Parallel Grey Wolf Optimization
High Performance Computing for Data Science Project 2023/2024

Munyaradzi Comfort Zhou
University of Trento
Trento, Italy
munyaradzi.zhou@studenti.unitn.it

Ebuka Nwafornso
University of Trento
Trento, Italy
ebuka.nwafornso@studenti.unitn.it

Yuhang Jiang
University of Trento
Trento, Italy
yuhang.jiang@studenti.unitn.it

ABSTRACT

The Grey Wolf Optimizer (GWO) is a widely used metaheuristic algorithm inspired by the hunting behavior of grey wolves. However, the original GWO heavily relies on the top three wolves (alpha, beta, and delta) and lacks additional information fusion, resulting in unsatisfactory convergence speed in many cases. This work addresses these challenges by introducing the History-Guided Trend-adjusted Grey Wolf Optimization (HGT-GWO), which combines global historical best positions and individual trend guidance to improve exploration and exploitation. Additionally, a novel master-worker island parallelization scheme is proposed, where worker processes operate semi-independently, significantly reducing sorting and communication overhead compared to standard parallelization methods. The new framework was implemented using MPI and MPI+OpenMP approach, validated on three benchmark functions. HGT-GWO demonstrated significant improvements in convergence speed, resource efficiency, and scalability, outperforming the original GWO.

ACM Reference Format:
Munyaradzi Comfort Zhou, Ebuka Nwafornso, and Yuhang Jiang. 2025. Parallel Grey Wolf Optimization High Performance Computing for Data Science Project 2023/2024. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Population-based metaheuristic optimization techniques, such as Swarm Intelligence (SI) algorithms, have gained significant popularity due to their simplicity, flexibility, and ability to avoid local optima [4]. Inspired by natural phenomena such as animal behaviour, physical processes, or evolutionary strategies, SI algorithms leverage the collective behaviour of agents, as seen in flocks, schools, and colonies, to explore and exploit the solution space effectively. This results in better performance in avoiding local optima and achieving greater exploration [7]. Examples include Genetic Algorithms (GA), Particle Swarm Optimization (PSO) [3], Ant Colony Optimization (ACO) [2], and Artificial Bee Colony (ABC) [6]. A key advantage of these techniques is their general applicability across diverse domains, as they operate without derivation and treat problems as black boxes [7], requiring no structural modifications.

Optimization algorithms are indispensable for solving complex problems aiming to find optimal solutions within feasible time frames. These algorithms are broadly classified into deterministic and stochastic methods. Deterministic algorithms guarantee exact results, but are computationally expensive, often impractical for high-dimensional problems. In contrast, stochastic approaches employ randomness to explore and exploit solution spaces, making them effective for real-world problems where derivative information is unavailable or expensive to compute.

Figure 1: Grey wolf pack hunting. [11]

Grey Wolf Optimization (GWO) is a recent addition to SI algorithms, inspired by the hunting behavior and social hierarchy of grey wolves. The GWO algorithm mimics the leadership structure of wolf packs, where alpha, beta, and delta wolves guide the search process while omega wolves explore the solution space. This structure ensures a balanced trade-off between exploration and exploitation, making GWO efficient in solving complex and high-dimensional optimization problems. This work explores the parallelization of GWO using a hybrid framework combining MPI [9] and OpenMP [1].

2 RELATED WORK

This work takes inspiration from the original GWO paper [7] , which implements the algorithm through a serial workflow. Firstly, we analyze the serial framework and suggest our improved serial workflow of GWO, History-Guided Trend-adjusted Grey Wolf Optimization (HGT-GWO). Lastly, we propose a hybrid parallelization approach that implements MPI and OpenMP .

2.1 Implementation

The GWO algorithm is inspired by the social hierarchy of grey wolves and their hunting technique. The social hierarchy comprises the alpha wolf,beta and delta wolves, representing the fittest solutions, respectively. The last in the pecking order in the pack is the omega wolf, and the first three wolves guide the optimization, the aplha representing the fittest solution.

According to [8], the mathematical representation of the Grey Wolf Optimizer (GWO) is modeled based on the main phases of grey wolf hunting, namely:

(1) Searching for prey (exploration)
(2) Encircling prey
(3) Hunting prey
(4) Attacking prey (exploitation)

Searching for Prey (Exploration)

In the GWO algorithm, the process of searching for prey is led by the top three wolves in the hierarchy: alpha, beta, and delta. These wolves guide the exploration of the search space to locate the prey. Two coefficient vectors, $\vec{A}$ and $\vec{C}$, are critical in this phase. The vector $\vec{A}$ ensures divergence by having values greater than 1 or less than -1, which encourages the wolves to explore away from the prey. On the other hand, $\vec{C}$, with random values in the range $[0, 2]$, introduces randomness into the optimization process. This randomness helps the GWO algorithm to avoid premature convergence and escape local optima, ensuring robust exploration of the search space. When $|\vec{A}| > 1$, exploration occurs.

Encircling Prey

After searching for prey the next phase is the gey wolves encircling their prey ready to attack. Wolves update their positions relative to the prey using the mathematical equations 1 and 2. Grey wolves encircle prey by updating their positions based on their distance $\vec{D}$ from the prey.

$$\vec{D} = |\vec{C} \cdot \vec{X}_p(t) - \vec{X}(t)| \tag{1}$$

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

where t indicates the current iteration, $\vec{A}$ and $\vec{C}$ are coefficient vectors, $\vec{X}_p$ is the position vector of the prey and $\vec{X}$ indicates the position vector of a grey wolf.
$\vec{A}$ and $\vec{C}$ are control parameters that adjust exploration and exploitation. The vector $\vec{a}$ decreases linearly from 2 to 0 over iterations, while $\vec{r}_1$ and $\vec{r}_2$ are random vectors in the range $[0, 1]$.

$$\vec{A} = 2 \cdot \vec{a} \cdot \vec{r}_1 - \vec{a} \tag{3}$$

$$\vec{C} = 2 \cdot \vec{r}_2 \tag{4}$$

Hunting Prey

The hunting process is driven by the collective intelligence of the pack, specifically the three best wolves: alpha, beta, and delta. These wolves represent the most promising solutions found so far. The rest of the wolves adjust their positions based on the weighted influence of these leaders, ensuring that the pack converges toward the prey (optimal solution). This step balances exploration and exploitation, as it incorporates guidance from the leaders while still allowing flexibility in movement.

The positions of the alpha ($\vec{X}_\alpha$), beta ($\vec{X}_\beta$), and delta ($\vec{X}_\delta$) wolves are used to guide the search process. The final position is the average of these three, which balances exploration and exploitation.

$$\vec{D}_\alpha = |\vec{C}_1 \cdot \vec{X}_\alpha - \vec{X}|,\ \vec{D}_\beta = |\vec{C}_2 \cdot \vec{X}_\beta - \vec{X}|,\ \vec{D}_\delta = |\vec{C}_3 \cdot \vec{X}_\delta - \vec{X}| \tag{5}$$

$$\vec{X}_1 = \vec{X}_\alpha - \vec{A}_1 \cdot (\vec{D}_\alpha),\ \vec{X}_2 = \vec{X}_\beta - \vec{A}_2 \cdot (\vec{D}_\beta),\ \vec{X}_3 = \vec{X}_\delta - \vec{A}_3 \cdot (\vec{D}_\delta) \tag{6}$$

$$\vec{X}(t+1) = \frac{\vec{X}_1 + \vec{X}_2 + \vec{X}_3}{3} \tag{7}$$

Attacking Prey (Exploitation)

As the wolves close in on the prey, the algorithm shifts its focus from exploration to exploitation. This is achieved by reducing the parameter $\vec{a}$, which decreases the influence of stochastic components in the position updates. The wolves perform a more localized search around the prey's estimated position to fine-tune the solution. This phase ensures convergence to the optimal solution by refining the search within the promising region. When $|\vec{A}| < 1$, exploitation occurs.

3 IMPLEMENTATION AND METHODOLOGY

The implementation of the serial GWO.

It initializes a population of search agents (wolves) and uses three key agents (X_α, X_β, and X_δ) to guide the rest. During each iteration, wolves update their positions based on the relative distances to these leaders and predefined parameters. This approach is simple and effective for optimization problems, relying entirely on the collective movement and exploration/exploitation balance of the wolves. An implementation is provided in our github repository [5].

We implemented the following step-by-step procedural framework in our approach to the problem, as illustrated in the figure below. First, we began with the original GWO serial implementation and enhanced it to the Serial HGT-GWO. This was followed by parallelizing it into the Parallel HGT-GWO, culminating in the Hybrid HGT-GWO implementation.

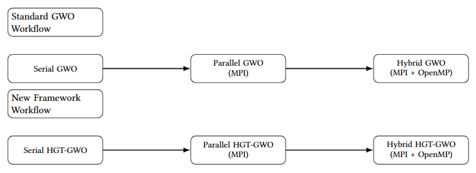

Figure 22: Workflow of Serial, Parallel, and Hybrid Optimization for Standard GWO and HGT-GWO.

Algorithm 1 Serial Implementation of GWO [10]

```
1: Initialize the grey wolf population $\vec{X}_i$ $(i = 1, 2, ..., n)$
2: Initialize $\vec{A}$, $\vec{a}$, and $\vec{C}$
3: Calculate the fitness of each search agent
4: $X_\alpha \leftarrow$ the best search agent
5: $X_\beta \leftarrow$ the second best search agent
6: $X_\delta \leftarrow$ the third best search agent
7: while $t <$ Max number of iterations do
8:     for each search agent do
9:         Update the position of the current search agent using Equation 7
10:    end for
11:    Update $\vec{a}$, $\vec{A}$, and $\vec{C}$
12:    Calculate the fitness of all search agents
13:    Update $X_\alpha$, $X_\beta$, and $X_\delta$
14:    $t \leftarrow t + 1$
15: end while
16: return $X_\alpha$
```

3.1 Historical-Guidance Trend-adjust GWO (HGT-GWO)

HGT-GWO incorporates information about previous iterations or historical positions of search agents to improve the optimization process by balancing exploration and exploitation. It takes step 9: updating the position of the current search agent using equation 7 and modifies it into the following equations:

$$\text{Standard Update (SU)} = \frac{\vec{X}_1 + \vec{X}_2 + \vec{X}_3}{3} \tag{8}$$

$$\text{Historical Guidance Update (HGU)} = \alpha \cdot \text{previous\_best\_pos} \tag{9}$$

$$\text{Trend Adjustment Update (TAU)} = \beta \cdot (\vec{X}_i - \text{previous\_pos}_i) \tag{10}$$

$$\vec{X}_{\text{new},i} = (1 - \alpha - \beta) \cdot \text{SU} + \alpha \cdot \text{HGU} + \beta \cdot \text{TAU} \tag{11}$$

$$\vec{X}_{\text{new},i} = (1-\alpha-\beta)\cdot\frac{\vec{X}_1 + \vec{X}_2 + \vec{X}_3}{3} + \alpha\cdot\text{prev\_best\_pos} + \beta\cdot(\vec{X}_i - \text{prev\_pos}_i) \tag{12}$$

Each component has a specific physical significance:

- $(1 - \alpha - \beta)$ · Standard Update: Preserves the standard GWO update logic and holds the primary weight.
- α · previous_best_pos: Guides the update using the historical global best position, enhancing memory ability.
- $\beta \cdot (\vec{X}_i - \text{previous\_pos}_i)$: Captures dynamic variation trends and strengthens local exploration capability.

The weights of each component (α, β) can be adjusted to balance the influence of different information in the update process.

3.2 Serial HGT-GWO

The Historical-Guidance Trend-ajusted Grey Wolf Optimization (HGT-GWO) extends the basic GWO by incorporating historical guidance and trend-based adjustments. Each wolf updates its position using a weighted combination of standard GWO updates, the best historical position (*previous_best_pos*), and momentum-based adjustments from prior iterations. This allows the algorithm to leverage past knowledge for more refined exploration and exploitation.

Algorithm 2 Serial Implementation of HGT-GWO

```
1: Initialize population $\vec{X}_i$, parameters $\vec{A}$, $\vec{a}$, $\vec{C}$, and bounds $(lb, ub)$
2: Evaluate fitness and identify $X_\alpha$, $X_\beta$, and $X_\delta$
3: Record initial positions in previous_best_pos and previous_positions
4: while $t < max\_iter$ do
5:     for each search agent $i$ do
6:         Calculate $\vec{X}_1$, $\vec{X}_2$, $\vec{X}_3$ based on distances to $X_\alpha$, $X_\beta$, and $X_\delta$
7:         Compute $standard\_update[i]$ as the average of $\vec{X}_1$, $\vec{X}_2$, $\vec{X}_3$
8:         Update $\vec{X}_i$ using:
             $\vec{X}_i = (1 - \alpha\_weight - \beta\_weight) \cdot standard\_update[i]$
             $+\alpha\_weight \cdot prev\_best\_pos + \beta\_weight \cdot (curr\_position - prev\_position)$
9:     end for
10:    Update parameters $\vec{a}$, $\vec{A}$, $\vec{C}$, and evaluate fitness
11:    Update $X_\alpha$, $X_\beta$, $X_\delta$, and historical positions
12:    $t \leftarrow t + 1$
13: end while
14: return $X_\alpha$
```

3.3 HGT-GWO: Parallelization with MPI

The parallel implementation of HGT-GWO distributes the computational workload across multiple processors using MPI. Each process evaluates a portion of the population's fitness locally, and results are aggregated using MPI operations. The root process performs updates to positions and parameters before broadcasting them back to all processes. This parallelization significantly reduces computation time for large-scale problems.

3.4 HGT-GWO: Hybrid Parallelization, MPI and OpenMP

The hybrid implementation combines MPI for inter-process communication with OpenMP for intra-process parallelism. Fitness evaluations and position updates within each process are parallelized using OpenMP threads, while MPI manages population distribution and aggregation across processes. This hybrid approach leverages both shared-memory and distributed-memory parallelism for optimal performance.

Algorithm 3 Parallel Implementation of HGT-GWO

```
 1: Initialize MPI environment, algorithm parameters, and memory
 2: Distribute population across MPI processes (Scatterv)
 3: if rank = 0 then
 4:     Initialize global population and previous_best_pos
 5: end if
 6: while $t < max\_iter$ do
 7:     Evaluate fitness of local population
 8:     Gather and process population on root (Gatherv)
 9:     if rank = 0 then
10:         Identify $\alpha$, $\beta$, $\delta$ wolves; update positions using:
            $X_i[d] = (1 - \alpha\_weight - \beta\_weight) \cdot standard\_update$
            $+\alpha\_weight \cdot previous\_best\_pos[d] + \beta\_weight \cdot (current\_pos - previous\_pos)$
11:         Update previous_best_pos and log results
12:     end if
13:     Broadcast and scatter updated population (MPI_Bcast, MPI_Scatterv)
14: end while
15: Log execution time and free memory
16: return global best position $X_\alpha$
```

Algorithm 4 Hybrid Implementation of HGT-GWO

```
 1: Initialize the grey wolf population $\vec{X}_i$ ($i = 1, 2, ..., n$), parameters $\vec{A}$, $\vec{a}$, and $\vec{C}$, and historical position $\vec{X}^{prev}$
 2: Calculate the fitness of each wolf
 3: $X_\alpha \leftarrow$ the best wolf, $X_\beta \leftarrow$ the second best, $X_\delta \leftarrow$ the third best
 4: while $t <$ Max number of iterations do
 5:     Scatter population among processes (MPI_Scatterv)
 6:     Evaluate fitness of local wolves (parallelized with OpenMP)
 7:     Gather population at the root (MPI_Gatherv)
 8:     Update positions of all wolves using historical guidance and Equation 7
 9:     Update $\vec{X}_\alpha$, $\vec{X}_\beta$, $\vec{X}_\delta$, and $\vec{X}^{prev}$
10:     Broadcast updated population (MPI_Bcast)
11:     $t \leftarrow t + 1$
12: end while
13: return $X_\alpha$
```

3.5 Benchmark test functions

To test our various algorithms we used the following three benchmark functions. They are ideal because of the following:
F1: Sphere Function

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

Unimodal: It has a single global minimum, making it straightforward to evaluate the algorithm's ability to converge to the optimal solution.Evaluates how well GWO can exploit a smooth and convex search space to converge to the global optimum.

F2: Schwefel 2.22 Function

$$f(x) = \sum_{i=1}^{n} |x_i| + \prod_{i=1}^{n} |x_i|$$

Non-Smooth: Includes absolute values and a product term, introducing sharp edges and testing the algorithm's handling of non-differentiable regions.

F3: Schwefel 1.2 Function

$$f(x) = \sum_{i=1}^{n} \left(\sum_{j=1}^{i} x_j \right)^2$$

Cumulative Nature: The function introduces dependencies between variables, making the landscape complex and testing the algorithm's ability to handle correlated variables.

3.6 Data Dependencies

The following code snippet demonstrates the implementation of the update positions function for the Hybrid Grey Wolf Optimization (HGT-GWO) algorithm, showcasing how positions of wolves are updated based on alpha, beta, and delta wolves:

```c
void update_positions(Wolf *population, Wolf alpha
    , Wolf beta, Wolf delta,
                      int pop_size, int dim,
                          double lb, double ub,
                          double a) {
    for (int i = 0; i < pop_size; i++) {
        for (int j = 0; j < dim; j++) {
            double r1 = (double)rand() / RAND_MAX;
            double r2 = (double)rand() / RAND_MAX;
            double A1 = 2.0 * a * r1 - a;
            double C1 = 2.0 * r2;
            double D_alpha = fabs(C1 * alpha.
                position[j] - population[i].
                position[j]);
            double X1 = alpha.position[j] - A1 *
                D_alpha;

            r1 = (double)rand() / RAND_MAX;
            r2 = (double)rand() / RAND_MAX;
            double A2 = 2.0 * a * r1 - a;
            double C2 = 2.0 * r2;
            double D_beta = fabs(C2 * beta.
                position[j] - population[i].
                position[j]);
            double X2 = beta.position[j] - A2 *
                D_beta;

            r1 = (double)rand() / RAND_MAX;
            r2 = (double)rand() / RAND_MAX;
            double A3 = 2.0 * a * r1 - a;
            double C3 = 2.0 * r2;
            double D_delta = fabs(C3 * delta.
                position[j] - population[i].
                position[j]);
            double X3 = delta.position[j] - A3 *
                D_delta;

            population[i].position[j] = (X1 + X2 +
                 X3) / 3.0;

            // Boundary check
            if (population[i].position[j] < lb)
                population[i].position[j] = lb;
            if (population[i].position[j] > ub)
                population[i].position[j] = ub;
        }
    }
}
```

Listing 1: Update Positions Function in C

Table 1: Data Dependencies

Location	Early/Late	Line	Iteration	Access	Carried	Type
population[i].position[j]	Early	9	j	Read	No	Flow
population[i].position[j]	Late	12	j	Write	No	Out
alpha.position[j]	Early	9	j	Read	No	-
beta.position[j]	Early	9	j	Read	No	-
delta.position[j]	Early	9	j	Read	No	-
X1	Early	9	j	Read	No	Flow
X2	Late	9	j	Read	No	Flow
X3	Late	9	j	Read	No	Flow
population[i].position[j]	Late	12	j	Write	No	Flow

NOTES

3.6.1 Data Dependency Breakdown. This section explains the rationale behind the data dependencies:

Read Dependencies: Variables such as `alpha.position[j]`, `beta.position[j]`, and `delta.position[j]` are accessed for reading to compute intermediate values (`X1`, `X2`, `X3`). These operations are independent across iterations, ensuring no loop-carried dependencies.

Write Dependencies: The final position updates (`population[i].position[j]`) occur after the computation of intermediate values and are scoped within each iteration. There is no dependency on previous iterations.

Flow Dependencies: Intermediate computations (`X1`, `X2`, `X3`) are locally scoped to the iteration and are used only for updating the corresponding position in the current loop iteration.

Parallelization Potential: The outer loop (`i` over wolves) and the inner loop (`j` over dimensions) are fully independent, making them ideal for parallelization. Each thread can safely handle a different `i` or `j` without conflicts or data races.

3.7 PBS Directives

For the PBS directives used to submit the jobs to the cluster, various python scripts were used found in the repository[5].

4 EXPERIMENTAL EVALUATION

4.1 High-Performance Computing Cluster Specifications

The High-Performance Computing (HPC) cluster at the University of Trento was utilized to test the algorithms as mentioned above. The main characteristics of the cluster are:

- Operating System: Linux CentOS 7
- Nodes: 126
- CPU Cores: 6092
- CUDA Cores: 37,376
- RAM: 53 TB
- Network:
  - All nodes are interconnected with a 10 Gb/s network
  - Some nodes have Infiniband 40 Gb/s connectivity
  - Others have Omnipath 100 Gb/s connectivity

4.2 Parallelization Results

Firstly for a detailed comparison and visualisation of the execution times of the standard serial implementation of the GWO and its MPI Parallelization, we have the detailed times in the GitHub folder [5]. To demonstrate the execution times, we shall analyse only the HGT-GWO MPI parallelization and the HGT-GWO Hybrid.

4.3 Algorithmic Evaluations for Parallelization of HGT-GWO using MPI

The performance of the Grey Wolf Optimization (HGT-GWO) algorithm when parallelized using MPI can be assessed through key metrics that evaluate computational efficiency and scalability:

- **Speedup (S):**
  - Measures the performance gain when executing the algorithm with N processes compared to a single process.
  - Defined as:
$$S = \frac{T_1}{T_N}$$
    where T_1 is the execution time on one process, and T_N is the time on N processes.
- **Efficiency (E):**
  - Evaluates how effectively the computational resources are utilized with N processes.
  - Defined as:
$$E = \frac{S}{N} = \frac{T_1}{N \cdot T_N}$$
- **Strong Scalability (E_{strong}):**
  - Assesses the ability of the algorithm to maintain efficiency when the size of the problem remains constant as N increases.
  - Defined as:
$$E_{\text{strong}} = \frac{S}{N}$$
- **Weak Scalability (E_{weak}):**
  - Examines the performance of the algorithm when the size of the problem increases proportionally to the number of processes, ensuring a constant workload per process.
  - Defined as:
$$E_{\text{weak}} = \frac{T_1}{T_N}$$
    where the workload per process remains constant.

These metrics provide insights into bottlenecks, communication overhead, and load-balancing challenges.

4.2.1 HGT-GWO MPI Parallelization

The dimension details for the MPI parallelizations are:

- `n_core` = {1, 2, 4, 8, 16, 32, 64}
- `dimensions` = {256, 512, 1024}

1. Execution times. The execution time plots for the Sphere, Schwefel 2.22, and Schwefel 1.2 functions (Figures 3, 4, and 5) demonstrate how the algorithm's computational performance scales with an increasing number of cores at different dimensionalities (256D, 512D, and 1024D). Across all three functions, execution time decreases as the number of cores increases, illustrating the effectiveness of parallelization. The Sphere function, being computationally simpler, achieves the lowest execution times, as reflected by its rapid decline in time with more cores. However, at higher dimensions, its execution time plateaus and shows a slight increase beyond 32 cores, likely due to communication overhead outweighing the computational benefits. Similarly, the Schwefel 2.22 and Schwefel 1.2 functions exhibit significant reductions in execution time with more cores, particularly for larger dimensions, where the increased workload ensures better utilization of resources. Nevertheless, a slight increase in execution time at 64 cores is noticeable for all functions, especially for Schwefel 1.2, which is the most computationally intensive. This anomaly can be attributed to parallel overheads such as increased communication and synchronization costs or suboptimal load balancing, which become more pronounced as the number of processes increases. The divergence between execution times at different dimensions highlights the role of problem complexity in maintaining computational efficiency, with higher dimensions benefiting more from parallelization.

Table 2: Sphere Function Execution Times (in seconds)

Cores	256 Dim.	512 Dim.	1024 Dim.
1	8.252094	9.534996	19.158959
2	3.666704	5.992016	10.594633
4	1.80743	3.01651	4.779819
8	0.917035	1.723238	2.525541
16	0.586738	0.862373	1.595241
32	0.501741	0.671417	1.2724
64	0.703353	0.975766	1.48192

Table 3: Schwefel 2.22 Function Execution Times (in seconds)

Cores	256 Dim.	512 Dim.	1024 Dim.
1	11.518391	13.985519	25.493448
2	4.466749	7.312965	12.203281
4	1.998745	3.349461	5.11318
8	0.970195	1.775817	2.643698
16	0.585455	0.880886	1.610919
32	0.51648	0.722308	1.374787
64	0.884081	1.130362	1.458286

HGT-GWO MPI Parallelization Speedup comparison

2. Speedup. The speedup graph shows that the Schwefel 2.22 function demonstrates good scalability up to 32 cores, particularly for 256D, which achieves the highest peak speedup of around 20 S=20. However, for higher dimensions like 512D and 1024D, the speedup values are lower, peaking closer to =15 S=15. After 32 cores, the speedup sharply declines across all dimensions. This diminishing speedup can be attributed to increasing MPI communication overhead, as the number of processes rises, leading to significant time spent on data exchange between processes. Additionally, smaller problem sizes, especially for 256D, exacerbate the imbalance in workload distribution across cores at higher core counts, further limiting performance improvements.

Table 4: Schwefel 1.2 Function Execution Times (in seconds)

Cores	256 Dim.	512 Dim.	1024 Dim.
1	57.895971	167.956992	755.74053
2	26.168104	97.587154	380.544003
4	13.091358	48.797115	168.845383
8	6.727586	25.244399	86.237104
16	3.672174	11.787053	46.195825
32	2.120744	6.184004	23.663539
64	1.507321	3.675589	12.928271

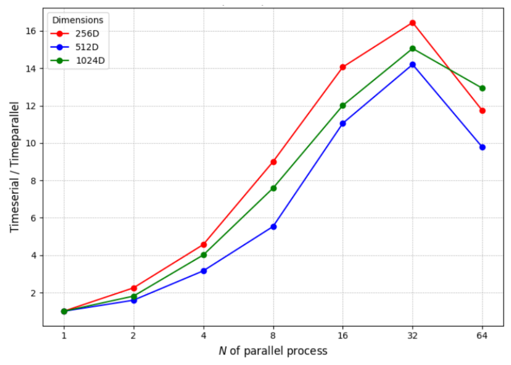

Figure 2: Sphere Function speedup.

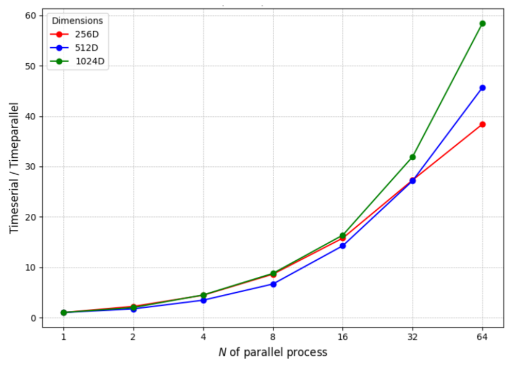

Figure 3: Schwefel 1.2 Function speedup.

HGT-GWO MPI Parallelization efficiency

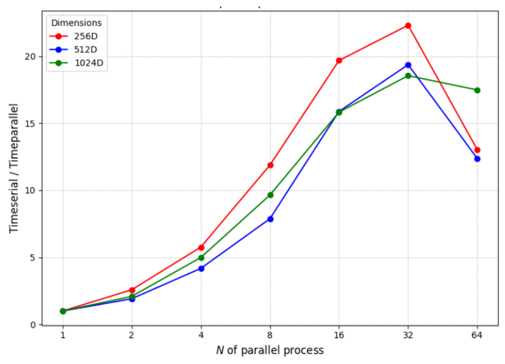

Figure 4: Schwefel 2.22 Function speedup.

3. Efficieny. The efficiency plots (Figures 9, 10, and 11) for the Sphere, Schwefel 1.2, and Schwefel 2.22 functions reveal how well the computational resources are utilized as the number of parallel processes increases. Across all three functions, efficiency is highest at lower core counts and diminishes as the number of processes increases, particularly beyond 16 cores. For the Sphere function, efficiency declines steeply for higher dimensions (e.g., 5120D), highlighting that the computational load per core reduces relative to the communication and synchronization overheads. The Schwefel 1.2 and Schwefel 2.22 functions show slightly better efficiency trends, particularly at higher dimensions, due to their greater computational complexity, which provides more substantial workloads per core and mitigates the relative impact of overhead. However, the efficiency decreases for all functions as the number of processes exceeds the workload scalability threshold, suggesting that inter-process communication costs and diminishing workload per core outweigh the benefits of parallelism.

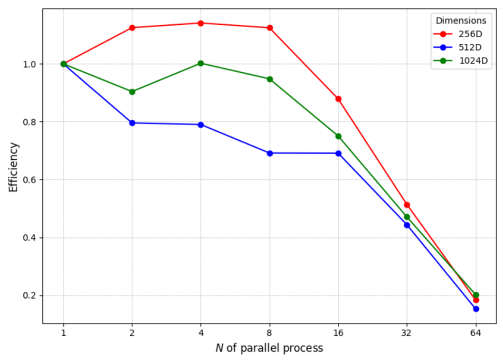

Figure 5: Sphere Function efficiency.

HGT-GWO MPI Parallelization strong scalability

4. Strong Scalability. The strong scalability plots (Figures 12, 13, and 14) illustrate how well the Parallel MPI HGT-GWO algorithm handles a fixed problem size as the number of parallel processes increases. At lower core counts, the algorithm achieves near-linear scalability for all three functions, especially at higher dimensions (e.g., 1024D). However, as the number of processes increases beyond 16, scalability begins to diminish for the Sphere function and to a lesser extent for Schwefel 2.22 and Schwefel 1.2. This degradation is more pronounced at lower dimensions, where the computational workload becomes insufficient to offset the communication and synchronization overheads. For example, the Sphere function at 256D exhibits a sharp decline in scalability, reflecting its simplicity and low computational cost. In contrast, the Schwefel functions maintain better scalability trends due to their higher computational complexity, which ensures a more balanced load among processes. The diminishing trends across all functions at high core counts suggest that the algorithm encounters bottlenecks such as excessive communication overhead, memory contention, or load imbalance, which reduce the effectiveness of parallel execution. Addressing these issues may require techniques such as improved workload distribution or optimized inter-process communication.

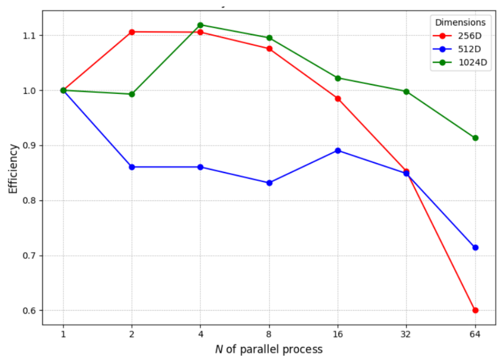

Figure 6: Schwefel 1.2 Function efficiency.

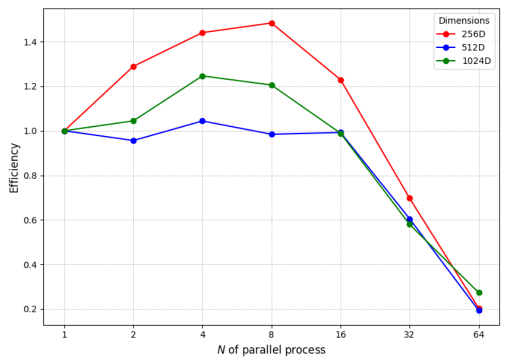

Figure 7: Schwefel 2.22 Function efficiency.

4. Weak Scalability. The weak scalability analysis of the three benchmark functions—Sphere, Schwefel 2.22, and Schwefel 1.2—provides insight into how the MPI HGT-GWO algorithm performs as the number of parallel processes increases while scaling the problem size proportionally. For the Sphere function, the plots indicate relatively stable weak scalability, with efficiency remaining near 1 for the first index but showing a slight decline for indices 2 and 3 as the number of cores increases. This behavior reflects the Sphere

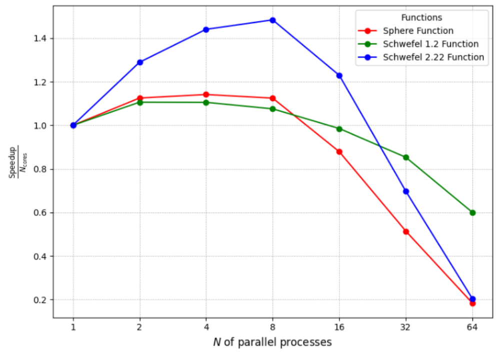

Figure 8: Strong scalability evaluation at 256 dimensions.

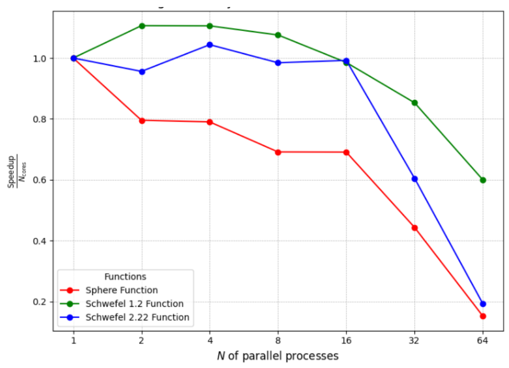

Figure 9: Strong scalability evaluation at 512 dimensions..

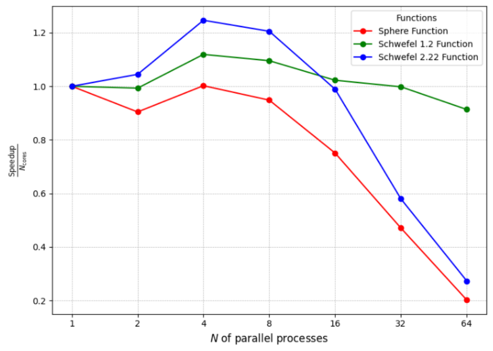

Figure 10: Strong scalability evaluation at 1024 dimensions.

function's simplicity, as its gradient-based nature results in uniform computational workload, allowing better load balancing. Conversely, Schwefel 2.22 exhibits notable performance variations, particularly for the first index, where efficiency initially improves before tapering off. This irregularity could stem from the function's sensitivity to initial positions and the computational cost associated with evaluating complex search spaces. Schwefel 1.2 shows a contrasting trend, with efficiency dropping significantly for indices 2 and 3 as the core count rises, suggesting that inter-process communication or synchronization overheads disproportionately impact this function due to its higher dependency on sequential cumulative evaluations. Overall, the observed anomalies highlight the interaction between function complexity, load distribution, and parallelization overhead, which are critical factors in scaling MPI HGT-GWO algorithm.

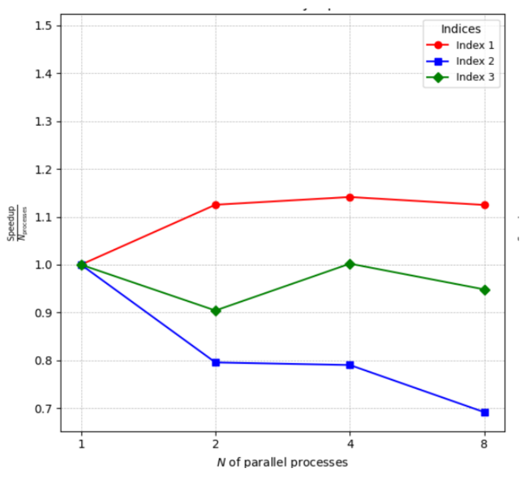

Figure 11: Sphere function weak scalability evaluation.

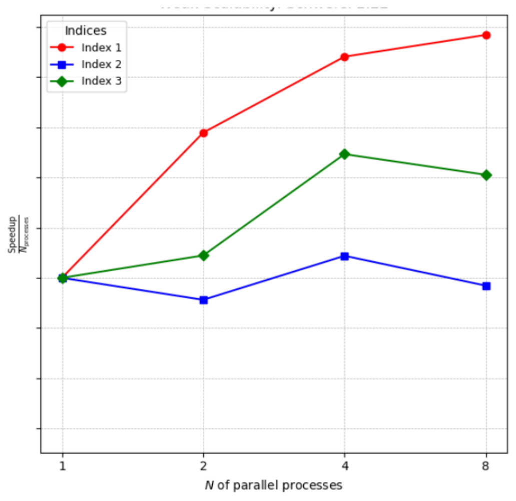

Figure 12: Schwefel 2.22 function weak scalability evaluation.

Index	N_cores	Dimensions
1	2	256
2	4	512
3	8	1024

Table 5: Index Table for Weak Scalability

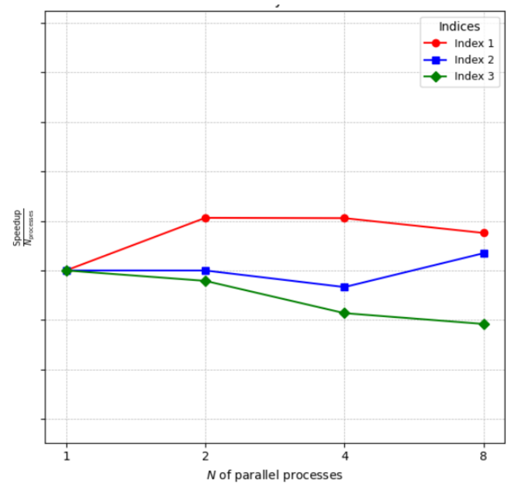

Figure 13: Schewefel 1.2 function weak scalability evaluation.

4.4 Algorithmic Evaluations for Parallelization of Hybrid HGT-GWO

4.2.2 HGT-GWO Hybrid Parallelization

The dimension details for the hybrid parallelizations are:

- `n_threads` = {1, 2, 4}
- `n_cores` = {2, 4, 8, 16}
- `dimensions` = {256, 512, 1024}

1. Execution times. The execution times for the Hybrid HGT-GWO algorithm across different numbers of cores for the Sphere, Schwefel 2.22, and Schwefel 1.2 functions at 256 dimensions reveal significant insights into the parallel performance of the algorithm. As the number of cores increases, the execution times decrease for all three functions, demonstrating the effectiveness of parallelization. The Sphere function, being computationally simple, shows the most substantial reduction in execution time, with a decrease from 4.799 seconds on 2 cores to 1.040 seconds on 16 cores—a near-linear improvement up to 8 cores, followed by a smaller reduction thereafter. Similarly, Schwefel 2.22 exhibits a comparable trend, with its execution time decreasing from 5.591 seconds on 2 cores to 0.917 seconds on 16 cores, indicating efficient parallel performance. The Schwefel 1.2 function, being the most computationally intensive, exhibits the largest initial execution times but also benefits significantly from parallelism, reducing from 29.929 seconds on 2 cores to 4.151 seconds on 16 cores. However, the rate of improvement diminishes as the core count increases, particularly beyond 8 cores, suggesting the presence of parallel overheads such as communication costs and synchronization delays. This diminishing trend highlights that while the algorithm benefits from parallel execution, its scalability is limited by the increasing impact of overheads as the number of processes grows.

2. Speedup. The speedup plots for the three functions—Sphere, Schwefel 2.22, and Schwefel 1.2—demonstrate the scalability of the Hybrid HGT-GWO algorithm. The results indicate that as the number of cores increases, the speedup improves for all three functions, which is expected due to the parallel nature of the algorithm. For the Sphere function, the speedup exhibits a near-linear growth initially, but it starts tapering off beyond 8 cores, highlighting diminishing returns in performance. A similar trend is observed for the Schwefel 2.22 and Schwefel 1.2 functions, though Schwefel 1.2 achieves slightly higher speedups at higher core counts, likely due to the inherent computational characteristics of the function making it more amenable to parallelism. However, the deviation from ideal linear speedup suggests the presence of overheads, such as inter-process communication and synchronization costs, which become more pronounced as the number of cores increases.

Table 6: Comparison of Execution Times (256 Dimensions) for Sphere, Schwefel 2.22, and Schwefel 1.2 Functions

Cores	Sphere Function (s)	Schwefel 2.22 (s)	Schwefel 1.2 (s)
2	4.799105	5.591696	29.928764
4	2.676503	2.702754	14.003771
8	1.332511	1.34737	6.368899
16	1.040307	0.916712	4.150996

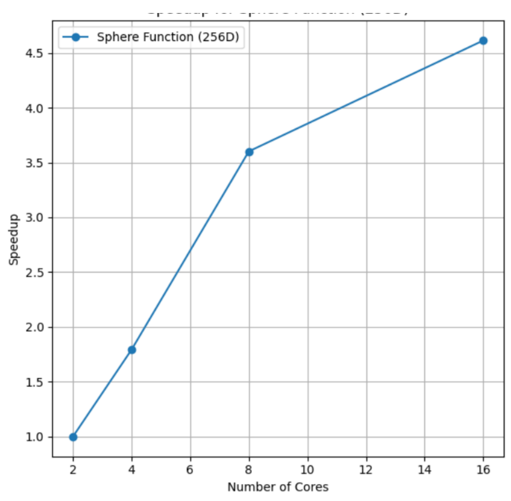

Figure 14: Sphere Function speedup.

3. Efficiency. The efficiency plots provide insight into how well the computational resources are utilized as the number of cores increases. For all three functions, efficiency decreases with an increasing number of cores, which is a typical behavior in parallel computing. The Sphere function shows a steady decline in efficiency, dropping below 0.3 for 16 cores, indicating significant parallel overheads. Schwefel 2.22 retains slightly higher efficiency compared to the Sphere function, particularly at lower core counts, suggesting a better load distribution or lower communication overhead for this function. Interestingly, Schwefel 1.2 exhibits the highest efficiency across the board, maintaining values above 0.4 up to 16 cores. This could be attributed to its problem-specific characteristics that align well with the Hybrid HGT-GWO algorithm, resulting in better parallel performance. Nonetheless, the decline in efficiency across all

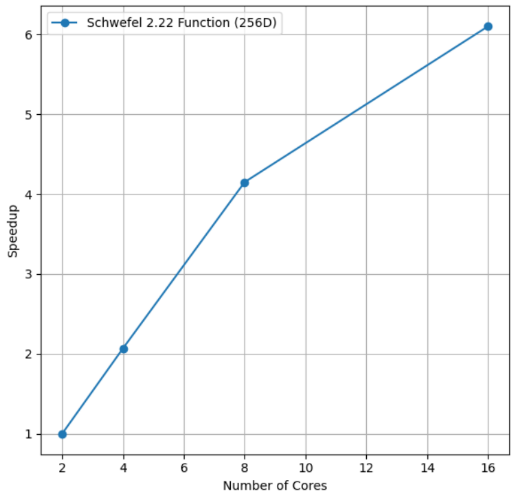

Figure 15: Schwefel 2.22 Function speedup.

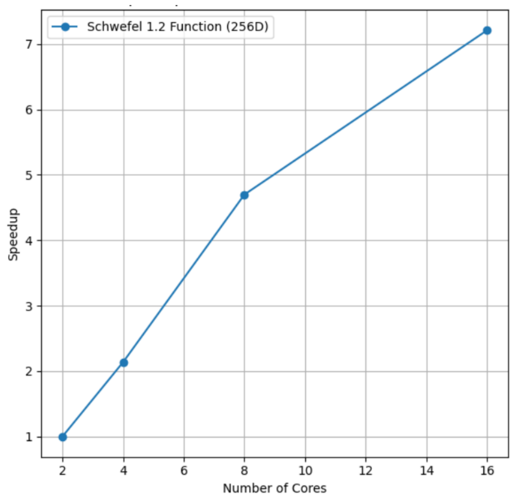

Figure 16: Schwefel 1.2 Function speedup.

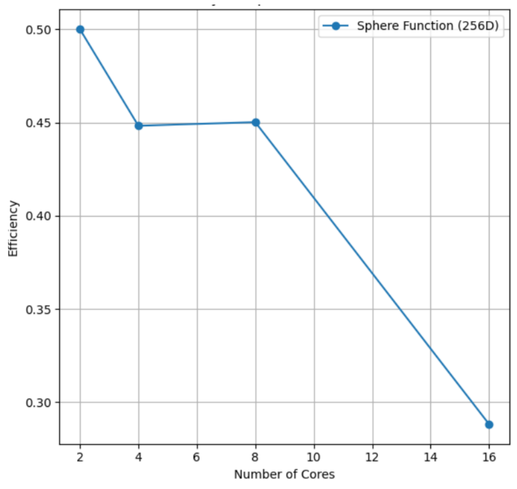

Figure 17: Sphere Function efficiency.

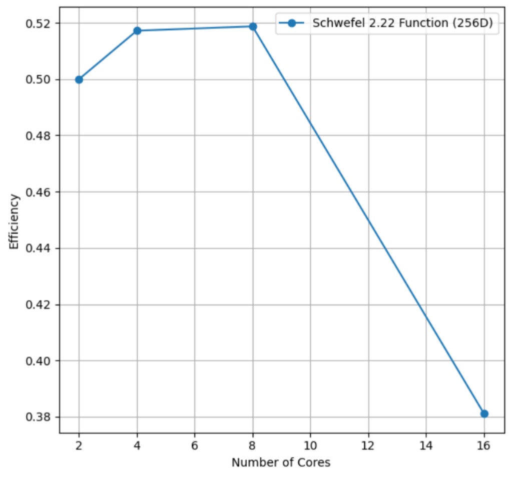

Figure 18: Schwefel 2.22 Function efficiency.

functions points to challenges in maintaining scalability, particularly for larger core counts.

4. Strong Scalability. The strong scalability analysis of the hybrid HGT-GWO algorithm, evaluated at 256 dimensions for the three benchmark functions—Sphere, Schwefel 2.22, and Schwefel 1.2—reveals critical insights into its performance as the number of cores increases. The plot indicates that Schwefel 1.2 achieves the highest speedup, followed by Schwefel 2.22, and finally, the Sphere function, which demonstrates the least improvement in speedup as the number of cores scales. This trend highlights the differing computational complexities of the functions, with Schwefel 1.2 benefiting the most from parallelization due to its inherently larger and more complex search space, where the algorithm's workload can be effectively distributed. In contrast, the Sphere function exhibits limited scalability, possibly due to its simplicity and the diminishing returns from increasing parallelism, as overheads such as inter-process communication and synchronization become more prominent relative to the computation time. The Schwefel 2.22 function shows a moderate scaling behavior, indicating a balance between computational complexity and communication overhead. The anomalies, such as the slower-than-expected speedup for the Sphere function at higher core counts, could be attributed to the fixed problem size (256 dimensions), where the diminishing workload per core exacerbates the impact of parallelization overheads, thereby limiting scalability.

5. Weak Scalability. There are no visualizations for weak scalability because, to plot such graphs, we need to ensure that the problem size increases proportionally with the number of cores. Specifically, for weak scalability, the execution times must be evaluated at the following configurations:

- 256 dimensions on 2 cores
- 512 dimensions on 4 cores

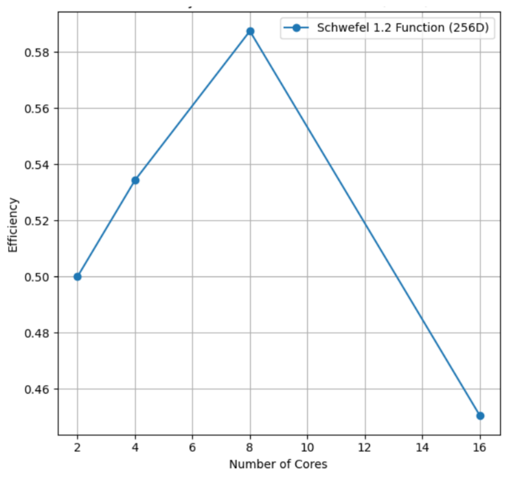

Figure 19: Schwefel 1.2 Function efficiency.

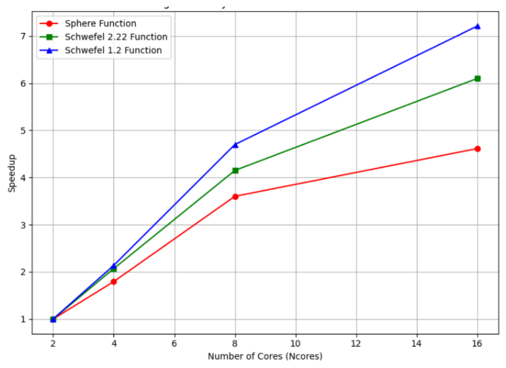

Figure 20: Strong Scalability evaluation of the three functions at 256 dimensions.

- 1024 dimensions on 8 cores

For the Hybrid HGT-GWO analysis, the analysis is only conducted at 256 dimensions.

5 COMPARISON OF EXECUTION TIMES

The three tables 7,8 and 9 provide a comparative analysis of the execution times for the standard GWO Hybrid (GWO) algorithm and its hybrid parallel variant, Hybrid (HGT-GWO), implemented with OpenMP and MPI. The results are presented for three benchmark functions: the Sphere function, Schwefel 2.22, and Schwefel 1.2, all at 256 dimensions, with varying numbers of cores (2, 4, 8, and 16) and threads (1, 2, and 4). Across all functions, HGT-GWO consistently outperforms the standard GWO in terms of execution time, particularly as the number of cores and threads increases, demonstrating the benefits of parallelism. For the Sphere function, HGT-GWO exhibits significant speedups, especially for higher cores and threads, where the execution time reduces from 4.80 seconds (1 thread, 2 cores) to 1.04 seconds (1 thread, 16 cores). A similar trend is observed for Schwefel 2.22 and Schwefel 1.2, with HGT-GWO achieving notable reductions in execution time. However, the efficiency gain is more pronounced for simpler functions like Sphere, while the more complex Schwefel functions see relatively smaller reductions. This highlights the varying computational demands of different objective functions and the adaptability of the hybrid parallel implementation.

These three tables show that the suggested new framework outperforms the standard GWO algorithm. This comparison is on both the hybrid versions of the two algorithms.

Table 7: Execution Times for F1: Sphere Function (256D)

Ncores	Threads	GWO	HGT-GWO
2	1	8.37	4.80
2	2	29.21	13.62
2	4	116.28	51.27
4	1	3.69	2.68
4	2	179.78	56.52
4	4	57.09	24.93
8	1	1.82	1.33
8	2	96.93	32.26
8	4	28.81	12.78
16	1	1.26	1.04
16	2	40.38	13.70
16	4	15.82	6.28

Table 8: Execution Times for F2: Schwefel 2.22 (256D)

Ncores	Threads	GWO	HGT-GWO
2	1	9.98	5.59
2	2	46.25	18.08
2	4	118.34	50.57
4	1	4.24	2.70
4	2	144.54	60.69
4	4	58.99	22.11
8	1	1.88	1.35
8	2	96.61	31.96
8	4	33.65	13.08
16	1	1.28	0.92
16	2	41.03	13.86
16	4	15.03	6.38

6 CONCLUSION

Although the Grey Wolf Optimizer (GWO) is effective, its reliance on the top three wolves (alpha, beta, and delta) leads to sorting and communication overhead, posing challenges in parallel implementations. This work introduced the History-Guided Trend-adjusted Grey Wolf Optimization (HGT-GWO) and a novel master-worker island parallelization scheme, enhancing exploration, exploitation, and scalability. Based on MPI, the framework demonstrated significant improvements in computational efficiency, showcasing HGT-GWO's potential for large-scale, high-dimensional optimization

Table 9: Execution Times for F3: Schwefel 1.2 (256D)

Ncores	Threads	GWO	HGT-GWO
2	1	33.28	29.93
2	2	40.33	31.68
2	4	130.98	49.05
4	1	15.18	14.00
4	2	186.17	66.57
4	4	48.41	23.36
8	1	6.79	6.37
8	2	99.53	35.47
8	4	33.21	13.61
16	1	4.38	4.15
16	2	41.44	14.95
16	4	16.87	7.33

problems. However, further optimization is needed for both GWO and HGT-GWO in MPI+OpenMP multi-threaded environments to fully leverage hardware performance.

6.1 Future Works

Future work will focus on optimizing synchronization strategies in the master-worker island scheme to further reduce communication overhead and improve scalability. Algorithm optimization in multi-threaded environments, extending the framework to ultra-large clusters, and integrating GPU acceleration will enhance computational efficiency for high-dimensional problems. Additionally, adaptive parameter tuning and benchmarking against state-of-the-art methods will further refine the algorithm. Finally, applying HGT-GWO to real-world tasks such as energy management and neural network training will validate its practical effectiveness.

REFERENCES

[1] R. Chandra (Ed.). 2007. *Parallel Programming in OpenMP.* Morgan Kaufmann, San Francisco, Calif.

[2] M. Dorigo, M. Birattari, and T. Stutzle. 2006. Ant Colony Optimization. *IEEE Computational Intelligence Magazine* 1, 4 (Nov 2006), 28–39. https://doi.org/10.1109/MCI.2006.329691

[3] R. Eberhart and J. Kennedy. 1995. A New Optimizer Using Particle Swarm Theory. In *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science.* IEEE, Nagoya, Japan, 39–43. https://doi.org/10.1109/MHS.1995.494215

[4] S. Gupta and K. Deep. 2020. A memory-based Grey Wolf Optimizer for global optimization tasks. *Applied Soft Computing* 93 (Aug 2020), 106367. https://doi.org/10.1016/j.asoc.2020.106367

[5] Y. Jiang, M. Zhou, and E. Nwafornso. 2025. Avalon-S/HPC4DS-Project-GWO. https://github.com/Avalon-S/HPC4DS-Project-GWO Accessed: Jan. 18, 2025.

[6] D. Karaboga and B. Basturk. 2007. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. *Journal of Global Optimization* 39, 3 (Oct 2007), 459–471. https://doi.org/10.1007/s10898-007-9149-x

[7] S. Mirjalili, S. M. Mirjalili, and A. Lewis. 2014. Grey Wolf Optimizer. *Advances in Engineering Software* 69 (Mar 2014), 46–61. https://doi.org/10.1016/j.advengsoft.2013.12.007

[8] C. Muro, R. Escobedo, L. Spector, and R. Coppinger. 2011. Wolf-pack (Canis Lupus) hunting strategies emerge from simple rules in computational simulations. *Behavioral Processes* 88, 3 (2011), 192–197. https://doi.org/10.1016/j.beproc.2011.09.006

[9] P. S. Pacheco. 1997. *Parallel Programming with MPI.* Morgan Kaufmann, San Francisco.

[10] Shubham. 2023. shubham-garad/GWO. https://github.com/shubham-garad/GWO. Accessed: Jan. 19, 2025.

[11] Wikipedia Contributors. 2025. File:Wolves and elk (cropped).jpg - Wikipedia. https://commons.wikimedia.org/wiki/File:Wolves_and_elk_(cropped).jpg Accessed: Jan. 05, 2025.