with every wolf's position update being computed simultaneously across multiple processes.

B. Parallel implementation

As we have seen the core component of the gwo algorithm are inherently parallelizable, opening up several possible parallelization strategies. In our MPI-based parallel solution, the overall population is partitioned into local subpopulations, with each MPI process managing its own subset of candidate solutions. This distributes the workload efficiently, ensuring all processes compute simultaneously. Each process independently generates and evaluates its local solutions. Crucially, the GWO's position update phase is executed concurrently across all processes. This parallelization significantly reduces execution time compared to a serial implementation.

However, the challenge in parallelizing a population-based algorithm lies not only in distributing the workload but also in ensuring that the best candidate solutions (alpha, beta, and delta) are consistently shared among all processes. To address this, we explored two distinct communication strategies.

C. Master–slave model

Our initial implementation employed a straightforward master-slave architecture. The n wolves are divided among the MPI processes, each responsible for initializing, evaluating fitness, identifying local best candidates, and updating positions within its assigned subset.

Synchronization of global best candidate positions is handled by a central process, in our case process 0, acting as the communicator. This master process gathers the local best candidates from each process in the pool, determines the global best among this restricted set and then broadcasts the global alpha, beta, and delta values back to each process for position updates. This workflow is better illustrated in the following Figure 3.

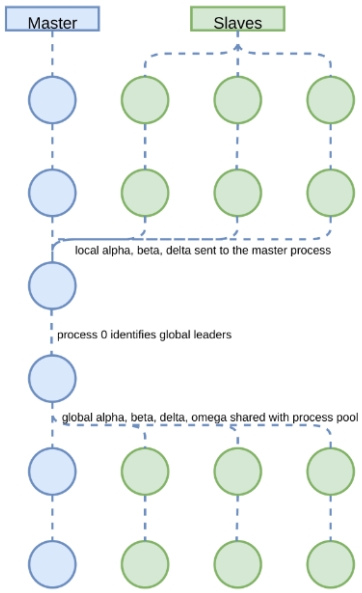

Fig. 3. Workflow of one iteration of the loop of the parallel GWO algorithm in the master-slave model focusing on the sharing of the alpha, beta, delta information across processes.

For efficient communication, we utilized collective communication operations. Specifically, *MPI_Gather* on process 0 collects all local best candidates (*local_best*) into a global collection (*global_best*). After process 0 identifies the global leaders, their positions are distributed to all processes using three *MPI_Broadcast* calls.

Despite its relative simplicity the initial master-slave approach presented robust results. However, we identified opportunities for performance gains, specifically by reducing communication overhead. Our first optimization targeted the data transferred. Initially, a struct containing both position and fitness was used. We explored using *MPI_Datatype* for increased efficiency, but ultimately, the simplest representation proved most effective: storing candidate positions and fitnesses in two separate arrays.

After considering various optimizations within the master-slave framework, we focused on alternative communication strategies to improve efficiency while preserving the core algorithmic structure. Since each process requires access to the global alpha, beta, and delta values for position updates, minimizing communication overhead was a key objective. To address the limitations of a centralized approach, the implementation shifted toward a more decentralized strategy, distributing the computation more evenly among processes. While exploring these alternatives, the island model was taken in consideration but ultimately discarded, as its evolutionary mechanics—where populations evolve independently with only occasional information exchange—diverge from the core principles of GWO. This investigation led to several implementation variations, ultimately resulting in the most efficient approach detailed in the following section.

D. Fully decentralized model

To address the limitations of our initial approach, we explored alternative communication patterns aimed at minimizing the number of communication calls and thus reducing overhead. This exploration led us to abandon the master-slave paradigm in favor of a fully distributed approach. This eliminated the performance bottleneck associated with a central coordinating process, where all other processes were forced to halt execution while waiting for process 0 to complete its calculations and distribute the results.

In this fully distributed scenario, each process possesses complete knowledge of the local best candidates from all other processes in the pool, enabling independent determination of the global best without further communication.

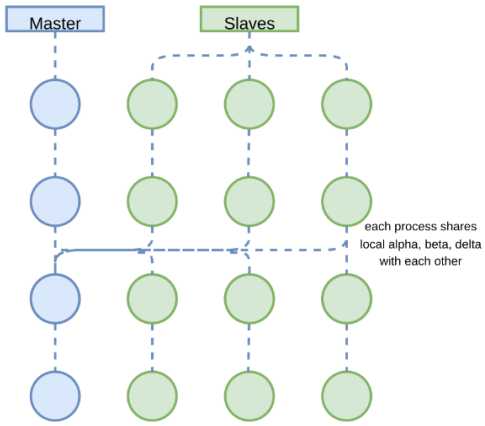

Fig. 4. Workflow of one iteration of the parallel GWO algorithm in the fully decentralized model focusing on the sharing of the alpha, beta, delta information across processes.

We implemented this concept using a single collective MPI call: *MPI_Allgather*. This call gathers data from all processes in the communicator and distributes it to all processes. Specifically, each process provides its local best candidates and *MPI_Allgather* collects all these local bests and distributes the complete set to every process. This allows each process to independently determine the global best.

Having identified the appropriate communication call and pattern, we then focused on optimizing the data structure for sending and receiving. Ultimately we used a send buffer (*local_candidates_buffer*) consisting of the fitnesses and the

corresponding positions of each process's three best candidates. The receive buffer (*global_candidates_buffer*) instead, is sized to accommodate all the local best candidates from every process in the pool. It will be used by each process to find the three best global fitness and the associated position used to update the wolves.

While this approach involves transferring a significant amount of data, we found out that, scaling with the problem size the trade-off of using a single collective communication operation resulted in a substantial performance improvement, as demonstrated by the results presented in the following section.

E. Hybrid Decomposition: Population and Dimension Distribution

Alongside partitioning the wolf population into subpopulations, we investigated an alternative data division strategy. Instead of distributing the population across processes, we explored decomposing the initial search space of DIM dimensions into the n MPI processes, with each process managing a population of n wolves within its assigned subspace. A strategy was devised to compute the fitness of each solution in the subspace by combining a candidate solution with the best wolves obtained by other processes in the remaining subspaces. However, in practice, testing with benchmark problems revealed that this decomposition did not produce significant improvements.

Partially inspired by the work of [8], we proposed a further solution that distributes both the population and the problem dimensions across processes. By concurrently reducing the number of wolves per process (i.e., setting the local population size to n/n_processes), we observed superlinear speedup.

It's important to recognize that while this hybrid decomposition approach delivers impressive performance gains, directly comparing it to population-only strategies is misleading. The latter significantly reduces the computational load per wolf, making direct comparisons inherently unfair. As a result, the observed speedups in the hybrid approach stem not only from improved parallelism but also from the reduced computational effort per wolf. To ensure an accurate evaluation, performance should be measured against a carefully selected baseline that accounts for these differences. For this reason, we chose not to conduct a detailed performance analysis of this implementation but included it for completeness and to highlight potential directions for future optimization strategies.

F. Data Dependencies

When parallelizing an algorithm, managing data dependencies becomes essential to ensure that the parallel processes function cohesively. The key areas impacted by these dependencies in the GWO case include the position updates based on the leaders (alpha, beta, and delta wolves), the fitness evaluation and leader selection, as well as the population-wide interactions required for accurate optimization. These steps rely on shared data across processes, and any inconsistency or race condition can severely affect the algorithm's performance. Proper synchronization and communication strategies are therefore necessary to ensure that all processes are working with up-to-date information, maintaining both the efficiency and correctness of the optimization.

Focusing on the leader selection part of the serial code.

```
for (i = 0; i < N_WOLVES; i++) {
    double fit = fitness[i];
    if (fit < alpha_fitness) {
        delta_fitness = beta_fitness;
        beta_fitness = alpha_fitness;
        alpha_fitness = fit;

        delta_index = beta_index;
        beta_index = alpha_index;
        alpha_index = i;

    } else if (fit < beta_fitness) {
        delta_fitness = beta_fitness;
        beta_fitness = fit;

        delta_index = beta_index;
        beta_index = i;
    } else if (fit < delta_fitness) {
        delta_fitness = fit;

        delta_index = i;
    }
}
```

The primary flow dependencies occur due to the sequential updates of `alpha_fitness`, `beta_fitness`, and `delta_fitness`, where each variable depends on the previous iteration's values. This cascading structure enforces strict execution order, preventing independent iteration execution. Additionally, anti-dependencies arise as `alpha_fitness`, `beta_fitness`, and `delta_fitness` are read before being updated, further restricting reordering possibilities. Output dependencies exist since these variables are overwritten in every iteration, causing repeated memory updates. Similar dependencies are observed for `alpha_index`, `beta_index`, and `delta_index`, as their values are directly linked to the fitness updates. These dependencies create a bottleneck, limiting the potential for parallel execution and requiring careful optimization strategies to improve efficiency. Table I lists some of the Data dependencies discussed.

Memory Location	Early Statement			Later Statement			Loop Carried?	Kind of Dataflow
	Line	Iteration	Access	Line	Iteration	Access		
alpha_fitness	3	i	Read	6	i	Write	no	Anti
delta_fitness	4	i	Write	4	i+1	Write	yes	Output
beta_fitness	4	i	Read	5	i	Write	no	Anti
alpha_fitness	6	i	Write	6	i+1	Write	yes	Output
beta_index	8	i	Read	9	i	Write	no	Anti
alpha_index	9	i	Read	10	i	Write	no	Anti

TABLE I
DATA DEPENDENCIES

Another important key aspect of GWO is on the population initialization process. In the sequential version, the initial positions of the wolves are generated using a standard random number generator, ensuring a diverse starting population. However, in the parallel implementation using MPI, special care must be taken to ensure that each process generates

a unique portion of the population without redundancy or correlation. To achieve this, the random number generator on each process is initialized using `srand(time(NULL) + rank)` where `rank` represents the unique identifier of the process. This approach ensures that different processes initialize their population with distinct random values, preventing overlap and maintaining diversity across the search space. Proper management of randomness in parallel execution is essential to preserving the exploratory capabilities of the algorithm while leveraging parallel computing for efficiency. In a OpenMP scenario, `rand_r` is the preferred choice for thread-safe generation of numbers and is crucial to avoid race conditions.

IV. EXPERIMENTAL RESULTS

This section presents the experimental results obtained from the implementation of our best performing parallel algorithm (i.e. fully decentralised model). The main objective is to evaluate the performance and efficacy of the proposed algorithm. By examining various metrics and comparing the results with those obtained with a serial version of GWO, this part aims to provide insights into the algorithm's strengths, weaknesses and overall effectiveness.

A. Hardware

To evaluate the performance of our algorithm, we leveraged the computing resources provided by the High-Performance Computing (HPC) cluster at the University of Trento, which operates on the Altaris PBS Professional cluster management software. The primary specifications of the cluster are outlined below:

- Operating System of the nodes: Linux CentOS7
- Number of nodes: 126
- CPU cores: 6092
- CUDA cores: 37,376
- RAM: 53 TB
- Connectivity: The nodes are interconnected via a 10 Gb/s network. Additionally, select nodes are equipped with high-speed connectivity options, including Infiniband at 40 Gb/s and Omnipath at 100 Gb/s.

B. Benchmark problems

In evaluating the parallelized and serial GWO algorithm, a set of benchmark functions is typically employed to assess its performance across different optimization landscapes. These benchmarks generally include both unimodal and multimodal functions to analyze the algorithm's convergence behavior and ability to escape local optima. A commonly used unimodal function is the Sphere function, defined as

$$f(x) = \sum_{i=1}^{n} x_i^2,$$

which has a single global minimum at the origin and is often utilized to measure exploitation capability. Since the Sphere function lacks local minima, it provides insight into the algorithm's convergence speed and precision. On the other hand, multimodal functions, such as the Rastrigin function, introduce numerous local optima, making them a more challenging test for an optimization algorithm. The Rastrigin function is defined as

$$f(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right],$$

where its high frequency of local minima requires a balance of exploration and exploitation to locate the global optimum effectively. The focus of this paper is on the efficiency of the algorithm when parallelized, particularly in terms of speedup and computational gain. As previously mentioned, the effectiveness of the parallelized GWO is expected to depend on the dimension of the problem, with higher dimensionalities presenting more significant challenges for parallelization in terms of both computational load and performance.

C. Experimental Setup and Parameters

Careful attention was devoted to hyperparameter tuning to ensure the Grey Wolf Optimizer converged to acceptable solutions. First of all the problem dimensionality was set to 10,000: this ensured the benchmark to be a large-scale optimization problem solvable by our serial implementation within a reasonable timeframe. We used 1,000 iterations, which were generally sufficient to reach the global minimum in most cases and the lower and upper bound of the search space were set repectively to -10 and 10.

To assess the parallel implementation's performance with larger populations, we scaled the number of wolves in the GWO. This scaling demonstrates the parallel approach's ability to handle more complex problems, as these problems often require larger populations to enhance the diversity of candidate solutions.

Moreover, since the GWO like all metaheuristics, is intrinsically stochastic, we ran the algorithm 10 times per experiment to ensure the robustness and consistency of the solutions. This approach helped account for the variability of stochastic algorithms, providing a more reliable measure of performance across multiple runs. Additionally, a warm-up run was executed before each benchmark to mitigate any potential initialization effects. This initial run allowed the algorithm to stabilize, ensuring that the timing and performance metrics were not influenced by any transient factors such as startup overhead, allowing for more accurate and consistent results in subsequent benchmark evaluations.

Finally, for optimal performance in all tests, a single node with n cores was exclusively allocated. A bash file presenting all the settins and parameters discussed is provided below.

```
#!/bin/bash
#PBS -l select=1:ncpus=4:mem=2gb -l place=pack:excl
#PBS -l walltime=0:30:00
#PBS -q short_cpuQ

module load mpich-3.2
mpirun.actual -n 4 ./gwo/gwo_parallel_distributed
    10000 1024 1000 -10 10
```

D. Results

Table II presents the results of both the parallel and serial algorithms for each execution conducted during the testing phase. As clearly shown, the table correlates the number of parallel processes used during the tests with the average execution time required to perform clustering on datasets of varying sizes. Therefore, our performance analysis focuses exclusively on execution time, as it allows us to accurately assess the benefits and drawbacks of adding parallelization to the GWO.

	128	**256**	**512**	**1024**
1	135.270	275.454	543.903	982.596
2	60.442	120.818	295.779	485.229
4	31.023	60.675	121.619	242.176
8	16.924	32.823	62.705	167.604
16	12.476	19.768	34.896	68.939
32	12.977	15.170	24.602	40.004

TABLE II
AVERAGE EXECUTION TIMES (IN SECONDS) OF PARALLEL GWO ALGORITHM WITH VARYING POPULATION SIZES (WOLVES) AND PARALLEL PROCESS COUNTS (PROCESSES)

The experimental results reveal significant trends in the performance of the parallel GWO algorithm as the population sizes and parallel process counts vary. In particular it can be immediately noticed that, as the number of processes increases, execution times decrease across all population sizes. For instance, with 1024 wolves, scaling from 1 to 32 processes reduces the runtime from 982.6 seconds to 40.0 seconds, showing strong scalability for larger population sizes. This provides a first evidence of the good use of parallelization of our algorithm.

However, the benefits of parallelization become less pronounced as we tested with higher number of processes, especially for smaller populations. For example, with 128 wolves, increasing the number of processes from 16 to 32 results in only a slight reduction in runtime, suggesting that communication and synchronization overheads may start to outweigh computational gains. This highlights the importance of balancing parallel resources with problem size to avoid inefficiencies in smaller-scale problems.

To better evaluate the impact of our parallelization approach, we rely on metrics like speedup and efficiency to analyze its scalability:

a) Speedup: Speedup is defined as the ratio between the execution time of the serial program and that of the parallel program. Ideally, when introducing parallelization into a sequential algorithm, we would like to observe near-linear speedup, meaning that doubling the number of processes should roughly halve the execution time.

From the speedup graph in figure 5, we can see that, overall GWO demonstrates good scalability. As we increase the number of parallel processes, the execution time decreases proportionally, leading to a near-linear increase in speedup. This is indicated by the general trend of the four curves, each representing different problem sizes (128, 256, 512, and 1024 wolves). However, we can observe some deviations from perfect linear scaling, particularly when increasing the number of processes from 8 to 16 and then to 32. This is due to the fact that as parallelization increases, so does the communication overhead among processes. When the workload per process becomes too small, the overhead of synchronizing data and sharing key information outweighs the benefits of further parallelization. This effect is particularly noticeable in the blue curve, which represents the smallest problem size (128 wolves). For this case, when using 32 processes, each process is handling only 4 wolves. At this point, the computational cost per process is minimal, and the benefit of distributing the workload is negated by the increased communication cost. Additionally, since GWO requires sharing the best three solutions (alpha, beta, and delta wolves) among processes, having only four wolves per process leaves little room for meaningful independent computation.

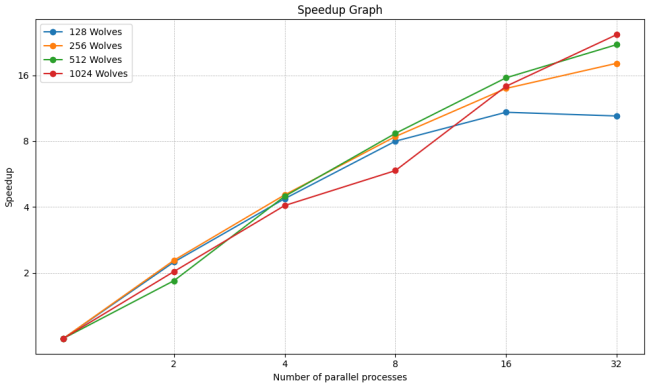

Fig. 5. Speedup performance for different numbers of wolves (128, 256, 512, and 1024) as the number of parallel processes increases. The graph shows near-linear scaling up to 16 processes, with diminishing returns and divergence beyond that point.

A particularly interesting case is the red curve (1024 wolves), which shows an unusual drop in speedup at 8 processes before continuing its upward trend. This anomaly is likely due to external factors such as fluctuations in the computational environment during the experiment. For instance, the cluster may have been under heavy load during that specific run, causing inconsistencies in execution time.S

b) Efficiency: Efficiency is defined as the speedup divided by the number of parallel processes. It measures how effectively the cores are utilized by the parallel algorithm. High efficiency indicates that the algorithm makes good use of the available resources and, ideally, should remain constant.

From the efficiency graph in Figure 6, we observe that for small numbers of parallel processes, efficiency remains close to or slightly above 1, indicating that the GWO scales well under moderate parallelization. This is particularly evident for 256, 512, and 1024 wolves, where efficiency remains high up to 8 processes.

However, as the number of processes increases beyond this point, a decline in efficiency is noticeable, especially for

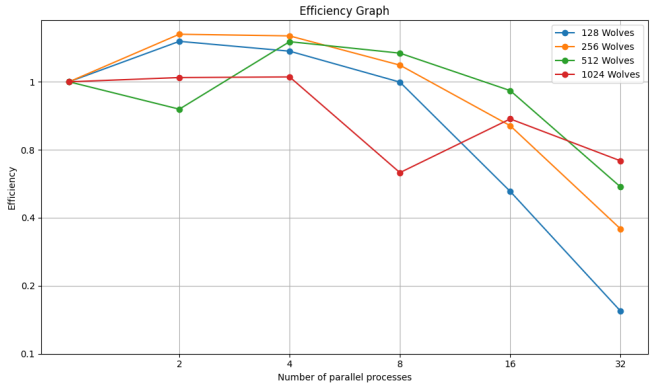

Fig. 6. Efficiency graph for different numbers of wolves (128, 256, 512, and 1024) as parallel processes increase. Efficiency remains high for small numbers of processes but declines as more processes are added, especially for smaller problem sizes. Larger problem sizes (512 and 1024 Wolves) maintain better efficiency at higher process counts, though all cases show a drop-off due to parallel overhead and diminishing returns.

smaller problem sizes. The blue curve (128 wolves) exhibits the steepest drop, particularly when moving from 16 to 32 processes. This behavior aligns with our observations from the speedup graph, when too few wolves are assigned per process, the benefits of parallelization are outweighed by communication overhead. In extreme cases, such as 128 wolves with 32 processes, efficiency falls even below 0.2, indicating scarce use of additional cores.

For larger problem sizes (512 and 1024 wolves), the decrease in efficiency is less pronounced, demonstrating that the GWO algorithm scales better when there is a sufficient workload per process. The green (512 wolves) and red (1024 wolves) curves show a more gradual decline, suggesting that for larger populations, parallelization remains beneficial even at higher process counts.

c) Scalability: To conclude the analysis, we investigated the scalability of the GWO. There are two key definitions of scalability: Strong scalability refers to an algorithm's ability to maintain efficiency when increasing the number of parallel processes while keeping the total problem size fixed. A strongly scalable algorithm will exhibit near-linear speedup, meaning that adding more computational resources effectively reduces execution time without excessive overhead. Weak scalability describes how well an algorithm maintains efficiency when both the problem size and the number of processes increase proportionally. A weakly scalable algorithm ensures that as the workload grows, parallel resources are utilized effectively, keeping execution time relatively stable. Our results indicate that GWO is not strongly scalable, particularly for small problem sizes. As we increase the number of processes while keeping the number of wolves constant, efficiency declines due to increased communication overhead and reduced workload per process. This effect is especially pronounced for smaller cases, where having too few wolves per process negates the benefits of parallel execution. The algorithm's reliance on global information sharing (top three wolves) exacerbates this issue, making synchronization costs more significant at higher process counts. However, GWO demonstrates good weak scalability, particularly for larger problem sizes. When both the number of wolves and processes increase proportionally, efficiency remains relatively stable, meaning that as long as the workload per process is sufficiently large, parallelization remains effective. While efficiency does decrease slightly at high process counts, this is expected due to increasing communication costs, but it does not negate the overall scalability trend. In conclusion, GWO benefits from parallelization but is best suited for large-scale problems, where a sufficient computational workload per process helps mitigate the impact of communication overhead.

V. CONCLUSION

In conclusion, this paper has explored the parallelization of the Grey Wolf Optimization (GWO) algorithm using the Message Passing Interface (MPI) framework, with a primary focus on improving efficiency in terms of scalability. The results demonstrate that the parallelization strategy is effective, significantly reducing computational time while maintaining or even enhancing the optimization performance. The experiments conducted confirm that MPI-based parallelism is well-suited for GWO, particularly in handling large-scale optimization problems where sequential execution becomes computationally expensive. The speedup and scalability analysis further highlight the advantages of this approach, proving that distributing the workload among multiple processing units leads to notable performance gains. Overall, the findings of this study reinforce the potential of parallelized GWO as a powerful optimization tool for complex real-world applications, making it a viable choice for tackling high-dimensional and computationally demanding problems.

VI. FUTURE WORKS

Future work could explore additional strategies to further enhance the efficiency of parallelized Grey Wolf Optimization. One promising direction is hybrid decomposition, where parallelization is applied not only by distributing the population across processing units but also by partitioning the search space dimensions, potentially leading to improved load balancing and convergence speed. Moreover, alternative parallel implementations, such as a hybrid OpenMP-MPI approach or GPU acceleration, could be investigated to leverage shared-memory and massively parallel architectures, respectively. While this paper has primarily focused on efficiency in terms of computational time using MPI, these alternative strategies could offer further optimizations and broaden the applicability of parallelized GWO to even more complex and large-scale optimization problems.

REFERENCES

[1] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671-680, 1983.
[2] F. Glover, "Tabu Search—Part I," *ORSA Journal on Computing*, vol. 1, no. 3, pp. 190-206, 1989.
[3] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097-1100, 1997.

[4] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.
[5] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942-1948, 1995.
[6] R. Storn and K. Price, "Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341-359, 1997.
[7] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in Engineering Software*, vol. 69, pp. 46-61, 2014.
[8] J. Smith and A. Jones, "A Novel Parallel Grey Wolf Optimizer for UAV Path Planning," *Journal of Unmanned Systems*, 2024.