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 29, 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 (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 clarify 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" ones. 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 single 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 assume 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 selects the best $n \cdot s$ 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. The final result of the algorithm is an `std::pair` containing the vector of coordinates of the minimum found and the value of the function at that point.

Both implementations regard the same algorithm but they use two different paradigm of parallel programming, so they have at the same time a lot of common points and differences. For example, both use a single class to handle the state of the algorithm (`src/GeneticAlgorithm.cpp` for the OpenMP implementation and `src/DistributedGeneticAlgorithm.cpp` for the MPI implementation) but, while for the shared-memory version an instance of this class represents the global state of the algorithm (which all threads access and use), for the distributed version this class represents just the data local to each process. Another example is the method used to store the population data: the OpenMP implementation uses an `std::vector` of `Creature` objects while the MPI one uses two `std::vector`s containing just the raw data.

3.1 OpenMP

As mentioned earlier, the OpenMP implementation uses an `std::vector` of `Creature` objects to store the population data. Each `Creature` (structure visible at `src/Creature.hpp`) is a simple object containing an `std::vector` representing its position and a single scalar, called `fitness`, which represents its fitness score. This AoS (Array-of-Structures) design is straightforward and improves the cache efficiency for operations involving single creatures such as the evaluation and the mutation phases. Also, keeping the fitness score together with the set of coordinates which generated it helps with the optimization mentioned in section 2.1 and allows implementing another simple feature: "marking" some creatures to be re-evaluated if they receive a mutation. This feature is implemented by giving the value of double-precision positive infinity (`std::numeric_limits<double>::infinity()`) to each creature which needs to be evaluated.

Evaluation As explained in section 2.1, the evaluation phase is embarrassingly parallel, so we use a single `pragma omp` directive on the `for` loop (shown in listing 1, extracted from `src/GeneticAlgorithm.cpp`) with the following parameters:

- `num_threads(n_threads)`: use all the available threads.
- `schedule(static)`: divide iterations evenly *a priori* between threads for minimum runtime overhead.
- `default(none)`: debugging option to disable implicitly declarating variables as shared inside the parallel region.

```cpp
void GeneticAlgorithm::evaluateCreatures() {
  #pragma omp parallel for schedule(static) \
    num_threads(n_threads) default(none)
  for (size_t i = 0; i < creatures.size(); i++) {
    if (std::isfinite(creatures.at(i).fitness)) {
      continue;
    }

    creatures.at(i).fitness = func(creatures.at(i).position);
  }
}
```

Listing 1: Parallel evaluation phase in OpenMP.

Selection and Crossover In this implementation the sorting of the creatures is performed serially before entering the crossover phase during which each thread creates its own random number generator with a unique seed before continuing. This can be seen in listing 2 in which each "discarded" creature is replaced by a new one.

```cpp
void GeneticAlgorithm::applyCrossover(const size_t seed) {
  // ...
  #pragma omp parallel num_threads(n_threads) default(none) \
    shared(seed, survived, n_creatures)
  {
    const size_t thread_id = omp_get_thread_num();

    std::mt19937 rnd{seed + thread_id};
    std::uniform_int_distribution<size_t> index_dist{
      0, survived - 1};
    std::bernoulli_distribution bool_dist{0.5};

    #pragma omp for schedule(static)
    for (size_t i = survived; i < n_creatures; i++) {
      const size_t father_index = index_dist(rnd);
      const Creature& father = creatures.at(father_index);

      size_t mother_index;
      do {
        mother_index = index_dist(rnd);
      } while (father_index == mother_index);
      const Creature& mother = creatures.at(mother_index);

      const size_t d = creatures.at(i).position.size();
      for (size_t j{0}; j < d; j++) {
        creatures.at(i).position.at(j) =
          bool_dist(rnd) ?
            father.position.at(j) :
            mother.position.at(j);
      }
      creatures.at(i).fitness =
        std::numeric_limits<double>::infinity();
    }
  }
  // ...
}
```

Listing 2: Parallel crossover phase in OpenMP.

The parents are selected with a uniform distribution among the creatures which survived and the resulting "son" is generated by using a Bernoulli distribution to select the parent to copy the coordinate value from.

In this phase, we use the same `pragma omp` options as before but, instead, we separate the creation of the thread pool (`parallel` region) from its usage in the `for` loop.

Mutation Listing 3 shows the parallel mutation phase in which, like the crossover phase, each thread creates its own random number generator in the `parallel` region before entering the `for` region. For each creature in the population, there is a chance of mutation based on the mutation rate, which is determined using a Bernoulli distribution. If a creature is selected for mutation, a random coordinate in its position vector is chosen using a uniform distribution, and its value is replaced with a new random value within the bounds of the search space (`lower_bound` and `upper_bound`).

```
void GeneticAlgorithm::applyMutation(const size_t seed) {
#pragma omp parallel num_threads(n_threads) default(none) \
  shared(seed)
  {
    const size_t thread_id = omp_get_thread_num();

    std::mt19937 rnd{seed + thread_id};
    std::bernoulli_distribution bool_dist{mutation_rate};
    std::uniform_real_distribution<double> position_dist{
      lower_bound, upper_bound};

    const size_t d = creatures.at(0).position.size();
    std::uniform_int_distribution<size_t> index_dist{0, d-1};

    #pragma omp for schedule(static)
    for (size_t i = 0; i < creatures.size(); i++) {
      if (!bool_dist(rnd)) {
        continue;
      }

      creatures.at(i).position.at(index_dist(rnd)) =
        position_dist(rnd);
      creatures.at(i).fitness =
        std::numeric_limits<double>::infinity();
    }
  }
}
```

Listing 3: Parallel mutation phase in OpenMP.

3.2 MPI

As mentioned earlier in this section, the distributed implementation of the Genetic Algorithm divides the global data already at the class level, meaning that each instance of the class `DistributedGeneticAlgorithm` represents just a portion of the whole problem local to a single process.

Initialization Unlike the OpenMP implementation, this version of the Genetic Algorithm requires a non-trivial amount of work for the initialization of the creatures of the first generation. Listing 4 (extracted from `src/Algorithm.cpp`) shows the scattering of the creatures data to all the processes right after serial initialization with random numbers. The root process (also called *master* or *leader*) sends contiguous portions of the population array to each process individually. This procedure is usually implemented through the MPI collective communication primitive `MPI_Scatter` but that would have implied the positions of all the creatures in the population to be stored contiguously in the same vector. Instead, we decided to implement the distributed population as an `std::vector<std::vector<double>>` keeping the access to each creature straightforward.

```
// ...
if (world_rank == 0) {
  for (int i = 1; i < world_size; i++) {
```

```
  const size_t start = i * local_size;
  const size_t end = (i + 1) * local_size;
  for (size_t j{start}; j < end; j++) {
   MPI_Send(creature_positions[j].data(),
    dimensions,
    MPI_DOUBLE,
    i,
    0,
    MPI_COMM_WORLD);
  }
 }
} else {
 creature_positions.resize(local_size);
 for (size_t i{0}; i < local_size; i++) {
  creature_positions[i].resize(dimensions);
  MPI_Recv(creature_positions[i].data(),
   dimensions,
   MPI_DOUBLE,
   0,
   0,
   MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
 }
}
// ...
```

Listing 4: Initialization of population with MPI.

Evaluation Here each process already has a portion of the population, so it iterates serially on it without additional procedures required.

Sorting and Crossover During this phase, all processes send their creatures to the root process, which sorts them according to their fitness score, then sends back the portions of the sorted population to the respective processes. Listing 5 shows how the initial gather and the final scatter of the population are implemented: the root process waits for all processes to send their creatures data along with the fitness scores (for the gather portion) while each process send this data and waits for it to come back after the sorting.

```
void DistributedGeneticAlgorithm::sortCreatures() {
  // ...
 if (world_rank == 0) {
  creature_positions.resize(total_creatures);
  creature_fitnesses.resize(total_creatures);
  for (int i = 1; i < world_size; i++) {
   const size_t start = i * local_size;
```

```
      const size_t end = (i + 1) * local_size;
      for (size_t j{start}; j < end; j++) {
        creature_positions.at(j).resize(dimensions);
        MPI_Recv(creature_positions.at(j).data(),
          dimensions,
          MPI_DOUBLE,
          i,
          0,
          MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
      }
      MPI_Recv(creature_fitnesses.data() + start,
        local_size,
        MPI_DOUBLE,
        i,
        0,
        MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    }
  } else {
    for (size_t i{0}; i < creature_positions.size(); i++) {
      MPI_Send(creature_positions.at(i).data(),
        dimensions,
        MPI_DOUBLE,
        0,
        0,
        MPI_COMM_WORLD);
    }
    MPI_Send(creature_fitnesses.data(),
      creature_positions.size(),
      MPI_DOUBLE,
      0,
      0,
      MPI_COMM_WORLD);
  }

  // Here the root process sorts the creatures

  if (world_rank == 0) {
    for (int i = 1; i < world_size; i++) {
      const size_t start = i * local_size;
      const size_t end = (i + 1) * local_size;
      for (size_t j{start}; j < end; j++) {
        MPI_Send(creature_positions.at(j).data(),
          dimensions,
          MPI_DOUBLE,
          i,
          0,
          MPI_COMM_WORLD);
      }
      MPI_Send(creature_fitnesses.data() + start,
        local_size,
        MPI_DOUBLE,
        i,
        0,
        MPI_COMM_WORLD);
    }
  } else {
    for (size_t i{0}; i < local_size; i++) {
      creature_positions[i].resize(dimensions);
      MPI_Recv(creature_positions[i].data(),
        dimensions,
        MPI_DOUBLE,
        0,
        0,
        MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    }
    MPI_Recv(creature_fitnesses.data(),
      local_size,
      MPI_DOUBLE,
      0,
      0,
      MPI_COMM_WORLD,
      MPI_STATUS_IGNORE);
  }

  // ...
}
```
Listing 5: Sorting of the population with MPI.

After the sorting, the crossover is implemented like the OpenMP one but, obviously, without the preprocessor directives since no shared-memory parallelism is needed here.

Mutation In this phase, each process creates its own random number generator from the given seed, which is supplied from the outside and is different at every iteration of the algorithm. This way each process can use a unique sequence of random numbers and the user may have reproducible executions of the algorithm.

3.3 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.

4 Results

In this section we show the results of some benchmarks we performed to compare the shared-memory implementation and the distributed version of the Genetic Algorithm.

4.1 Benchmark setup

All benchmark results reported in the following paragraphs have been measured on the same machine for which the relevant hardware specification are reported in table 1, while 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

CPU Vendor	Intel
CPU Model	Core i7 7700
CPU core frequency	3.60 GHz
CPU physical cores	4
HyperThreading	enabled
L3 cache size	8 MiB
RAM capacity	16 GB
RAM frequency	2400 MHz

Table 1: Hardware specifications of the machine used for benchmark measurements.

Both benchmark groups used Rosenbrock as objective function, with 10 dimensions, 100 maximum iterations, mutation rate 0.2 and survival rate 0.5 while they vary the number of threads/processes used and the population size.

Both benchmark groups do not use more than 4 threads/processes in order to have a fair comparison between OpenMP and MPI. This limitation is due to the limited number of physical cores on the machine used which did not allow more than 4 MPI processes to be created.

4.2 OpenMP results

Tables 2 and 3 show the total execution time of the OpenMP implementation, along with the average time per iteration, with 10^5 and 10^6 creatures respectively and an increasing number of threads used.

The OpenMP implementation shows a good speedup, without any evident plateau, peaking with 4 cores and 10^6 creatures at 2,32. This result matches our expectations since most of the algorithm is embarrassingly parallel and can therefore use all the available threads.

Threads used	Population size	Total time	Time per iteration
1	100'000	2,456 s	24,56 ms
2	100'000	1,556 s	15,56 ms
3	100'000	1,267 s	12,67 ms
4	100'000	1,143 s	11,43 ms

Table 2: Execution times with 100'000 creatures and increasing number of threads for the OpenMP implementation.

Threads used	Population size	Total time	Time per iteration
1	1'000'000	38,340 s	383,40 ms
2	1'000'000	23,536 s	235,36 ms
3	1'000'000	18,842 s	188,42 ms
4	1'000'000	16,478 s	164,78 ms

Table 3: Execution times with 1'000'000 creatures and increasing number of threads for the OpenMP implementation.

4.3 MPI results

Tables 4 and 5 show the total execution time of the MPI implementation, along with the average time per iteration, with 10^5 and 10^6 creatures respectively and an increasing number of processes used.

It is clear that this implementation of the algorithm, unlike the OpenMP one, has a great bottleneck which does not allow any speedup at all. This may point towards a high communication overhead, since the difference in execution time increases with the number of processes used.

Processes used	Population size	Total time	Time per iteration
1	100’000	3,997 s	39,97 ms
2	100’000	4,446 s	44,46 ms
3	100’000	5,211 s	52,11 ms
4	100’000	5,750 s	57,50 ms

Table 4: Execution times with 100’000 creatures and increasing number of processes for the MPI implementation.

Processes used	Population size	Total time	Time per iteration
1	1’000’000	53,442 s	534,42 ms
2	1’000’000	54,872 s	548,72 ms
3	1’000’000	60,380 s	603,80 ms
4	1’000’000	69,860 s	698,60 ms

Table 5: Execution times with 1’000’000 creatures and increasing number of processes for the MPI implementation.