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* is an optimization algorithm for nonlinear functions based on bird swarms. 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 through 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.

A. Parametrization

PSO requires the following parameters to be set:

- *Swarm size*: typically 20 particles for problems with dimensionality between 2 and 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):

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

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);
- ϕ_1, ϕ_2 are acceleration coefficients/learning rates (cognitive and social, respectively);
- U_1 and U_2 are uniform random numbers in range $[0, 1]$.

Finally, each particle updates its position (equation 2):

$$x' = x + v' \tag{2}$$

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 though 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 versions have been implemented. Hence, some hands on is still mandatory.

In the following table we list some of the implementations we have found online.

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` 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*. The main reason behind this choice is the amount of parallelization this algorithm can provide.

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 presented in figure 1.

The *all-to-all* parallel pattern is implemented using `MPI_Allgather` and it is characterized by the exchange of individual messages from every process to any other process.

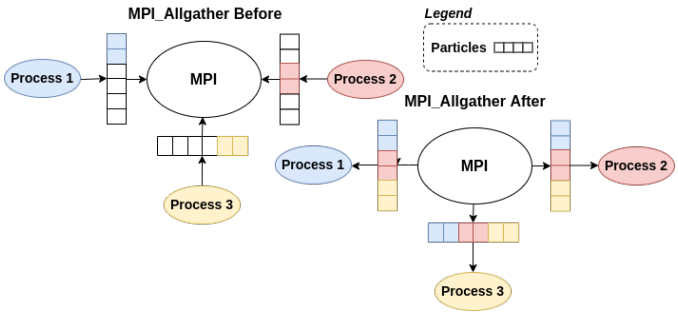

Figure 1. Communication schema.

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 database, 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 the 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 splitted 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 1, `MPI_Allgather` is suitable for the problem since it is an *all-to-all* communication channel and 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 it 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, fitnessGoal = minimum

The amount of particles and the neighborhood population are unreasonable for any known problem but they were chosen to show the clear advantage 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 several days.

The number of tests we have ran in total is around 1280, 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]`;

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.

Figure 2 shows the amount of jobs we have run and the associated time exceeded rate.

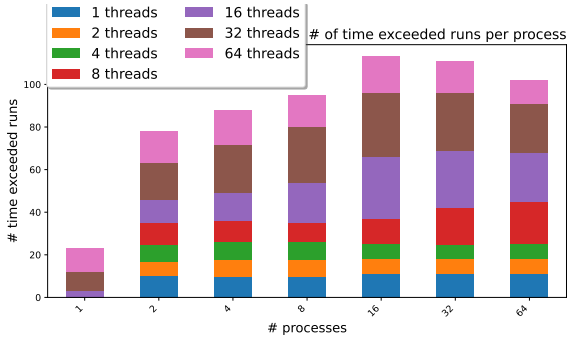

Figure 2. Number of time exceeded runs per process.

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

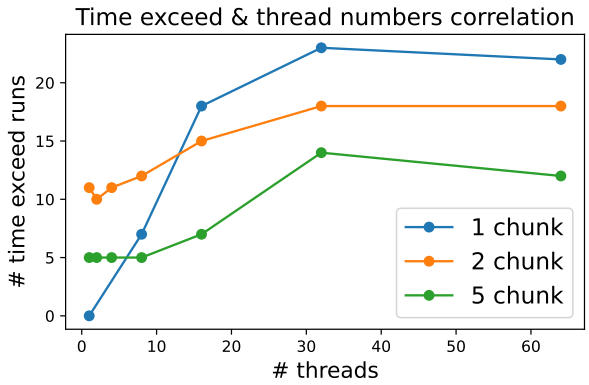

Figure 3. Threads and time exceed runs.

This proof of concept highlights how the overhead paid for a continuos context switch introduced by OpenMP is 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. From there, it is possible to see that the execution time 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 to compute the average.

A part from the efficiency issues in the the multi-threaded scenario, we have deepen our benchmark analysis by considering

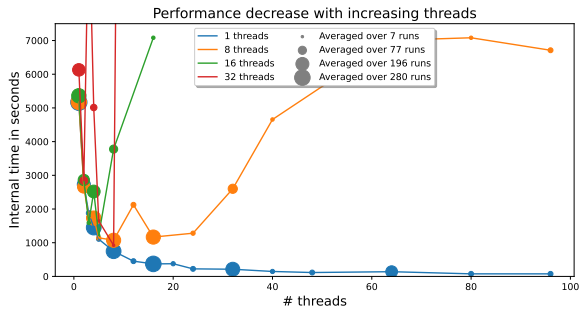

Figure 4. Thread and time exeuction runs.

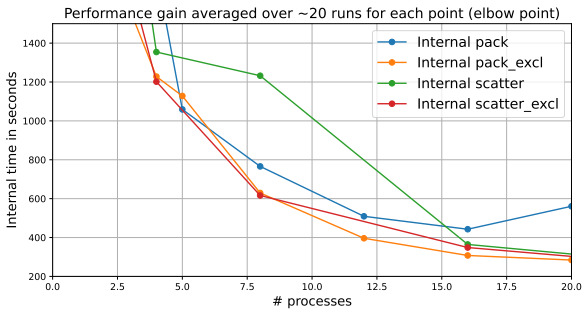

Figure 6. Processes and time execution runs elbow point.

single threaded jobs. Figure 5 shows the execution time difference between various configuration places, where *excl* means exclusive.

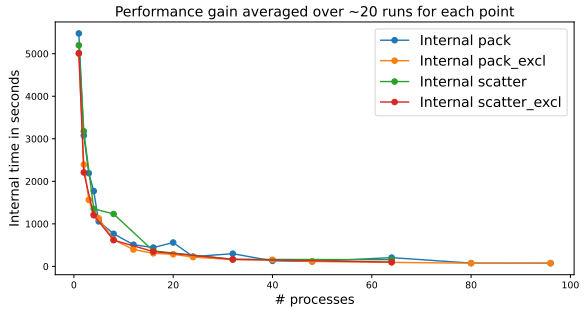

Figure 5. Processes and time execution runs.

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 implies that the computation time of the job is markedly higher then 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. This spot identifies the best visual tradeoff between number of processes and the execution time of the parallel solution. Indeed, for the problem configuration used for our tests, more than 16 processes do not bring an enough gain in order to motivate the expenses associated to the PBS request.

As mentioned in section I-C, the analysis of the available similar works has required to directly manage others' code. Unfortunately, since the intricate problem was resolved in a matter of seconds, we claim that the findings are deceptive. We believe that the main reasons are the way some PSO instances perform the iterations since some of them stop whenever the solution is below an error threshold, hence performing fewer iterations than those required; as well as our difficulty in comprehending the actual behavior of other people's code, which has resulted in the development of an inconsistent algorithm.

Based on the prior results, we have chosen to take into account only the multi-process solution, and we have deepened our investigation by examining the parallel performance improvement using speedup and efficiency graphs. To begin with, the notion of scalability cannot be directly analyzed considering the problem we are optimizing. The reason for that regards the notion of *problem size*, which cannot be trivially defined. Naively, one would conclude that the problem size is doubled when the problem dimension is doubled, however, the parallelization influence is limited only in the time for the position and velocity update. On the other hand, we cannot argue that the problem size is doubled when the number of particles is doubled, since the problem persists, but holds in the opposite direction, namely only some portions of the code benefit from the parallelization. Therefore, we claim that the concept of problem size is represented by a tight coupling between the problem dimension and the number of particles. Due to this non-trivial correlation, we have decided to focus only on one hard problem configuration and support our results with hundreds of runs.

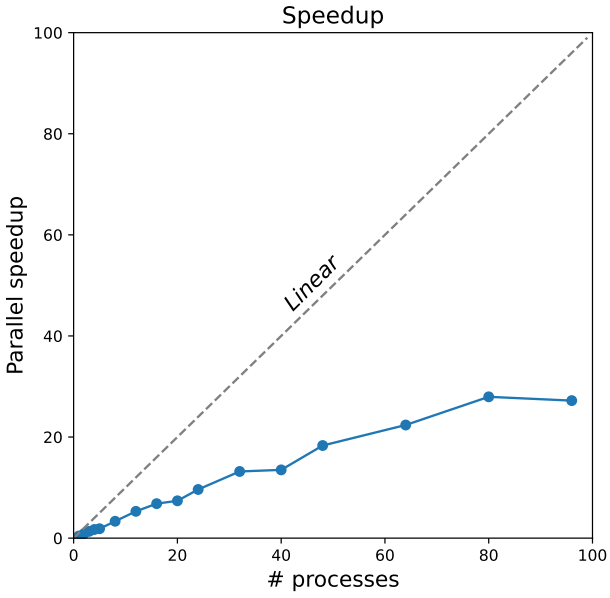

Figure 7. Speedup

From figure 7 we can see that the speedup is very limited. This can be seen as a consequence of the time needed for

communication between multiple processors, as at the end of each iteration all the processes must be synchronized and the number of exchanged messages is considerably high. Therefore, we claim that the overhead time we pay for the parallelization plays a relevant role, however, parallelization is still capable of providing a massive improvement in terms of time. This inevitably implies that the perfect parallelization and the ideal speedup cannot be achieved, as highlighted in the plot.

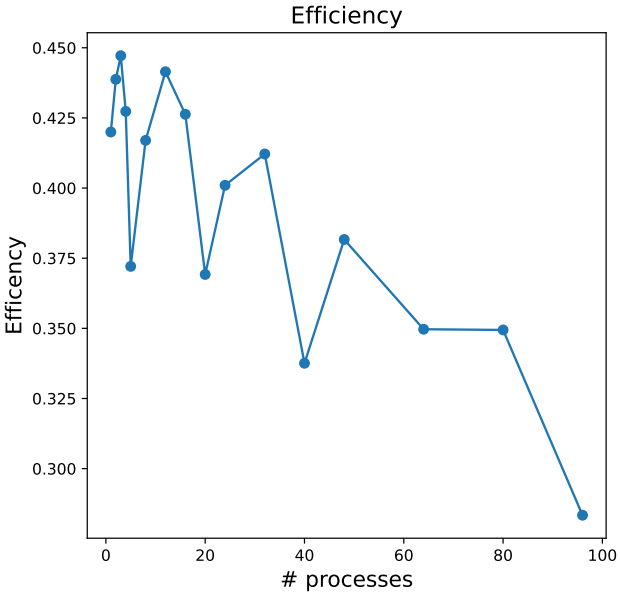

Figure 8. Efficiency

Likewise figure 7, figure 8 shows the efficiency curve considering only the jobs run with MPI, which is the reason why the efficiency does not start with 1. As a first consideration, the single process MPI job requires additional time compared to the serial version of the program to complete the execution, showing that the MPI, if not used properly introduces a non-negligible overhead. The efficiency plot, on the other hand, shows that the best trade-off between the number of processes employed and the speedup gained has two peaks, respectively around 3 and 12 processes. Furthermore, we can see that the more the number of processes the less the efficiency is, which means that despite taking less time to execute, it is not convenient to employ a huge number of processes. Moreover, figure 8 highlights an irregular curve, we believe that this is a consequence of an inhomogeneous cluster, hence, there are nodes which are slower compared to others.

To conclude, table I provides a complete overview of the program analysis.

# Pr.	Seconds	Diff	Speedup	Efficiency	Type
1	2099	0	1	1	Serial
1	4997	-2898	0.41	0.41	OpenMPI
2	2392	-293	0.87	0.43	OpenMPI
3	1564	535	1.34	0.44	OpenMPI
4	1227	872	1.70	0.42	OpenMPI
5	1128	971	1.86	0.37	OpenMPI
8	629	1470	3.33	0.41	OpenMPI
12	396	1703	5.29	0.44	OpenMPI
16	307	1792	6.82	0.42	OpenMPI
20	284	1815	7.38	0.36	OpenMPI
24	218	1881	9.62	0.40	OpenMPI
32	159	1940	13.18	0.41	OpenMPI
40	155	1944	13.50	0.33	OpenMPI
48	114	1985	18.32	0.38	OpenMPI
64	93	2006	22.37	0.34	OpenMPI
80	75	2024	27.95	0.34	OpenMPI
96	77	2022	27.20	0.28	OpenMPI

Table I
Speedup and efficiency table.

IV. Final discussion

Up until this point, we produced a hybrid OpenMP-MPI algorithm to solve complex continuous optimization problems, equipped with an efficient and reproducible DevOps pipeline.

We have realized that thread parallelization does not fit well all the problems. Indeed, due to the high overhead implied by the thread generation, we have observed that using OpenMP worsen the result, not providing the much-wanted speed benefit.

Benchmarking in the case of thread parallelization is a task which is far from trivial. Every system may perform differently in the presence or absence of threads. Moreover, it is hard to decide whether to parallelize or not some piece of code based on general assumptions. As an effective parallelization, we started our project by parallelizing each for loop in the code. This has resulted in a waste of resources and a worsening of performances for small-size problems. Unfortunately, the same has happened even in the case when the threads acted on the most time-consuming region of the code.

To conclude, the program we provided is suitable for single-threaded process parallelization and, as shown in the efficiency and speedup plots, it provides the best result when the number of processes is limited, as even if the computational time decreases, the more the processes the more the overhead required for the MPI communication to take place is.

A. Future Work

As a further work, it would be interesting to complement the already present architecture with different type of neighborhood and analyze which configuration brought the best results in presence of parallelization, and in terms of quality of the provided solutions. However, the scope of our project was to implement the above described parallel algorithm, which already posed significant challenges, especially because we could not base our implementation on pre-existing works.

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/cpso, 2019.

[3] souusouho, "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.