Particle Swarm Optimization: a parallelized approach

Bortolotti Samuele, Izzo Federico

UNIVERSITÀ DI TRENTO

2022-11-22

Contents

1 Abstract

Particle Swarm Optimization is an optimization algorithm for nonlinear function based on birds swarm. It falls back into the sub-field of *Bio-Inspired Artificial Intelligence* and it was designed from a simplified social model inspired from the nature.

A key concept associated with PSO is the role of genetic algorithms and evolution, the functioning is based on several iterations that aim to identify the best possible position represented as a point in a landscape (Figure 1).

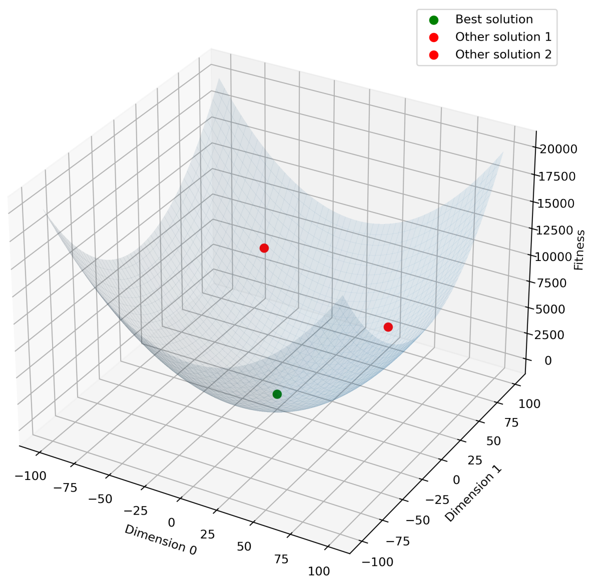


Figure 1: Solution landscape with best possible solution represented in green

PSO is originally attributed to James Kennedy and Russell Eberhart and was first intended for simulating social behavior in 1995.

The goal of this project is to design a parallelized implementation capable of exploring the solution space in a faster way. This is done though the usage of two main libraries for *High Performance Computing (HPC)*: *OpenMPI* and *OpenMP*.

The effectiveness of the proposed solution is tested using the HPC cluster of the University of Trento among other implementations found online.

2 Introduction

2.1 Particle Swarm Optimization

In order to deeply understand the reasons behind the report design choices, it is fundamental to understand comprehensively *Particle Swarm Optimization*.

2.1.1 Generalities

Particle Swarm Optimization focuses on main definitions: the notion of *particle* and the one of *particle perception*.

A particle can be seen as an entity which is characterized by:

- a position x depicting the *candidate solution* for our optimization problem;
- a velocity component v, which is used in order to *perturb* the particle;
- a performance measure $f(x)$, also called *fitness* value, which quantify the quality of the candidate solution.

The entire set of particles is referred as *swarm*.

Under the expression *particle perception*, we define how each particle communicate with each other. In practice, a particle needs to perceive the positions along with the associated performance measures of the *neighboring particles*. Thanks to this communication pattern, each particle remembers the position z associated to the best performance of all the particles within the neighborhood, as well as its own position where it obtained the best performance so far y.

There are different structures of neighborhood which can be considered, and they usually depend on the type of optimization problem one has to face.

The most relevant types of neighborhood are:

- *Global*: the best individual in the neighborhood is also the *global* best in the entire swarm.
- *Distance-based*: based on a proximity metric (e.g. euclidean distance)
- *List-based*: based on a predetermined topology arranging the solution indexes according to some order or structure, and a given neighborhood size.

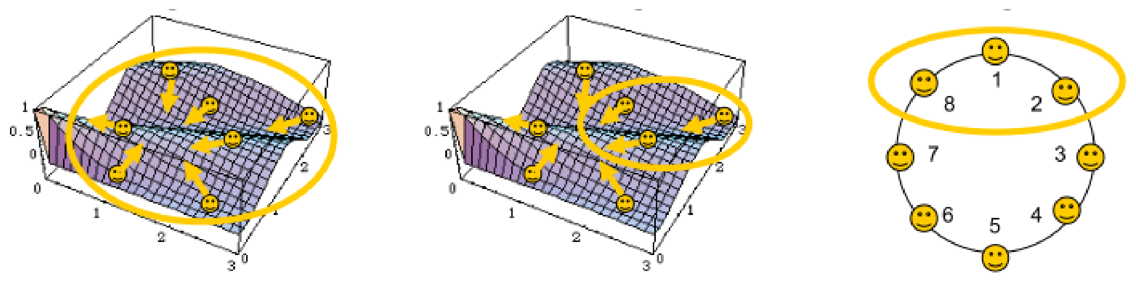


Figure 2: Different neighborhood structures in PSO

This project implements a version of PSO considering *distance-based* neighborhood in a nearest neighbor fashion. In details, each particle has a fixed number of neighbors, which depend dynamically on the particle position on the landscape. The program offers the user the possibility to modify the number of particles to consider within a particle neighborhood.

2.1.2 Parametrization

In order to assess a solution for an optimization problem, PSO requires the following parameters ot be set:

- *Swarm size*: typically 20 particles for problems with dimensionality 2-200;
- *Neighborhood size*: typically 3 to 5, otherwise global neighborhood;
- *Velocity update factors*.

2.1.3 Continuous Optimization

Once the algorithm has been parametrized, a swarm of particles is initialized with random positions and velocity.

At each step, each particle updates first its velocity:

$$v' = w \cdot v + \phi_1 U_1 \cdot (y - x) + \phi_2 U_2 \cdot (z - x)$$

where: - x and v are the particle current position and velocity, respectively; - y and z are the personal and social/global best position, respectively; - w is the inertia (weighs the current velocity)ϕ_1, ϕ_2 are acceleration coefficients/learning rates (cognitive and social, respectively); U_1 and U_2 are uniform random numbers in $[0, 1]$.

Finally, each particle updates its position:

$x' = x + v'$

and in case of improvement, update y (and eventually z).

The loop is iterated until a given stop condition is met.

The pseudocode of the algorithm is shown below:

Algorithm 1 Initialize

```
1:  procedure INITIALIZE($\mathcal{S}, \mathcal{D}, f, v, x, x_{min}, x_{max}, v_{max}$)
2:      for each particle $i \in \mathcal{S}$ do
3:          for each dimension $d \in \mathcal{D}$ do
4:              $x_{i,d} \leftarrow Rnd(x_{min}, x_{max})$                ▷ Initialize the particles' positions
5:              $v_{i,d} \leftarrow Rnd(-v_{max}/3, v_{max}/3)$          ▷ Initialize the particles' velocity
6:          end for
7:      end for
8:      $pb_i \leftarrow x_i$                                             ▷ Initialize the particle best position
9:      $gb_i \leftarrow x_i$                                             ▷ Update the particle's best position
10: end procedure
```

Algorithm 2 Particle Swarm Optimization (Nearest Neighbors)

```
1:  function PSO($\mathcal{S}, \mathcal{D}, MAX\_IT, n, f, v, x, x_{min}, x_{max}, v_{max}$)
2:      INITIALIZE($\mathcal{S}, \mathcal{D}, f, v, x, x_{min}, x_{max}, v_{max}$)        ▷ Initialize all the particles
3:      $it = 0$
4:      repeat
5:          for each particle $i \in \mathcal{S}$ do
6:              if $f(x_i) < f(pb_i)$ then
7:                  $pb_i \leftarrow x_i$        ▷ Update the particles' best position
8:              end if
9:          end for
10:         $\mathcal{S}' = $ COPY($\mathcal{S}$)        ▷ Copy the particle's vector
11:         for each particle $i \in \mathcal{S}$ do
12:             $\mathcal{S}'$ = SORT(S', i)        ▷ Sort the particles w.r.t. $i$th particle
13:             for each particle $j \in \mathcal{S}'$ do
14:                 if $f(x_j) < f(gb_i)$ then
15:                     $gb_i \leftarrow x_j$
16:                 end if
17:             end for
18:         end for
19:         for each particle $i \in \mathcal{S}$ do
20:             for each dimension $d \in \mathcal{D}$ do
21:                 $v_{i,d} = v_{i,d} + C_1 \cdot Rnd(0,1) \cdot [pb_{i,d} - x_{i,d}] + C_2 \cdot Rnd(0,1) \cdot [gb_d - x_{i,d}]$
22:                 $x_{i,d} = x_{i,d} + v_{i,d}$        ▷ Update the velocity and positions
23:             end for
24:         end for
25:         $it \leftarrow it + 1$        ▷ Advance iteration
26:     until it < MAX_ITERATIONS
27:     return x
28: end function
```

2.2 OpenMPI

OpenMPI library is used to convey information across processes running on different nodes of a cluster. The basic information unit is composed as a broadcast message shared over the whole network, in this way all particles of Particle Swarm Optimization (PSO) are able to know all information associated to other members of the swarm.

The process that produces the message sends the message using a gather function because all particles must know the positions of other individuals of the population at every step. In the following picture it is presented a simple schema of the communication.

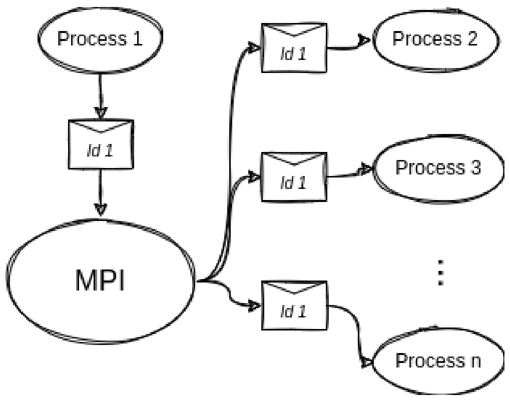


Figure 3: Communication schema

2.3 OpenMP

A process can have the task of computing the algorithm for one or more particles, it is divided in several threads that optimize the execution time of the process.

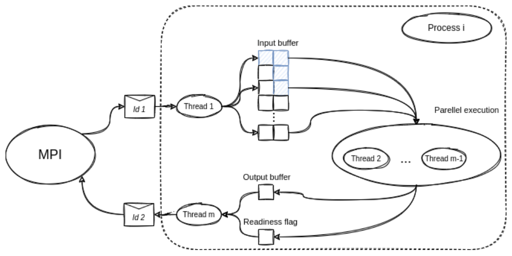


Figure 4: Execution schema

OpenMP is an application programming interface (API) which supports multi-platform shared memory multiprocessing programming.

2.4 Project generalities

In the following sections, the report address how to setup and run the program.

2.4.1 Libraries

The project requires few libraries in order to work properly. As it is mandatory for the course, OpenMP and MPI were employed. Along with the compulsory libraries, the following libraries were exploited:

- sqlite: SQLite is a C-language library that provides a SQL database engine that is tiny, quick, self-contained, high-reliability, and full-featured. The choice of `sqlite` was made in order to save particles' information at each iteration in an simple and fast way, avoiding dealing with race conditions.
- argp: `argp` is a parsing interface for unix-style argument vectors. The argp features include, as defined in the GNU coding standards, automatically producing output in response to the '–help' and '–version' options and the possibility for programmers to explicitly define the input of the script. This library was employed in order to allow the user to explore the possible configurations made available by the software.
- check: `check` is a unit testing framework written in C. It has a straightforward interface for defining unit tests helping the developer to build robust software. This library was included in the application in order to perform unit-testing on the structure we have created. This choice implication are a more robust software.

2.4.2 Build

In order to build the executable file of our project, as well as the binary file needed to run the project unit test, we have employed GNU Make.

GNU Make is a tool which manages the creation of executables and other non-source files from a program's source files. Make learns how to create your software using a file called the `Makefile`, which lists each non-source file and how to compute it from other files.

Thanks to the definitions of rules, Make enables the user to build and install packages without knowing the details on how that is done.

Moreover, thanks to wildcards, it is easy to automatize the application building process. Indeed, it first allow to assemble each `C` source file in order to create the object files. Then, all of the object files are linked together, along with other libraries, in order to produce the final executable file. If the building rule is called multiple times, Make is smart enough to understand whether an object file needs to be recreated or no, making use of the already assembled objects, thus speeding up the building process.

Furthermore, Make can do much more than compiling software, for instance, the project contains rules which allow to build and open the code documentation written by the means of Doxygen.

In order to get the right flag for linking the needed external, the project employs pkg-config. This package collects metadata about the installed libraries on the system and easily provides it to the user.

2.4.2.1 Compile

To compile the project, it is possible to call the Makefile by typing:

```
make build
```

In this way, the executable `bin/particle-swarm-optimization` is ready to be launched.

Instead, to build the unittest, it is possible to execute the following command.

```
make test
```

The artifact is located in the `bin` directory and it is called `test`.

Along with the executable files, there are also scripts used in order to run the program within the University cluster. Each job in the cluster is handled by *PBS (Portable Bash Script)* which submits them to the scheduler. By means of a script, it is possible to tell the scheduler what resources the job requires in order to complete (e.g. number of processors, amount of memory, time to complete etc.) and the application the user wants to run.

The `run.sh` file in the `scripts` folder of the repository allows the user to submit the application to the cluster. The script has three parameters: number of processes, path of the ini file containing the program configuration and the number of threads. Once submitted with the `qsub` command, the script generates a number of docker containers equal to the number of specified processes thanks to the `mpiexec` binary. Each container runs the application in a shared network, therefore each process is able to communicate with each other. The details of the program deployment is discussed in the section dedicated to DevOps.

The `generate_cluster_run.sh` file, contained in the `scripts` folder, is employed in order to generate specific runs in order to benchmark the application. In details, the shell file considers several combinations of processes, threads, nodes and places. More details are provided in the section dedicated to the application benchmark.

2.4.3 Execute

The executable file can be invoked with or without `OpenMP` and with or without `OpenMPI`. However, to fully exploit `OpenMPI`, it is recommended to execute the program `mpiexec` to spawn multiple processes of the multi-process application.

The executable file requires several arguments. Below there is an excerpt of the program output when the `--help` flag is called.

```
A Cooperating parallelized solution for Genetic Algorithm. A tool that
    takes a set of continuous or discrete variables and an optimization
problem designed to work with them. The goal is to find the optimal
    solution by
exploiting Genetic Algorithms and the computational power offered by
    the cluster

  -m, --number-of-threads[=COUNT]
                             Number of threads for process
  -u, --use-openmpi          Use OpenMPI
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

In order to run, the application requires three parameters, two which are optional, while one is mandatory.

The compulsory parameter is the configuration file, which needs to be provided in an `INI file`. This file, takes care of all the parameters which are needed by the Particle Swarm Optimization algorithm to run and which have been fully discussed during the introduction to the problem. The repository provides a standard `INI` file, called `pso-data.ini`, which can be modified in order to configure algorithm so as to solve the target problem.

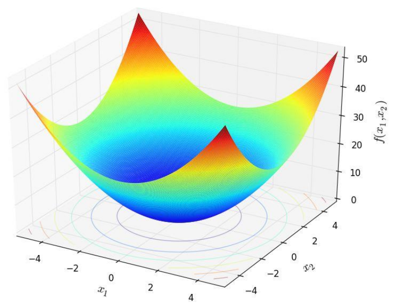
It is possible to specify the number of threads the program is allowed to spawn with the `-m` flag, and whether to employ `MPI` primitives or not with the `-u` flag.

3 Problem Analysis

As explained during the introductory part, the main focus of the PSO algorithm is to find an approximate solution of a continuous optimization problem. Therefore, we have relied on some of the most relevant benchmark functions for continuous optimization. The experiments focuses mostly on six of them, which are listed below:

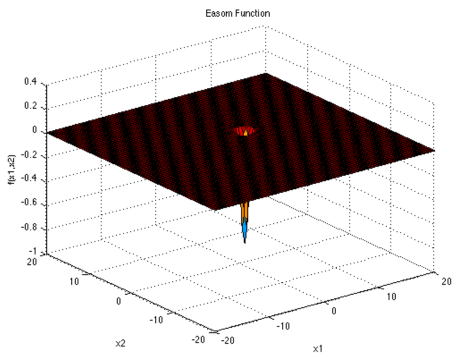
- `sphere function`: unimodal function suitable for single objective optimization. The single optimum is located in $\vec{x} = \vec{0}$. The sphere function is defined as follows:

$$\vec{x} \text{ argmin} f(x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} x_i^2$$



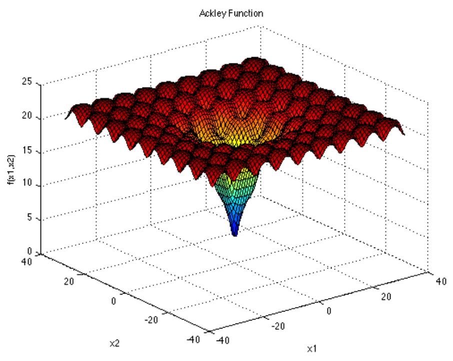
- `wave function`: TODO
- `Easom function`: ultimodal function suitable for single objective optimization. The single optimum is located in $\vec{x} = \vec{\pi}$. The Easom function is defined as:

$$f(x) = -\cos(x_1)\cos(x_2)\exp(-(x_1-\pi)^2-(x_2-\pi)^2)$$



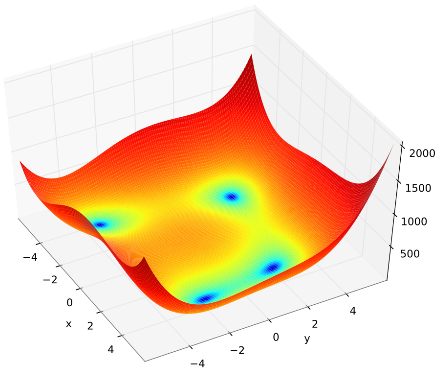
- `Ackley function`: unimodal function suitable for single objective optimization. The single optimum is located in $\vec{x} = \vec{0}$. The Ackley function is defined as:

$$f(x) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1} d\cos(cx_i)\right) + a + \exp(1)$$



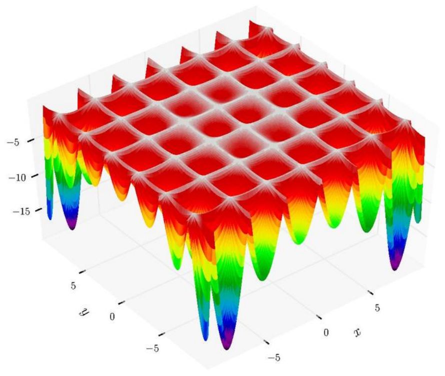
- `Himmelblau function`: multimodal function suitable for single objective optimization. The function presents four identical minima: $f(\vec{x}) = f(3.0, 2.0) = f(-2.805118, 3.131312) = (-3.779319, -3.283186) = f(3.584428, -1.848126) = 0.0$ The function is defined as:

$$f(x, y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2$$



- `Holder table function`: multimodal function suitable for single objective optimization. The function presents four identical minima: $f(\vec{x}) = f(8.05502, 9.66459) = f(8.05502, 9.66459) = (8.05502, 9.66459) = f(8.05502, 9.66459) = -19.2085$. The function is defined as:

$$f(x) = -\left|\sin(x_1)\cos(x_2)\exp\left(1 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi}\right)\right|$$



For the sake of the explainability, the functions presented above shows the two-dimensional case. Of course, such functions can scale up to as many dimensions as one desires.

Moreover, our particle swarm optimization implementation can handle also other functions. Indeed, it is possible to define the proper single objective function to optimize in the `problems.h` and `problems.c` files, specifying it in the configuration along with the *fitnessGoal*, namely, whether the function needs to be maximized or minimized.

3.1 Serial version of the algorithm

As can be seen from the PSO pseudocode shown in the introduction, the main steps the algorithm has to face are:

1. initialize the particles in the swarm according to the problem dimensionality;
2. exchange particles' positions among within the swarm;
3. sort the particles according to a distance measure (euclidean distance) in ascending order;
4. update the particles position and velocity.

3.1.1 Serial algorithm optimization

As mentioned in the introduction section, the program provides the possibility to either run on a single thread or on multiple threads. The multithreading functionalities exploited by the application are offered by the OpenMP library.

As a first approach, we have tried to use OpenMP directives in order to generate a thread for each loop iteration whenever it was possible.

To prevent contaminating the spaces of other threads, the actions inside the *for* loop worked on distinct data structures and variables.

For example, we inserted the relative OpenMP directive every time there was the need to loop though all the particles.

However, OpenMP *fork-join* model requires a non negligible overhead so as to spawn multiple threads which are eventually joined into the master at the end of the OpenMP block. For relatively small problems, this operation was a time-consuming procedure which leads to a significant rise in execution time with respect to the single thread model. Moreover, during the experiments we have not been able to observe the threads advantage we were hoping for. We assume that the main reason behind this non-tangible advantage are the optimization provided by `gcc` during at compile time and the non.optimal thread allocation patterns performed on the cluster. Indeed, it is not rare to observe different threads being executed on the same computational unit, which clearly slows down the computation due to the overhead required by the context switching operation.

In the final version of the application, we have included the OpenMP directives only in the portion of the code where we thought it was needed, even if the advantage in terms of time were not satisfactory compared to the single threaded application.

3.1.1.1 Exchange particles' information In the serial version of the application, each particle is stored within an array data structure, therefore with a two-level nested loop, it is possible to make every particle contribution to each other. In this case, as each particle handles a different portion of the memory, a *pragma for* directive is included.

3.1.1.2 Sorting algorithm Concerning the sorting algorithm, the program relies on *quicksort*. The main reason behind this choice is the amount of parallelization this algorithm can provide. Indeed, merge sort has a better worst-case performance $\mathcal{O}(n \log n)$ with respect to quicksort having $\mathcal{O}(n^2)$ but requires synchronization in order to merge the partial solutions which is not required in quicksort. Moreover, its average performance is $\mathcal{O}(n \log n)$ as for merge sort.

The parallel quicksort main working loop can be described as follows. Before going into the implementation details, in the base settings we have a pool of threads provided by the OpenMP library which can be called whenever is needed, and an array of items that needs to be sorted.

Initially, one thread selects a pivot and moves the elements of the array which are smaller than the pivot value to the left and the elements which are bigger than the pivot value to the right. The resulting two portions of the array wait until there is a free thread ready to process it.

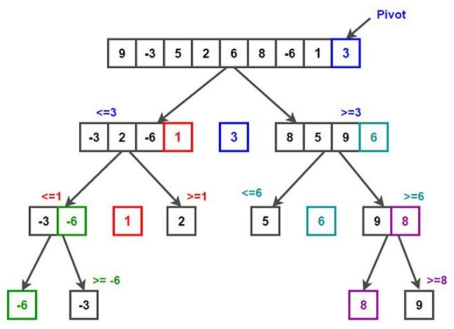


Figure 5: Quicksort

3.2 Parallel version of the algorithm

3.2.1 Architecture

3.2.2 Communication pattern

3.2.3 Message

3.2.4 Logs

3.2.5 Output and SQLite

4 Benchmarking

5 Conclusion

5.1 Is parallelization always a good choice?

Well yes, but actually no.

5.2 Thread allocation pattern