STOCHASTIC TUNNELLING with OPENMP and MPI

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

Giuseppe Starnini 252540 - Group 02

26th June 2025

Contents

1 Stochastic Tunnelling Approach **3**

2 Analysis **4**
2.1 Variables 4
2.2 Beta Management 4
2.3 Movement Conditions 4

3 Implementation **5**
3.1 Objective Function 5
3.2 OpenMP 5
3.3 MPI 6

4 Results **6**

1 Stochastic Tunnelling Approach

Stochastic Tunnelling (ST) is an optimization algorithm designed for determining the global minima of complex and rugged energy landscapes. It is a generic physically motivated generalization of simulated annealing. This approach circumvents the freezing problem which arises when the energy difference between "adjacent" local minima on the energy surface is much smaller than the energy of intervening transition states separating them.The physical idea behind the stochastic tunneling method is to allow the particle to "tunnel" forbidden regions, once it has been determined that they are irrelevant for the low-energy properties of the problem. This can be accomplished by applying the transformation:

$$f_{STUN}(x) = 1 - \exp[-\gamma(f(x) - f_0)],$$

where f_0 is the lowest minimum encountered thus far. The effective potential preserves the locations of all minima, but maps the entire energy space from f_0 to the maximum of the potential onto the interval [0, 1]. The degree of steepness of the cutoff of the high-energy regions is controlled by the tunneling parameter γ. To illustrate the physical content of the transformation we consider a Monte Carlo process at some fixed inverse temperature β. A step from x_1 to x_2 with $\Delta = f(x_1) - f(x_2)$ is accepted with probability $\tilde{w}_{1\to 2} = \exp(-\tilde{\beta}\Delta)$. In the process the temperature rises rapidly when the local energy is larger than f_0 and the particle diffuses (or tunnels) freely through potential barriers of arbitrary height. As better and better minima are found, ever larger portions of the high-energy part of the function surface are flattened out. In analogy to the simulated annealing approach this behavior can be interpreted as a self-adjusting cooling schedule that is optimized as the simulation proceeds. Parameter β is adjusted during the simulation. If a short-time moving average of the f_{stun} exceeds the threshold f_{thresh}, β is reduced by some fixed factor, otherwise it is increased. β is updated in the following manner: The average number of times a particle changes its position over the last N time steps is computed. If this average exceeds a predetermined threshold, the parameter is increased by an adjustment factor. Conversely, if the average falls below the threshold, is decreased. This adaptive mechanism ensures an appropriate balance between periods of search and periods of tunneling, optimizing the particle's exploration strategy.

2 Analysis

2.1 Variables

- positions: number of particles
- dimension : dimension of the domain
- lower bound : starting point of the domain for each dimension
- upper bound : ending point of the domain for each dimension
- max iterations : number of iteration to compute
- beta : The parameter that determines the tunneling probability
- beta threshold : threshold which determine the changing of beta
- beta adjust factor: factor which divide or multiply beta
- tunnelling: number of step used to compute the average of tunnelling of a particle
- gamma : parameter which determine the flattening of the function
- sigma: each particle moves with a random movement which follows a normal distribution with variance sigma. It start from sigma max and decrease at each step linearly till sigma min

2.2 Beta Management

At each time step, if a particle undergoes a change in position, its internal tunnelling array is updated by appending a value of 1; otherwise, a value of 0 is appended. This array stores the particle's movement status for the last N time steps. The average of these stored values, representing the frequency of movement, is then computed. Should this average exceed a globally defined beta thresholding variable, the β parameter is incremented by beta adjust factor. Conversely, if the average falls below this threshold, β is decreased by the same factor.

2.3 Movement Conditions

A particle always moves to a better point. If the function has a worst value in the new point the particles moves with a probability $\tilde{w}_{1\to 2} = \exp(-\tilde{\beta}\Delta)$

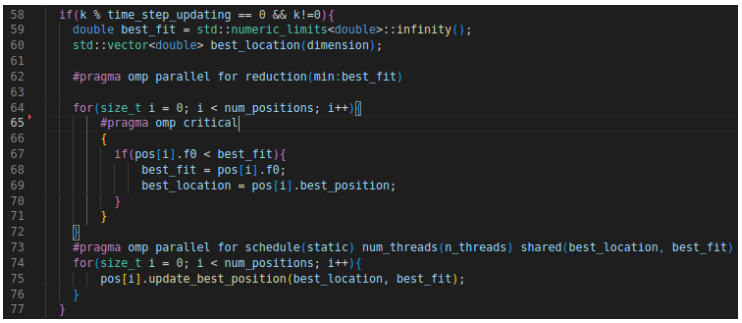
3 Implementation

3.1 Objective Function

Each algorithm requires a pointer to an object implementing the objective function to be minimized. This is implemented with a pure virtual class (include/ObjectiveFunction.hpp) which declares a single method: the overloaded operator(). This const method takes as input the array of coordinates to be evaluated in the form of an std::vector and returns a single scalar being the value of the function. This design allows the user to minimize parameterized functions, such as Rosenbrock and Rastrigin, by declaring their parameters as fields.

3.2 OpenMP

I decided to parallelize all movements of the particle. At each iteration each particle can update its position and change its own paramater in a full parallel way. The only critical moment is when every N iteration, the global minimum is computed and shared to all particles.



3.3 MPI

The same critical situation appeared in MPI implementation:

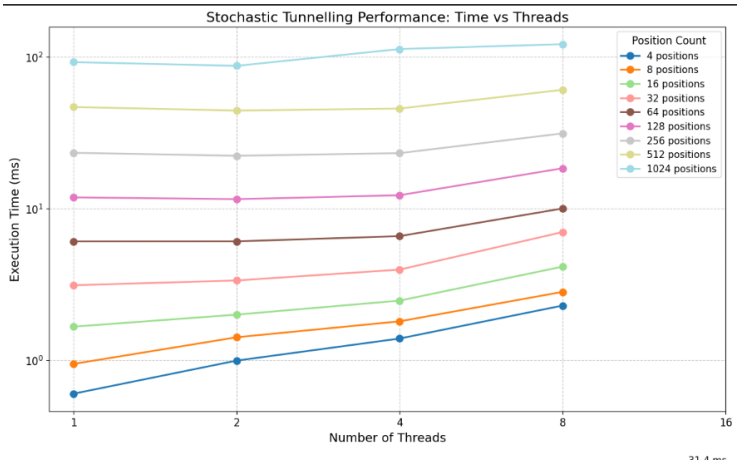
```
if(k % time_step_updating == 0 && k != 0) {
    double local_best_fit = std::numeric_limits<double>::infinity();
    std::vector<double> local_best_location(dimension);

    for(size_t i = start; i < end; i++) {
        if(pos[i].f0 < local_best_fit) {
            local_best_fit = pos[i].f0;
            local_best_location = pos[i].best_position;
        }
    }

    if(world_rank == 0) {
        std::vector<double> all_fits(world_size);
        std::vector<std::vector<double>> all_locations(world_size, std::vector<double>(dimension));

        for(int i = 1; i < world_size; i++) {
            MPI_Recv(&all_fits[i], 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            MPI_Recv(all_locations[i].data(), dimension, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        all_fits[0] = local_best_fit;
        all_l
            std::vector<double>::iterator min_it
        auto min_it = std::min_element(all_fits.begin(), all_fits.end());
        int best_rank = std::distance(all_fits.begin(), min_it);
        std::vector<double> global_best = all_locations[best_rank];
        double global_best_fit = *min_it;

        for(int i = 1; i < world_size; i++) {
            MPI_Send(&global_best_fit, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            MPI_Send(global_best.data(), dimension, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
        }

        for(size_t i = start; i < end; i++) {
            pos[i].update_best_position(global_best, global_best_fit);
        }

    } else {
        MPI_Send(&local_best_fit, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        MPI_Send(local_best_location.data(), dimension, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

        double global_best_fit;
        std::vector<double> global_best(dimension);
        MPI_Recv(&global_best_fit, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv(global_best.data(), dimension, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        for(size_t i = start; i < end; i++) {
            pos[i].update_best_position(global_best, global_best_fit);
        }
    }
}
```

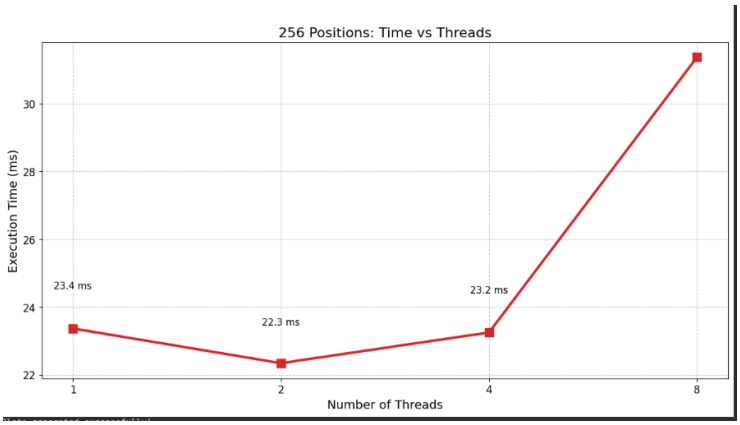
4 Results

In this section we show the results of some benchmarks we performed with the shared-memory implementation All benchmark results reported in the following paragraphs have been measured on the same machine. The versions of various software used for the benchmarks are reported here:

- CMake 3.27.5
- GNU Make 4.3
- GNU g++ 11.4.0
- OpenMPI 5.0.5

The computing time is increased with number of threads probably for huge overheading. Here are presented the resuts obtained with google benchmark.


Stochastic Tunnelling Performance: Time vs Threads
Position Count
4 positions
8 positions
16 positions
32 positions
64 positions
128 positions
256 positions
512 positions
1024 positions
Execution Time (ms)
Number of Threads


256 Positions: Time vs Threads
Execution Time (ms)
23.4 ms
22.3 ms
23.2 ms
Number of Threads

The convergence results in the considered functions are good. Here is presented a custum suite test with OpenMp applied to different surfaces:

Test Setup

- **Framework:** GoogleTest
- **Dimensions:** 2
- **Population size:** 100 candidates
- **Max iterations:** 1 000
- **Random seed:** 42
- **Search bounds:** [-10, 10]
- **Threads:** 5
- **Starting value of sigma (sigma_max)** = 1.0
- **Final value of sigma (sigma_min)** = 5.e-5
- **Gamma** = 0.0001
- **Beta_adjust_factor** = 0.9
- **Number of step used to compute avg of movements (tunnelling)** = 10
- **Threshold (beta_tresholding)** = 0.2
- **Frequency of best-position exchange among particles (time_step_updating)** = 100

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from TunnellingConvergence
[ RUN      ] TunnellingConvergence.Sphere
[       OK ] TunnellingConvergence.Sphere (53 ms)
[ RUN      ] TunnellingConvergence.Rosenbrock
[       OK ] TunnellingConvergence.Rosenbrock (24 ms)
[ RUN      ] TunnellingConvergence.Rastrigin
[       OK ] TunnellingConvergence.Rastrigin (28 ms)
[ RUN      ] TunnellingConvergence.EuclideanDistance
[       OK ] TunnellingConvergence.EuclideanDistance (42 ms)
[----------] 4 tests from TunnellingConvergence (149 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (149 ms total)
[  PASSED  ] 4 tests.
```

Summary of test duration

Test	Duration
Sphere	53 ms
Rosenbrock	24 ms
Rastrigin	28 ms
Euclidean Distance	42 ms
Total elapsed time	280 ms