Parallel Genetic Algorithm with OpenMP and MPI

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

Filippo Barbari 259411 - Group 02

4th February 2025

Last update: January 28, 2025.

Contents

1 Genetic Algorithm

A Genetic Algorithm (GA, for short) is a type of derivative-free optimization algorithm which aims to mimic the evolutionary behavior of animal species in order to find the optimum of a given function. In the GA jargon, a **creature** is a certain solution (i.e. a point in the n-dimensional search space) while the **population** is the collection of all solutions currently handled by the algorithm. Also, to further explain the analogy, it is really common to see Genetic Algorithm descriptions using the term **generation** to indicate iterations and **offspring** to indicate the population for the next generation.

A Genetic Algorithm, at a high level, is divided in 4 steps, to be repeated as many times as needed:

1. **Evaluation**: Each creature/solution is evaluated and assigned a **fitness score**.

2. **Selection**: Based on the results of the evaluation phase, a process is applied to either select the "best" creatures (the ones with better results) or to discard the "worst" creatures. This process emulates the natural selection applied from a certain environment to the animal species which live in it.

3. **Crossover/Reproduction**: During this phase, the selected creatures are randomly combined to create new offspring (new solutions) to fill the next generation.

4. **Mutation**: The solutions in the new generation are randomly modified to introduce variety in the population.

A Genetic Algorithm needs a pair of parameters to control its behavior:

- The **survival rate**: a coefficient between 0 and 1 representing the relative amount of creatures surviving from a generation to the next one.

- The **mutation rate**: a coefficient between 0 and 1 representing the probability of a mutation appearing in a creature.

These two parameters help to control the algorithm's behavior and to balance it between exploration and convergence. A low survival rate means that a lot of solutions are discarded at each generation and, therefore, the offspring is generated from a small pool of creatures resulting in a fast convergence towards an optimum. A high mutation rate, on the other hand, makes the algorithm more similar to a random search since many of the creatures of each generation are randomly modified. As usual, the best combination of these parameters changes from one problem to another, especially when dealing with high dimensionality.

Usually, a Genetic Algorithm deals with arrays of bits as solutions, or arrays of discretized values, indicating the "genome" of each creature. In our case, however, we are dealing with the continuous landscapes of n-dimensional mathematical functions such as Rosenbrock: this different application domain required some modifications in certain key parts of the algorithm which we will discuss in the following sections.

2 Analysis

In our implementation of the GA, we decided to use the selection technique often referred to as **elitism** which is really straightforward: if we say that n is the size of the population and s is the survival rate, this technique sorts all the creatures based on their fitness score and discards all the worst $n \cdot (1 - s)$ creatures.

2.1 Evaluation

The evaluation phase requires to compute the value of the objective/fitness function for each creature in the population. We can assume that the evaluation of the objective function on a given creature is independent from all the other creatures in the population so this phase is embarrassingly parallel. We can also optimize this phase by avoiding to re-compute the fitness for creatures which did not change since the last generation.

2.2 Selection

In this phase, as mentioned at the beginning of the section, the algorithm needs to sort all creatures in the current generation based on their fitness score and discard the worst ones. In both implementations, this phase is called `sortCreatures` since most of the computational complexity of this phase comes from the sorting required.

2.3 Crossover

During the crossover phase the next generation of creatures is created by randomly combining some of the creatures which survived in the last generation. There are many variations in the literature of this process: some involve different numbers of parents while others mix random values during the crossover. In our implementations, only two unique parents are used and their "genome", their respective solutions are randomly mixed without introducing any modification. To clarify, one can think of this procedure as flipping a fair coin for each coordinate in the solution and choosing the value from the first parent if it's heads or the second parent if it's tails.

In order to save memory and execution time, this technique is implemented, in both versions of the algorithm, by overwriting the worst solutions.

The only dependency each thread/process may have with others is the array of creatures on which they operate. The surviving creatures are only read for the crossover and we can divide evenly the remaining "slots" in the population between the threads/processes. For this reason, this phase is embarrassingly parallel.

2.4 Mutation

Lastly, each creature has a certain probability (the mutation rate) to be randomly modified. In the continuous domain of our implementations, this random modification means that a random coordinate (chosen uniformly) is replaced by a new random value in the whole domain.

This phase is embarrassingly parallel since mutating a creature is independent from all other creatures.

3 Implementation

In this section we will discuss the implementation details and choices we made to overcome some challenges we faced during development.

The API to use these algorithms is inside `src/Algorithm.cpp` which handles the initialization of the input structures, performs the actual loop and extracts the result at the end.

Both implementations regard the same algorithm, so they have a lot in common

3.1 OpenMP

3.2 MPI

4 Results

4.1 Benchmark setup

4.2 OpenMP results

4.3 MPI results