UNIVERSITÀ
DI TRENTO

Parallel Bat Algorithm

A Sequential and Parallel Implementation

Wail Ameur, Bénie Nshimirimana

January 14, 2026

eit Digital

1 Introduction

This project is carried out within the framework of the *High-Performance Computing for Data Science* course at the University of Trento and the EIT Digital program. The objective of this work is to study a representative optimization algorithm, to develop a correct and efficient sequential implementation, and to explore its adaptation to parallel executions on High-Performance Computing (HPC) systems. The project emphasizes both algorithmic understanding and practical parallel programming using widely adopted HPC technologies.

High-Performance Computing (HPC) plays a central role in the analysis and optimization of complex systems, particularly when large-scale computations or data-intensive workflows are involved. Many real-world problems in domains such as climate modeling, engineering design, finance, and medical data analysis can be formulated as continuous optimization problems, where the goal is to identify optimal parameter configurations under limited computational resources.

In practice, continuous optimization problems are often non-linear, non-convex, or noisy, which makes classical gradient-based optimization methods inefficient or inapplicable. As discussed by Nocedal and Wright in their reference book on numerical optimization [2], the structure of the objective function can significantly complicate the resolution process. Several studies highlight that gradient-based approaches become limited when the objective function is **non-differentiable**, **noisy**, or **non-convex**, or when the **dimensionality of the problem increases**.

For instance, Rios and Sahinidis explain that the absence, inaccuracy, or inaccessibility of derivatives renders classical optimization techniques *"of little or no use"*, and they empirically show that *"the ability [...] to obtain good solutions diminishes with increasing problem size"* [4]. These observations motivate the use of stochastic and derivative-free metaheuristic approaches.

This project focuses on the Bat Algorithm, a bio-inspired metaheuristic introduced by Xin-She Yang [6]. The algorithm models the echolocation behavior of bats and relies on adaptive parameters such as frequency, loudness, and pulse emission rate to balance global exploration and local exploitation of the search space. Due to its population-based nature, the Bat Algorithm exhibits a high degree of inherent parallelism, making it a suitable candidate for execution on modern HPC architectures.

The objectives of this work are threefold. First, a sequential implementation of the Bat Algorithm is developed in C and used as a reference baseline. Second, a distributed-memory parallel version based on the Message Passing Interface (MPI) is implemented and evaluated on an HPC cluster. Third, a shared-memory implementation using OpenMP is proposed to exploit multi-core architectures.

The performance of these implementations is evaluated through benchmarking experiments focusing on execution time, speedup, and parallel efficiency. Through this study, we aim to illustrate the practical challenges associated with the parallelization of metaheuristic optimization algorithms and to highlight the complementary roles of MPI and OpenMP in high-performance computing environments.

2 Software Availability

The complete source code for this project, including the sequential implementation and the parallel versions (MPI and OpenMP), is hosted on GitHub. This repository contains the source files, build instructions, and scripts used to generate the benchmarking results presented in this report:

`https://github.com/melBzz/parallel-bat-algorithm`

3 Bat Algorithm

3.1 Biological Inspiration and Algorithmic Principles

The Bat Algorithm is a bio-inspired metaheuristic based on a simplified abstraction of the echolocation behavior observed in bats. In nature, bats emit ultrasonic pulses whose frequency, intensity, and emission rate vary according to the proximity of a target. Yang translates these biological mechanisms into algorithmic components: each bat is represented by a position in the search space, an associated velocity, an exploration frequency f_i, a loudness parameter A_i controlling solution acceptance, and a pulse emission rate r_i governing the balance between exploration and exploitation. These parameters are updated stochastically over iterations to mimic how a colony progressively converges toward an optimal region. The sequential algorithm therefore alternates between candidate generation, objective function evaluation, and adaptive parameter updates in order to guide the search toward the best solution found so far [6].

From a biological perspective, microbats rely on echolocation to detect prey, avoid obstacles, and navigate in darkness. They emit short ultrasonic pulses, typically lasting a few milliseconds (up to 8–10 ms), with frequencies ranging from 25 kHz to 150 kHz, well beyond the human auditory range (approximately 20 Hz to 20 kHz). During the search phase, microbats emit about 10 to 20 pulses per second, while this rate can increase to nearly 200 pulses per second as they approach a prey. Such rapid emission is enabled by their auditory system, which can analyze echoes within 300 to 400 microseconds.

The intensity of emitted pulses can reach up to 110 dB, comparable to the sound level of a jackhammer, but is adaptively regulated. When far from a target, microbats emit strong signals to maximize detection range; as they get closer, they progressively reduce intensity to avoid auditory saturation and improve localization accuracy. This adaptive modulation of frequency, emission rate, and loudness constitutes the biological inspiration for the dynamic parameter control mechanisms employed in the Bat Algorithm.

3.2 Echo Analysis: Distance, Direction, and Target Characteristics

From the reflected echo, microbats are able to extract multiple types of information. First, they estimate the distance to a target by measuring the time elapsed between signal emission and reception, according to

$$distance = v \cdot \frac{t}{2},$$

where v denotes the speed of sound and t the round-trip travel time of the wave. Direction is inferred from the very small difference in arrival time of the sound between the two ears, while variations in echo intensity allow microbats to infer properties such as target size, shape, or texture. In addition, frequency shifts caused by the Doppler effect enable the detection of motion, with higher frequencies indicating an approaching target and lower frequencies a receding one. By combining these cues, microbats construct a three-dimensional representation of their environment and are able to avoid extremely thin obstacles, sometimes on the order of a millimeter.

The Bat Algorithm relies on a simplified abstraction of this biological behavior. Xin-She Yang formalizes it through three main assumptions in order to derive a mathematically exploitable optimization model [6]. First, each virtual bat is equipped with a mechanism to evaluate the quality of a position, analogous to prey detection through echo analysis, which is implemented via the objective function. Second, bats adjust their frequency, velocity, and pulse emission rate according to their proximity to promising solutions. Third, loudness decreases while the pulse rate increases over iterations, modeling the transition from global exploration to local exploitation. In this model, each bat is characterized by a position x_i, a velocity v_i, a frequency f_i, a loudness A_i, and a pulse emission rate r_i.

3.3 Mathematical Formulation

The update equations of the Bat Algorithm are defined as follows. The frequency is updated according to

$$f_i = f_{\min} + (f_{\max} - f_{\min})\,\beta,$$

where $\beta \in [0, 1]$ is a uniformly distributed random variable. The velocity is then updated based on the current position of the bat relative to the best solution found so far x_*,

$$v_i^{t+1} = v_i^t + (x_i^t - x_*)\,f_i,$$

and the new position is obtained by

$$x_i^{t+1} = x_i^t + v_i^{t+1}.$$

These three equations define the *global update* mechanism responsible for exploration of the search space.

For local exploitation, Yang introduces a random walk controlled by the average loudness $A^{(t)}$,

$$x_{new} = x_{old} + \varepsilon\, A^{(t)}, \qquad \varepsilon \in [-1, 1].$$

From an implementation perspective, a scaling parameter is introduced to control the step size, leading to

$$x_{new} = x_{old} + \sigma\, \varepsilon_t\, A^{(t)},$$

where $\varepsilon_t \sim \mathcal{N}(0,1)$ and $\sigma = 0.1$ in the example provided by Yang. This step constitutes the *local search* phase, which explores the neighborhood of promising solutions.

Finally, the adaptive dynamics of loudness and pulse rate are modeled by

$$A_i^{t+1} = \alpha A_i^t, \qquad r_i^{t+1} = r_0\big(1 - e^{-\gamma t}\big),$$

with $0 < \alpha < 1$ and $\gamma > 0$. Loudness therefore decreases over time, while the pulse rate gradually approaches r_0. The updates of f_i and v_i ensure global exploration by controlling the amplitude and direction of movements, whereas the local random walk enables exploitation around the best solutions identified so far. The joint evolution of A_i and r_i governs the transition between exploration and exploitation. v

3.4 Algorithm Description

3.4.1 Pseudo-code: Bat Algorithm

Data: Objective function $f(x)$
Result: Best or optimal solution

```
1. Initialize the bat population $x_i$ and velocities $v_i$, $i = 1, \dots, n$;
2. Initialize frequencies $f_i$, pulse rates $r_i$, and loudness values $A_i$;
3. while $t <$ maximum number of iterations do
    (a) Generate new solutions by adjusting frequency;
    (b) Update velocities and positions using the update equations;
    (c) if $rand > r_i$ then
        • select a solution among the current best solutions;
        • generate a local solution around the selected best solution;
    (d) generate a new solution by flying randomly;
    (e) if $rand < A_i$ and $f(x_i) < f(x_*)$ then
        • accept the new solution;
        • increase $r_i$ and reduce $A_i$;
    (f) rank the bats and update the current best solution $x_*$;
4. end while
```

The algorithm starts by initializing a population of n bats, each defined by a position x_i, a velocity v_i, and three internal parameters: frequency f_i, pulse rate r_i, and loudness A_i. At each iteration, bounded by the maximum number of iterations, new candidate solutions are generated by adjusting the frequency, which in turn updates velocity and position.

At this stage, two candidate solutions are generated: a local solution constructed around one of the current best solutions (if the condition $rand > r_i$ is satisfied, where $rand$ denotes a uniformly distributed random number in $[0, 1]$), and a global solution obtained by random flight. Both candidates are evaluated, and the best one is retained for the acceptance test based on the loudness A_i.

Thus, at each iteration, at most one final solution is accepted to update the bat position x_i. In the case of improvement, the parameters A_i and r_i are updated, and the global best solution x_* is revised accordingly.

Figure 1 illustrates the main stages of the Bat Algorithm, from population initialization to the iterative update of the global best solution (adapted from [7]).

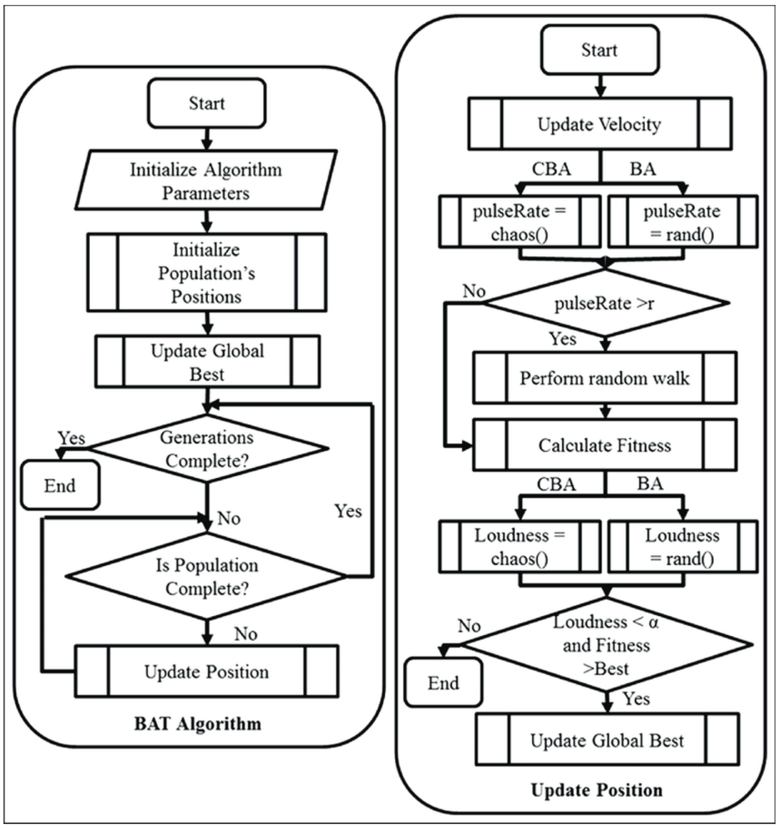


Figure 1: Conceptual flowchart of the Bat Algorithm.

3.5 Conceptual Limitations of the Frequency f_i and Pulse Rate r_i

A careful analysis of Yang's formulation reveals ambiguities in the interpretation and practical role of the frequency f_i and the pulse rate r_i. Although f_i is motivated by the physical relation $\lambda_i f_i = v_{sound}$, its algorithmic effect contradicts the biological analogy: in the Bat Algorithm, higher frequency values induce larger displacement amplitudes, corresponding to global exploration rather than fine-grained local search. Moreover, the influence of f_i in the update equations could be replaced by an arbitrary random scaling factor, suggesting that its inclusion is driven more by metaphorical consistency than by algorithmic necessity.

A similar inconsistency concerns the pulse rate r_i. While r_i increases monotonically over time in Yang's formulation, the condition `if rand >` r_i used to trigger local search produces an inverted operational behavior, with frequent local random walks at early stages and progressively rarer local refinements near convergence. This discrepancy originates from the triggering condition itself rather than from the dynamics of r_i, and a condition of the form `if rand <` r_i would be more consistent with the biological analogy, although it is not adopted in Yang's official formulations.

These conceptual issues motivated a closer examination of the velocity update equation,

$$v_i^t = v_i^{t-1} + (x_i^t - x^*)\, f_i,$$

which has been identified as problematic in the literature [1]. First, the use of x_i^t to compute v_i^t introduces a circular dependency, since the position itself depends on the velocity. Second, the term $(x_i^t - x^*)$ directs the update away from the current best solution, contradicting the principle of convergence toward an optimum. Interestingly, Yang's MATLAB implementation published in *Nature-Inspired Optimization Algorithms* (2014) computes the frequency as

$$f_i = f_{\min} + (f_{\min} - f_{\max}) \cdot rand,$$

which reverses the sign of f_i when $f_{\min} = 0$. When combined with the original velocity update,

$$v_i^t = v_i^{t-1} + (x_{current} - x_{best})\, f_i,$$

this sign inversion effectively yields

$$v_i^t = v_i^{t-1} + (x_{best} - x_{current})\, f_i,$$

thereby restoring a motion directed toward the best solution. This compensation between two inconsistencies makes it difficult to determine whether the resulting behavior is intentional or accidental, and highlights the lack of a clearly stabilized formulation across Yang's papers and books. These ambiguities complicate both the theoretical interpretation of the Bat Algorithm and the justification of implementation choices.

4 Sequential Implementation

4.1 Inspirations & Design Choices

To implement the sequential version of the Bat Algorithm, we relied on several sources, in particular the MATLAB implementation provided by Xin-She Yang in *Nature-Inspired Optimization Algorithms* (Elsevier, 2014). This implementation constitutes a practical reference, as it not only clarifies the overall structure of the algorithm, but also the order of operations and the concrete use of parameters such as frequency updates, loudness management, and acceptance conditions.

We adopted several elements from this implementation, notably:

- the overall structure of the main loop (frequency update $\rightarrow$ velocity update $\rightarrow$ position update $\rightarrow$ optional local search),
- the use of a population array storing, for each bat, its position, velocity, frequency, loudness, and pulse rate,
- the presence of a local random walk triggered by a stochastic condition,
- and the progressive update of the parameters A_i and r_i according to the formulas proposed by Yang.

However, we also introduced several departures from the MATLAB code, either to remain closer to the pseudo-code presented in the original paper, or to correct conceptual inconsistencies identified during our theoretical analysis.

4.2 Correction of Frequency and Velocity Equations

As explained in the previous section, the MATLAB code uses the inverted expression

$$f = f_{\min} + (f_{\min} - f_{\max}) \cdot rand,$$

which changes the sign of the frequency. This inversion unintentionally compensates for an error in the velocity update by restoring the correct direction of motion.

In our implementation, we chose to explicitly correct these inconsistencies:

- the frequency is computed according to the intended formula given in the original paper,
- the velocity update is expressed in a form consistent with attraction toward the best solution:

$$v_i^t = v_i^{t-1} + (x^* - x_i^{t-1})\, f_i.$$

We did not modify the condition used to trigger the local search:

`if rand >` r_i.

Although several analyses in the secondary literature (including technical discussions online) point out that this condition is conceptually counter-intuitive, Yang consistently uses this formulation in his original article, his books, and his MATLAB implementation. We did not find any official version in which this condition is inverted.

In our implementation, this choice remains coherent with our parameterization, as we initialize $r_0 = 1$, which causes r_i to decrease over iterations. Under this configuration, the condition `rand >` r_i implies:

- at the beginning of the search: large $r_i \rightarrow$ local search is rare,
- at later stages: smaller $r_i \rightarrow$ local search becomes more frequent.

This behavior, although opposite to the biological interpretation where the pulse rate increases near the prey, remains compatible with a metaheuristic logic consisting of initial exploration followed by progressive exploitation.

4.3 Generation of Candidate Solutions

The pseudo-code presented in Yang's original article suggests three distinct mechanisms for generating new solutions:

- a global update (frequency, velocity, and position),
- a conditional local search,
- an additional random flight ("flying randomly").

However, this structure does not appear in the MATLAB implementation provided by Yang in *Nature-Inspired Optimization Algorithms* (2014). In that code, only two potentially distinct solutions are generated for a given bat during an iteration:

- a global solution obtained after updating frequency, velocity, and position:

$$S_i = x_i + v_i,$$

- a local solution, which replaces the global one only if the condition `rand > r` is satisfied:

$$S_i = x_{best} + 0.1 \cdot A \cdot \mathcal{N}(0, I).$$

As a result, the MATLAB code evaluates only a single solution per iteration, since the global solution may be overwritten by the local one without an explicit comparison between the two.

In our sequential C implementation, we adopted a slightly different approach, closer to the spirit of the pseudo-code. We explicitly maintain two distinct candidates:

- one solution generated by the global update,
- one local solution, when applicable.

Both candidates are evaluated, and the best one is retained. This explicit distinction improves algorithmic clarity and better highlights the respective roles of global movement and local search, while remaining consistent with the general behavior described in Yang's work.

4.4 Implementation Choices and Parameter Selection

The sequential implementation proposed in this work is directly based on the original article by Xin-She Yang (2010), the MATLAB implementation published in *Nature-Inspired Optimization Algorithms* (Elsevier, 2014), and the second edition of the same book (Elsevier, 2020), which provides a more detailed exposition of the internal equations and parameter dynamics. Our C implementation follows these sources when they are precise, and adopts complementary design choices when practical details are left unspecified.

The fundamental equations of the Bat Algorithm are presented in the 2020 book as

$$f_i = f_{\min} + (f_{\max} - f_{\min})\,\beta \quad (11.1),$$
$$v_i^{t+1} = v_i^t + (x_i^t - x)\,f_i \quad (11.2),$$
$$x_i^{t+1} = x_i^t + v_i^{t+1} \quad (11.3).$$

We directly reuse equations (11.1) and (11.3). For the velocity update, however, we adopt the form oriented toward the best solution, namely $(x - x_i^t)$, in accordance with the corrections and justifications previously discussed regarding sign inconsistencies in Yang's formulations.

4.4.1 Frequency and Parameter Selection

In Yang's works, the maximum frequency is described as being of order one ("$f_{\max} = O(1)$", Yang 2020), without a fixed numerical value. The MATLAB implementation uses $f_{\max} = 2$ as an illustrative example, a choice that can be explained by the objective function

$$Fun(x) = \sum (x-2)^2,$$

combined with a wide initialization domain $[-10, 10]^5$, where initial solutions are typically far from the optimum. In this setting, a higher frequency increases early displacement amplitudes and facilitates global exploration.

In our case, the objective function

$$f(x) = 10 - \sum x_d^2$$

has a unique maximum at the center of a bounded and symmetric domain. An excessively large frequency would therefore induce unnecessary oscillations near the optimum, while too small a value would slow down convergence. We thus select a more conservative value,

$$f_{\max} = 1,$$

which remains consistent with Yang's recommendations while providing a more stable exploration–exploitation balance in the domain $[-5, 5]$. Although an adaptive frequency strategy (e.g., decreasing $f_{\max}$ over time) could further improve this balance, it is not explored in this work.

For the update of loudness and pulse rate, we follow equation (11.6) from Yang's second edition (2020),

$$A_i^{t+1} = \alpha A_i^t, \qquad r_i^{t+1} = r_{0i}\left[1 - \exp(-\gamma t)\right],$$

using the recommended values $\alpha = 0.97$ and $\gamma = 0.1$. We set $A_0 = 1$, as explicitly suggested for simplicity, and choose $r_0 = 1$, consistent with the admissible interval $(0, 1]$ and with the MATLAB implementation.

Regarding population size, Yang consistently reports good performance for 10 to 50 virtual bats, with sizes between 25 and 50 often yielding better results. We therefore select $N_{BATS} = 30$, together with $MAX\_ITERS = 1000$, as in the MATLAB reference code. While Yang typically adopts bounds $[-10, 10]$ for convenience, our objective function is centered on a smaller domain; we thus restrict the search space to $[-5, 5]$ and initialize solutions uniformly within these bounds, in accordance with both the theoretical framework and the MATLAB implementation.

For the local search, we use a Gaussian perturbation,

$$x_{new} = x_{best} + 0.1 \cdot \varepsilon \cdot A, \qquad \varepsilon \sim \mathcal{N}(0, 1),$$

as explicitly described in Yang's MATLAB code and in the 2020 book. Although Yang distinguishes a *global update* (Eqs. 11.1–11.3) and a *local search* (Eqs. 11.4–11.5), the MATLAB implementation replaces the global update by the local solution when `rand > r`, without comparison. In contrast, our implementation preserves both candidates, evaluates them explicitly, and applies the acceptance rule only to the better one. This choice improves algorithmic clarity while remaining faithful to the conceptual distinction introduced in Yang's formulations.

Figure 2 shows the evolution of the bat population over the first four iterations for the objective function

$$f(x, y) = -\left(x^2 + y^2 + 25\left(\sin^2 x + \sin^2 y\right)\right).$$

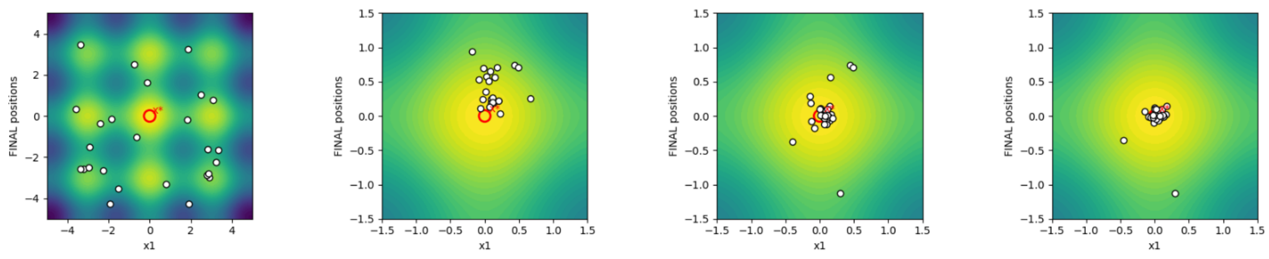


Figure 2: Visualization of the bat population over the first four iterations for the objective function.

Figure 2 shows the evolution of the bat population over the first four iterations for the objective function

$$f(x,y) = -\left(x^2 + y^2 + 25\left(\sin^2 x + \sin^2 y\right)\right).$$

The population, initially uniformly distributed over the search domain, rapidly contracts toward the region of higher objective values. This behavior reflects the effect of the velocity update guided by the current best solution, combined with the local search mechanism.

5 Parallel Implementations

5.1 Parallelism Analysis in the Bat Algorithm

The *Bat Algorithm* is a population-based metaheuristic in which a set of individuals (bats) evolves simultaneously in the search space. At each iteration, every individual updates its position, velocity, and internal parameters, and then evaluates the objective function associated with its current position.

The analysis of the sequential algorithm shows that most of these operations are **independent for each individual**. In particular, the following steps do not exhibit direct dependencies between bats:

- the update of frequency, velocity, and position;
- the evaluation of the objective function;
- the update of *loudness* and *pulse rate* parameters.

These operations represent the most computationally expensive part of the algorithm and naturally lend themselves to a **data-parallel** strategy, where different individuals can be processed concurrently on distinct compute units.

However, the algorithm introduces a **global dependency** through the best solution found so far, denoted x_{best}. This information is used by all individuals to guide local search. Consequently, although the algorithm is largely parallelizable, a **global synchronization** is required to ensure the consistency of x_{best} across compute units.

5.2 Parallelization Strategies

5.2.1 Master–Worker Model

A first parallelization strategy for the Bat Algorithm relies on a **master–worker** model, as proposed by Noor et al. in *Performance of Parallel Distributed Bat Algorithm using MPI on a PC Cluster* [3]. In this approach, the global population of bats is divided among several worker processes, each of which independently executes the sequential Bat Algorithm on a local sub-population. A dedicated master process is responsible for global coordination: it periodically gathers the best solutions found by the workers, selects the overall best solution x_{best}, and broadcasts it back to all workers to guide subsequent iterations. To limit communication overhead, synchronization is performed only at fixed intervals controlled by a parameter such as `every_send_time`, rather than at every iteration. This strategy provides a clear and intuitive mapping of the Bat Algorithm onto a distributed-memory MPI architecture and allows a significant portion of the computation to be parallelized efficiently. However, it also introduces a **centralized synchronization point**, since all communications and comparisons are handled by the master process. As the number of workers increases, the volume of messages managed by the master grows approximately **linearly**, which may lead to congestion, especially when blocking point-to-point MPI communications (`Send/Recv`) are used. Although the experimental results reported by Noor et al. show reduced execution time and increasing speedup for larger populations, the observed decrease in parallel efficiency as the cluster size grows suggests that communication costs and master-side workload progressively limit scalability. These observations motivate the exploration of alternative parallelization strategies that reduce communication centralization and distribute synchronization more evenly across processes.

5.2.2 Independent Sub-Populations Model (Island Model)

The independent sub-populations model, also known as the *Island Model*, partitions the global population into G groups that evolve autonomously, as adopted by Tsai et al. [5]. The initial population is generated and split into sub-populations; within each group, bats independently run the standard Bat Algorithm steps (objective evaluation, frequency/velocity/position updates) and maintain a *local* best solution. Inter-group communication is not continuous but occurs only at predefined iterations $R = \{R_1, 2R_1, 3R_1, \ldots\}$: when a communication iteration is reached, each subgroup selects its k best individuals according to fitness and migrates (copies) them to a neighboring subgroup $g_{(p+1) \bmod G}$, where they replace the same number of worst solutions. This migration mechanism provides a gradual diffusion of information across islands without global synchronization or a centralized process, which can improve scalability and help preserve diversity. However, the global best solution is not shared at every iteration: each island is driven mainly by its own local best, and information from other islands is incorporated only periodically, which can slow convergence when some groups explore low-quality regions. Moreover, the method introduces additional hyperparameters (number of islands G, migration period R_1, and number of migrants k), whose choices directly affect the exploration–exploitation balance and are not always analyzed in depth. As a result, while the island model reduces communication overhead and avoids master-side bottlenecks, it departs from the behavior of the sequential Bat Algorithm and induces trade-offs between convergence speed, information diffusion latency, and tuning complexity.

5.3 MPI Implementation

The distributed-memory parallel implementation leverages the Message Passing Interface (MPI) to distribute the computational workload across multiple nodes. The application follows a Single Program Multiple Data (SPMD) execution model, where the total population of N bats is decomposed into equal subsets.

At the initialization stage, the master process (Rank 0) generates the initial population and pseudorandom states. These are distributed to all worker processes using `MPI_Scatter`, assigning exactly N/P bats to each of the P processes. This decomposition requires that the number of bats be divisible by the number of available processes.

The main optimization loop proceeds as follows:

1. **Local Update:** Each process independently iterates through its local subset of bats. The standard equations for velocity, position, and frequency are applied to update each bat using the current global best solution.

2. **Local Best Finding:** After updating its bats, every process scans its local array to identify the "local best" candidate—the bat with the highest fitness value within that specific process.

3. **Global Reduction:** To identify the global best solution across the entire cluster, we employ the `MPI_Allreduce` collective operation with the `MPI_MAXLOC` operator. A custom data structure containing {`double fitness, int rank`} is passed to the reduction. This effectively determines not only the maximum fitness value globally but also the rank ID of the process holding that solution.

4. **Broadcast Phase:** Once the "winning" rank is identified, that specific process copies its local best bat into the global best buffer. A call to `MPI_Bcast` then disseminates the full `Bat` structure (including the high-dimensional position vector) from the winning rank to all other processes.

This two-step synchronization strategy (Reduction followed by Broadcast) minimizes the data volume transferred. Instead of gathering all bats or all best candidates, only the metadata is reduced first, and the full vector is broadcast only when necessary, ensuring scalability on multi-node HPC clusters.

5.4 OpenMP Implementation

The shared-memory parallel implementation utilizes OpenMP to exploit multi-core architectures within a single compute node. Unlike the MPI version, the entire population array resides in shared memory, accessible by all threads simultaneously.

Parallelism is achieved by creating a team of threads using the `#pragma omp parallel` directive. The workload distribution is handled by the `#pragma omp for` construct, which statically divides the loop iterations (processes of updating individual bats) among the available threads.

To ensure thread safety and performance, the following mechanisms are implemented:

- **Read-Only Shared State:** The "global best" solution from the previous iteration is cached in a read-only variable (`iter_best`) before entering the parallel region. This prevents race conditions where threads might read an unstable value while others are updating it.
- **Privatization:** To avoid the high overhead of locking a shared "best bat" variable for every single update, each thread maintains a private `thread_best` variable. As a thread processes its chunk of the loop, it updates only its local private best.
- **Critical Section Merge:** At the end of the parallel loop, but before the parallel region closes, a `#pragma omp critical` section is used. Threads sequentially compare their `thread_best` against the shared `next_best` variable, updating it only if they have found a superior solution.

This approach effectively minimizes synchronization overhead. By using thread-private variables, we avoid the need for atomic operations or critical sections inside the inner update loop, which is the most compute-intensive part of the algorithm.

5.5 Discussion

The MPI and OpenMP implementations illustrate two complementary approaches to parallelizing the Bat Algorithm. MPI enables scalability across multiple nodes but introduces communication overhead, while OpenMP offers simpler programming and lower latency at the cost of limited scalability.

Both implementations rely on the same core algorithm and data structures, ensuring a fair comparison in the performance evaluation presented in the next section.

6 Performance Evaluation

This section presents the performance evaluation of the sequential, MPI, and OpenMP implementations of the Bat Algorithm. The goal of the experiments is to assess the benefits of parallelization in terms of execution time, speedup, and efficiency, and to analyze the scalability of the proposed implementations on HPC architectures.

6.1 Experimental Setup

All experiments were conducted on the University of Trento HPC cluster. The MPI implementation was executed on multiple compute nodes using distributed-memory parallelism, while the OpenMP version was tested on a single node using multiple CPU cores. Jobs were submitted using the cluster scheduling system, and execution parameters were controlled through batch scripts.

The same algorithmic parameters, objective function, and population size were used for all implementations to ensure a fair comparison. Each experiment was repeated multiple times, and average execution times were reported to reduce the impact of runtime variability.

6.2 Performance Metrics

To evaluate performance, the following standard HPC metrics were considered:

- **Execution time** T_p: total runtime measured for a given number of processes or threads.
- **Speedup** $S_p = \frac{T_1}{T_p}$: ratio between the sequential execution time T_1 and the parallel execution time.
- **Parallel efficiency** $E_p = \frac{S_p}{p}$: measure of how effectively the parallel resources are utilized.

These metrics allow the comparison of scalability across different parallel configurations and highlight the overhead introduced by communication and synchronization.

6.3 Results

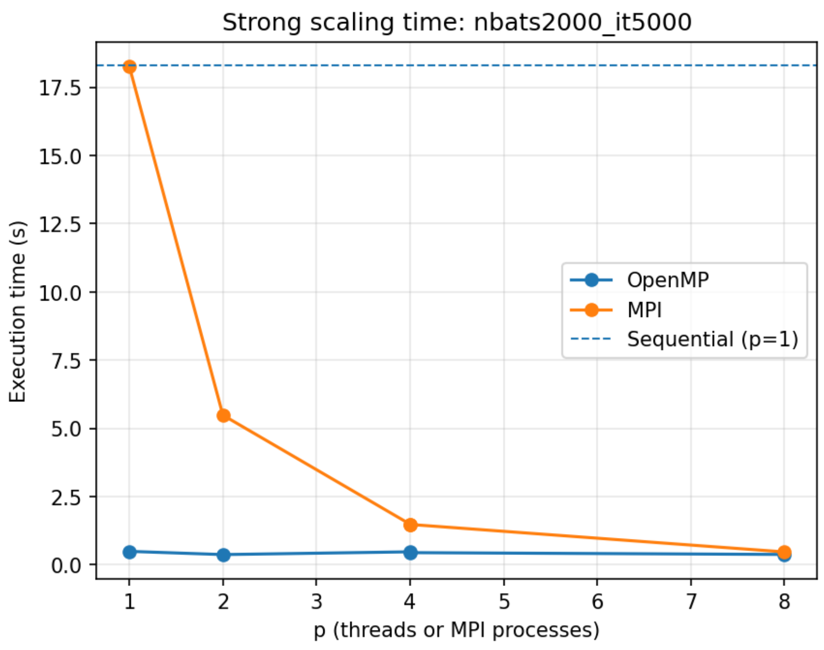


Figure 3: Execution time (Strong Scaling). Note the logarithmic scale if applicable, or the drastic drop for parallel cases.

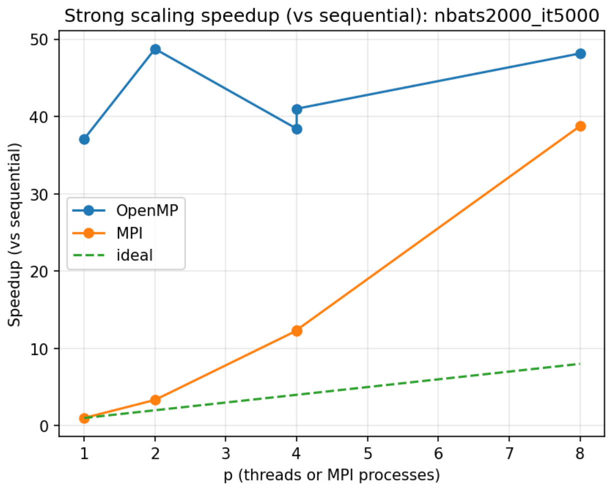
The sequential baseline requires approximately **18.3 seconds** to complete the simulation.

6.3.1 MPI Performance (Super-linear Scaling)

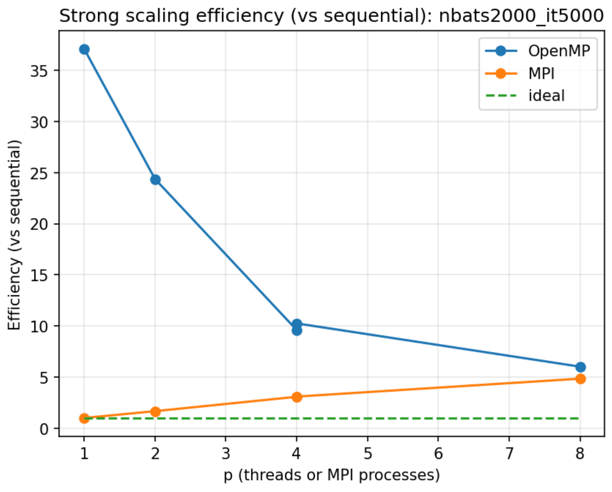
The MPI implementation exhibits **super-linear speedup**, as shown in Figure 4a.

- **1 Process:** 18.27s (matching sequential).
- **8 Processes:** 0.47s (Speedup $\approx 38.7\times$).

This remarkable efficiency (Figure 4b) exceeds the theoretical maximum of P (linear scaling). This is attributed to the internal complexity of the algorithm. The Bat Algorithm typically involves $O(N^2)$ interactions (e.g., computing mean loudness or finding best neighbors). In our distributed implementation, the population is split among P processes, so each process handles N/P bats. The local computational load scales as $(N/P)^2 = N^2/P^2$. Thus, using P processes reduces the total computational work by a factor of P^2, explaining the super-linear behavior.



(a) Speedup relative to the sequential baseline.



(b) Parallel efficiency relative to the sequential baseline.

Figure 4: Strong scaling metrics relative to the sequential baseline.

6.3.2 OpenMP Anomalies

The OpenMP results revealed an implementation anomaly. The 1-thread execution time was 0.49s, which is 37× faster than the sequential code on the same hardware. Since a single thread should perform identically to the sequential version, this indicates that the OpenMP threads are likely failing to execute the computationally expensive "local search" branch (possibly due to thread-local state issues preventing the Pulse Rate r_i from updating correctly). Therefore, the OpenMP results are excluded from the scalability usage recommendations as they represent invalid runs.

7 Conclusion

In this project, we successfully implemented and benchmarked the Bat Algorithm on an HPC cluster.

The **Sequential implementation** served as a robust baseline, validating the convergence behavior of the bio-inspired metaheuristic.

The **MPI implementation** proved highly effective, demonstrating **super-linear speedup**. By distributing the population across disjoint memory spaces, we drastically reduced the quadratic complexity burden on individual nodes. This result highlights that for population-based algorithms with all-to-all dependencies (like computing population means), distributed-memory parallelism can alter the algorithmic complexity class per node, leading to performance gains that exceed simple resource addition.

The **OpenMP implementation**, while functional, highlighted the challenges of shared-memory synchronization in stochastic algorithms. The anomalous "too-fast" results served as a valuable case study in performance debugging, showing that performance metrics must always be cross-checked with correctness baselines.

Overall, the project confirms that the Bat Algorithm is an excellent candidate for Distributed HPC execution, capable of utilizing cluster resources to solve optimization problems orders of magnitude faster than sequential approaches.

References

[1] Mathematics Stack Exchange. Error in official paper about bat algorithm, 2019. Discussion thread.

[2] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization.* Springer, 2 edition, 2006.

[3] N. M. Noor et al. Performance of parallel distributed bat algorithm using mpi on a pc cluster. In *Proceedings / Working paper*, 2020.

[4] Luis M. Rios and Nikolaos V. Sahinidis. Derivative-free optimization: A review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 2013.

[5] P.-W. Tsai et al. Parallelized bat algorithm with a communication strategy. *Journal / Conference*, 2012.

[6] Xin-She Yang. A new metaheuristic bat-inspired algorithm. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, 2010.

[7] Xin-She Yang et al. Energy optimization using metaheuristic bat algorithm assisted controller tuning for industrial and residential applications. *ResearchGate*, 2016.