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 by 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).

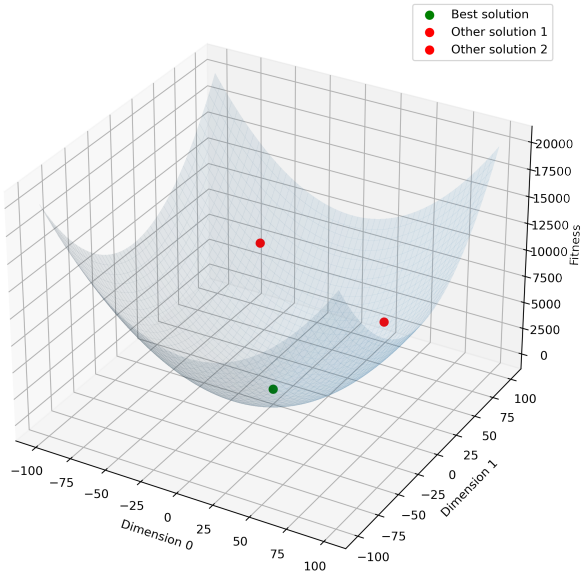

Figure 1: Solution landscape with best possible solution represented in green

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.

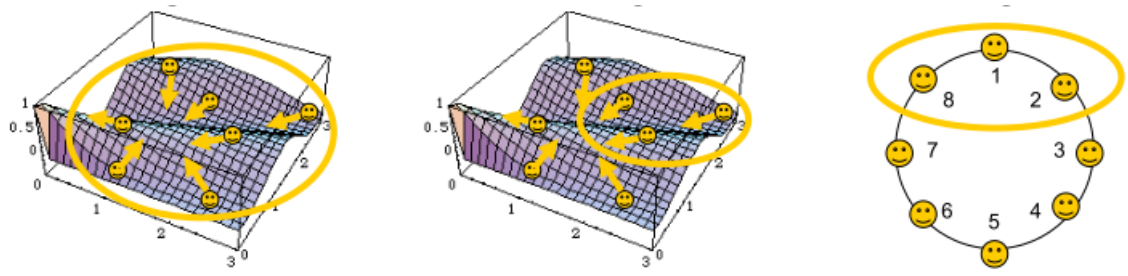

Figure 2: Different neighborhood structures in PSO

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

procedure INITIALIZE($\mathcal{S}$, $\mathcal{D}$, f, v, x, x_{min}, x_{max}, v_{max})
  for each particle $i \in \mathcal{S}$ **do**
    for each dimension $d \in \mathcal{D}$ **do**
      $x_{i,d} \leftarrow Rnd(x_{min}, x_{max})$ ▷ Initialize the particles' positions
      $v_{i,d} \leftarrow Rnd(-v_{max}/3, v_{max}/3)$ ▷ Initialize the particles' velocity
    end for
  end for
  $pb_i \leftarrow x_i$ ▷ Initialize the particle best position
  $gb_i \leftarrow x_i$ ▷ Update the particle's best position
end procedure

Algorithm 2 Particle Swarm Optimization (Nearest Neighbors)

function PSO($\mathcal{S}$, $\mathcal{D}$, MAX_IT, n, f, v, x, x_{min}, x_{max}, v_{max})
  INITIALIZE($\mathcal{S}$, $\mathcal{D}$, f, v, x, x_{min}, x_{max}, v_{max}) ▷ Initialize all the particles
  $it = 0$
  repeat
    for each particle $i \in \mathcal{S}$ **do**
      if $f(x_i) < f(pb_i)$ **then**
        $pb_i \leftarrow x_i$ ▷ Update the particles' best position
      end if
    end for
    $\mathcal{S}' = $ COPY($\mathcal{S}$) ▷ Copy the particle's vector
    for each particle $i \in \mathcal{S}$ **do**
      $\mathcal{S}'$ = SORT(S', i) ▷ Sort the particles w.r.t. ith particle
      for each particle $j \in \mathcal{S}'$ **do**
        if $f(x_j) < f(gb_i)$ **then**
          $gb_i \leftarrow x_j$
        end if
      end for
    end for
    for each particle $i \in \mathcal{S}$ **do**
      for each dimension $d \in \mathcal{D}$ **do**
        $v_{i,d} = v_{i,d} + C_1 \cdot Rnd(0,1) \cdot [pb_{i,d} - x_{i,d}] + C_2 \cdot Rnd(0,1) \cdot [gb_d - x_{i,d}]$
        $x_{i,d} = x_{i,d} + v_{i,d}$ ▷ Update the velocity and positions
      end for
    end for
    $it \leftarrow it + 1$ ▷ Advance iteration
  until it < MAX_ITERATIONS
  return x
end function

2.2 MPI

The MPI (Message Passing Interface) library is used to convey information across processes running on different nodes of a cluster.

In the scenario described by the application, 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.

2.3 OpenMP

OpenMP is an API which supports multi-platform shared memory programming.

In the program scenario, a process is delegated to handle the computing regarding one or more particles. The process job is divided in several threads which optimize the execution time of the process.

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 not, 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. Hence, `pkg-config` takes care of where a library is located regardless of the distribution simplifying the application building process.

2.4.2.1 Compile

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

```
1 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.

```
1 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:

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

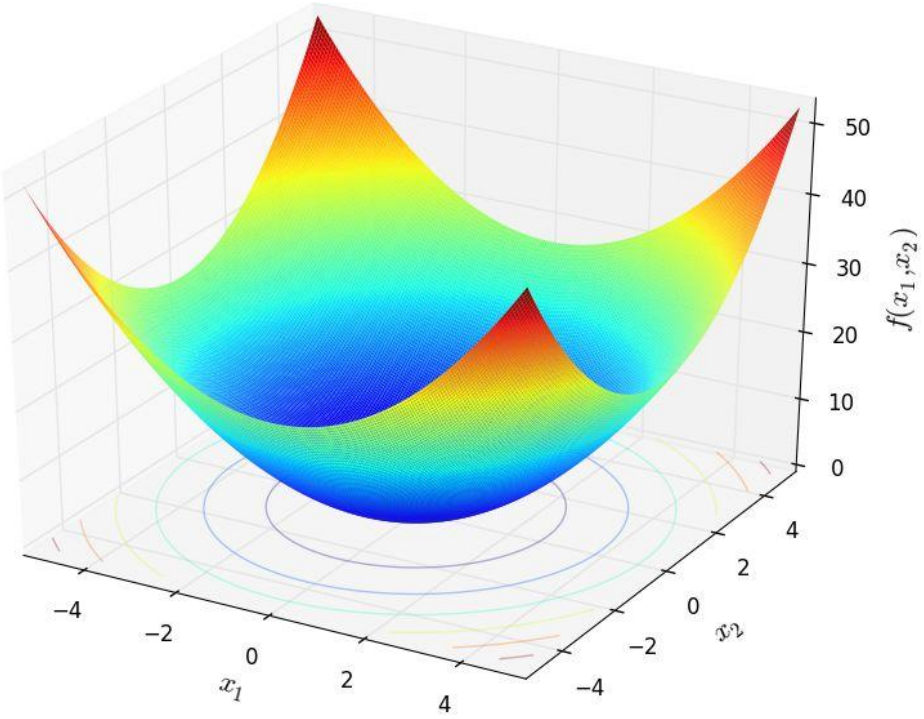

Figure 3: sphere function

- `wave function`: unimodal function suitable for single objective optimization. The function does not admit a single optimum, however the fitness increases as long as x approaches to $-\infty$. The wave function is defined as follows:

$$f(x, y) = x^3 + y^2$$

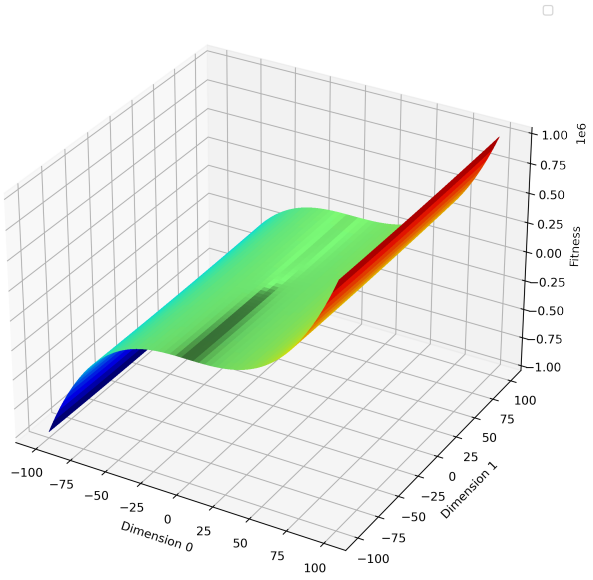

Figure 4: Wave function

- `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:

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

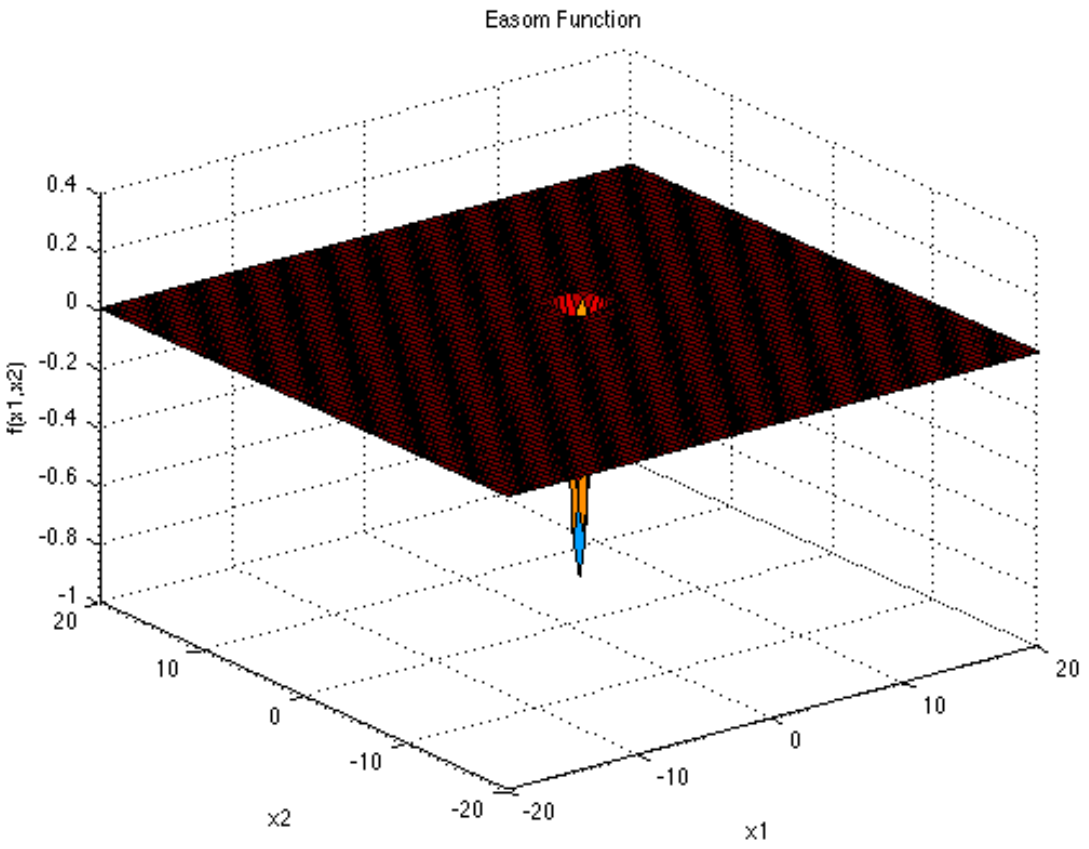

Figure 5: Easom function

- `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:

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

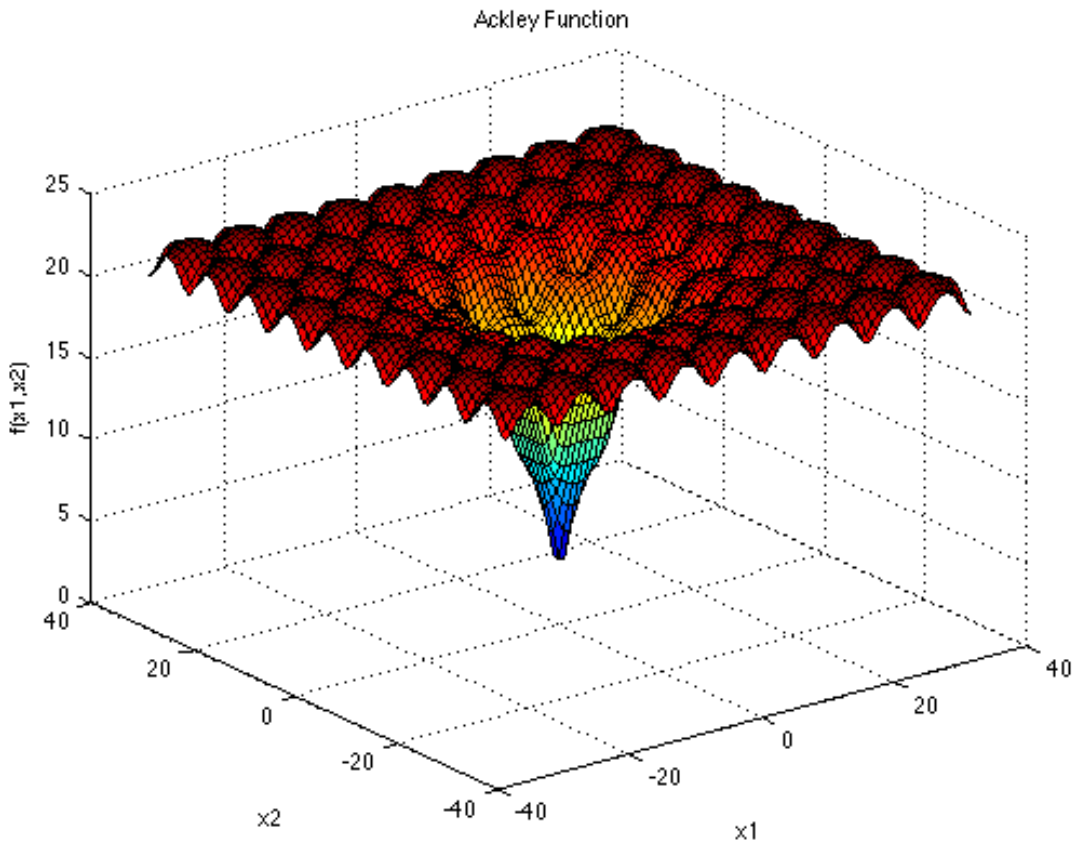

Figure 6: Ackley function

- `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:

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

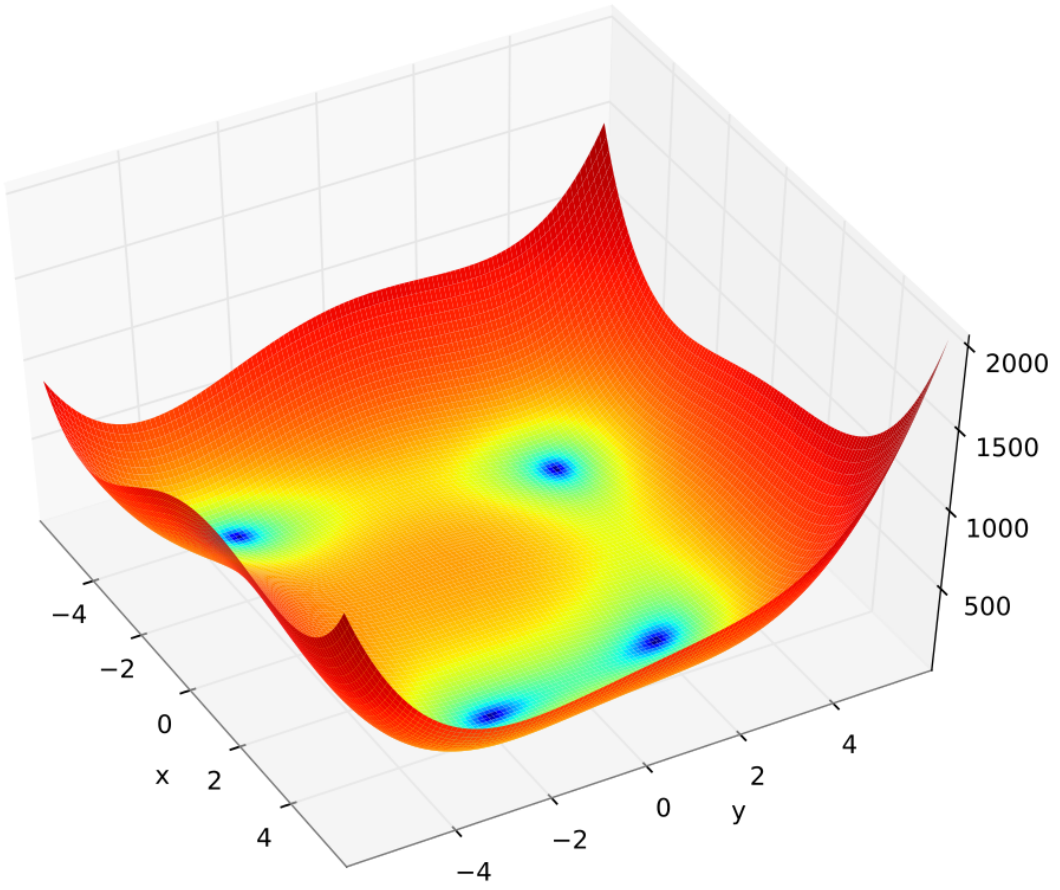

Figure 7: Himmelblau function

- `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:

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

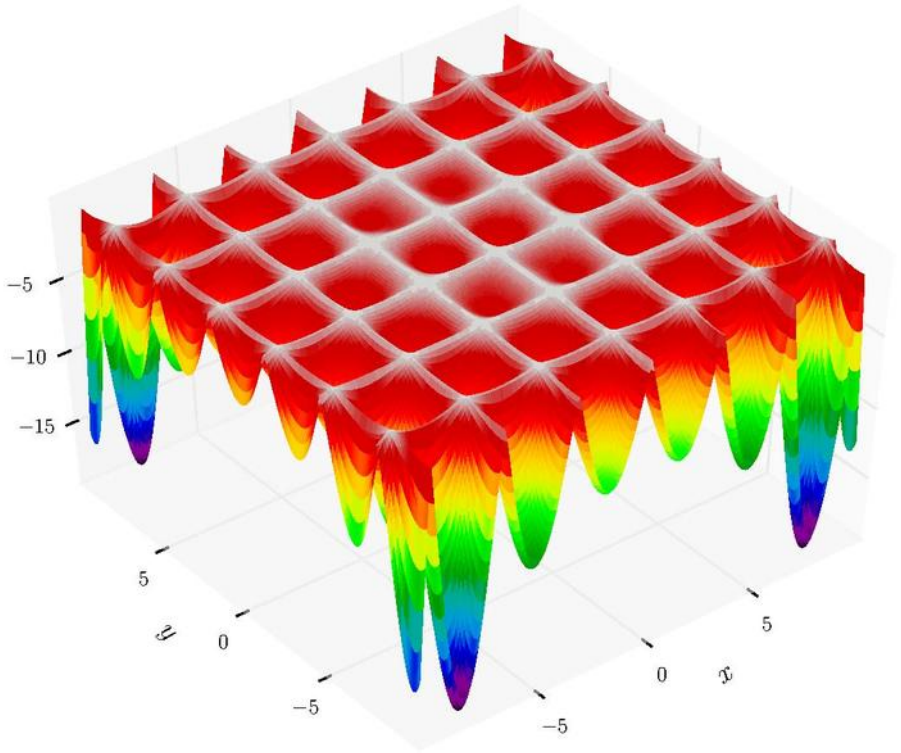

Figure 8: Holder table function

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 Program configuration

3.2 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.

3.2.1 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.

3.2.1.1 Particles' initialization Since only one process is involved, the initialization step is trivial, as it requires to create a given number of particles sequentially, which are immediately stored within an array data structure.

3.2.1.2 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.

3.2.1.3 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.

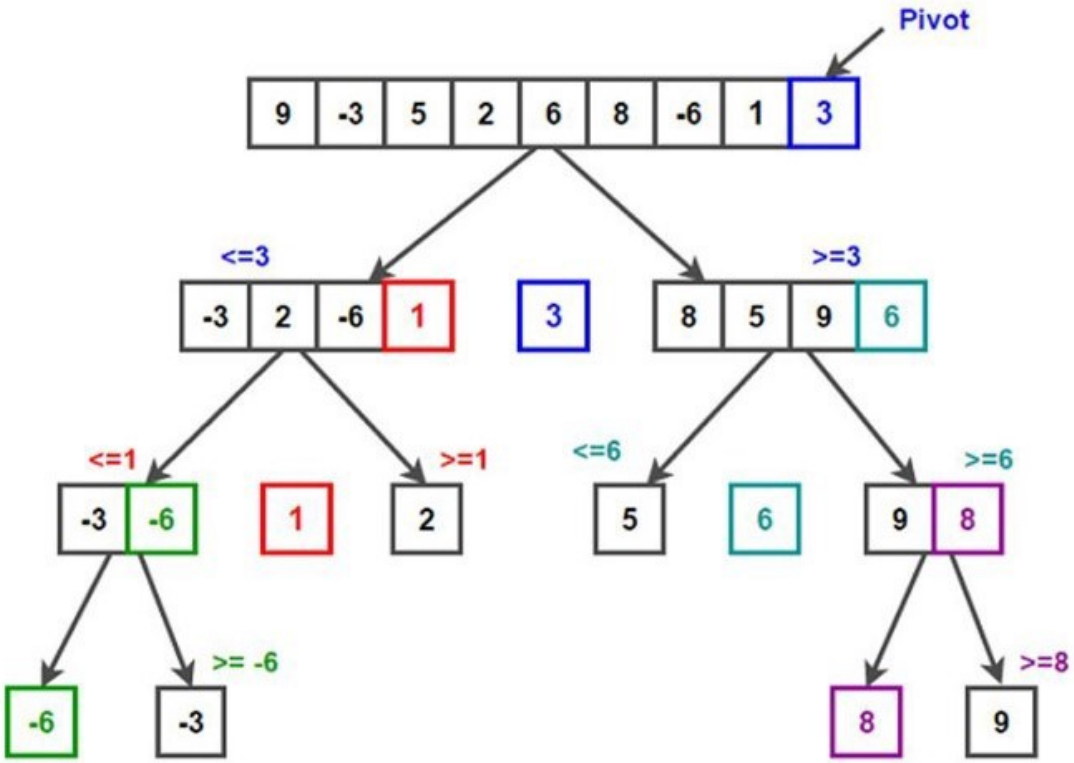

Figure 9: Quicksort

3.2.1.4 Velocity and position update As for the initialization step, the process consists in a for loop which scans all the particles' data, and applies the formula according to the algorithm list in the introduction section.

3.3 Parallel version of the algorithm

In this section the report discuss how we have parallelized the algorithm in order to speed up the performance.

In practice, we have distributed the workload among N different processes in the cluster using the *MPI* library and we have exploited multiprocessing via OpenMP for a couple of different shared-memory tasks.

3.3.1 Architecture

In order to subdivide the work and to carry out the final computation, the architecture proposed by the report focuses on the *all-to-all* parallel computational pattern.

All-to-all parallel pattern is characterized by the exchange of individual messages from every process to any other processor. In this way, the program effectively uses all the processes in order to carry out the computation, as the coordination operations are handled by MPI.

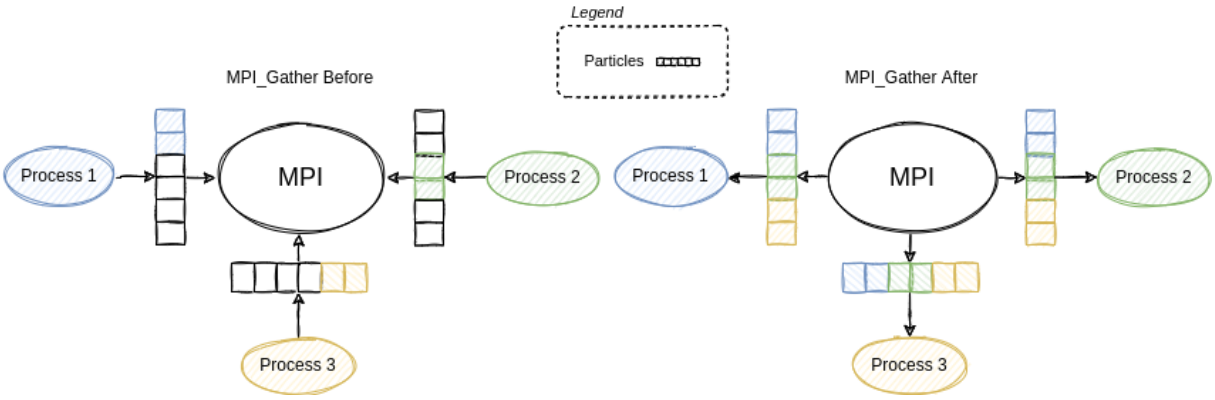

Figure 10: Communication schema.

3.3.2 Message

To send a message between different processes we created a custom MPI data type called broadcastMessage_t. Its purpose is to inform the receiver process about the particles' position and fitness of the sender. The structure is composed by a timestamp, which is needed for logging purposes on the sqlite, the current iteration of the algorithm, the identifier of the particle, the sender rank and the current solution.

On the other hand, solution is another structured datatype, which has been defined in order to carry a specific particle information. This structured datatype contains the problem dimensionality, the fitness value of the candidate solution and the vector depicting the current particle position within the fitness landscape.

So as to create a variable of the previously mentioned message data type, we have defined a proper function called define_datatype_broadcast_message. This function, in turn, calls the function required to defined a message carrying the a solution type variable and a timestamp type variable.

The stratification above has simplified the MPI_Datatype definition process.

3.3.3 Communication pattern

The communication between the different processes is synchronous.

Firstly, each process takes charge of a given number of particles. In details, let N be the number of particles the user has requested to program to manage and let p be the number of processes available to *MPI*. Without the need of synchronization nor of message exchange, each process creates N/p particles and the remaining $N\%p$

ones are split among the remaining processes. This was possible by exploiting the number of processes and the process rank in the following way:

```
int particlesNumberPerProcess = particlesNumber / n_processes;
int particlesNumberReminder = particlesNumber % n_processes;
int processToNumberOfParticles[n_processes];
for (int i = 0; i < n_processes; i++) {
  processToNumberOfParticles[i] = particlesNumberPerProcess;
  if (i != 0 && i <= particlesNumberReminder)
    processToNumberOfParticles[i]++;
}
```

In this way, the i-th rank process has `processToNumberOfParticles[i]` particles to handle and, at the same time, knows how many particles the other processes have to manage which will be useful later when handling message communication. The possibility of coordinate the processes without synchronization calls or messages provides a relevant improvement in the application performances.

The most interesting part in the algorithm parallelization is the program segment related to the message exchange among multiple processes.

To carry out this operation, each process embeds its own particles in an array of `define_datatype_broadcast_message` . Then, the particle information exchange happens with an `MPI_Allgather` communication primitive.

In principle, at the beginning of the algorithm execution, the set of all the particles have been distributed across all processes. However, this operation has been carried out by each process alone without the need for a `MPI_Scatter` call.

`MPI_Allgather` primitive is suitable for the problem since it is an *all-to-all* communication primitive and since it allow to reunite all the particles of each process into a single vector, which, at the end of the communication, will be equal for each process. A scheme illustrating the working behavior of the communication primitive employed is shown below:

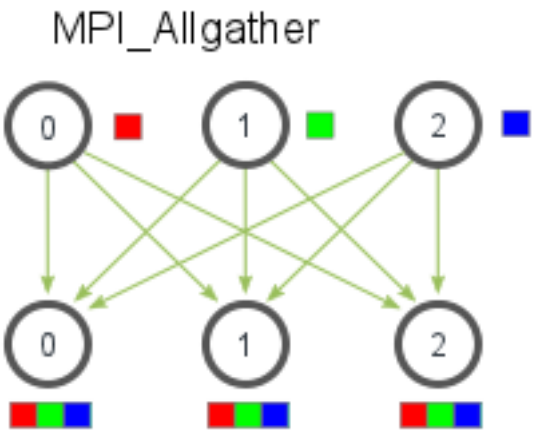

Figure 11: Allgather

Once each process knows everything about the others, the application needs to consider the neighbor contributions in order to update the process particles' position and velocity.

At this point, each process can sort all the particles, whose position is known thanks to the `MPI_Allgather` communication, with respect to all particles proper to the process, according to the euclidean distance. In this way, for each process particle is possible to identify the k-th nearest neighbors.

Finally, by applying the position and velocity update formulas listed in the PSO pseudocode it was possible to evolve the algorithm and approach the target function optima.

Moreover, with the help of OpenMP we have parallelized the computation of the sorting algorithm as well as the loop needed in order to update the algorithm variables.

We have observed that the workload split implied by the algorithm enhance the PSO performances. A first, and trivial way to observe the advantage offered by *MPI* and optimization is to give a look at some of the expensive computation the serial algorithm has to carry out, in particular, let n be the number of particles and let m be the problem dimensionality. Moreover, during this reasoning process, we consider the average performance case and the single threaded scenario.

1. the distance computation is quadratic in the number of particles, while the euclidean distance is liner in the vectors' number of dimensions, hence the complexity is $\mathcal{O}(mn^2)$. In the application scenario, the distance needs to be computed a $\mathcal{O}(n^2)$ number of times, as it is required to know the relative position of each particle with respect to all the others. Hence, the complexity grows to $\mathcal{O}(mn^4)$;
2. the sorting algorithm, in the average performance scenario has a complexity which is given by $\mathcal{O}(n \log n)$, where n is the number of particles. This operation has to be performed with respect to all the particle in the swarm, increasing the complexity to $\mathcal{O}(n^2 \log n)$;
3. finally, the particle's update is linear in the dimensionality of the problem, hence the complexity is $\mathcal{O}(m)$, which has to be performed for all the particle in the swarm, bringing the complexity to $\mathcal{O}(nm)$.

All those operations, are quite expensive in terms of time-complexity. Thanks to the workload sharing we have set up, the number each process has to manage is drastically reduced, and decreases the more processes MPI has at its disposal. Let p be the number of processes. On average, each process has to manage n/p particles, while the problem dimensionality remains untouched.

The complexity then decreases as follows:

1. $\mathcal{O}(m * (n/p)^4)$ for the distance computation;
2. $\mathcal{O}((n/q)^2 \log(n/q))$ for the sorting operation;
3. $\mathcal{O}((n/q)m)$ for the particles' variables update.

Despite this results being positive, we have to consider the time needed for each process to exchange their particle to each process. However, most HPC systems use *InfiniBand* interconnection, an high throughput, low latency connection among nodes in the cluster, therefore we claim that the advantage in terms of computational complexity remains legitimate since, in this scenario, the network has little impact on the application performance.

A visual proof of this statement is deeply discussed in the benchmarking section of the report.

All the previously described operations are executed for a specific number of times specified by the user.

3.3.4 Logs

In order to provide a more effective way of visualizing the program behavior, we have employed a thread-safe logging utility library.

In this way, we were able to always know each process state. The logging library provides six different logging level: trace, debug, info, warn, error and fatal.

An excerpt of the logging library output is shown below:

TODO qua vanno i logs

Moreover, all the logs have been formatted in order to comply to a common standard. In this way, during the benchmarking phase, it was possible to extract and manage logs information.

3.3.5 Output and SQLite

The final output of the program are is best particles fitness value found by every process in the system.

In a real case scenario, one would be more interested in the candidate solution found by the application rather than in the fitness value. However, the fitness function is a suitable value to analyze in order to have a clear understanding on whether the model is improving its solution or not. Moreover, the particles position at every iteration can be stored within the SQLite database.

In this way, it is possible to recover the path the program has followed in order to build the final solution, and eventually choosing the most suitable point for the user application.

4 Benchmarking

After implementing the algorithm, we wanted to understand how our parallelization impacts the performance of the algorithm.

Hence, we devised a full analysis of our algorithm performance changing the number of threads, the number of processes and the pBS process allocation pattern, in order to understand how the running time would have been affected.

4.1 Problem configuration

In order to understand how our parallelization improved the running time, we devised a configuration file which is the same for every run, so as to have a common baseline.

The configuration is listed below:

```
[global]
problemDimension = 50
particlesNumber = 5000
iterationsNumber = 500
neighborhoodPopulation = 5000
```

```
6
7  [velocity]
8  w = 0.8
9  phi_1 = 0.3
10 phi_2 = 0.3
11
12 [randomBounds]
13 initMaxPosition = 500.0
14 initMinPosition = -500.0
15 initMaxVelocity = 100.0
16 initMinVelocity = -100.0
17
18 [functions]
19 fitness = sphere
20 distance = euclidean
21 fitnessGoal = minimum
```

The amount of particles and the neighboord population are unreasonable for any known problem but they were chosen to show clearly the advantages brought by a multi process solution.

4.2 Cluster jobs

In order to have high-quality and trustworthy results to examine, as indicated in the repository structure, we created a script that allowed us to send thousands of tasks to the University's HPC cluster over the course of many days. The script keeps a limit on the user current submitted job in order to do not monopolize the cluster and comply with the cluster policies. Precisely, we have set a 15 process limit and every 10 seconds the script checks whether the user has 15 or more running processes in the cluster. If it is the case, then the script waits, otherwise it submits a new job to the scheduler.

The total number of tests we have ran in total is around around 960, in particular we tried every possible combinations of different parameters:

- processes: chosen between `[1 2 4 8 16 32 64]`
- threads: chosen between `[1 2 4 8 16 32 64]`
- select: chosen between `[1 2 3 4 5]`
- places: chosen between `[pack scatter pack:excl scatter:excl]` TODO: spiegare select and places

select equivale al numero di chunk (un chunk un insieme core/socket non necessariamente sulla stessa macchina). ncpus e' il numero di core che voglio per ogni chunk. pack mette tutti i chunk sulla stessa macchina, scatter li divide su macchine diverse SEMPRE mentre le versioni excl riservano gli interi ?nodi/chunk? alla tua creazione

We decided to increase the processes and threads number by power of 2 because the most complex algorithm (following the asymptotic notation) should be $\mathcal{O}(nlogn)$ (TODO verificare questa complessita')

Since it takes 50 minutes on average for each run, we had to stretch out the submission of jobs across many days (two weeks).

4.3 Results

During the execution of the bencmarking phase two problems occured:

- one student saturates the available space in the home of the cluster and for this reason some of our runs were not able to write results on disk at the end of the execution;
- we decided to sumbit all jobs to the short job queue of the cluster, and some jobs failed for *time exceed* error or missing resources.

All configurations were launched by both of them in order to validate results and reduce possible noise. In Table 1 it is presented the amount of jobs and the associated fail rate (it is shown only the amount of cores for presentation reasons)

Processes number	Total	Failed	Fail rate
1	110	18	16.36
2	109	51	46.79
4	104	52	50.00
8	105	59	56.19
16	105	64	60.95
32	91	63	69.23
64	56	43	76.79

5 DevOps

In order to automate the build process of documentation, executable, and report a *Continuous Integration* workflow was designed.

5.1 Nix

The first step of the workflow is composed of a package manager called *Nix*. It is designed to create environments ensuring reproducibility across different machines.

The behavior of nix commands is described inside the `flake.nix` file, it also contains all the dependencies required for the execution of each possible scenario. To enforce the reproducibility in the future all required packages are fixed inside the `flake.lock` file.

The build entry points offered by nix are:

- `nix build .#report`: builds this report from Markdown files using pandoc and a latex template;
- `nix build .#particle-swarm-optimization`: builds the final binary;
- `nix build .`: same as `nix build .#particle-swarm-optimization`.

There exists also a development entrypoint that can be called using the command `nix shell`, it provides a new shell containing all dependencies required for build, testing, and report.

5.2 Docker

The reproducibility during execution of the final binary is provide by *containers*. A container is sand box that contains all runtime dependencies required by the executable placed within it.

A manager for the life-cycle of containers is required, we opted for *Docker*, a state-of-the-art software used in many environments. It builds an image from the descriptor file `Dockerfile` placed in the root of the project, then from the image it is possible to create several containers that are independent of each other.

The main problem encountered was the root permission required by docker for the build operation. This prevented us to use it inside the cluster. To overcome this requirement we had to setup other technologies described in the following sections.

5.3 GitHub actions

A *GitHub action* is a list of commands executed when a specific event is triggered within a repository. It is possible to define multiple actions (also on different files) inside the `.github/workflows` directory.

We defined two workflows, one for the container creation and the other for the documentation compilation. They are triggered on each commit.

5.3.1 Container creation

The container creation workflow uses the `Dockerfile` to build up a docker image (as explained in Section 6.2) that is then pushed to a *container registry* called *DockerHub*. The process also includes an automation that executes unit tests, if only one them is marked as failed then the whole process is interrupted to avoid runtime errors in production.

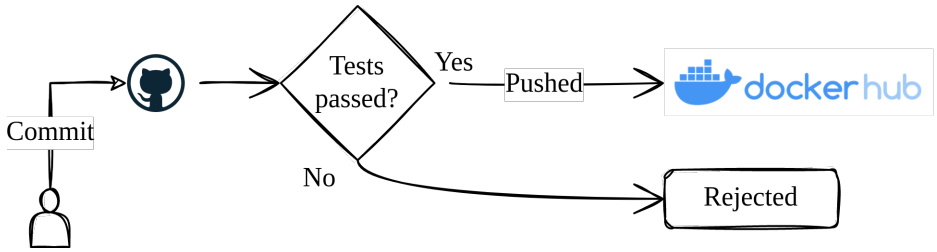

Figure 12: Container creation workflow

5.3.2 Documentation compilation

The documentation is compiled in two formats:

- **html**: it includes all the content of this report and in addition it also provides code documentation extracted with *Doxygen* following rules defined in the `Doxyfile` file placed in the root of the project;
- **pdf**: this report.

Both formats are generated using a github action and the html website is hosted on *GitHub Pages* at this link.

5.4 Udocker

The final step of the workflow is represented by *Udocker*, it is a container manager like Docker but it does not require root permission for the execution phase.

During the usage inside the cluster we encountered two main problems:

- the build phase;
- the OpenMPI communication between independent sandboxes.

5.4.1 Build phase

Unfortunately *udocker* does not offer primitives for containers building, it only provides one intake operation that is a pull from a container registry. For this reason we created the workflow described in Section 6.3.1.

The pulled image created by GitHub and hosted on DockerHub can be used to create a container within the cluster. The sandbox is executed using a special environment thanks to *Fakechroot*.

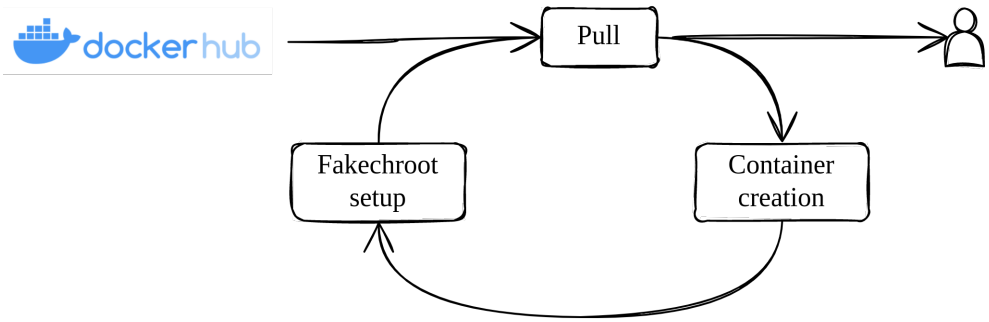

Figure 13: Container pull worflow

5.4.2 OpenMPI communication

Normally the OpenMPI communication between different processes is not compatible with sandbox systems like udocker, some additional steps are required to do that:

- the OpenMPI version installed inside the container must match the version used by the host;
- the executed container must share environment variables with the host, in particular the one that points to the dynamic library of OpenMPI.
- the executed container must share the host authentication and network.

6 Conclusion

6.1 Is parallelization always a good choice?

During our benchmarking analysis we have surprisingly realized that the thread parallelization is not always a good choice.

Due to the high overhead implied by the thread generation, we have observed that using OpenMP does not always result in a guaranteed speed benefit.

In cases in which the parallel region took little time to execute, it would be preferable to avoid parallelization and proceed with the straightforward execution of the code in a serial manner.

Benchmarking in the case of thread parallelization is a task which if far from being trivial. In fact, every system may perform differently in presence or in absence of threads. Moreover, it is hard to decide whether to parallelize or not sime piece of code based on general assumptions.

As an effective parallelization, we have started our project by parallelizing each for loop in the code. This has resulted in a waste of resource and a worsening of performances for small size problems. On the other hand, employing parallelization implies a significant performance boost for big dimensionality problems.

6.2 Thread allocation pattern