Parallel Simulated Annealing with OpenMP and MPI

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

Sophia Peritz 10778839 - Group 02

June 2025

Last update: June 21, 2025.

Contents

1 Simulated Annealing

Simulated Annealing (SA) is a well-established stochastic optimization method inspired by the physical process of annealing in solids. Originally introduced in the early 1980s, SA has since become one of the most widely used metaheuristic techniques for global optimization problems where the objective function is expensive to evaluate, non-differentiable, or analytically unavailable.

The core idea of SA is to emulate the cooling of a physical system to reach a low-energy crystalline state.

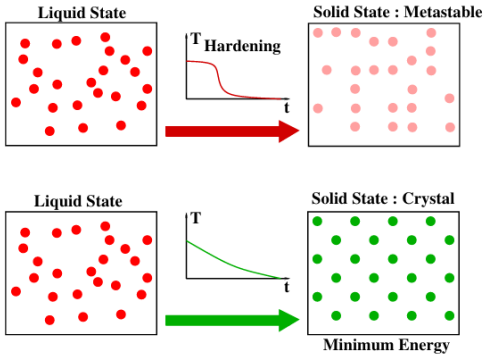


Figure 1: [1] When temperature is high, the material is in a liquid state (left). For a hardening process, the material reaches a solid state with non-minimal energy (metastable state; top right). In this case, the structure of the atoms has no sym metry. During a slow annealing process, the material reaches also a solid state but for which atoms are organized with symmetry (crystal; bottom right).

In this analogy, the optimization problem is modeled by defining:

- the **state space** S as the set of all possible solutions;
- the **objective function** $f : S \to \mathbb{R}$ as an energy function to be minimized;
- a **temperature parameter** T, which controls the system's ability to escape local minima by accepting transitions to higher-energy (worse) states.

1.1 Mathematical Foundation

The algorithm iteratively generates new candidate solutions $x' \in N(x)$, where $N(x) \subseteq S$ denotes the neighborhood of the current solution x. If the candidate solution improves the objective function (i.e., $f(x') < f(x)$), it is always accepted. Otherwise, it is accepted with probability:

$$P(\text{accept } x') = \exp\left(-\frac{f(x') - f(x)}{k_B T}\right)$$

where:

- $\Delta f = f(x') - f(x)$ is the change in the objective function,
- k_B is a constant analogous to Boltzmann's constant,
- T is the current temperature.

This acceptance rule, known as the *Metropolis criterion*, allows SA to probabilistically accept worsening moves, enabling the algorithm to escape local minima during early (high-temperature) stages. As $T \to 0$, the algorithm becomes increasingly greedy, converging to a local optimum.

1.2 Algoritmic Structure

The `SimulatedAnnealing` class implements the Simulated Annealing optimization algorithm. The main features and methods of the class are described as follows:

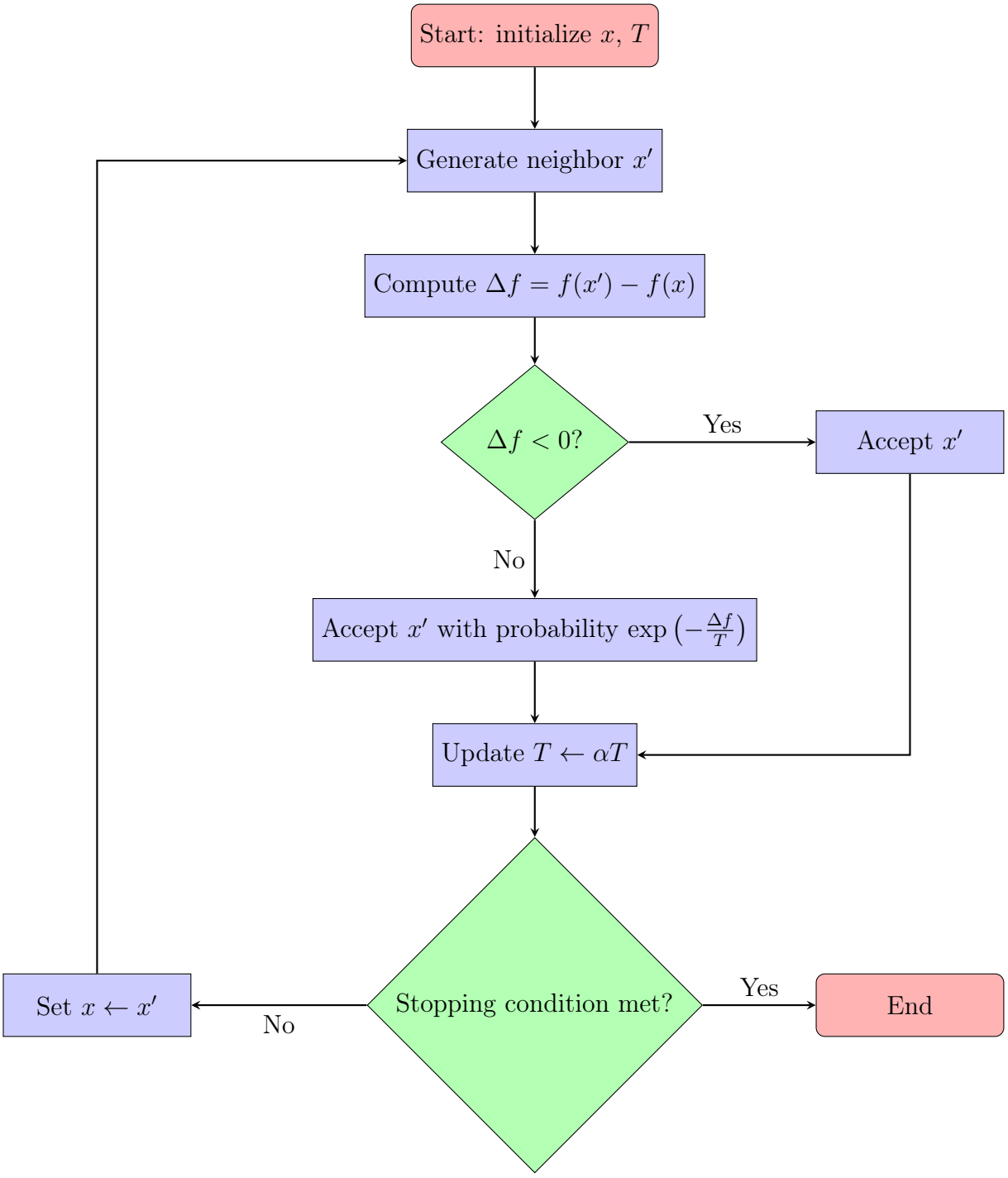
- **Constructor**: Initializes the algorithm parameters including the objective function, problem dimension, maximum number of iterations, dwell iterations, initial temperature and cooling rate, initial step size and its scaling factor, Boltzmann constant, search bounds, initial guess, and random number generator seed.
- **New Solution Proposal** (`proposeNewSolution`): Generates a candidate solution by perturbing the current solution with a random value drawn from a uniform distribution within the interval $[-stepSize, stepSize]$. The candidate solution is then clamped to lie within the specified lower and upper bounds.
- **Equilibration** (`equilibrate`): Performs a number of iterations where the candidate solution is accepted or rejected according to the Metropolis criterion. Better solutions are always accepted; worse solutions are

accepted probabilistically based on the current temperature and the difference in cost. This step simulates thermal equilibrium at a given temperature.

- **Melting Phase** (`melt`): Increases the current temperature iteratively until the system "melts". This phase ensures the system starts in a sufficiently randomized state.
- **Annealing Process** (`anneal`): Executes the main annealing loop for the maximum number of iterations, performing equilibration at each temperature, then reducing the temperature and step size according to their scaling factors, thereby gradually focusing the search towards optimal solutions.
- **Temperature Update** (`updateTemperature`): Multiplies the current temperature by the cooling factor to simulate gradual cooling.
- **Step Size Update** (`updateStepSize`): Scales down the step size to allow for finer search as the algorithm progresses.
- **Best State Accessor** (`getBestState`): Returns the best solution state found so far during the annealing process.

The `State` class represents a candidate solution in the optimization process. Its main functionalities are:

- **Constructor**: initializes the state with a vector of values and computes the associated cost by evaluating the objective function on these values.
- **Method** `update`: updates the state values with a new vector and recalculates the associated cost using the objective function.


Start: initialize x, T
Generate neighbor x'
Compute $\Delta f = f(x') - f(x)$
$\Delta f < 0$?
Yes
Accept x'
No
Accept x' with probability $\exp\left(-\frac{\Delta f}{T}\right)$
Update $T \leftarrow \alpha T$
Stopping condition met?
Yes
End
No
Set $x \leftarrow x'$

2 Parallel Implementation with OpenMP

To parallelize the Simulated Annealing process, I implemented a multi-threaded version using OpenMP. Each thread runs a separate, independent annealing process, allowing for concurrent exploration of the solution space. The main section of the implementation is shown below:

Listing 1: OMP-based Simulated Annealing

```
#pragma omp parallel num_threads(n_threads)
{
    int tid = omp_get_thread_num();
    size_t local_seed = seed + tid;

    SimulatedAnnealing sa(
        *func, dimensions, max_iterations / n_threads,
        dwell_iterations / n_threads, initial_temperature,
        temperature_scale, initial_step_size, step_size_scale,
        boltzmann_constant, initial_guess,
        lower_bound, upper_bound, local_seed
    );

    sa.melt();
    sa.anneal();

    bestStates[tid] = std::make_unique<State>(sa.getBestState());
    bestCosts[tid] = bestStates[tid]->cost;
}
```

Key implementation choices:

- **One particle per thread:**
  - Simple and efficient design.
  - Each thread explores the space independently.
  - Avoids synchronization overhead or contention.
- **Different random seeds:**
  - Ensures diversity of solutions.
  - Reduces the risk of all threads converging to the same local minimum.

- **Preallocated vectors:**
  - Thread-safe writes via indexing: `bestStates[tid]`.
  - No need for `#pragma omp critical` sections.
- **Final reduction:**
  - The best solution is selected by comparing all local minima in the following way:

Listing 2: Best Solution Selection

```
auto min_it = std::min_element(bestCosts.begin(), bestCosts.end());
size_t best_idx = std::distance(bestCosts.begin(), min_it);
```

Note on parallel strategy

Assigning one particle per thread offers simplicity and efficient load balancing, but may limit solution diversity if the number of threads is low. In more complex scenarios, using multiple particles per thread could improve global search performance, at the cost of increased memory usage and slightly more complex thread management.

2.1 Convergence Tests

This section documents a set of tests used to verify the **convergence behavior of the Simulated Annealing algorithm** on various classical objective functions. The tests are implemented using `GoogleTest` and check both the **accuracy of the solution** and the **closeness of the estimated minimum position** to the known global optimum.

Each test asserts that the algorithm achieves a result sufficiently close to the known global minimum, with different error thresholds tailored to the specific characteristics of each objective function.

Parameter Sensitivity

The algorithm parameters, such as initial temperature, cooling rate, step size, etc., are **not one-size-fits-all**. They are tuned for each function individually, due to the unique challenges posed by each:

- **Local curvature:** e.g., Rastrigin has many sharp local minima, while Sphere is smooth and convex.
- **Problem conditioning:** Rosenbrock has a narrow, curved valley that requires fine-grained control.
- **Search space scale and multimodality:** Parameters like `step_size` and `temperature_scale` greatly impact the balance between exploration and convergence.

Choosing appropriate parameters involves balancing exploration (via temperature and step size) with convergence (via cooling and decay), and often requires empirical tuning.

Tested Functions

Objective Function	Convergence Criterion
Sphere	$\lvert f(x) - 0\rvert \leq 10^{-3}, \quad \lVert x - 0\rVert \leq 5 \times 10^{-2}$
Euclidean Distance	$\lvert f(x) - 0\rvert \leq 3 \times 10^{-3}, \quad \lVert x - 0\rVert \leq 3 \times 10^{-3}$
Rosenbrock	$\lvert f(x) - 0\rvert \leq 10^{-3}, \quad \lVert x - x^*\rVert \leq 10^{-2}$
Rastrigin	$\lvert f(x) - 0\rvert \leq 10^{-2}, \quad \lVert x - 0\rVert \leq 2 \times 10^{-2}$

Table 1: Summary of convergence criteria for each test.

Test Setup

All tests use a shared core structure, invoking:

```
algorithm::run_simulated_annealing(
    dimensions,
    max_iterations,
    dwell,
    initial_temperature,
    temperature_scale,
    initial_step_size,
    step_size_scale,
    boltzmann_k,
    initial_guess,
    lower_bound,
    upper_bound,
    objective_function,
    seed,
```

```
    n_threads,  // 4
    true        // verbose output
);
```

Sample Output

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from SaConvergence

[ RUN      ] SaConvergence.Sphere
Minimum found:
  f(3.490406e-04, -9.462030e-04) = 1.017129e-06
[       OK ] SaConvergence.Sphere (58 ms)

[ RUN      ] SaConvergence.EuclideanDistance
Minimum found:
  f(5.057432e-06, -1.274359e-04) = 1.275363e-04
[       OK ] SaConvergence.EuclideanDistance (115 ms)

[ RUN      ] SaConvergence.Rosenbrock
Minimum found:
  f(9.994925e-01, 9.991756e-01) = 6.201195e-07
[       OK ] SaConvergence.Rosenbrock (92 ms)

[ RUN      ] SaConvergence.Rastrigin
Minimum found:
  f(4.373832e-04, 1.087775e-03) = 2.727006e-04
[       OK ] SaConvergence.Rastrigin (330 ms)

[----------] 4 tests from SaConvergence (596 ms total)
[==========] 4 tests from 1 test suite ran. (596 ms total)
[  PASSED  ] 4 tests.
```

2.2 Benchmarking

To evaluate the performance and scalability of the parallel implementation of Simulated Annealing using OpenMP, a set of benchmarks was conducted on a machine with 8 logical cores. The tested objective function was the Rosenbrock function in two dimensions, with a fixed initial guess and optimization bounds. The number of threads and the maximum number of iterations were varied to assess the impact on both performance and scalability.

Benchmark Configuration

The benchmark was implemented using the `Google Benchmark` framework. The following parameters were kept constant across all tests:

- **Initial temperature**: 15.0
- **Temperature scale**: 0.93
- **Initial step size**: 0.5
- **Step size scale**: 0.99
- **Dwell iterations**: 300
- **Boltzmann constant**: 1.0

The tested configurations involved varying the number of threads `(1, 2, 4, 8)` and the maximum number of iterations `(200, 512, 1000, 2000)`. For each configuration, we measured:

- **Real execution time (in seconds)**
- **Effective throughput in iterations per second**

Results and Discussion

Figure 2 shows the results of the benchmarks. As expected, increasing the number of threads consistently reduces execution time and improves throughput. The performance gain is most notable when increasing from 1 to 4 threads. However, the improvements become sublinear beyond 4 threads, indicating the onset of parallel overhead and diminishing returns due to task granularity or synchronization costs.

The highest throughput was observed with 8 threads and 2000 iterations, reaching nearly **98,000 iterations per second**. In contrast, the same configuration with a single thread achieved less than **2,000 iterations per second**, confirming a significant speedup thanks to parallelization. The improvement, however, is not perfectly linear, especially at higher thread counts, due to the fixed cost of overheads and the limited dimensionality of the problem.

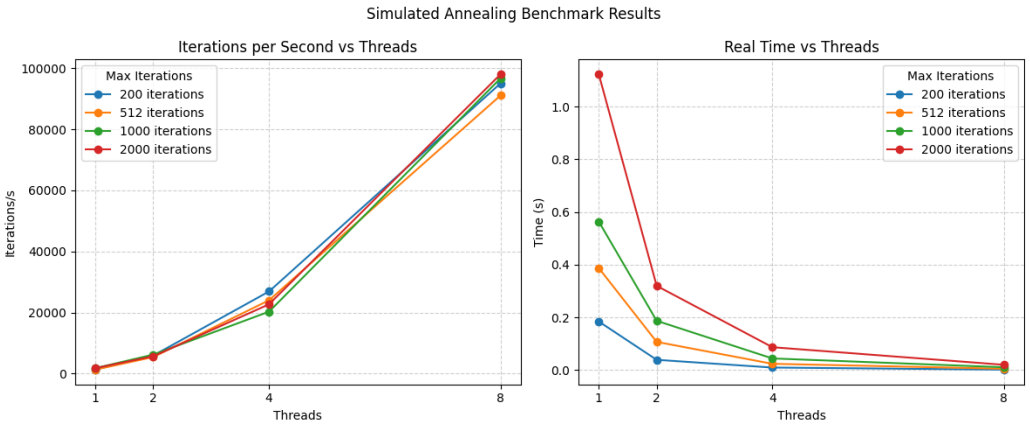


Figure 2: Benchmark results for parallel Simulated Annealing. (Left) Iterations per second vs. number of threads. (Right) Real execution time vs. number of threads.

Scalability

The results demonstrate good scalability for low to moderate thread counts. The diminishing efficiency at 8 threads suggests that for this relatively small optimization problem (2D Rosenbrock), the parallel workload becomes less balanced, and thread contention begins to affect performance.

3 Parallel Implementation with MPI

To enable distributed memory parallelism, the Simulated Annealing algorithm was parallelized using `MPI` (Message Passing Interface). The core parallel execution is implemented in the `run_sa_mpi` function:

Listing 3: MPI initialization

```
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

Each MPI process generates a different random initial guess, using a process-specific seed to ensure search space diversification:

Listing 4: Random initialization

```
std::mt19937 local_gen(seed + world_rank * 99991);
std::uniform_real_distribution<double> dist(lower_bound, upper_bound);
for (size_t i = 0; i < dimensions; ++i) {
    my_guess[i] = dist(local_gen);
}
```

Then, each process constructs its own instance of the Simulated Annealing object with its local initialization:

Listing 5: Local SA initialization

```
DistributedSimulatedAnnealing sa(*func,
    dimensions,
    max_iterations,
    dwell_iterations,
    initial_temperature,
    temperature_scale,
    initial_step_size,
    step_size_scale,
    boltzmann_constant,
    my_guess,
    lower_bound,
    upper_bound,
    seed + world_rank * 1000);
```

Each process then independently executes the annealing routine. After the search, the best result from each process is gathered using `MPI_Allreduce` with the `MPI_MINLOC` operation to identify the global minimum and the process that found it:

Listing 6: Reduction to global best

```
std::pair<double, int> local_result = {local_best.cost, world_rank};
std::pair<double, int> global_result;

MPI_Allreduce(
    &local_result,
    &global_result,
    1,
    MPI_DOUBLE_INT,
    MPI_MINLOC,
    MPI_COMM_WORLD
);
```

Finally, the globally best solution is broadcast to all processes to ensure consistency:

Listing 7: Broadcast of best solution

```
MPI_Bcast(global_best_position.data(), static_cast<int>(dimensions),
          MPI_DOUBLE, global_result.second, MPI_COMM_WORLD);
```

3.1 Convergence Tests

The parallel MPI version was tested using 4 processes on a suite of benchmark functions. Execution logs showed all processes completed successfully and produced valid results:

```
[==========] 4 tests from 1 test suite ran. (1678 ms total)
[  PASSED  ] 4 tests.
```

- **No significant speedup** was observed: execution time remained nearly constant regardless of the number of MPI processes.
- This is due to the **embarrassingly parallel structure** of the algorithm:
  - Each process performs its own Simulated Annealing search independently.
  - There is no inter-process communication during computation.
  - Communication occurs only once at the end to:
    1. Collect the best solution from each process via `MPI_Allreduce`.
    2. Broadcast the globally best solution to all processes via `MPI_Bcast`.

- As a result, the MPI version:
  - **Does not reduce wall-clock time** for a single search.
  - **Improves robustness and reliability** by exploring the search space from different initial guesses.

References

[1] Daniel Delahaye, Supatcha Chaimatanan, Marcel Mongeau.
Simulated Annealing: From Basics to Applications.
In: Gendreau M., Potvin J.Y. (eds) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol 272. Springer, 2019.
DOI: 10.1007/978-3-319-91086-4_1.

[2] Ding-Jun Chen, Chung-Yeol Lee, Cheol-Hoon Park, Pedro Mendes.
Parallelizing Simulated Annealing Algorithms Based on High-Performance Computer.
Journal of Global Optimization, 39(3), pp. 261–289, 2007.
DOI: 10.1007/s10898-007-9138-0.

[3] Patrick Siarry, Gérard Berthiau, François Durbin, Jacques Haussy.
Enhanced Simulated Annealing for Globally Minimizing Functions of Many Continuous Variables.
C.E.A. and École Centrale de Paris.

[4] A. Corana, M. Marchesi, C. Martini, S. Ridella.
Minimizing Multimodal Functions of Continuous Variables with the 'Simulated Annealing' Algorithm.
ACM Transactions on Mathematical Software (TOMS), Vol. 13, No. 3, pp. 262–280, 1987.