

# Particle Swarm Optimization: a parallelized approach

Bortolotti Samuele, Izzo Federico University of Trento  
Trento, Italy  
samuele.bortolotti@studenti.unitn.it, federico.izzo@studenti.unitn.it

**Abstract**—*Particle Swarm Optimization (PSO)* 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.

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.

**Index Terms**—ParticleSwarmOptimization; PSO; OpenMPI; OpenMP; C; Bio-Inspired; HPC; Parallelization

## I. INTRODUCTION

PSO focuses on two 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*.

Each particle needs to perceive the positions along with the associated performance measures of the *neighboring particles*. In this way, each agent remembers the position  $z$  associated to the best performance of all the particles within the neighborhood, as well as its own best performance so far  $y$ .

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. Moreover, the program offers the user the possibility to modify the number of particles to consider within a particle neighborhood.

### A. Parametrization

PSO requires the following parameters to 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*.

### B. Continuous Optimization

A swarm of particles is initialized with random positions and velocity.

At each step, each particle updates first its velocity (equation 1):

$$v' = w \cdot v + \phi_1 U_1 \cdot (y - x) + \phi_2 U_2 \cdot (z - x) \quad (1)$$

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);
- $\phi_1, \phi_2$  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 (equation 2):

$$x' = x + v' \quad (2)$$

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

### C. State-of-the-art analysis

As a first approach to the problem, we have surfed the web in order to look for pre-existing PSO implementations.

Based on what we have found, the approaches can be divided into three main categories:

1. those ones which aim to change the behavior of the algorithm introducing new features;
2. those ones which aim to solve a real world problem using PSO as main algorithm;
3. those ones which aim to optimize the runtime execution speed.

In our study we have decided to exclude the second category of PSO algorithms since these solutions are strictly problem dependent. Thus, a comparison would produce meaningless results.

On the other hand, all those approaches which belong to first category of problems can be employed as case studies for our benchmarking analysis. However, it is strictly required to change some implementation aspects by modifying directly the code. In some cases, this requires a deep understanding of others' code, most of the time a tough job due to the absence of documentation.

The third category is our perfect competitor, since they share our same objective. However, there are several cases in which

different PSO version have been implemented. Hence, some hands on is still required.

In the following table we list some of the implementations we have decided to consider during the benchmarking phase.

| Ref. | Year | Type          | Code | Note |
|------|------|---------------|------|------|
| [1]  | 1995 | Serial        | No   | -    |
| [2]  | 2019 | Serial        | Yes  | 1    |
| [3]  | 2019 | Serial        | Yes  | 1    |
| [4]  | 2020 | Serial        | Yes  | 1    |
| [5]  | 2020 | Serial        | Yes  | 1    |
| [6]  | 2014 | MPI           | No   | -    |
| [7]  | 2017 | MPI/MP        | No   | -    |
| [8]  | 2019 | MPI/MP,CUDA   | Yes  | 1    |
| [9]  | 2020 | OpenMP        | Yes  | 2    |
| [10] | 2021 | Serial,OpenMP | Yes  | 1    |

The indexes in the notes refer to:

1. provides only global neighborhood implementation. Thus, the comparison would be untruthful as those implementations have a clear advantage in the execution time due to a favorable topology;
2. provides PSO with different neighborhood versions but without a distance based approach. Hence, the implication are the same as for the point 1.

According to the previous statements, we claim that we have implemented a PSO version which differ from the ones we have decided to consider since it has a different notion of neighborhood which makes it harder to parallelize.

## II. MAIN STEP TOWARDS PARALLELIZATION

In this section the report provides a detailed description of the major contribution we have provided to the serial parallelization in order to move towards an efficient hybrid OpenMP-MPI solution.

### A. Serial version of the algorithm

The main steps of the algorithm 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 (eq. 1 and 2).

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.

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.

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.

For the neighborhood sort, the program relies on *quicksort* (figure ??). The main reason behind this choice is the amount of parallelization this algorithm can provide.

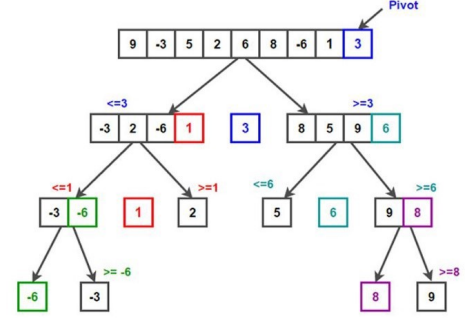


Figure 1. Quicksort

### B. Parallel version of the algorithm

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.

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 (figure 2).

*All-to-all* parallel pattern implemented using *MPI\_Allgather* is characterized by the exchange of individual messages from every process to any other process.

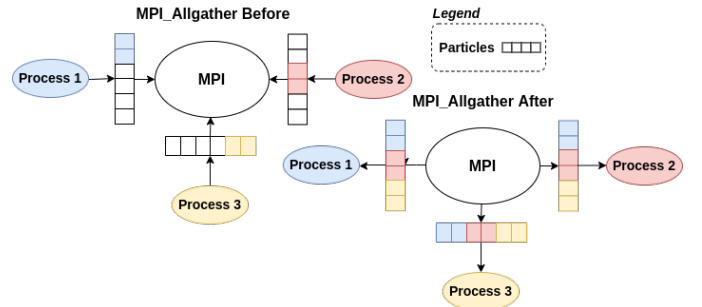


Figure 2. Communication schema.

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.

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.

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.

As presented in figure 2, `MPI_Allgather` is suitable for the problem since it is an *all-to-all* communication channel and since it allows 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.

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 equations 1 and 2 it is possible to evolve the algorithm and approach the target function optima.

### III. BENCHMARKING

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.

#### A. Problem configuration

We devised a configuration file which is the same for every run, so as to have a common baseline.

The configuration is listed below:

- `problemDimension` = 50
- `particlesNumber` = 5000
- `iterationsNumber` = 500
- `neighborhoodPopulation` = 5000
- `weights`: `w` = 0.8, `phi_1` = 0.3, `phi_2` = 0.3
- `functions`: `fitness` = sphere, `distance` = euclidean, `fitness-Goal` = minimum

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

#### B. 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 number of tests we have ran in total is around 980 (TODO write correct number), in particular we tried every possible combination 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`];

*Select* refer to the number of chunks requested to the PBS. A *chunk* can be seen as group of sockets, which are not necessarily on the same machine. *Place* can take one of the following values:

- **pack**: all the chunks are allocated within the same nodes;
- **scatter**: the chunks are located on different nodes;
- **pack:excl**: the scenario is the same as **pack**, but the chunks are reserved entirely for the user application;
- **scatter:excl**: the scenario is the same as **scatter**, but the chunks are reserved entirely for the user application.

#### C. Results

All the job configurations were tested by both members of the group in order to validate and reduce possible noise of the results.

Table I shows the amount of jobs we have run and the associated fail rate (we have decided to show only the amount of processes and not the threads one for presentation purposes).

| 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     |

Table I  
JOB RESULTS.

The table highlights a correlation between the failure rate and the number of processes.

As a result of a first analysis, none of the failed runs were caused by an error in the code, most of them were failed due to the time exceed error. Thus, we have tried to investigate the main reason behind this weird behavior.

To begin with, we have kept constant the number of processes and we have increased the number of chunks for our jobs.

Figure 3 shows the number of failed runs associated with the corresponding number of threads. As a matter of fact, the more the requested chunks, the more the cores for the job are. Hence, since the number of MPI processes is always the same, unused cores can host threads, which could be a reasonable explanation for the low amount of failed jobs in higher chunks requests.

This proof of concept highlights how the overhead paid for a continuous context switch introduced by OpenMP is dramatically higher than the performance gain due to the parallelization. Therefore, we came to the conclusion that since several optimizations are already included within modern compilers such as GCC, OpenMP introduces only an unwanted overhead for the problem that we are facing. Hence, the optimal scenario is represented by the single threaded multi-process case.

The previously described phenomena is also observable from figure 4. There, it is possible to see that the execution time

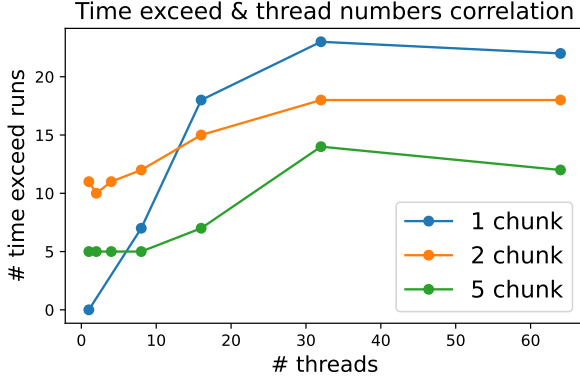


Figure 3. Threads and time exceed runs.

increases when the number of threads increases. Specifically, the dots in the plot represent the executed jobs while the size of the dots expresses the number of correctly terminated runs used for to compute the average.

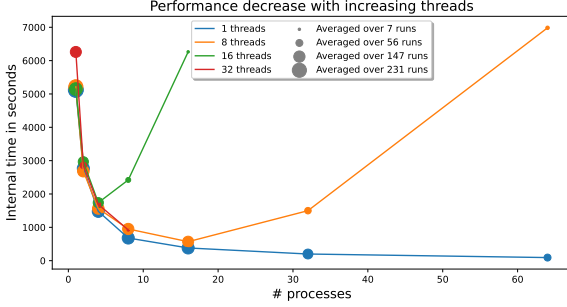


Figure 4. Thread and time execution runs.

A part from the efficiency issues in the the multi-threaded scenario, we have deepened our benchmark analysis by considering single threaded jobs. Figure 5 shows the execution time difference between various configuration places, where *excl* means exclusive.

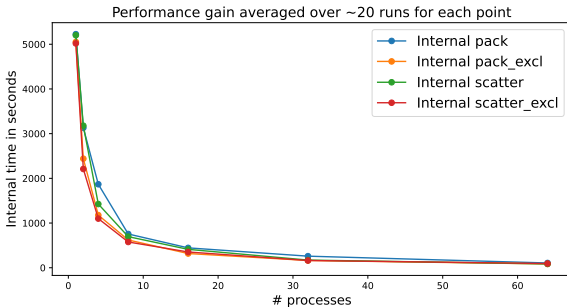


Figure 5. Processes and time execution runs.

The plot suggests that the difference in terms of execution time between exclusive chunks and shared ones is marginal with respect to the entire time needed for the job execution. This inevitably implies that the computation time of the job

is markedly higher than the scheduling time between other users' processes. For the same reason, we can claim that the overall computation time does not suffer from the network overhead. The last statement can be appreciated from the small differences in term of execution time between pack and scatter jobs executions.

Furthermore, we have highlighted an elbow point in figure 6, visible in the right plot. This spot identifies the best visual tradeoff between number of processes and the execution time of the parallel solution. Indeed, with more than 16 processes the gain does not motivate the expenses associated to the PBS request.

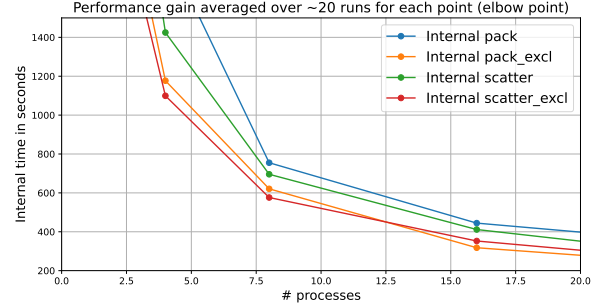


Figure 6. Processes and time execution runs elbow point.

To validate this hypothesis we have created speedup and efficiency plots with respect to SOTA serial solution presented in section I-C.

#### IV. FINAL DISCUSSION

##### REFERENCES

- [1] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95 - international conference on neural networks*, 1995, vol. 4, pp. 1942–1948 vol.4.
- [2] toddguant, "PSO library for c." <https://github.com/toddgaunt/cps>, 2019.
- [3] sousouho, "Succing PSO." <https://github.com/sousouhou/succinctPSO>, 2019.
- [4] kkentzo, "Pso." <https://github.com/kkentzo/pso>, 2020.
- [5] fisherling, "Pso." <https://github.com/fisherling/pso>, 2020.
- [6] A. O. S. Moraes, J. F. Mitre, P. L. C. Lage, and A. R. Secchi, "A robust parallel algorithm of the particle swarm optimization method for large dimensional engineering problems," *Applied Mathematical Modelling*, vol. 39, no. 14, pp. 4223–4241, 2015.
- [7] N. Nedjah, R. de Moraes Calazan, and L. de Macedo Mourelle, "A fine-grained parallel particle swarm optimization on many-core and multi-core architectures," in *Parallel computing technologies*, 2017, pp. 215–224.
- [8] abhi4578, "Parallelization-of-PSO." <https://github.com/abhi4578/Parallelization-of-PSO>, 2019.

- [9] LaSEEB, “Openpso.” <https://github.com/abhi4578/openpso>, 2020.
- [10] pg443, “Particle-swarm-optimization-OpenMP.” <https://github.com/pg443/Particle-Swarm-Optimization-OpenMP>, 2021.