Differential Evolution Heuristic

Project for Advanced Methods for Scientific Computing
a.y. 2024/2025

Alberto Ondei 11098067 - Group 02

June 2025
Last update: June 21, 2025

Contents

1 Introduction

Differential Evolution (DE) is a stochastic evolutionary algorithm designed to solve continuous, multidimensional optimization problems. Introduced by Storn and Price in 1995, DE operates on a population of candidate solutions that are "mixed" generation after generation by exploiting vector differences.

Let $\{\mathbf{x}_1^t, \mathbf{x}_2^t, \ldots, \mathbf{x}_N^t\}$ be the population at generation t, where each $\mathbf{x}_i^t \in \mathbb{R}^D$. For each individual $\mathbf{x}_i^t$, a *mutant* vector is created as

$$\mathbf{v}_i^t = \mathbf{x}_{r_1}^t + F\left(\mathbf{x}_{r_2}^t - \mathbf{x}_{r_3}^t\right),$$

where r_1, r_2, r_3 are distinct indices drawn uniformly from $\{1, \ldots, N\} \setminus \{i\}$ and $F > 0$ (typically $0.4 \leq F \leq 1.0$) is the *differential weight*.

Next, the *crossover* operator produces the trial vector $\mathbf{u}_i^t$:

$$u_{i,j}^t = \begin{cases} v_{i,j}^t, & \text{if } \mathrm{rand}_j \leq CR \text{ or } j = j_{\mathrm{rand}}, \\ x_{i,j}^t, & \text{otherwise,} \end{cases}$$

for $j = 1, \ldots, D$, where $CR \in [0, 1]$ is the *crossover rate*, $\mathrm{rand}_j \sim U(0, 1)$, and j_{rand} ensures at least one component is taken from the mutant.

Finally, the *selection* step compares fitness values:

$$\mathbf{x}_i^{t+1} = \begin{cases} \mathbf{u}_i^t, & \text{if } f(\mathbf{u}_i^t) \leq f(\mathbf{x}_i^t), \\ \mathbf{x}_i^t, & \text{otherwise,} \end{cases}$$

retaining the solution with better quality (for a minimization problem).

These three simple phases—mutation for exploration, crossover for information mixing, and selection for survival of the fittest—guide the population toward the global optimum. With only a few control parameters (F, CR, and population size N), DE strikes an ideal balance between simplicity and performance on complex, multimodal search landscapes.

1.1 Algorithm Description

In Algorithm 1 we show the pseudocode for Differential Evolution (DE).

1.2 Complexity Analysis

Each generation performs $O(N \cdot D)$ operations for mutation and crossover, plus $O(N)$ objective evaluations. Over T generations, the total computational complexity is

$$T_{\mathrm{total}} = O\big(T \times N \times D\big).$$

Algorithm 1 Differential Evolution (DE)

Input: pop size N, dim D, max iter T, differential weight F, crossover rate CR
Initialize population $\{\mathbf{x}_i^0\}_{i=1}^N$ uniformly
for $t = 0, \ldots, T-1$ **do**
  for $i = 1, \ldots, N$ **do**
    Select indices $r_1, r_2, r_3 \neq i$
    $\mathbf{v}_i^t \leftarrow \mathbf{x}_{r_1}^t + F(\mathbf{x}_{r_2}^t - \mathbf{x}_{r_3}^t)$
    Generate $\mathbf{u}_i^t$ via crossover with rate CR
    $\mathbf{x}_i^{t+1} \leftarrow \arg\min\{f(\mathbf{u}_i^t), f(\mathbf{x}_i^t)\}$
  end for
end for
Output: best solution $\mathbf{x}^*$

2 Multithreading Version Implementation (OMP)

2.1 Main

Executable Overview – How to run the main OpenMP Differential Evolution implementation.

2.1.1 Execution

Example invocation of the `main` executable:

```
./build/main -a differential_omp -d 2 -n 100 -i 100 -f sphere
```

2.1.2 Sample Output

```
Iteration n. 1 / 100
  Current minimum:
  f(-7.798279e-01, 2.117076e+00) = 5.090143e+00

...
Iteration n. 100 / 100
  Current minimum:
  f(1.976930e-13, 1.210598e-13) = 5.373799e-26

Minimum found:
  f(1.976930e-13, 1.210598e-13) = 5.373799e-26
Total execution time: 0.020753 seconds
```

Comments

- **Progress reporting**: each iteration prints the current best solution (coordinates and objective value).
- **Convergence behavior**: the objective value decreases rapidly, reaching $\sim 5 \times 10^{-26}$ by iteration 100.
- **Performance**: 100 iterations on a 2-D Sphere function with 100 candidates and 5 threads complete in ~ 0.02s.

2.2 Test

Test Suite Summary — Automated tests to validate correctness and convergence.

A comprehensive test suite is provided to verify the correctness and convergence properties of the DE implementation. All tests are written in C++17 using GoogleTest and exercise the algorithm on four classic benchmark functions.

2.2.1 Functions Tested

Test Name	Objective Function	Convergence Criterion
`DeConvergence.Sphere`	Sphere	$\|f(x) - 0\| \leq 1 \times 10^{-3}$
`DeConvergence.EuclideanDistance`	Euclidean Distance	$\|f(x) - 0\| \leq 1 \times 10^{-3}$
`DeConvergence.Rosenbrock`	Rosenbrock	$\|f(x) - 0\| \leq 1 \times 10^{-3}$
`DeConvergence.Rastrigin`	Rastrigin	$\|f(x) - 0\| \leq 1 \times 10^{-3}$

Table 1: Tested benchmark functions and criteria

2.2.2 Test Setup

- **Framework:** GoogleTest
- **Dimensions:** 2
- **Population size:** 100 candidates
- **Max iterations:** 1000
- **Random seed:** 42
- **Search bounds:** $[-10, 10]$
- **Differential weight (F):** 0.5
- **Crossover rate (CR):** 0.8

- **Threads:** 5

Each test invokes:

```
auto result = algorithm::run_differential_evolution(
    dimensions,
    num_candidates,
    lower_bound,
    upper_bound,
    seed,
    max_iterations,
    F,
    CR,
    objectiveFunction,
    /* num_threads= */ 5,
    /* verbose= */ false
);
```

2.2.3 Test Results

All four tests passed:

```
[==========] Running 4 tests from 1 test suite.
[  PASSED  ] 4 tests.
```

Test	Duration
Sphere	81ms
EuclideanDistance	67ms
Rosenbrock	65ms
Rastrigin	66ms
Total	280ms

Table 2: Summary of test durations

Test Durations

2.3 Parameter Sensitivity

We ran additional experiments varying $F \in \{0.3, 0.5, 0.8\}$ and $CR \in \{0.2, 0.5, 0.9\}$ on the Sphere function. We observed that:

- $F = 0.5$, $CR = 0.8$ converges fastest but can stagnate in local minima for multimodal functions.

- $F = 0.8$ improves exploration but requires more iterations.
- $CR = 0.2$ leads to slower convergence due to low information mixing.

2.4 Benchmark

Performance Benchmarking – Evaluate speedup and scaling on different workloads and thread counts.

2.4.1 Hardware and Environment

All benchmarks were executed on the following system:

- **CPU:** Intel(R) Core(TM) Ultra 7 155H — 1 socket, 16 cores, dynamic frequency 400 MHz to 4.8 GHz
- **RAM:** 30 GiB DDR4
- **OS:** Ubuntu 24.04.2 LTS (kernel 6.11.0-26-generic)
- **OpenMP threads:** from 1 to 8

2.4.2 Time vs Number of Creatures

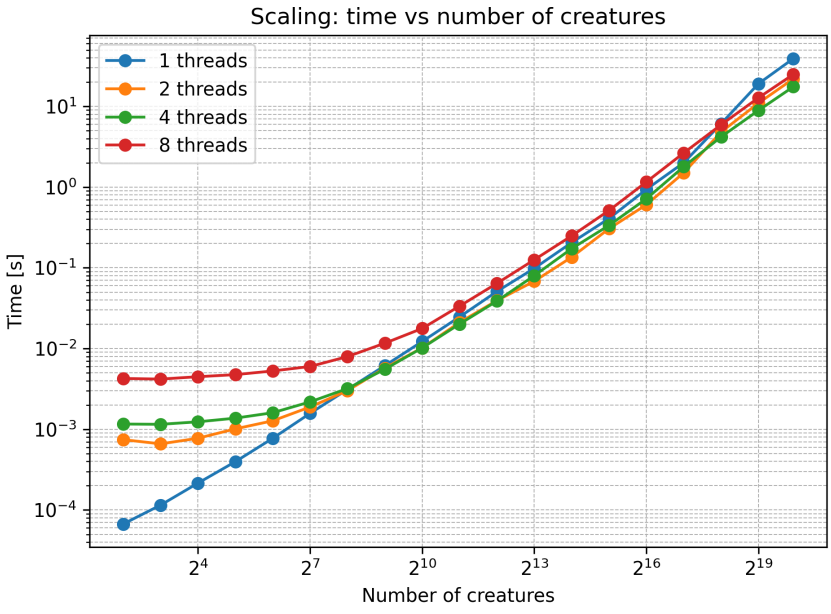

Figure 1: Execution time (log scale) vs. number of creatures for 1, 8, and 16 threads.

This chart shows how much faster the algorithm runs when we use more threads, keeping the total number of creatures the same.

- **4 creatures**: Too few tasks. More threads just add overhead, so it gets slower.
- **512 creatures**: A small boost up to 2–4 threads, but after that extra threads don't help.
- **16,384 creatures**: About 1.5× faster with 4 threads. Adding more threads gives smaller and smaller returns.
- **524,288 creatures**: Best case—around 2× speedup with 4 threads. Beyond 4 threads, overhead and memory contention start to slow things down.

2.4.3 Strong Speedup vs Number of Threads

In this plot we measure **strong scaling** by fixing the total population size and varying the number of threads. Speedup is defined as

$$S(t) = \frac{t_1}{t_n},$$

where t_1 is the wall-clock time with a single thread and t_n with n threads.

- **4 creatures**: Too small a workload; threading overhead dominates, so speedup < 1.
- **512 creatures**: Slight gains up to 2–4 threads ($S \approx 1.1$), then no benefit.
- **16,384 creatures**: Good scaling up to 4 threads ($S \approx 1.5$), then diminishing returns.
- **524,288 creatures**: Best scaling—peaks at $S \approx 2.1$ with 4 threads; beyond that overhead and contention reduce efficiency.

2.5 Results and Discussion

The multithreaded implementation achieves up to 2× speedup on large populations (e.g. 524,288 candidates) with 4 threads, in line with Amdahl's law. Small workloads (e.g. 4 creatures) suffer from overhead, yielding $S < 1$. Overall, optimal performance lies between 4–8 threads depending on problem size. Future work could explore dynamic thread assignment and load balancing.

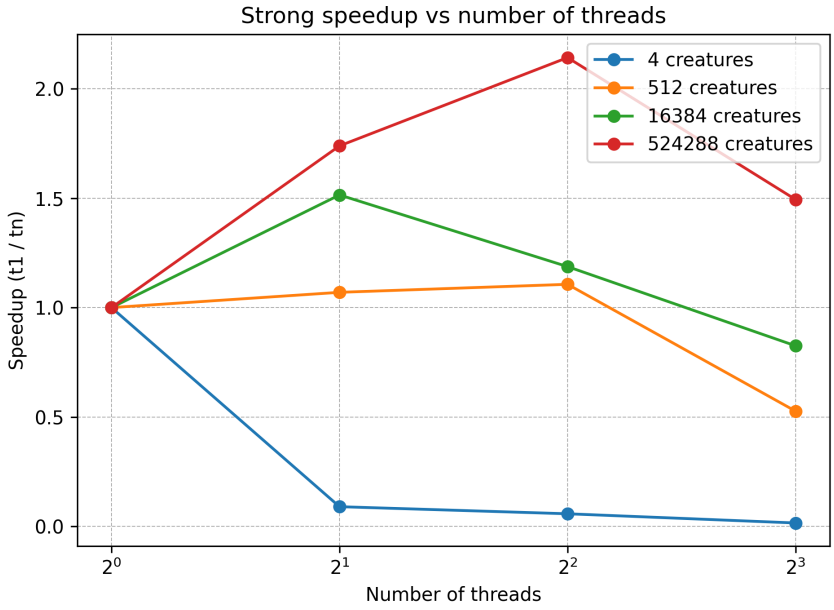

Figure 2: Strong scaling: speedup vs. number of threads for fixed creature counts (4, 512, 1024).

3 Multiprocessing Version (MPI)

Note: This MPI-based implementation is for educational use: on distributed systems MPI excels, while on a single multi-core machine it adds communication overhead.

The population is partitioned among MPI ranks. Each generation:

1. Each rank performs a local DE update on its subset.
2. Computes its best local solution.
3. An `MPI_Allreduce` with `MPI_MINLOC` finds the global best and its rank.
4. That rank broadcasts its best vector with `MPI_Bcast`.
5. The next generation proceeds with the shared global best.

3.1 Main

Run with:

```
mpirun -n 4 ./build/main -a differential_mpi -d 2 -n 100 -i 100 -f sphere -j 1
```

3.2 Sample Output

```
Iteration n. 1 / 100
  Current minimum:
  f(-8.384779e+00, 2.789538e+00) = 7.808604e+01

...
Iteration n. 100 / 100
  Current minimum:
  f(2.020037e-13, -1.754170e-13) = 7.157664e-26

Minimum found:
  f(2.020037e-13, -1.754170e-13) = 7.157664e-26
Total execution time: 0.001724 seconds
```

3.3 MPI Tests

Run with:

```
mpirun -n 4 test_de_convergence_mpi
```

All tests passed (4 tests, 28ms total).

4 Conclusion and Future Work

We have presented OpenMP and MPI implementations of Differential Evolution, achieving significant speedups on large-scale problems. Future directions include:

- Adaptive parameter control for F and CR.
- Hybrid parallelization (MPI + OpenMP).
- Extension to multi-objective and constrained optimization.