UNIVERSITÀ
DI TRENTO

Parallel Bat Algorithm

A Sequential and Parallel Implementation

Wail Ameur, Bénie Nshimirimana

January 13, 2026

1 Introduction

This project is carried out in the context of the course *High-Performance Computing for Data Science* at the **University of Trento**. The aim of the assignment is to study a representative optimization algorithm and to explore its adaptation from a sequential implementation to parallel executions on high-performance computing (HPC) systems. In particular, 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 objective is to identify optimal parameter configurations under limited computational resources.

In practice, these optimization problems are often non-linear, non-convex, or noisy, making classical gradient-based methods inefficient or inapplicable. To address such challenges, population-based metaheuristic algorithms have been widely adopted. These methods rely on stochastic exploration mechanisms and cooperative search strategies to approximate global optima without requiring derivative information.

This project focuses on the **Bat Algorithm**, a bio-inspired metaheuristic introduced by *Xin-She Yang*. The algorithm models the echolocation behavior of bats to balance global exploration and local exploitation through adaptive parameters such as frequency, loudness, and pulse emission rate. 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 objective of this work is threefold. First, a correct and efficient **sequential implementation** of the Bat Algorithm is developed in C and used as a reference baseline. Second, a **distributed-memory parallel version** is implemented using the Message Passing Interface (**MPI**) and evaluated on an HPC cluster. Third, an additional **shared-memory implementation** based on **OpenMP** is proposed to exploit multi-core architectures.

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

2 Bat Algorithm

The Bat Algorithm (BA) is a population-based metaheuristic optimization method proposed by Xin-She Yang in 2010. It is inspired by the echolocation behavior

of microbats, which emit ultrasonic pulses and analyze the returning echoes to navigate and locate prey. Algorithmically, this behavior is translated into a cooperative search process that balances global exploration of the search space with local exploitation of promising regions.

Due to its stochastic nature and the independence of most operations applied to individuals, the Bat Algorithm is particularly well suited for parallel execution on modern computing architectures.

2.1 Biological Inspiration and Intuition

In nature, bats dynamically adjust the characteristics of their emitted sound pulses depending on their distance from a target. When far from prey, bats emit loud pulses at low rates to explore the environment. As they approach a target, pulse emission becomes more frequent while loudness decreases, allowing for more precise localization.

In the Bat Algorithm, these behaviors are abstracted through adaptive parameters that control the movement of candidate solutions within the search space. This mechanism allows the algorithm to gradually shift from exploration to exploitation as the optimization progresses.

2.2 Algorithm Description

In the Bat Algorithm, each bat represents a candidate solution to the optimization problem. A bat i is characterized by the following variables:

- a position vector $x_i \in R^d$, representing a solution in the search space;
- a velocity vector v_i, controlling the bat's movement;
- a frequency f_i, which influences the step size;
- a loudness A_i, regulating solution acceptance;
- a pulse emission rate r_i, controlling the probability of local search.

At each iteration, bats update their frequency, velocity, and position while being attracted toward the current global best solution. In addition, a local random walk around the best solution is performed with a probability related to the pulse emission rate. New solutions are accepted based on their quality and the bat's loudness.

2.3 Mathematical Model

The frequency of bat i is updated as:

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

where $\beta \sim U(0, 1)$.

The velocity and position updates are defined as:

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

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

where x^* denotes the best solution found so far.

With a probability $1 - r_i$, a local search is performed around the global best solution:

$$x_{local} = x^* + \epsilon A_{mean},$$

where $\epsilon \sim \mathcal{N}(0, 1)$ and A_{mean} is the average loudness of the population.

A new solution is accepted if it improves the objective function and satisfies a loudness-based acceptance criterion. After acceptance, the loudness and pulse emission rate are updated as:

$$A_i^{t+1} = \alpha A_i^t,$$

$$r_i^{t+1} = r_0 \left(1 - e^{-\gamma t}\right),$$

where α and γ are algorithm parameters controlling convergence behavior.

2.4 Algorithm Workflow

The overall workflow of the Bat Algorithm can be summarized as follows:

1. Initialize a population of bats with random positions and velocities.
2. Evaluate the objective function for all bats and determine the initial global best solution.
3. Iteratively update bat parameters and positions.
4. Perform local search and solution acceptance based on loudness and pulse rate.
5. Update the global best solution.
6. Repeat until a stopping criterion is met.

A conceptual illustration of the algorithm is shown in Figure 1, highlighting the interaction between individual bats and the global best solution.

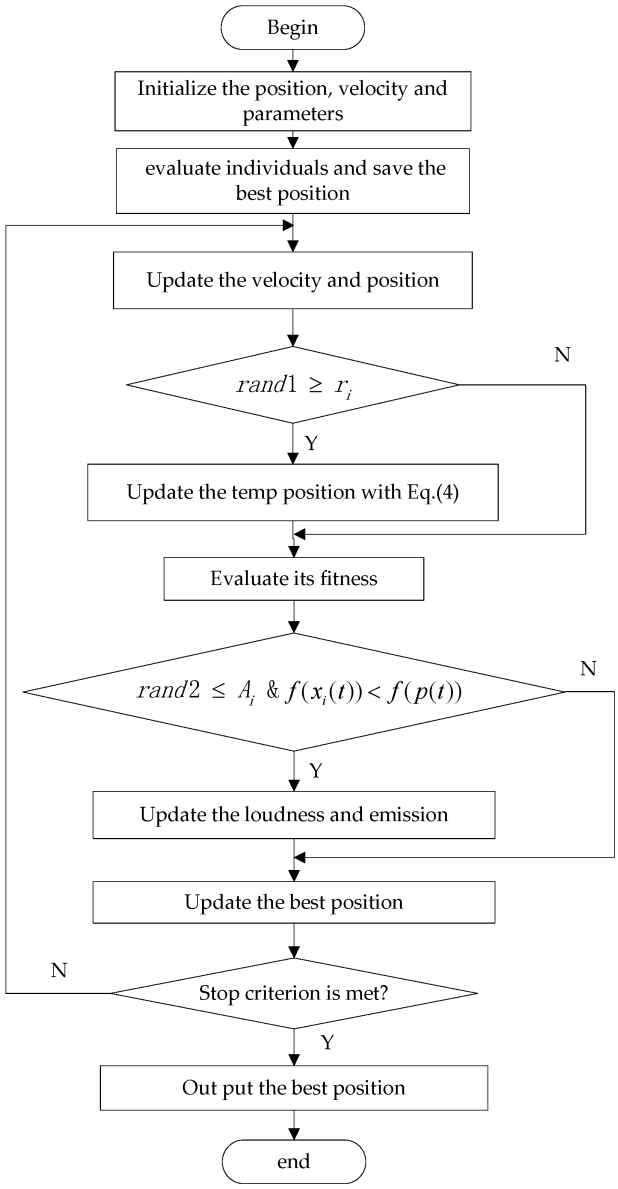

Figure 1: Conceptual flowchart of the Bat Algorithm. The population of bats explores the search space while being progressively attracted toward the best solution through adaptive exploration and exploitation mechanisms.

3 Sequential Implementation

Before introducing parallelism, a sequential version of the Bat Algorithm was implemented in the C programming language. This implementation serves as a reference baseline to validate correctness and to evaluate the performance gains achieved by parallel execution.

3.1 Implementation Overview

The sequential implementation closely follows the mathematical formulation of the Bat Algorithm described in Section 2. The algorithm operates on a fixed-size population of bats, each represented by a structure containing its position, velocity, frequency, loudness, pulse emission rate, and objective function value.

The implementation is modular and consists of separate source files for the core algorithm, utility functions, and data structures. This design facilitates code reuse and simplifies the transition to parallel implementations. In particular, the same data structures and objective function are reused in both the MPI and OpenMP versions.

3.2 Initialization and Parameters

At the beginning of execution, the bat population is initialized with random positions uniformly distributed within predefined bounds of the search space. Velocities are initialized to zero, while loudness and pulse emission rate are set to their initial values. The objective function is evaluated for each bat, and the initial global best solution is identified.

Algorithm parameters such as population size, frequency range, loudness decay factor, and pulse emission growth rate are fixed throughout the execution. These parameters were selected to ensure stable convergence behavior while keeping the computational workload representative for benchmarking purposes.

3.3 Main Iterative Loop

The algorithm proceeds iteratively for a fixed number of iterations. At each iteration, all bats are updated sequentially. For each bat, frequency, velocity, and position updates are performed, followed by a probabilistic local search around the current global best solution. Candidate solutions are evaluated using the objective function and accepted based on the loudness-based acceptance criterion.

After each bat update, the global best solution is updated if a better solution is found. This sequential dependency ensures correctness but also highlights a potential bottleneck when considering parallel execution.

3.4 Objective Function and Output

For validation and benchmarking purposes, a simple continuous objective function is used. Intermediate results, such as the positions of bats at selected iterations, are stored to enable visualization of the algorithm's convergence behavior. Additionally, the best objective value is periodically printed to monitor convergence during execution.

This sequential implementation provides a correct and reproducible baseline against which the MPI and OpenMP implementations are compared in terms of execution time and scalability.

4 Parallel Implementations

The Bat Algorithm is well suited for parallel execution due to its population-based structure, where most operations applied to individual bats are independent. However, the algorithm also introduces global dependencies, most notably the need to maintain and update the globally best solution found so far. This section describes the parallelization strategy adopted in this work and presents the distributed-memory and shared-memory implementations based on MPI and OpenMP, respectively.

4.1 Parallelization Strategy

At each iteration of the Bat Algorithm, the update of each bat involves local computations such as frequency update, velocity update, position update, and objective function evaluation. These operations can be performed independently for different bats, making the algorithm naturally amenable to data parallelism.

The main challenge arises from the dependency on the global best solution, which must be consistently shared among all processing elements. After each iteration, the best solution identified locally must be compared with those found by other processes or threads to determine the global optimum. This requirement introduces synchronization and communication overhead, which must be carefully managed to achieve good scalability.

In this work, the population of bats is partitioned across parallel workers, and each worker is responsible for updating a subset of the population. Synchronization is performed once per iteration to update the global best solution.

4.2 MPI Implementation

The distributed-memory parallel implementation is based on the Message Passing Interface (MPI) and follows a Single Program Multiple Data (SPMD) execution model. Each MPI process executes the same program but operates on a different subset of the bat population.

At initialization, the total population is divided evenly among the available MPI processes. Each process initializes and updates its local bats independently.

During each iteration, every process determines the best solution found among its local bats.

To compute the global best solution across all processes, an `MPI_Allreduce` collective operation is used. This operation allows all processes to contribute their local best values and receive the global best solution in a single synchronization step. By using `MPI_Allreduce`, the need for explicit master–worker coordination is avoided, leading to a simpler and more symmetric design.

Once the global best solution is known, it is broadcast implicitly to all processes through the collective operation, allowing each process to proceed with the next iteration using a consistent global state. This approach ensures correctness while limiting communication to a single collective operation per iteration.

The MPI implementation is designed to execute on multi-node HPC clusters, where processes run on separate memory spaces and communicate explicitly through message passing.

4.3 OpenMP Implementation

In addition to the MPI version, a shared-memory parallel implementation using OpenMP is proposed. In this case, all threads operate within the same memory space and share access to the bat population and the global best solution.

Parallelism is introduced by parallelizing the loop that updates the bats at each iteration using OpenMP `parallel for` directives. Since multiple threads may attempt to update the global best solution concurrently, synchronization mechanisms such as critical sections or reductions are required to ensure thread safety.

Compared to the MPI implementation, the OpenMP version benefits from lower communication overhead due to shared memory access. However, scalability is limited by the number of available cores on a single node and by contention on shared data structures.

The OpenMP implementation is particularly suitable for multi-core systems and serves as a complementary approach to the MPI version, highlighting the differences between shared-memory and distributed-memory parallel programming models.

4.4 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.

5 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.

5.1 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.

5.2 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.

5.3 Results

Figure 2 shows the execution time of the Bat Algorithm for the sequential, MPI, and OpenMP implementations ($N = 2000, iters = 5000$).

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

5.3.1 MPI Performance (Super-linear Scaling)

The MPI implementation exhibits **super-linear speedup**, as shown in Figure 3.

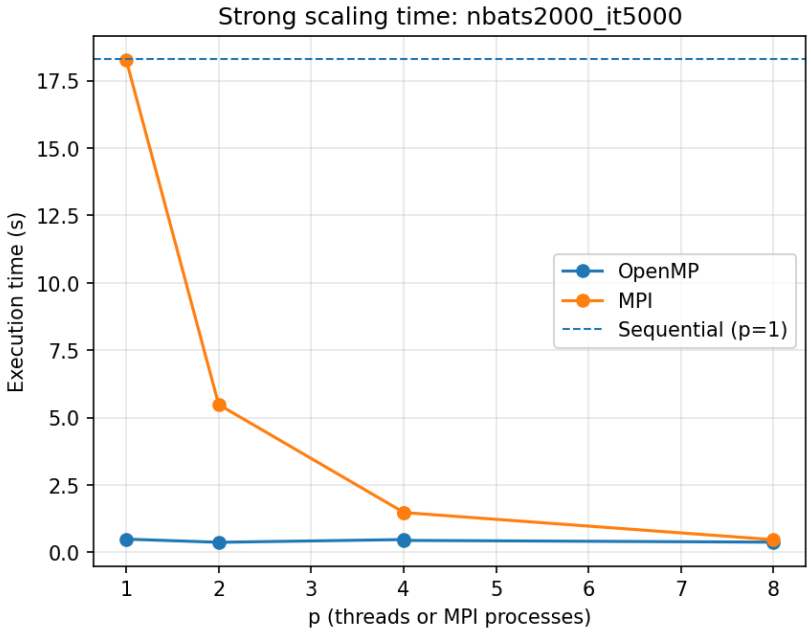

Figure 2: Execution time (Strong Scaling). Note the logarithmic scale if applicable, or the drastic drop for parallel cases.

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

This remarkable efficiency (Figure 4) 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.

5.3.2 OpenMP Anomalies

The OpenMP results revealed an implementation anomaly. The 1-thread execution time was 0.49s, which is $37\times$ 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,

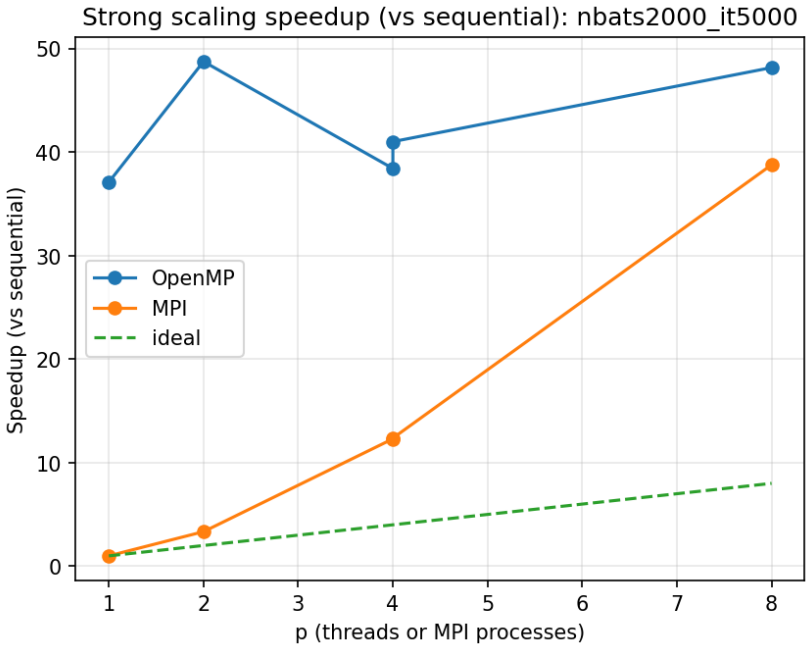

Figure 3: Speedup relative to the Sequential baseline. MPI achieves super-linear speedup.

the OpenMP results are excluded from the scalability usage recommendations as they represent invalid runs.

6 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,

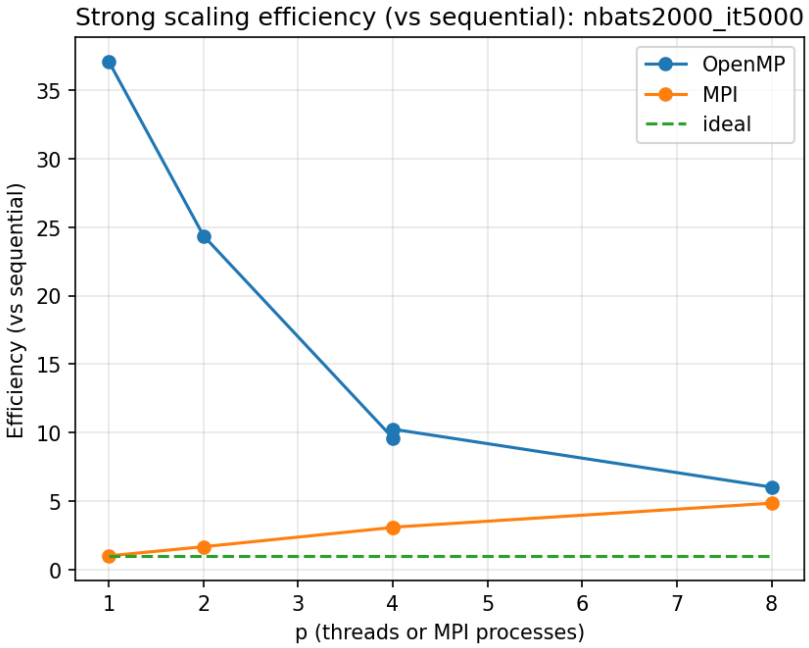

Figure 4: Parallel Efficiency. MPI and OpenMP shows efficiency well above 1.0 due to algorithmic decomposition.

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.