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. Terminology 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].

B. Movement of ravens

The movement of ravens in the algorithm is defined by the position update equation $\boldsymbol{p}_{i,t} = \boldsymbol{p}_{i,t-1} + \boldsymbol{d}_{i,t}$, where $\boldsymbol{d}_{i,t}$ represents the distance vector for raven i at iteration t. The distance vector $\boldsymbol{d}_{i,t}$ is constructed through two key steps in the implementation. First, the direction is determined by converting the previous direction vector (v) into a magnitud using norm:

$$\|\boldsymbol{v}\| = d(\boldsymbol{x}) = \sqrt{\sum_{i=1}^{n} (x_i)^2} \tag{2}$$

where v holds the n number of parameters (dimensions) in the search space. This is later used in the computation of a unit vector scaled using:

$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} = \frac{\boldsymbol{v}}{d(\boldsymbol{v}, \boldsymbol{0})} \tag{3}$$

This vector represents the exploration direction for the current iteration, influenced by the previous movement patterns. Second, the step size is calculated adaptively based on the remaining distance to the target location. The implementation computes the Euclidean distance between the current position and the best known location:

$$d(\boldsymbol{p}_i, \boldsymbol{p}_{\text{best}}) = \sqrt{\sum_{j=1}^{n} (p_{i,j} - p_{\text{best},j})^2} \tag{4}$$

then generates a random step size s_t as a fraction of this remaining distance using a uniform random value. This creates a Lévy flight-like behavior where ravens take larger steps when far from the target and smaller steps when approaching it. The actual displacement vector is therefore $\boldsymbol{d}_{i,t} = \text{direction\_unit\_vector} \times s_t$, resulting in an adaptive movement strategy that balances exploration through randomness with exploitation through proximity-based step sizing. The final position update follows $\boldsymbol{p}_{i,t} = \boldsymbol{p}_{i,t-1} + (\text{direction} \times s_t)$, where the direction is influenced by historical movement and the magnitude adapts dynamically to the distance from the target.

III. Serial Implementation

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. [illegible]

d) *Communication Patterns:*

Leader Selection:

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

$$\text{local\_best}_p = \arg\min_{i \in \text{local\_ravens}_p} f(\boldsymbol{x}_i) \tag{5}$$

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:

$$T_{\{\max\}} = \max_{p \in \{0,\ldots,k-1\}} T_p \tag{6}$$

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

$$P_{\{\text{local}\}} = \left\lfloor \frac{P}{k} \right\rfloor \tag{7}$$

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

$$s_p = s_{\{\text{base}\}} + p \tag{8}$$

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

V. Optimizations

VI. Conclusion

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.