Raven Roosting Optimization with MPI: A Parallel Metaheuristic Approach

Hector Carlos Flores Reynoso
Department of Computer Science
University of Trento
Student ID 266140
hector.floresreynoso@unitn.it

***Abstract*—This paper presents the implementation and parallelization of the Raven Roosting Optimization (RRO) algorithm [1], a nature-inspired metaheuristic optimization technique based on the foraging behavior of ravens. We describe both a serial implementation in C and a parallel version using Message Passing Interface (MPI) for distributed memory systems. The algorithm simulates raven behavior including roosting, foraging, and social hierarchy with leader-follower dynamics. Our parallel implementation distributes the population across multiple processes, achieving significant speedup through domain decomposition and collective communication patterns. We present detailed pseudocode for both implementations, discuss key optimizations including early stopping mechanisms and efficient bound checking, and analyze the communication overhead and scalability characteristics of the MPI-based approach.**

***Index Terms*—MPI, Parallel Computing, Metaheuristic Optimization, Raven Roosting**

I. Introduction

A. Background

Metaheuristic optimization algorithms have emerged as powerful tools for solving complex, NP-hard optimization problems where traditional exact methods become computationally intractable. These nature-inspired algorithms offer several key advantages: they require no gradient information, can escape local optima through stochastic mechanisms, and provide near-optimal solutions with significantly less computational effort than classical approaches. Their versatility and problem-agnostic nature have led to widespread applications across diverse domains including production scheduling, engineering design, and resource optimization [2]. The Raven Roosting Optimization (RRO) algorithm represents a nature-inspired metaheuristic, mimicking the intelligent foraging behavior and social coordination of ravens.

B. Algorithm Overview

The RRO algorithm is based on three key behavioral patterns observed in ravens [1]:

- **Roosting Behavior**: Ravens return to a common roosting site where they rest and plan their next foraging expedition.
- **Foraging Behavior**: Ravens explore the search space looking for food sources (optimal solutions) through a series of flight steps with exploratory lookouts.
- **Social Hierarchy**: Ravens follow a leader-follower dynamic where the best-performing individual becomes the leader, and a subset of the population acts as followers who explore in the vicinity of the leader.

In real life, ravens has shown a social behaviour mainly through communal roosting that functions as an information-sharing system about food. At night, juvenile, non-breeding ravens gather to successful foragers to scout for short-lived carcasses as described by the Information Center Hypothesis. This social system is flexible: different individuals are knowledgeable at different times, and ravens combine socially acquired information with their own private knowledge when deciding whether to follow others or search independently [3].

C. Problem Statement

The algorithm aims to minimize an objective function $f(\boldsymbol{x})$ where $\boldsymbol{x} \in \mathbb{R}^n$ represents a position in the search space with n dimensions (features). Each raven in the population represents a candidate solution, and the algorithm iteratively improves these solutions through exploration and exploitation phases.

The algorithm maintains:

- A population of ravens, each with a current position and discovered food source
- Fitness values for each raven based on the objective function
- A global leader (best solution found)
- A roosting site that is kept throught each iteration.

D. Motivation for Parallelization

Given the computationally intensive nature of the algorithm, especially with large populations and high-dimensional search spaces, parallelization using MPI becomes essential for:

- Reducing execution time through distributed computation
- Handling larger problem instances
- Scaling across multiple compute nodes in HPC environments

II. Term Discussion

A. Objective Function

The Griewank function is a multimodal optimization benchmark function widely used to test the performance of optimization algorithms. It is particularly challenging due to

its many local minima that can trap optimization algorithms, while having a single global minimum. This function serves as the objective function in the Raven Roost Algorithm (RRA) implementation, providing a rigorous test case for evaluating the algorithm's exploration and exploitation capabilities [1].

The mathematical formulation of the Griewank function is:

$$f(\boldsymbol{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \tag{1}$$

where $\boldsymbol{x} = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ represents a point in the n-dimensional search space.

a) *Function Characteristics:*

Domain: The function is typically evaluated on the hypercube $x_i \in [-600, 600]$ for all dimensions $i = 1, ..., n$ [4].

Global Minimum: The unique global minimum occurs at $\boldsymbol{x}^* = (0, 0, ..., 0)$ with $f(\boldsymbol{x}^*) = 0$.

Multimodality: The function exhibits many widespread local minima that are regularly distributed across the search space. The cosine component creates numerous local optima that can trap gradient-based methods, while the quadratic term creates a parabolic basin structure. This dual nature makes the Griewank function particularly effective for testing an algorithm's balance between exploration (escaping local minima) and exploitation (converging to the global optimum) [4].

III. Serial Implementation

A. Algorithm Structure

The serial implementation follows a straightforward execution model where a single process handles the entire population of ravens.

a) *Main Parameters:*

The algorithm accepts the following parameters:

- **Dataset Path**: CSV file containing initial population positions (NxM matrix)
- **Lower/Upper Bounds**: Search space boundaries for each dimension
- **Iterations**: Number of complete optimization cycles
- **Flight Steps**: Number of steps each raven takes toward its destination
- **Lookout Steps**: Number of exploratory searches per flight step
- **Radius**: Looking radius for exploratory searches ($r_{\{\text{pcpt}\}}$)

b) *Core Algorithm Pseudocode:*

```
Algorithm: Raven Roosting Optimization (Serial)
Input: pop_size, features, iterations, flight_steps, lookout_steps, bounds, radius
Output: Best solution found
Initialize population from dataset
Evaluate fitness for all ravens
Set roosting_site as centroid
current_leader ← argmin(fitness)
followers ← select 20% of population (excluding leader)
for iter = 1 to iterations do
    for i = 1 to pop_size do
        current_position[i] ← roosting_site
        if is_follower[i] then
            target ← random_point_near(leader, r_pcpt)
        else
            target ← food_source[i]
        end if
        for step = 1 to flight_steps do
            direction ← normalize(target - current_position[i])
            remaining ← distance(current_position[i], target)
            step_size ← uniform(0, remaining)
            current_position[i] ← current_position[i] + step_size × direction
            enforce_bounds(current_position[i])
            for lookout = 1 to lookout_steps do
                candidate ← random_point_near(current_position[i], r_pcpt)
                enforce_bounds(candidate)
                if f(candidate) < fitness[i] then
                    food_source[i] ← candidate
                    fitness[i] ← f(candidate)
                    if random() < 0.1 then
                        break
                    end if
                end if
            end for
        end for
    end for
    current_leader ← argmin(fitness)
    followers ← random_select(20%, excluding leader)
end for
return food_source[current_leader]
```

Fig. 1. Pseudocode for serial RRO implementation

c) *Key Implementation Details:*

Initialization Phase:

The initialization phase loads the population from a CSV dataset, evaluates initial fitness values, and calculates the roosting site as a reference point. The looking radius parameter $r_{\{\text{pcpt}\}}$ is computed based on the specified radius and search space bounds.

Distance Calculation:

Euclidean distance is computed between positions in the search space. For two positions $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$, the distance is calculated as:

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum_{j=1}^{n} \left(b_j - a_j\right)^2} \tag{2}$$

Leader Selection:

The leader is determined as the raven with the minimum fitness value, representing the best solution found so far:

$$\text{leader} = \arg\min_{i \in \{1,\ldots,P\}} f(\boldsymbol{x}_i) \tag{3}$$

Flight Behavior:

Each raven performs a series of flight steps toward its target. The movement follows:

$$\boldsymbol{x}_{i,t+1} = \boldsymbol{x}_{i,t} + s_t \cdot \hat{\boldsymbol{d}} \tag{4}$$

where

$$\hat{\boldsymbol{d}} = \frac{\boldsymbol{t}_i - \boldsymbol{x}_{i,t}}{\|\boldsymbol{t}_i - \boldsymbol{x}_{i,t}\|} \tag{5}$$

is the normalized direction toward target $\boldsymbol{t}_i$, and

$$s_t \sim \mathcal{U}\big(0, \|\boldsymbol{t}_i - \boldsymbol{x}_{i,t}\|\big) \tag{6}$$

is the step size uniformly sampled from the remaining distance. At each step, the raven performs lookout steps to search for better food sources within radius $r_{\{\text{pcpt}\}}$.

d) *Memory Management:*

The serial implementation carefully manages memory for:

- Population matrices (`food_source`, `current_position`)
- Fitness array
- Temporary vectors (`direction`, `n_candidate_position`, `prev_location`, `final_location`)
- Leader position and follower flags

e) *Time Complexity:*

For a population of size P, N features, I iterations, F flight steps, and L lookout steps:

- **Overall**: $O(I \cdot P \cdot F \cdot L \cdot N)$
- **Objective function evaluations**: $O(I \cdot P \cdot F \cdot L)$

IV. Parallel Implementation with MPI

A. Parallelization Strategy

The parallel implementation in the `parallel/` directory uses MPI to distribute the raven population across multiple processes. The key parallelization approach is **domain decomposition** [5], where each MPI process manages a subset of the total population.

a) *MPI Context and Configuration:*

The implementation initializes an MPI context structure that tracks process rank and world size, communicator, and global configuration parameters broadcast to all processes. The context is initialized at program startup to establish the distributed computing environment.

b) *Data Distribution:*

Each process handles a local subset of ravens:

- Process 0 reads parameters and broadcasts to all processes
- Population is divided among processes
- Each process maintains local copies of: food sources, positions, fitness values
- Global leader information is shared across all processes

c) *Parallel Algorithm Pseudocode:*

```
Algorithm: Raven Roosting Optimization (Parallel MPI)
Input: Same as serial + MPI context
Output: Best solution found
MPI_Init()
rank ← MPI_Comm_rank()
size ← MPI_Comm_size()
if rank == 0 then
    Parse command-line arguments
end if
MPI_Bcast(parameters, root=0)
local_rows ← pop_size / size
Initialize local population segment
Evaluate local fitness values
Set roosting_site (shared across all processes)
MPI_Barrier()
local_best ← argmin(local_fitness)
global_best ← MPI_Allreduce(local_best, op=MINLOC)
MPI_Bcast(leader_position, root=global_best.rank)
num_local_followers ← 0.2 × local_rows
local_followers ← random_select(num_local_followers)
for iter = 1 to iterations do
    for i = 1 to local_rows do
        current_position[i] ← roosting_site
        if is_follower[i] then
            target ← random_point_near(leader, r_pcpt)
        else
            target ← food_source[i]
        end if
        for step = 1 to flight_steps do
            direction ← normalize(target - current_position[i])
            remaining ← distance(current_position[i], target)
            step_size ← uniform(0, remaining)
            current_position[i] ← current_position[i] + step_size × direction
            enforce_bounds(current_position[i])
            for lookout = 1 to lookout_steps do
                candidate ← random_point_near(current_position[i], r_pcpt)
                enforce_bounds(candidate)
                if f(candidate) < fitness[i] then
                    food_source[i] ← candidate
                    fitness[i] ← f(candidate)
                end if
            end for
        end for
    end for
end for
```

Fig. 2. [illegible] MPI RRO [illegible] MAX, root=0) random_select(num_local_followers)

d) *Communication Patterns:*

Leader Selection:

Finding the global leader requires collective communication. Each process first finds its local minimum:

$$\text{local\_best}_p = \arg\min_{i \in \text{local\_ravens}_p} f(\boldsymbol{x}_i) \tag{7}$$

Then a global reduction determines the overall minimum across all processes, followed by broadcasting the leader position from the owning process to all others. This requires $O(\log\ k)$ communication time for k processes.

Parameter Broadcasting:

Configuration is broadcast from rank 0 to all processes. The global parameter structure is transmitted as a byte buffer, distributing algorithm configuration uniformly across the distributed system in $O(\log\ k)$ time.

Timing Reduction:

Maximum execution time across all processes is gathered using a reduction operation:

$$T_{\{\max\}} = \max_{p \in \{0,\ldots,k-1\}} T_p \tag{8}$$

This determines the wall-clock time as the slowest process duration.

e) *Synchronization Points:*

The parallel implementation uses several synchronization barriers:

- Before starting computation
- After initialization
- Before timing measurements

f) *Load Balancing:*

Population distribution aims for balanced workload. Ravens are divided as evenly as possible with each process handling

$$P_{\{\text{local}\}} = \left\lfloor \frac{P}{k} \right\rfloor \tag{9}$$

ravens (where P is total population and k is number of processes). Each process handles the same number of iterations, and computational work scales linearly with local population size.

g) *Scalability Considerations:*

Strong Scaling: For fixed problem size, speedup is limited by:

- Communication overhead (leader updates, broadcasts)
- Load imbalance if population doesn't divide evenly
- Synchronization barriers

Weak Scaling: Increasing problem size proportionally with processes maintains efficiency better, as communication-to-computation ratio remains favorable.

h) *Random Number Generation:*

Each process uses an independent random number generator seeded with rank-dependent values. The seed for process p is computed as

$$s_p = s_{\{\text{base}\}} + p \tag{10}$$

where $s_{\{\text{base}\}}$ is the initial seed. This ensures reproducibility while maintaining statistical independence across processes.

V. Optimizations

A. Serial Optimizations

a) *Early Stopping Mechanism:*

The serial implementation includes an early stopping heuristic that terminates the lookout phase when a better solution is found. With probability $p = 0.1$, the algorithm breaks out of both the lookout and flight loops upon finding an improvement. This optimization reduces unnecessary computation when good solutions are discovered early, potentially saving

$$(1 - p) \cdot L \cdot F \tag{11}$$

lookout operations per successful discovery, where L is lookout steps and F is flight steps.

b) *Efficient Memory Access Patterns:*

The implementation uses contiguous memory layouts for population data, enabling better cache locality, vectorization opportunities for modern compilers, and efficient memory operations. Population matrices are stored in row-major order with linear indexing:

$$\text{population}[i \cdot n + j] \tag{12}$$

for raven i and dimension j, where n is the number of features.

c) *Bounds Checking:*

A dedicated bounds checking function ensures solutions remain within feasible space without redundant checks. For each dimension j, positions are clamped:

$$x_j = \max\big(L_j, \min\big(x_j, U_j\big)\big) \tag{13}$$

where L_j and U_j are the lower and upper bounds respectively.

d) *Random Number Generation:*

Uses PCG (Permuted Congruential Generator) algorithm [2] which provides:

- High statistical quality
- Fast generation
- Small state size
- Good period length

B. Parallel-Specific Optimizations

a) *Minimizing Communication:*

The parallel implementation strategically places communication:

- **Once per iteration**: Leader selection and broadcast
- **Once at start**: Parameter broadcasting
- **Once at end**: Timing reduction

This design minimizes the frequency of collective operations, keeping most computation local.

b) *Asynchronous Overlapping Potential:*

While the current implementation uses blocking collectives, the structure allows future optimization through:

- Non-blocking leader broadcast (`MPI_Ibcast`)
- Overlapping computation of independent ravens with communication
- Pipelined iteration execution

c) *Local-First Processing:*

Each process computes its local leader before global reduction, minimizing data transfer:

- **Without local-first**: Transfer all fitness values (P doubles)
- **With local-first**: Transfer only local minimum (1 struct with rank and value)

d) *Reduced Early Stopping in Parallel:*

The parallel version conditionally disables early stopping when measuring speedup. When benchmarking mode is enabled, the probabilistic early termination is suppressed to ensure deterministic execution for fair performance comparisons across different process counts.

e) *Memory Optimizations:*

Reusable Buffers:

The implementation uses reusable buffers for temporary computations:

- `direction`, `n_candidate_position`, `prev_location`, `final_location`

These are allocated once per process and reused across all iterations, avoiding repeated allocation overhead.

Structure Packing:

MPI communication of configuration uses byte-level broadcasting, transmitting the entire structure as a contiguous byte array. This avoids overhead of creating custom MPI datatypes for simple structs.

C. Algorithmic Optimizations

a) *Follower Selection Strategy:*

Followers are selected randomly with 20% of population. The number of followers is computed as:

$$N_{\{\text{followers}\}} = \lceil 0.2 \cdot P - 1 \rceil \tag{14}$$

This balances exploration (non-followers search their own best) and exploitation (followers search near global leader).

b) *Adaptive Step Sizing:*

Step size is proportional to remaining distance. At each flight step t, the step size is:

$$s_t = d_{\{\text{remaining}\}} \cdot u \tag{15}$$

where $d_{\{\text{remaining}\}} = \|\boldsymbol{t}_i - \boldsymbol{x}_{i,t}\|$ is the distance to target and $u \sim \mathcal{U}(0, 1)$. This provides large steps when far from target (exploration) and small steps when near target (exploitation).

c) *Roosting Reset:*

Ravens return to roosting site at the start of each iteration. For all ravens i:

$$\boldsymbol{x}_{i,0} \leftarrow \boldsymbol{r} \tag{16}$$

where $\boldsymbol{r}$ is the roosting site. This reset mechanism prevents premature convergence, maintains diversity in search trajectories, and models natural raven behavior.

D. Performance Considerations

a) *Computational Bottlenecks:*

Profile analysis would likely show hotspots in:

1) **Objective function evaluation** (`lookout_steps` × `flight_steps` × `pop_size` calls per iteration)
2) **Distance calculations** (repeated square root operations)
3) **Random number generation** (multiple calls per step)

b) *Potential Further Optimizations:*
Future improvements could include:

- **Vectorization**: SIMD operations for distance calculations
- **GPU acceleration**: Parallel evaluation of objective function
- **Hybrid MPI+OpenMP**: Thread-level parallelism within each node
- **Adaptive parameters**: Dynamic adjustment of flight_steps and lookout_steps based on convergence rate
- **Communication aggregation**: Batch multiple MPI operations

VI. Conclusion

A. Summary

This paper presented a comprehensive study of the Raven Roosting Optimization algorithm, including both serial and parallel MPI implementations. The algorithm successfully models the foraging behavior of ravens through a metaheuristic approach that balances exploration and exploitation via leader-follower dynamics.

The serial implementation provides a clear baseline with optimizations such as early stopping and efficient memory access patterns. The parallel MPI version achieves scalability through domain decomposition, distributing the population across multiple processes while maintaining algorithmic correctness through collective communication for leader selection.

B. Key Contributions

- Detailed pseudocode for both serial and parallel implementations
- Analysis of parallelization strategy using MPI domain decomposition
- Identification of optimization techniques including early stopping, adaptive step sizing, and efficient memory management
- Discussion of communication patterns and synchronization requirements
- Performance considerations for scalability

C. Implementation Highlights

The implementations demonstrate several best practices:

- Clean separation between computation and communication logic
- Rank-specific random number seeding for reproducibility
- Configurable parameters for flexible experimentation
- Comprehensive timing instrumentation
- Proper memory management with cleanup routines

D. Scalability Analysis

The parallel implementation exhibits:

- **Good strong scaling potential** for large populations where computation dominates communication
- **Excellent weak scaling characteristics** as problem size grows proportionally with process count
- **Communication bottleneck** at leader selection phase, occurring once per iteration
- **Load balance** maintained through even population distribution

E. Future Work

Several avenues for enhancement remain:

a) *Hybrid Parallelism:* Combining MPI with OpenMP could exploit both distributed and shared memory parallelism, particularly beneficial on modern multi-core cluster nodes.

b) *Advanced Communication:* Non-blocking collectives and asynchronous leader updates could overlap communication with computation, potentially improving strong scaling efficiency.

c) *Adaptive Mechanisms:* Dynamic parameter adjustment based on convergence metrics could reduce unnecessary computation in later iterations when the population converges.

d) *Alternative Objective Functions:* The modular design supports easy integration of different objective functions, enabling application to diverse optimization problems.

e) *Performance Modeling:* Analytical performance models could predict scalability and guide optimal process count selection for given problem sizes.

F. Concluding Remarks

The Raven Roosting Optimization algorithm demonstrates the effectiveness of nature-inspired metaheuristics for complex optimization problems. The MPI parallelization successfully distributes computational workload while maintaining the algorithm's essential characteristics. Both implementations provide a solid foundation for further research and practical applications in high-performance computing environments.

The careful balance between local computation and global coordination exemplifies effective parallel algorithm design, achieving meaningful speedup while preserving solution quality. The documented code structure, complete with pseudocode and implementation details, facilitates reproducibility and extension by the research community.

References

[1] A. Brabazon, W. Cui, and M. O'Neill, "The Raven Roost Algorithm: A social foraging-inspired algorithm for optimisation," *Journal of Metaheuristic Algorithms*, vol. 1, pp. 1–20, 2020.

[2] X.-S. Yang, *Nature-Inspired Metaheuristic Algorithms*, 2nd ed. Luniver Press, 2010.

[3] J. Wright and others, "Communal Roosts as Structured Information Centres in Ravens," *Animal Behaviour*, 2003.

[4] M. Jamil and X.-S. Yang, "A Literature Survey of Benchmark Functions For Global Optimization Problems," *International Journal of Mathematical Modelling and Numerical Optimisation*, vol. 4, no. 2, pp. 150–194, 2013.

[5] E. Alba, "Parallel Metaheuristics: Recent Advances and New Trends," *International Transactions in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.