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 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 $f(\boldsymbol{x})$ 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

Throughout this paper the following notation is used:

- t represents the current iteration number and T is the maximum number of iterations;
- P 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;
- $RS(t)$ is the roosting site across all 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:

$$s_{t+1} = a \times s_t + c \bmod m \tag{1}$$

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:

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

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 $i = 1, ..., n$ [5].
- **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) [5].

C. 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 a random distance vector for raven i at iteration t. The distance vector $\boldsymbol{d}_{i,t}$ is constructed through two key steps in the implementation. A direction, and a random amplifier:

$$\boldsymbol{d}_{i,t} = \boldsymbol{s}_{i,t} \times D(t) \tag{3}$$

To mimic the real behavoir, $D(t)$ 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.

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

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

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

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:

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

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 $X(t)$:

$$d(\boldsymbol{a}, \boldsymbol{b}) = \sqrt{\sum_{i=1}^{n} (a-b)^2} \tag{7}$$

a and 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:

$$\boldsymbol{s}_{i,t} = r_i \times R(T) \tag{8}$$

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 $\boldsymbol{d}_{i,t} = s_t \times D(t)$, 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 $\boldsymbol{d}_{i,t} = s_t \times D(t)$. This so that each point they stop they can scout the area with a Hypersphere vision across n dimensions. The looking radii (r) is governed by:

$$r = \frac{R}{3.6 \times \sqrt[F]{P}} \tag{9}$$

When a raven finds a promising food source, he updates it and decides to continue or report it back to the rooster immediately.

$$\boldsymbol{p}_{i,t+1} = \begin{cases} p_{i,t} + d_{i,t} & U(0,1) \leq 0.1 \\ RS(t) & U(0,1) > 0.1 \end{cases} \tag{10}$$

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)
Initialize population with clamping
Evaluate best food source with Objective Function
Set roosting site and current leader
Define followers (excluding leader)
for t= 1 to iterations do
    for n = 1 to pop_size do
        Set it's current position to the roosting site
        Determine it's destination based on if is he a follower
        for step = 1 to flight steps do
            Move to p_{i,t+1} with clamping
            for lookout = 1 to lookout steps do
                Pick a random spot with clamping <= radii
                if finds a better food source then
                    Set it as personal best
                    if p < 0.1 then
                        Stop early and go back to the roosting site
                    end if
                end if
            end for
        end for
    end for
    Set current new leader
    Define followers (excluding leader)
end for
return current leader
```

Fig. 1. Pseudocode for serial RRO implementation

c) *Key Implementation Details:*

Leader Selection:

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

$$\text{leader} = \min(FS(t)) \tag{11}$$

d) *Time Complexity:*

For a iteration of size I, population of size P with F features. Each on taking F flight steps and L lookout steps:

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

Note: The early stopping mechanism (p < 0.1) can reduce the number of objective function evaluations by terminating lookout steps prematurely when a significantly better solution is found, potentially reducing evaluations by up to 10% in practice.

IV. Parallelization Strategy with MPI

The following extra terminilogy is introduced:

- **WR**: An MPI process responsible for managing a subset of the total population.
- **WS**: Number of total MPI processes.

Using parallelization we are able to achieve **domain decomposition** [6], where the problem is broken down into smaller tasks called ranks. Each **rank** (WR) is an MPI process, who is in charge of managing a subset of the total population. Ranks are list a list of numbers $\boldsymbol{x} \in [0, \infty]$.

Using this approach we are able to achieve faster convergence and exploration of the search space, since the program can run multiple calculations in parallel. In contrast the serial needs to iterate each calculation sequentially, which can be time-consuming for large populations. During the implementation of this algorithm, the same aspects where taken into as if the program was running sequentially. Each rank, recieves a set of global variables such as the **leader** ($L(t)$) that impact the algorithm in the same way as it was running sequentially.

a) *MPI Data Distribution:*

As previously mentioned, 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

Where each process is divided using:

$$FS_{\text{WR}}(t) = \frac{P}{\text{WS}} \tag{12}$$

$$\text{extra} = P \bmod \text{WS} \tag{13}$$

$$\text{number\_of\_rows} = \begin{cases} FS_{\text{WR}}(t) + 1 & \text{WR} < \text{extra} \\ FS_{\text{WR}}(t) & \text{WR} \geq \text{extra} \end{cases} \tag{14}$$

b) *Parallel Algorithm Pseudocode:*

The following algorithm in Fig. 2 shows the approach used to run the algorithm in a parallel manner using MPI. In order to facilitate reading, statments in this color symbolize that the process is shared across all WS.

The lifecycle of MPI is defined in it's initialization and closing arguments. Inside the algorithm it can be assumed multiple calls to MPI are made to **Reduce**, **Scatter** and **Broadcast** the results of the global variables.

Raven Roosting Optimization (Parallel MPI)

```
MPI Initialization
Initialize local population with clamping
Evaluate local best food source with Objective Function
Set roosting site and current leader
Define followers (excluding leader)
for iter = 1 to iterations do
  for i = 1 to local_rows do
    Set it's current position to the roosting site
    Determine it's destination based on if is he a follower
    for step = 1 to flight steps do
      Move to p_{i,t+1} with clamping
      for lookout = 1 to lookout steps do
        Pick a random spot with clamping <= radii
        if finds a better food source then
          Set it as personal best
          if p < 0.1 then
            Stop early and go back to the roosting site
          end if
        end if
      end for
    end for
  end for
  Set current global leader
  Define followers (excluding leader)
end for
return current leader
Close MPI
```

Fig. 2. Pseudocode for parallel MPI RRO implementation

c) *Key Implementation Details:*

Leader Selection:

Finding the global leader requires collective communication. Each process first evaluates the quality of it's solutions locally and determines its local best candidate by computing the minimum fitness value (Objective Function):

$$\text{local\_best}_p = \min(\text{local\_ravens}) \tag{15}$$

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 $O(\log \text{ WS})$ communication time for WS processes.

Timing Reduction:

To accurately measure the total execution time of the parallel algorithm, the maximum execution time across all participating processes is computed using a global reduction

operation. Each process records its own elapsed execution time T_p and the overall wall-clock time is then obtained as:

$$T_{\{\max\}} = \max_{p \in \{0,\ldots,\mathrm{WS}-1\}} T_p \tag{16}$$

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

d) *Scalability Considerations:*

Strong scalability and weak scalability describe how well a parallel algorithm performs as you change the number of processors, both with static or dynamic population sizes. This speedup is denotaed by Amdahl's Law [7] where:

$$S(p) = \texttt{frac}\{1\}\{, s + d\texttt{frac}\{1 - s\}\{p\}, \} \tag{17}$$

Where p denotes the number of parallel workers (processes, threads, or cores) that are employed to execute the program. The term s represents the proportion of the total execution time that is strictly serial—that is, the part of the code that cannot be divided among multiple processors and must run on a single core.

Strong Scaling:

Assumuing a static problem size (i.e., population size and dimensionality) but the number of processes increases, the achievable speedup is constrained by several algorithm-specific factors:

- Collective communication overhead becomes more pronounced, particularly during global leader selection and the subsequent broadcast of the leader's position vector.
- Load imbalance may arise when the population cannot be evenly partitioned across processes, resulting in some ranks owning fewer individuals and spending proportionally more time waiting at synchronization points.
- Global synchronization operations, such as barriers and collective reductions, introduce idle time that grows with the number of processes, further limiting strong scaling efficiency.

Weak Scaling:

Assuming a linear increment betwee the problem size and the number of processes can the computational workload should remain balanced as relative impact of communication overhead stays stable. This allows the algorithm to preserve a high level of parallel efficiency, as most operations are performed locally and collective communications scale logarithmically with the number of processes.

V. Parallelization Strategy with OpenMP

MPI allows distribute memory into different process (ranks) allowing it to execute lower volumes of data so then a master process can join it back to together. However, further breakdown can be achieved using OpenMP.

When an OpenMP instruction is specified using the **#pragma omp parallel** instruction. Iterations blocks or sections are broken down among team members according to a specified schedule. Each thread executes only the part of the loop that was mapped to it, so a single loop is "broken down" into as many independent chunks, this is partically usefull among independent members such as [8]:

Parallel OpenMP Implementation

```
1 int seeds = malloc(nthreads * sizeof(unsigned int));
2 #pragma omp for
3 for iter = 1 to iterations do
4     int tid = omp_get_thread_num();
5     seeds[tid] = time(NULL) ^ (tid * 0x9e3779b9);
6 }
```

Fig. 3. OpenMP example to initialize seeds

a) *OpenMP Data Distribution:*

OpenMP doesn't segment information as MPI. Instead of physically diving data as shown in MPI Data distribution it creates threas inside the same ejecution, by taking advantage of the of the number of threads inside each core. So for each instruction you parallelize under OpenMP, each thread is goign to executed as many "threads" define during the exeuction.

Each chunk is can be divided equally using:

$$\text{chunk\_size} = \min\left(\frac{\text{rows}}{\text{threads}}, 1\right) \tag{18}$$

However, we can do more complex scenarios can arrise. Using the **dynamic** scheduling we can provided smaller chunk sizes (ej: 5) and have the process call multiple times the queue to get continous segements of 5 chunks. This is partically useful when dealing with memory contrains as you can process less information in a given point in time.

VI. Optimizations

VII. 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.

[7] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *AFIPS Conference Proceedings*, 1967, pp. 483–485.

[8] S. Thibault, F. Broquedis, B. Goglin, R. Namyst, and P. Wacrenier, “An Efficient OpenMP Runtime System for Hierarchical Arch,” *arXiv preprint arXiv:0706.2073*, 2007.