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 social behavior mainly through communal roosting that functions as an information-sharing system about food. At night, juvenile, non-breeding ravens gather with 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 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;
- 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 $t+1$;
- $L(t)$ represents the leader, with the best location of the source food (global);
- $FS(t)$ is the best local food source known to a raven. These will be replaced by the current positions $X(t)$ if they yield better fitness values;
- $R(t)$ is the remaining distance to get to the final destination represented as a scalar. This can be $L(t)$ or $FS(t)$, see below.
- $U(a,b)$ 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 real behavior, $D(t)$ is the direction of each bird to find the best food source. At the start of each t, each raven decides if it is going to pursue its 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 magnitude 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 in 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 by a random step size r_i to get a 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 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, ravens move a fraction of the total remaining distance governed by $\boldsymbol{d}_{i,t} = s_t \times D(t)$. This is so that at each point they stop, they can scout the area with a hypersphere vision across n dimensions. The looking radius (r) is governed by:

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

When a raven finds a promising food source, it updates it and decides to continue or report it back to the roost 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 its destination based on if it is 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 an iteration of size I, population of size P with F features, each one 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 terminology 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, which is in charge of managing a subset of the total population. Ranks are 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 version needs to iterate each calculation sequentially, which can be time-consuming for large populations. During the implementation of this algorithm, the same aspects were taken into account as if the program was running sequentially. Each rank receives 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) *MPI 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, statements in this color symbolize that the process is shared across all WS.

The lifecycle of MPI is defined in its 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
Compute local indices
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 its current position to the roosting site
    Determine its destination based on if it is 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 local and global leader
  Define followers (excluding leader)
end for
return current leader
Close MPI
```

Fig. 2. Pseudocode for parallel MPI RRA implementation

c) *MPI Key Implementation Details:*

Leader Selection:

Finding the global leader requires collective communication. Each process first evaluates the quality of its 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 denoted by Amdahl's Law [7] where:

$$S(p) = \frac{1}{(1-p) + \frac{p}{N}} \tag{17}$$

Where p represents the the fraction of the program that is parallelizable. N denotes the number of parallel workers (processes, threads, or cores) that are employed to execute the program and $S(N)$ be the speedup with N processors.

Strong Scaling:

Assuming 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 between the problem size and the number of processes, the computational workload should remain balanced as the 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 distributing memory into different processes (ranks) allowing it to execute lower volumes of data so that a master process can join it back together. However, further breakdown can be achieved using OpenMP.

When an OpenMP instruction is specified using the **#pragma omp parallel** instruction, iteration 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 particularly useful 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 does. Instead of physically dividing data as shown in MPI Data distribution, it creates threads inside the same execution, by taking advantage of the number of threads inside each core. So for each instruction you parallelize under OpenMP, each thread is going to execute as many "threads" defined during the execution.

Each chunk can be divided equally using:

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

However, more complex scenarios can arise. Using the **dynamic** scheduling we can provide smaller chunk sizes (e.g., 5) and have the process call the queue multiple times to get continuous segments of 5 chunks. This is particularly useful when dealing with memory constraints as you can process less information at a given point in time.

b) *OpenMP Parallel Algorithm Pseudocode:*

OpenMP statements are usually encapsulated around **for** and block variables. Only 3 locations were picked to implement OpenMP.

1) Evaluating the Objective Function
2) Set roosting site
3) Compute local rows

These locations were picked because they are isolated processes that can be computed individually. So the implementation of the parallelization doesn't affect other members.

The following algorithm in Fig. 4 shows the approach used to run the algorithm in a parallel manner using OpenMP. In order to facilitate reading, statements in this color symbolize the key structures where OpenMP was implemented. MPI structures are still taken into account, but omitted for visibility purposes.

RAVEN ROOSTING OPTIMIZATION (PARALLEL OPENMP)

```
Initialize local population with clamping
Evaluate local best food source with Objective Function
Set roosting site and current leader
for iter = 1 to iterations do
  for i = 1 to local_rows do
    Set its current position to the roosting site
    Determine its destination based on if it is 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 local and global leader
  Define followers (excluding leader)
end for
return current leader
```

Fig. 4. Pseudocode for parallel OpenMP RRA implementation

c) *OpenMP Key Implementation Details:*

Instruction indications:

The decision to implement parallelization when computing local rows instead of parallelizing the entire iterations is due to the fact that **global** computations have to be done prior to updating the iteration.

When starting a new iteration, the previous global leader's position has to be known. This makes it particularly difficult since you have data dependencies between iterations. Instead, an alternative is parallelizing the computation of the current iteration to find the best value for the Objective Function. This process is an excellent candidate for parallelization since you have to await for all "ravens" to compute the value, but each raven can achieve it individually.

Timing Reduction:

Similar to MPI, OpenMP can measure given

$$T_{\text{OMP}(p)} = t_{\text{end}} - t_{\text{start}} \tag{19}$$

Compared to a specific "serial" block of code. However, similarly to MPI, the process is divided into execution processes (p), where each process computes the same work. This allows execution into multiple R rounds according to a chunk size C capacity. Each thread t executes a portion of the work during round r, with execution time $T_{p,r}$. The duration of a round is determined by the slowest participating thread:

$$T_r = \max_{t \in \{0,\ldots,p-1\}} T_{p,r} \tag{20}$$

So the total OpenMP execution time is the sum of all round durations:

$$T_{\text{OMP}(p,C)} = \sum_{r=0}^{R-1} \max_{t \in \{0,\ldots,p-1\}} T_{p,r} \tag{21}$$

d) *Scalability Considerations:*

This type of reduction is dependent on the total size of rows. Serial implementation can lead to better process timings given the overhead it takes to split and join threads.

VI. CONSIDERATIONS

A. Considerations

B. Dataset

In order to evaluate the speedup of the implementation, a controlled dataset was used to measure both a baseline and an implementation difference. Additionally, additional computation-intensive datasets were used to test the parallelization limits. Each dataset is defined by a dynamic number of rows (population size) and fixed number of columns (feature dimensionality), except for the controlled set. All the tests mentioned in the following pages are tested with the same dataset configurations while varying MPI ranks and OpenMP threads.

The following table shows the configurations used:

TABLE I
CONFIGURATION TABLE

Set ID	Rows	Columns	Controlled?
1	128	100	Yes
2	256	200	No
3	512	200	No
4	1024	200	No
5	2048	200	No

C. Efficient Memory Access Patterns

The implementation uses a contiguous memory layout to enable better cache locality and efficient memory operations. Population matrices are stored in row-major order with linear indexing:

$$\text{population}[i \cdot n + j] \tag{22}$$

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

D. Reducing overhead by minimizing communication

The parallel implementation is designed to minimize overhead by limiting communication to essential phases of the algorithm. By broadcasting parameters only during initialization and performing timing reductions at the end, it ensures minimum communication.

A "local-first" model is employed to reduce the volume of data transmitted across the network. Instead of transferring a complete set of fitness values from every process for global evaluation, each process first identifies its own local leader. This only requires a single reduction to communicate the result. This optimization reduces the data transfer and avoids local minima as the search space is more thoroughly examined compared to a global implementation where a single leader is followed.

E. Memory Considerations

The algorithm uses a list of matrices to store global and local information of ravens' whereabouts, food sources and fitness values. However, the total memory consumption is dominated by the two matrices, as they are dependent on their population and features. The following table shows the composition size:

Data Structure	Size (Bytes)	Algorithmic Role
Population Matrices (Global State)		
food_source	pop_size × features × 8	Primary coordinate with best food location per raven
current_position	pop_size × features × 8	Raven's location after each iteration.
Global Vectors (Shared Reference)		
leader	features × 8	Best raven position.
roosting_site	features × 8	Convergence point where ravens gather.
fitness	pop_size × 8	The objective scores to each raven.
Auxiliary Search Vectors (Scratchpad)		
prev_location	features × 8	Sub-step location for each raven.
final_location	features × 8	Final destionation they want to reach
n_candidate_ position	features × 8	Step size direction correction.
direction	features × 8	Unit vector for the next iterative step.

Each section has a specific role when parallelizing. **Global Vectors (Shared Reference)** are shared across MPI; instead of having `total_population_sizes` allocated, the array collapses into smaller `local_population_sizes` where only the dominant leader is shared. This means that the communication never exceeds features × 8 where 8 is the bytes required to send a double number.

In contrast, OpenMP has to allocate separate **Auxiliary Search Vectors (Scratchpad)** as these memory references are not thread-safe. This yields a higher memory increase since you are allocating p arrays instead of a single reusable instance.

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.

VII. Performance Analysis

A. Experimental Setup

To assess the scalability of the parallel Raven Roosting Algorithm implementations, a series of experiments were conducted on the HPC cluster.

B. SpeedUp

The first implementation sought to assess the speedup using a single dataset and multiple CPU counts (1, 2, 4, 16, 32, and 64) and thread counts (1, 2, 4). A unified RAM configuration (4GB) was used and two different placement strategies **pack:excl** and **scatter:excl**.

> **Note**
> The pack:excl consolidates processes onto the minimum number of nodes possible, whereas scatter:excl distributes chunks across separate nodes. Both strategies ensure exclusive node access during execution.

For this implementation, conditions such as early stop were removed to maintain data consistency.

a) *SpeedUp Evolution:*

A total of 5 experiments were executed in this phase across three different trials to ensure reproducibility. To fully take advantage of and understand how each module contributes to the speedup, the following phases were tested:

1) Serial Implementation used for baseline (Serial Implementation)
2) Initial implementation of MPI, basic data handling (Trial 1)
3) Parallelize CSV ingestion and boundary logic and added MPI structs for data handling. Introduced a global, local struct division (Trial 2)
4) Final full MPI implementation (Trial 3)
5) OpenMP implementation (Trial 4)

b) *SpeedUp Results and Discussion:*

The following tables demonstrate the results for each configuration (not all configurations are available) across each phase.

TABLE II
AVERAGE TOTAL IN SERIAL IMPLEMENTATION

NP	Pack *(Total Time)*	Scatter *(Total Time)*
1	195.00	195.33
2	192.00	195.00
4	191.33	196.33
16	193.00	210.00
32	210.33	210.67

In this implementation, the total time remains relatively flat (around 191–196 seconds) regardless of the number of processes. This coincides with the implementation, since the process doesn't attempt to split up anything.

TABLE III
AVERAGE TOTAL AND COMPUTATION TIMES IN TRIAL 0

NP	Pack		Scatter	
	Total	**Comp**	**Total**	**Comp**
1	235.72	235.70	213.17	213.16
2	214.21	214.19	179.50	179.48
4	214.61	214.59	190.69	190.67
16	242.63	242.62	214.25	214.24
32	245.40	245.39	228.14	228.12
64	219.50	219.48	259.98	259.97

Results here represent worse timing than the baseline, since total times are consistently higher than the baseline (e.g., 235s vs 195s at np=1).

This is expected because of the added overhead. Interestingly, the computation_time is almost identical to total_time; this suggests that the setup/teardown of the MPI environment is adding a fixed cost to the parallel logic.

TABLE IV
AVERAGE TOTAL AND COMPUTATION TIMES IN TRIAL 1

NP	Pack		Scatter	
	Total	**Comp**	**Total**	**Comp**
1	240.53	240.52	214.07	214.04
2	214.73	214.71	217.26	217.25
4	200.80	200.77	223.41	223.41
16	211.60	211.59	235.09	235.06
32	246.44	246.43	262.96	262.92
64	244.29	244.36	261.30	261.27

This trial is similar to Trial 0 but begins to highlight the impact of parallelization. Here there is high variability between pack and scatter. At np=32, pack is faster (246s) than scatter (262s); these results are expected since pack reduces the communication overhead by placing nodes together.

Within this implementation, the algorithm is still bound by a serial bottleneck, since it is able to initialize in a distributed way but there is still a serial implementation when processing each row.

TABLE V
AVERAGE TOTAL AND COMPUTATION TIMES IN TRIAL 2

NP	Pack		Scatter	
	Total	**Comp**	**Total**	**Comp**
1	236.98	236.95	181.71	181.69
2	99.47	99.44	96.38	96.36
4	47.09	47.07	46.37	46.34
16	13.17	13.15	13.63	13.61
32	6.79	6.77	7.25	7.23
64	3.24	3.19	4.12	4.09

With this implementation there is evident **Strong Scaling**. The execution time tends to drop linearly as np increases. When doubling the processes from np=2 (99s) to np=4 (47s), results show almost a perfect 2x speedup, proving that strong scaling exists.

With this implementation, the algorithm is 60x faster than the baseline (np=64).

TABLE VI
AVERAGE TOTAL AND COMPUTATION TIMES IN TRIAL 3

NP	Threads	Pack		Scatter	
		Total	Comp	Total	Comp
1	1	253.76	253.71	215.73	215.72
	2	208.00	207.91	170.76	170.75
	4	188.53	188.46	139.80	139.79
2	1	113.98	113.93	108.75	108.73
	2	99.29	99.23	95.72	95.71
	4	101.32	101.27	106.97	106.93
4	1	58.09	58.04	59.66	59.66
	2	40.03	40.00	36.75	36.74
	4	28.49	28.45	26.51	26.49
16	1	16.82	16.74	16.29	16.28
	2	15.62	15.54	14.82	14.79
	4	11.59	11.47	20.67	20.64
32	1	7.48	7.36	9.19	9.16
	2	6.33	6.25	5.86	5.78
	4	15.48	15.44	35.44	35.36
64	1	4.52	4.34	5.12	5.04
	2	22.74	22.66	7.56	7.50
	4	84.55	84.16	4.64	4.22

Results with OpenMP are scattered. At a lower np count, adding threads is highly effective, as denoted in np=1, where 4 threads reduce the time from 253s to 188s. This tends to be the trend until np=64.

When this state is reached, we start to see an inflection point where negative scaling starts to occur using multiple threads, especially in a pack configuration (84s with 4 threads vs 4.5s with 1 thread).

c) *Further Analysis:*

Even though evidence exists that by implementing parallelism regardless of the implementation (MPI or OpenMP) the results compared to the baseline are always better, as suggested in the literature, Strong Scaling Saturation [9] can occur in instances where the sub-problems become so small that thread management and cache contention dominate the runtime. This can be seen better using the following graph.

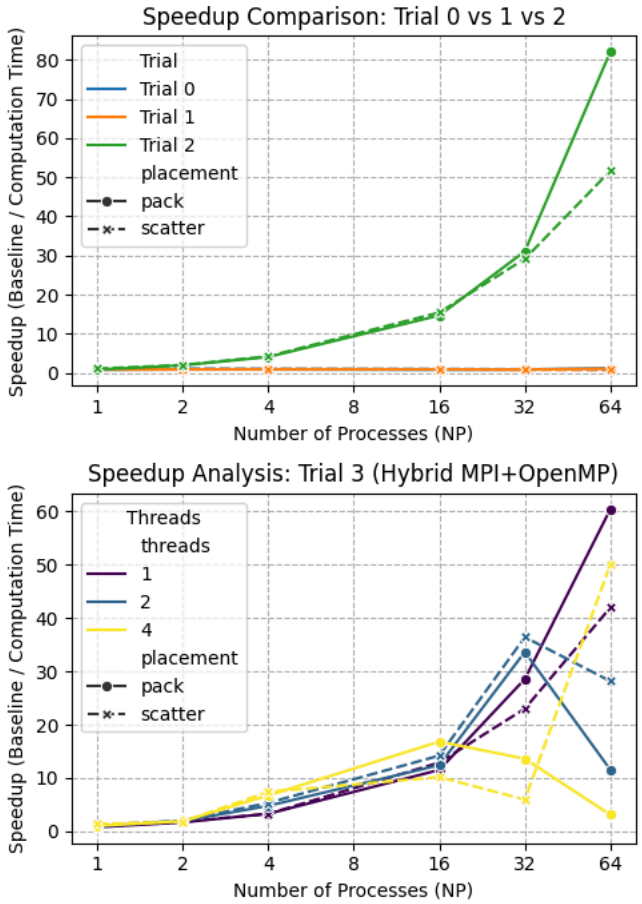

Fig. 5. SpeedUp tendency

As mentioned above, strong scaling occurs as the runtime falls as the number of compute resources (cores, MPI ranks, threads…) increases; however, this is only true when $\text{np} = 1 \rightarrow 4, \text{threads} \leq 4$, since each iteration has a 30% reduction when doubling cores. This is consistent except for np=2, threads=4, where saturation starts to show as we see greater results compared with the 2-thread case, most probably a bad run due to some network constraint. The behavior then deviates back to $\text{np} = 16 \rightarrow 32, \text{threads} \leq 2$, where speedup still occurs even at a more inclined rate, but only lasts for 1 iteration of threads since the tendency continues to grow at 1 thread, which indicates OpenMP is no longer in use.

These results showcase the limits of the execution. For the implementation of the Raven Roosting Algorithm, the best configuration can be achieved with either 64 cores at 4 threads (further analysis has to be done to account for any system flukes) or with 32 CPUs and 2 threads. There we can see a tendency that as the number of CPUs and threads increase, so does the speedup. This is a consistent indicator of Strong Scaling because the time tends to go down instead of remaining flat, showing the code can successfully decompose a fixed problem.

C. Parallelization Implementation

Now that we have proven that the code is able to achieve Strong Scaling, the second implementation assesses parallelization prioritizing a more diverse dataset (256, 512, 1024 and 2048) and a simpler configuration of CPUs (16, 32, 64, 128) and thread counts (2, 4, 6). Conditions such as memory or placement location were kept the same.

While measuring conditions can be different since early stopping mechanisms are in place, the purpose of the experiment is to demonstrate weak scalability as well as strong scalability.

a) *Parallelization Implementation Evolution:*

Two implementations were tested: one with the full MPI product and the other with a full OpenMP implementation. With this implementation we prioritize convergence results; at a greater number of processes we expect to see a much larger convergence rate.

b) *Parallelization Implementation Results and Discussion:*

The following convergence data denotes the time it takes to converge against the total dataset size.

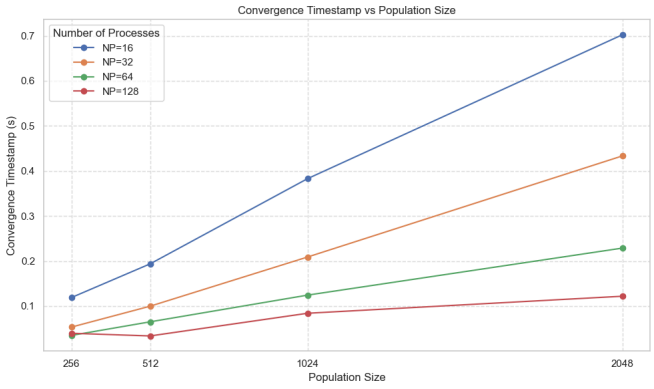

Fig. 6. Convergence of MPI processes

As expected, the greater the amount of data, the longer it takes to converge; however, the NP size does affect the time it takes to converge. We can still see reductions of almost 0.6 seconds between np=16 and np=128, confirming again Strong Scaling.

However, if we analyze the implementation of OpenMP Fig. 7, we can see a different result. When using $\text{thread} = 2$, 128 threads take a while to converge; this behavior seems to be constant regardless of the total number of threads. This means that at np=128 we start to see saturation.

Interestingly enough, convergence times are much larger when implementing threading. In the pure MPI implementation, the slowest convergence peaked at 0.7 seconds while the slowest on the OpenMP is at 26 minutes. This indicates a massive overhead in the OpenMP implementation, likely due to thread synchronization, memory locking, or "false sharing"; however, it's hard to tell since the experiment conditions are not the same.

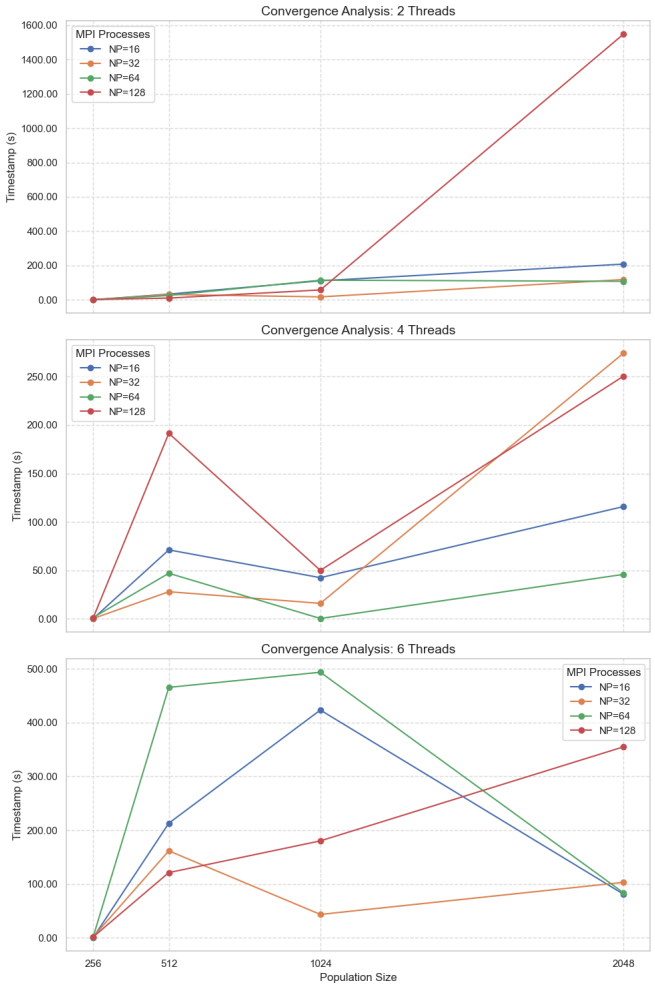

Fig. 7. Convergence of OpenMP processes

In this configuration, the most stable graph is shown to be "2 threads", showing a generally expected trend where larger populations take longer to converge. In all the configurations, NP=32 demonstrates to be consistent, yielding the most reliable results as it remains the lowest when $\text{thread} = 2$ and low on $\text{np} = 16, 64$, suggesting that it is the better balance when combining both technologies.

c) *Further Analysis:*

While the pure MPI implementation demonstrates weak scaling, maintaining constant execution times of 0.12s as population size and process count increase, the OpenMP implementation suffers from severe performance degradation, with execution times jumping between orders of magnitude. A possible analysis could be that the solution is bottlenecked by thread management overhead that outweighs the parallel benefit, making pure MPI a better alternative.

VIII. Conclusion

A. Concluding Remarks

The Raven Roosting Optimization algorithm demonstrates a good tendency to be a strong candidate for parallelization.

MPI parallelization yielded expected results based on the computational workload. However, OpenMP research could be characterized as unconclusive, as some results had erratic behavior that could be an invitation for further investigation.

However, the purpose of the project was achieved—I successfully implemented an algorithm and observed how parallel computing in an HPC scenario does have drastic effects on execution times. Further analysis of this project can be done by implementing alternative pieces of code such as different objective functions or parallelization techniques such as Island or Master-Slave configurations that could affect the results of the experiment.

This project proved to be a nice introduction to the topic and I would personally like to keep researching other techniques and implementations.

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.

[9] M. Lange, G. Gorman, M. Weiland, and L. Mitchell, "Achieving Efficient Strong Scaling with PETSc using Hybrid MPI/OpenMP Optimisation," *arXiv preprint arXiv:1303.5275*, 2013, [Online]. Available: https://arxiv.org/abs/1303.5275