

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 have 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 gather 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 $f(x)$ where $x \in \mathbb{R}^n$ 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 throughout 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

Throughout this paper the following notation is used:

- t represents the current iteration number and T is the maximum number of iterations;
- N is the total population of Ravens and F the number of features for each Raven;
- $X(t)$ is the position vector of ravens in iteration t ;
- $D(t)$ is direction where the raven is moving on $t + 1$;
- $L(t)$ Represent the leader, with the best location of source food (global);
- $FS(t)$ is the best local food source know to a raven. These will be replace by the current positions $X(t)$ if they yield better fitness values;
- $R(t)$ is the remaining distance to get the final destination represented as a scalar. Which can be $L(t)$ or $FS(t)$, see bellow.
- $U(a, b)$ denotes a random number drawn uniformly in the range $[a, b]$ based on PCG;
- UB and LB are the upper and lower bounds of the search space;
- r_i are random numbers drawn from $U(0, 1)$;
- R is the defined looking radius (in a hypersphere) from a specific position $X(t)$. Usually is equal to the upper bound UB ;

A. PCG Implementation

PCG (Permuted Congruential Generator) is a family of simple, fast, space-efficient, and statistically good random number generators developed by O'Neill [4]. Unlike traditional linear congruential generators (LCGs), PCG applies a permutation function to the output of an LCG to improve statistical quality while maintaining computational efficiency.

The basic PCG algorithm uses a linear congruential generator as its core:

$$s_{t+1} = a \times s_t + c \bmod m \quad (1)$$

where s_t is the internal state, a is the multiplier, c is the increment, and m is the modulus. The output is then passed through a permutation function that scrambles the bits to eliminate the statistical weaknesses inherent in LCGs. This permutation typically involves bit shifts, rotations, and XOR operations, which are computationally inexpensive but dramatically improve the randomness quality.

PCG generators offer several advantages: they pass stringent statistical test suites (including TestU01's BigCrush), have small state sizes, are extremely fast, and provide strong statistical properties suitable for scientific simulations and optimization algorithms. In this implementation, PCG is used to generate the uniform random numbers $U(a, b)$ that drive the stochastic behavior of the Raven Roost Algorithm.

B. 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:

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

where $x = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$ 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 $i = 1, \dots, n$ [5].
- **Global Minimum:** The unique global minimum occurs at $x^* = (0, 0, \dots, 0)$ with $f(x^*) = 0$.
- **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) [5].

C. Movement of ravens

The movement of ravens in the algorithm is defined by the position update equation $p_{i,t} = p_{i,t-1} + d_{i,t}$, where $d_{i,t}$ represents a random distance vector for raven i at iteration t . The distance vector $d_{i,t}$ is constructed through two key steps in the implementation. A direction, and a random amplifier:

$$d_{i,t} = s_{i,t} \times D(t) \quad (3)$$

To mimic the real behaviour, $D(t)$ is the direction of each bird to find the best food source. At the start of each t , each raven decides if he is going to pursue his best food source point or follow the leader and scout the area.

$$\vec{d} = \begin{cases} L(t) - X(t) & U(0, 1) \leq 0.2 \\ FS(t) - X(t) & U(0, 1) > 0.2 \end{cases} \quad (4)$$

First we calculate \vec{d} into a magnitude using norm:

$$\|v\| = \sqrt{\sum_{i=1}^n (x_i)^2} \quad (5)$$

where v is any given vector (\vec{d} in this case) that holds the n number of parameters (dimensions) in the search space. This is later used the computation of a unit vector using:

$$D(t) = \hat{v} = \frac{v}{\|v\|} \quad (6)$$

This vector represents the exploration direction for the current iteration, influenced by the previous movement patterns.

Second, we calculated a random modifier based on the remaining distance to the target location using **Euclidean distance** to get a scalar value between the final destination $FS(t)$ or $L(t)$ and the current position $X(t)$:

$$d(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^n (a_i - b_i)^2} \quad (7)$$

\mathbf{a} and \mathbf{b} represent any vector of n number of parameters (dimensions) in the search space. We multiply this distance to random step size r_i to get fraction of this remaining distance using a uniform distribution value, giving the following formula:

$$s_{i,t} = r_i \times R(T) \quad (8)$$

This creates a Lévy flight-like behavior, where ravens take larger steps when they are far from the target and smaller steps as they approach to the target. The actual displacement vector is therefore $\mathbf{d}_{i,t} = s_t \times D(t)$, resulting in an adaptive movement strategy that balances exploration through randomness with exploitation through proximity-based step sizing.

D. Lookout function

As mentioned earlier, raven's move a fraction of the total remaining distance governed by $\mathbf{d}_{i,t} = s_t \times D(t)$. This so that each point they stop they can scout the area with a Hypersphere vision across n dimensions. The looking radii is governed by:

$$r = \frac{R}{3.6 \times N^{\frac{1}{F}}} \quad (9)$$

When a raven finds a promising food source, he can decide to ignore it and continue or set it as it personal best and report it back to the rooster immediately.

$$\vec{d} = \begin{cases} \text{RETURN HOME} & U(0,1) \leq 0.1 \\ \text{CONTINUE} & U(0,1) > 0.1 \end{cases} \quad (10)$$

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** [6], 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:

```

1 Algorithm: Raven Roosting Optimization (Parallel MPI)
2 Input: Same as serial + MPI context
3 Output: Best solution found
4 MPI_Init()
5 rank ← MPI_Comm_rank()
6 size ← MPI_Comm_size()
7 if rank == 0 then
8   | Parse command-line arguments
9 end if
10 MPI_Bcast(parameters, root=0)
11 local_rows ← pop_size / size
12 Initialize local population segment
13 Evaluate local fitness values
14 Set roosting_site (shared across all processes)
15 MPI_Barrier()
16 local_best ← argmin(local_fitness)
17 global_best ← MPI_Allreduce(local_best, op=MINLOC)
18 MPI_Bcast(leader_position, root=global_best.rank)
19 num_local_followers ← 0.2 × local_rows
20 local_followers ← random_select(num_local_followers)
21 for iter = 1 to iterations do
22   for i = 1 to local_rows do
23     | current_position[i] ← roosting_site
24     if is_follower[i] then
25       | target ← random_point_near(leader, r_pcpt)
26     else
27       | target ← food_source[i]
28     end if
29     for step = 1 to flight_steps do
30       | direction ← normalize(target - current_position[i])
31       remaining ← distance(current_position[i], target)
32       step_size ← uniform(0, remaining)
33       current_position[i] ← current_position[i] + step_size × direction
34       enforce_bounds(current_position[i])
35       for lookout = 1 to lookout_steps do
36         | candidate ← random_point_near(current_position[i], r_pcpt)
37         enforce_bounds(candidate)
38         if f(candidate) < fitness[i] then
39           | food_source[i] ← candidate
40           | fitness[i] ← f(candidate)
41         end if
42       end for
43     end for
44   end for
45   num_local_followers ← random_select(num_local_followers)
46   root ← MPI_Allreduce(num_local_followers, op=MAX, root=0)

```

Fig. 1. Pseudocode for parallel MPI-BRO optimization. MPI_COMM_WORLD is the MPI context, MPI_MAX is the MPI datatype for integer, MPI_MINLOC is the MPI operation for minimum, MPI_BCAST is the MPI collective communication for broadcast, MPI_ALLREDUCE is the MPI collective communication for all-reduce, MPI_RANK is the MPI attribute for rank, MPI_SIZE is the MPI attribute for size, MPI_COMM_SIZE is the MPI function for getting the size of the MPI group, MPI_COMM_RANK is the MPI function for getting the rank of the process in the MPI group, MPI_INIT is the MPI function for initializing the MPI environment, MPI_FINALIZE is the MPI function for finalizing the MPI environment, MPI_BARRIER is the MPI function for barrier, MPI_ARGV is the MPI attribute for command-line arguments, MPI_ARGC is the MPI attribute for the number of command-line arguments, MPI_ARGV_0 is the MPI attribute for the first command-line argument, MPI_ARGV_1 is the MPI attribute for the second command-line argument, MPI_ARGV_2 is the MPI attribute for the third command-line argument, MPI_ARGV_3 is the MPI attribute for the fourth command-line argument, MPI_ARGV_4 is the MPI attribute for the fifth command-line argument, MPI_ARGV_5 is the MPI attribute for the sixth command-line argument, MPI_ARGV_6 is the MPI attribute for the seventh command-line argument, MPI_ARGV_7 is the MPI attribute for the eighth command-line argument, MPI_ARGV_8 is the MPI attribute for the ninth command-line argument, MPI_ARGV_9 is the MPI attribute for the tenth command-line argument, MPI_ARGV_10 is the MPI attribute for the eleventh command-line argument, MPI_ARGV_11 is the MPI attribute for the twelfth command-line argument, MPI_ARGV_12 is the MPI attribute for the thirteenth command-line argument, MPI_ARGV_13 is the MPI attribute for the fourteenth command-line argument, MPI_ARGV_14 is the MPI attribute for the fifteenth command-line argument, MPI_ARGV_15 is the MPI attribute for the sixteenth command-line argument, MPI_ARGV_16 is the MPI attribute for the seventeenth command-line argument, MPI_ARGV_17 is the MPI attribute for the eighteenth command-line argument, MPI_ARGV_18 is the MPI attribute for the nineteenth command-line argument, MPI_ARGV_19 is the MPI attribute for the twentieth command-line argument, MPI_ARGV_20 is the MPI attribute for the twenty-first command-line argument, MPI_ARGV_21 is the MPI attribute for the twenty-second command-line argument, MPI_ARGV_22 is the MPI attribute for the twenty-third command-line argument, MPI_ARGV_23 is the MPI attribute for the twenty-fourth command-line argument, MPI_ARGV_24 is the MPI attribute for the twenty-fifth command-line argument, MPI_ARGV_25 is the MPI attribute for the twenty-sixth command-line argument, MPI_ARGV_26 is the MPI attribute for the twenty-seventh command-line argument, MPI_ARGV_27 is the MPI attribute for the twenty-eighth command-line argument, MPI_ARGV_28 is the MPI attribute for the twenty-ninth command-line argument, MPI_ARGV_29 is the MPI attribute for the thirtieth command-line argument, MPI_ARGV_30 is the MPI attribute for the thirty-first command-line argument, MPI_ARGV_31 is the MPI attribute for the thirty-second command-line argument, MPI_ARGV_32 is the MPI attribute for the thirty-third command-line argument, MPI_ARGV_33 is the MPI attribute for the thirty-fourth command-line argument, MPI_ARGV_34 is the MPI attribute for the thirty-fifth command-line argument, MPI_ARGV_35 is the MPI attribute for the thirty-sixth command-line argument, MPI_ARGV_36 is the MPI attribute for the thirty-seventh command-line argument, MPI_ARGV_37 is the MPI attribute for the thirty-eighth command-line argument, MPI_ARGV_38 is the MPI attribute for the thirty-ninth command-line argument, MPI_ARGV_39 is the MPI attribute for the fortieth command-line argument, MPI_ARGV_40 is the MPI attribute for the forty-first command-line argument, MPI_ARGV_41 is the MPI attribute for the forty-second command-line argument, MPI_ARGV_42 is the MPI attribute for the forty-third command-line argument, MPI_ARGV_43 is the MPI attribute for the forty-fourth command-line argument, MPI_ARGV_44 is the MPI attribute for the forty-fifth command-line argument, MPI_ARGV_45 is the MPI attribute for the forty-sixth command-line argument, MPI_ARGV_46 is the MPI attribute for the forty-seventh command-line argument, MPI_ARGV_47 is the MPI attribute for the forty-eighth command-line argument, MPI_ARGV_48 is the MPI attribute for the forty-ninth command-line argument, MPI_ARGV_49 is the MPI attribute for the fiftieth command-line argument, MPI_ARGV_50 is the MPI attribute for the fifty-first command-line argument, MPI_ARGV_51 is the MPI attribute for the fifty-second command-line argument, MPI_ARGV_52 is the MPI attribute for the fifty-third command-line argument, MPI_ARGV_53 is the MPI attribute for the fifty-fourth command-line argument, MPI_ARGV_54 is the MPI attribute for the fifty-fifth command-line argument, MPI_ARGV_55 is the MPI attribute for the fifty-sixth command-line argument, MPI_ARGV_56 is the MPI attribute for the fifty-seventh command-line argument, MPI_ARGV_57 is the MPI attribute for the fifty-eighth command-line argument, MPI_ARGV_58 is the MPI attribute for the fifty-ninth command-line argument, MPI_ARGV_59 is the MPI attribute for the sixtieth command-line argument, MPI_ARGV_60 is the MPI attribute for the sixty-first command-line argument, MPI_ARGV_61 is the MPI attribute for the sixty-second command-line argument, MPI_ARGV_62 is the MPI attribute for the sixty-third command-line argument, MPI_ARGV_63 is the MPI attribute for the sixty-fourth command-line argument, MPI_ARGV_64 is the MPI attribute for the sixty-fifth command-line argument, MPI_ARGV_65 is the MPI attribute for the sixty-sixth command-line argument, MPI_ARGV_66 is the MPI attribute for the sixty-seventh command-line argument, MPI_ARGV_67 is the MPI attribute for the sixty-eighth command-line argument, MPI_ARGV_68 is the MPI attribute for the sixty-ninth command-line argument, MPI_ARGV_69 is the MPI attribute for the seventieth command-line argument, MPI_ARGV_70 is the MPI attribute for the seventy-first command-line argument, MPI_ARGV_71 is the MPI attribute for the seventy-second command-line argument, MPI_ARGV_72 is the MPI attribute for the seventy-third command-line argument, MPI_ARGV_73 is the MPI attribute for the seventy-fourth command-line argument, MPI_ARGV_74 is the MPI attribute for the seventy-fifth command-line argument, MPI_ARGV_75 is the MPI attribute for the seventy-sixth command-line argument, MPI_ARGV_76 is the MPI attribute for the seventy-seventh command-line argument, MPI_ARGV_77 is the MPI attribute for the seventy-eighth command-line argument, MPI_ARGV_78 is the MPI attribute for the seventy-ninth command-line argument, MPI_ARGV_79 is the MPI attribute for the eightieth command-line argument, MPI_ARGV_80 is the MPI attribute for the eighty-first command-line argument, MPI_ARGV_81 is the MPI attribute for the eighty-second command-line argument, MPI_ARGV_82 is the MPI attribute for the eighty-third command-line argument, MPI_ARGV_83 is the MPI attribute for the eighty-fourth command-line argument, MPI_ARGV_84 is the MPI attribute for the eighty-fifth command-line argument, MPI_ARGV_85 is the MPI attribute for the eighty-sixth command-line argument, MPI_ARGV_86 is the MPI attribute for the eighty-seventh command-line argument, MPI_ARGV_87 is the MPI attribute for the eighty-eighth command-line argument, MPI_ARGV_88 is the MPI attribute for the eighty-ninth command-line argument, MPI_ARGV_89 is the MPI attribute for the ninetieth command-line argument, MPI_ARGV_90 is the MPI attribute for the ninety-first command-line argument, MPI_ARGV_91 is the MPI attribute for the ninety-second command-line argument, MPI_ARGV_92 is the MPI attribute for the ninety-third command-line argument, MPI_ARGV_93 is the MPI attribute for the ninety-fourth command-line argument, MPI_ARGV_94 is the MPI attribute for the ninety-fifth command-line argument, MPI_ARGV_95 is the MPI attribute for the ninety-sixth command-line argument, MPI_ARGV_96 is the MPI attribute for the ninety-seventh command-line argument, MPI_ARGV_97 is the MPI attribute for the ninety-eighth command-line argument, MPI_ARGV_98 is the MPI attribute for the ninety-ninth command-line argument, MPI_ARGV_99 is the MPI attribute for the hundredth command-line argument.

d) *Communication Patterns:*

Leader Selection:

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

$$\text{local_best}_p = \arg \min_{i \in \text{local_ravens}_p} f(x_i) \quad (11)$$

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:

$$T_{\{\max\}} = \max_{p \in \{0, \dots, k-1\}} T_p \quad (12)$$

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

$$P_{\{\text{local}\}} = \left\lfloor \frac{P}{k} \right\rfloor \quad (13)$$

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

$$s_p = s_{\{\text{base}\}} + p \quad (14)$$

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. E. O'Neill, "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation," technical report HMC-CS-2014-0905, Sept. 2014. [Online]. Available: <https://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf>
- [5] 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.
- [6] E. Alba, "Parallel Metaheuristics: Recent Advances and New Trends," *International Transactions in Operational Research*, vol. 20, no. 1, pp. 1–48, 2013.