Democratic and Popular Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Mohamed Khider - BISKRA
Faculty of Exact Sciences, Natural Sciences and Life
Computer Science Department

Order Number: GLSD5/M2/2019

Thesis

Presented to obtain the diploma of academic Master in

Computer Science

Option: **Software Engineering and Distributed Systems**

Parallelization of Spider Monkey Optimization (SMO) algorithm

By:
Said Bousnane

Defended the 07/06/2019, in front of the jury composed of:

Dr. Sahli Sihem	MAA	President
Dr. Kahloul Laid	MCA	Supervisor
Dr. Chami Djazia	MAA	Examiner

University Year: 2018/2019

Acknowledgements

Praise to ALLAH, the Compassionate, the Merciful. Peace and blessing on the Messenger of Allah, Muhammad the prophet (Peace Be Upon Him). I would like to express my gratitude to ALLAH for His blessing and inspiration leading me to finish this work.

My special thanks and appreciation to my supervisor ***Dr. Kahloul Laid*** for his continuous encouragement, guidance and for his endless patience and precious advice, although the surgery that he made; We ask ALLAH to heal him soon. Without forgetting to thank a lot the PhD student ***Leila Belaiche*** for her support, following-up my work and advice all the time, especially during the time when my supervisor was recovering from the surgery that he made.

I would like to express my deepest thanks to the members of the jury: *Mrs. Sahli Sihem* and *Mrs. Chami Djazia* for reading and evaluating my dissertation.

I never forget to thank all my teachers at the Computer Science Department who taught me the basic principles of computer science, with a special mention of the head of the department *Prof. Mohamed Chaouki Babahenini.*

Finally, my gratitude is deeply paid to my mother and my father, and to all members of my family in Ghardaia, which were patient on me during the five years of my studying here in Biskra away from them. Without forgetting the family "Bousnane" herein Biskra which I was staying with during my study period.

Abstract

The parallelization is the adopted programming approach to reduce the execution time and increase the performance of programs in many fields. This approach makes the processors treat many processes/tasks in one moment of time, in contrast to the classical method, which is "the sequential approach" that treats a single problem at a moment of time. For this reason, we have implemented an application called ParSMO and it contains the implementation of a swarm intelligence based algorithm (SI), which is called: Spider Monkey Optimization (SMO) algorithm for numerical optimization. This thesis is composed of two main parts, the first part is the implementation of the SMO algorithm sequentially, whereas the second part is the implementation of the first proposed parallel SMO algorithm in the literature, using the Multiprocessing package in Python 3.7. The aim of this work is comparing the sequential SMO with the parallel one, in terms of the execution time, the near-optimum solution and the objective space density of the last generation. The experimental study that was realized on ParSMO application using the two test problems "Dekkers and Aarts" (DA) and "Six-hump Camelback" (ShC) illustrates that the parallel SMO algorithm outperforms the sequential SMO in terms of gaining a shorter execution time and a better objective space density in the last generation.

Résumé

Le parallélisme est l'approche de programmation adoptée pour réduire le temps d'exécution et augmenter la performance des programmes dans plusieur domaines. Cette approche permet à un ensemble de processeurs de traiter de nombreux processus/tâches en même temps, contrairement à la méthode classique "l'approche séquentielle", qui traite un problème unique en un instant donné. Pour cette raison, nous avons implémenté une application appelée ParSMO, et qui est l'implémentation d'un algorithme basé sur l'intelligence essaim, cette algorithme est appelé : "L'algorithme de Spider Monkey Optimization (SMO) pour l'optimisation numérique". Ce mémoire est composé de deux parties principales, la première partie est l'implémentation séquentielle de l'algorithme SMO, tandis que la seconde partie est l'implémentation du premier algorithme parallèle SMO, en utilisant le package Multiprocessing dans Python 3.7. Le but de ce travail est de comparer le SMO séquentiel avec celui parallèle, en termes de temps d'exécution, de solution quasi-optimale et de densité de l'espace objectif de la dernière génération. L'étude expérimentale appliquée à l'application ParSMO utilisant les deux problèmes de test "Dekkers and Aarts" (DA) et "Six-hump Camelback" (ShC) montre que l'algorithme parallèle SMO surpasse celui séquentielle en termes de gagner un temps d'exécution plus court et une meilleure densité de l'espace objectif de la dernière génération.

Contents

List of Tables

List of Figures

List of Algorithms

Introduction

Taking inspiration from nature to develop computationally efficient algorithms is one of the ways to deal with real-world optimization problems. This approach is called swarm intelligence and it is defined as the emergent collective intelligence of groups of simple agents. The fields of application of this approach are many and various, including robotics, artificial intelligence (IA), process optimization, telecommunication...etc.

Researchers have developed many algorithms by simulating the swarming behavior of various creatures like ants, honey bees, monkeys, birds ... etc. This kind of algorithms which are based on Swarm Intelligence has great potential to find a near-optimum solution of a real-world optimization problem. Hence, a recent algorithm for numerical optimization based on swarm intelligence is proposed by modeling the foraging behavior of spider monkeys (SMs) and it is called: Spider Monkey Optimization (SMO) algorithm for numerical optimization (Jagdish Chand Bansal 2014).

SMO algorithm was the selected one among other swarm intelligence based algorithms for several reasons, including being the spider monkeys society an organized one in foraging, where they live in groups of specified numbers and be led by female leaders, and has a stable behavior in foraging represented in following the fission-fusion system, further to the communication between the group members in various ways. Thus, the SMO algorithm is based on changing the number of groups in the swarm (from larger to smaller and vice-versa) through the algorithm steps, in order to have a good balance between exploration and exploitation to achieve a better optimization performance.

In the light of the high complexity of this kind of algorithms, and in pursuit of better performance of the SMO algorithm in terms of execution time, the near-optimum solution, and better objective space, we propose the first parallel SMO in the literature. Using Python programming language and Multiprocessing package, we implemented the sequential version of SMO, as well as the proposed parallel version of SMO in an application, we called it ParSMO. As expected, and after several Experiments of the SMO algorithm with its two versions, the sequential version, and the parallel one, the results illustrate that the parallel SMO outperforms the sequential SMO in terms of gaining the computational time and improving the objective space.

This thesis is organized in two parts:

Part I is a theoretical part (State of the art) of two chapters. **chapter 1** defines swarm intelligence and some related terms and gives a detailed view and explanation of the SMO algorithm. **chapter 2** presents the basic concepts of parallel computing, further the parallel architectures and the parallel programming models.

Part II is designed for explaining our contributions, and it is composed of two chapters. **chapter 3** illustrates the analysis of our application in general, and the global design of it, further to detailed design which contains explaining diagrams. **chapter 4** introduces all the tools that we worked with and the implementation of our application, moreover, the results that we got from different experiments, finishing the thesis by a discussion of these results and giving the conclusions.

Part I

State of the art

Chapter 1

Spider Monkey Optimization (SMO) algorithm

Chapter 1

Spider Monkey Optimization (SMO) Algorithm

Introduction

Optimization is common in almost all spheres of human activities such as Industry, agriculture...etc. Earlier, optimization problems are mainly based on personal experiences to solve. After that, some classical or traditional optimization techniques and methods are founded, represented in some primary algorithms, but it wasn't useful for solving the complex real-world optimization problems. Since the end of the twentieth century, the fast development of computer and artificial intelligence technologies has provided new efficient methods to solve the complex optimization problems. From those methods, a nature-inspired approach, which is called "Swarm Intelligence". Researchers have developed many algorithms by simulating the swarming behavior of various creatures like ants, honey bees, fish, spiders... etc, and the findings are very motivating.

In this chapter, we will provide definitions for some terms in the optimization field such as optimization problems (see 1.1), swarm intelligence (see 1.2). After that, we will mention the algorithms which belong to the swarm intelligence (SI) approach (see 1.2.3). Then, we will take the chance to explain more specifically spider monkey optimization (SMO) algorithm, which we implemented it, and the motivation to use it (see 1.3.1), its process and its performance among the other algorithms in the same field (see 1.3.2)...etc.

1.1 Optimization Problems

Definition (Yujun Zheng n.d., Rothlauf 2011) : An optimization problem is to find an optimal or a near-optimal solution among large set of feasible candidates. The objective function of the problem determines the best solution among the others.
Formally, in a domain P, and subject to an objective function f, there is an optimal solution ($x^* \in P$), where:

- **max *f(x)* :** $f(x) \leq f(x^*)$, $\forall\ x \in P$

$\Longrightarrow$ Or

- **min *f(x)* :** $f(x) \geq f(x^*)$, $\forall\ x \in P$

There are two very useful approaches that are primary used to solve optimization problems, which are: Swarm Intelligence and Evolutionary Computation.

1.2 Swarm Intelligence (SI)

Definition: (Jagdish Chand Bansal 2019, Amrita Chakraborty n.d.)
The word "Swarm" can be considered as a set of interacting homogeneous individuals. Whereas the term "Swarm Intelligence" represents a discipline that deals with natural and artificial systems composed of many individuals that coordinate based on the collective and self-organized cooperative behavior of social entities such as flock of birds, school of fishes or ant colonies...etc.
The SI also is a meta-heuristic approach inspired by the collective behavior of social insect colonies and other animal societies, which is used to solve optimization problems.

1.2.1 The conditions of swarm intelligence

As shown in Figure 1.1, self-organization and Division of labor are two necessary and sufficient conditions for obtaining intelligent swarming behaviors (KARABOGA 2005).

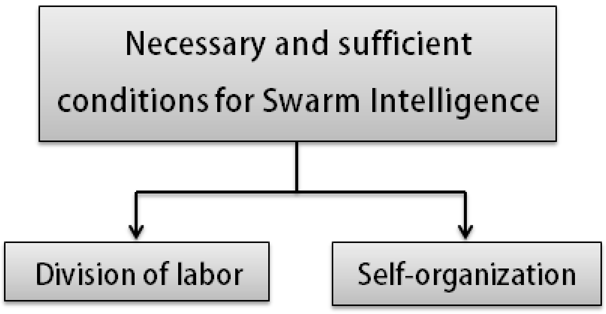


Figure 1.1: Conditions for intelligent swarming

a) Self-organization:
The property self-organization based on four characteristics, which are (Bonabeau E n.d.):

1) **Positive feedback:** Information extracted from the output system and revealed to the input to promote formation of appropriate structures. Positive feedback provides diversity in swarm intelligent.

2) **Negative feedback:** Compensates the effect of positive feedback. It helps the stability of the collective pattern.

3) **Fluctuation:** The rate of the randomness in the system. It helps to get rid of stagnation.

4) **Multiple interactions:** Provides the method of learning from the individuals within a society, and also improves the overall intelligence of the swarm.

b) Division of labor:
This helps different tasks to be performed simultaneously by different specialized individuals.

1.2.2 Representation and fitness evaluation

a) Representation of individuals: (Jagdish Chand Bansal 2014, Leila 2017)

The individual is represented by a vector of real values, where the size of the vector is the dimension of the objective function.
- Example:
For the objective function: $f(x) = x_1^2 + x_2^2$, the individuals will be a couple (a value for x_1 and an other for x_2).
b) Fitness evaluation: (Leila 2017)

The evaluation should provide a value that determines an ordering between different individuals.
Adaptation function (*fitness*): this function evaluates the quality of each individual in a population.

1.2.3 Swarm intelligence algorithms

(Jagdish Chand Bansal 2014) Figure 1.2 Represents the hierarchy of swarm intelligence-based algorithms. Those algorithms are divided into two categories: “insect based” and “animal based”.

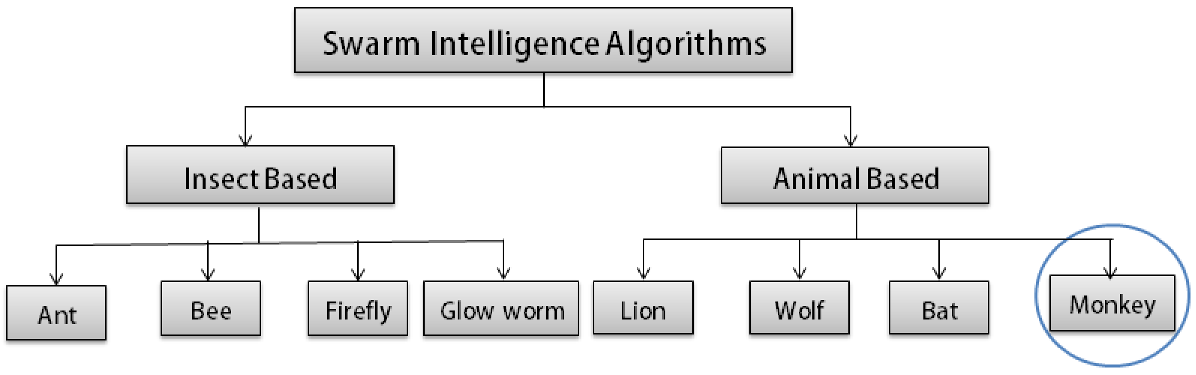


Figure 1.2: Hierarchy of swarm intelligence-based algorithms

1.3 Spider Monkey Optimization (SMO) algorithm

1.3.1 Motivation

(Jagdish Chand Bansal 2019)

a) Emergence of Fission-Fusion society structure:

The concept of fission-fusion society is introduced by the biologist “Hans Kummer” In a “fission-fusion” society and in case of shortage of food, the competition for food among the parent group members leads to divide themselves into sub-groups (fission) in order to forage, and at night they return to join the primary group (fusion) to share the food and to take part in other activities.

b) Foraging behavior of Spider Monkeys:

Spider monkeys live in the tropical rain forests of Central and South America and exist as far north as Mexico. They are called spider monkeys because they look like spiders when they are suspended by their tails. Spider monkeys live in a unit group called “parent group”, and based on the food scarcity or availability they split themselves or combine. Communication between them depends on their gestures, positions and whooping.

c) Social organization and behavior:

The social organization and behavior of spider monkeys can be understood through the following steps:

- Spider monkeys live in groups of 40 - 50 individuals.
- Generally, a female leads the group (global Leader) and decides the forage route.
- If the leader does not find sufficient food then she divides the group into smaller groups (size varies from 3 to 8 members) and these groups forage independently. In the night and at their habitat everybody share the foraging experience.
- Sub-groups are also supposed to be led by a female (local leader).

d) Communication:

- Spider monkeys share their intentions and observations using positions and postures.
- During traveling, they interact with each other over long-distances using particular sounds such as whooping or chattering.
- This long-distance communication permits spider monkeys to get-together, stay away from enemies, share food and gossip.
- Each individual has its own discernible sound so that other members can easily identify who is calling.

- In order to interact to other group members, they generally use visual and vocal communication.

The above discussed about foraging behavior of spider monkeys is shown in Figure 1.3.

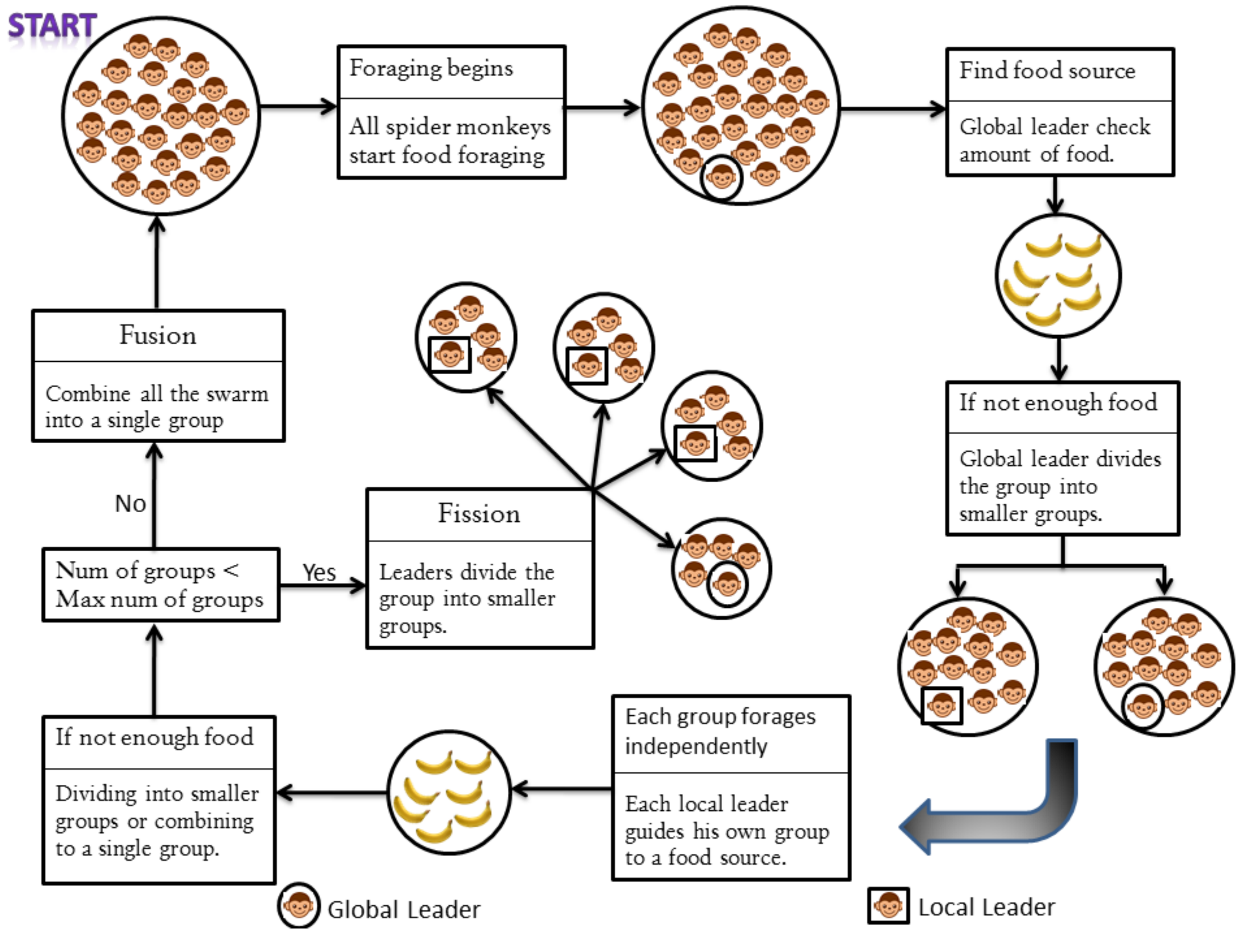

START
Foraging begins
All spider monkeys start food foraging
Find food source
Global leader check amount of food.
If not enough food
Global leader divides the group into smaller groups.
Each group forages independently
Each local leader guides his own group to a food source.
If not enough food
Dividing into smaller groups or combining to a single group.
Num of groups < Max num of groups
Yes
Fission
Leaders divide the group into smaller groups.
No
Fusion
Combine all the swarm into a single group
Global Leader
Local Leader

Figure 1.3: Foraging behavior of Spider Monkeys

1.3.2 SMO algorithm process

(Jagdish Chand Bansal 2014) SMO algorithm consists of six phases: Local Leader Phase, Global Leader Phase, Global Leader Learning Phase, Local Leader Learning Phase, Local Leader Decision and Global Leader Decision Phase. These phases are explained next:

a) Initialization of the population:

At the beginning, SMO generates a uniformly distributed initial swarm of N spider monkeys (Algorithm 1), where SM_i (the i^{th} monkey in the swarm) is a D-dimensional vector, and D is the number of variables in the optimization problem. Each spider monkey SM represents the potential solution of the optimization problem. Each SM_i is initialized using the equation 1.1 below:

$$SM_{ij} = SM_{minj} + U(0,1) \times (SM_{maxj} - SM_{minj}) \tag{1.1}$$

Where, SM_{minj} and SM_{maxj} are the bounds of SM_i in j^{th} dimension, and U(0, 1) is a uniformly distributed random number in the range [0, 1].

Algorithm 1 Initialization of the population

```
1: procedure INITIALIZATION-OF-POPULATION($size$)
2:     for $i = 1$ to $size$ do
3:         for each $j \in \{1...D\}$ do
4:             $SM_{ij} = SM_{minj} + U(0,1) \times (SM_{maxj} - SM_{minj})$
5:         end for
6:     end for
7: end procedure
```

b) Local Leader Phase (LLP):

In this phase, the local leader experience and the local group experience are the information that each spider monkey SM based on to modifies its position. The position update equation for j^{th} dimension of the i^{th} SM in the k^{th} local group is:

$$SM_{newij} = SM_{ij} + U(0,1) \times (LL_{kj} - SM_{ij}) + U(-1,1) \times (SM_{rj} - SM_{ij}) \quad (1.2)$$

Where LL_{kj} is the j^{th} dimension of the local leader of the group k, and SM_{rj} is the j^{th} dimension of a SM_r chosen randomly from the group k such that ($r \neq i$).
The Equation 1.2 clears that the members of a group which are going to update their position are attracted towards their local leader.
Algorithm 2 shows position update process in LLP.

Algorithm 2 Position update process in Local Leader Phase (LLP)

```
1: procedure LOCAL-LEADER-PHASE
2:     for each $k \in \{1...NumGp\}$ do
3:         // $NumGp$ is the number of groups in the iteration
4:         for each member $SM_i \in k^{th}$ group do
5:             for each $j \in \{1...D\}$ do
6:                 if $U(0,1) \geq pr$ then
7:                     $SM_{newij} = SM_{ij} + U(0,1) \times (LL_{kj} - SM_{ij}) + U(-1,1) \times (SM_{rj} - SM_{ij})$
8:                 else
9:                     $SM_{newij} = SM_{ij}$
10:                end if
11:            end for
12:        end for
13:    end for
14: end procedure
```

c) Global Leader Phase (GLP):

In GLP phase, all the SMs update their position using experience of the global leader also the experience of the local group members. The position update equation 1.3 is as follows:

$$SM_{newij} = SM_{ij} + U(0,1) \times (GL_j - SM_{ij}) + U(-1,1) \times (SM_{rj} - SM_{ij}) \tag{1.3}$$

Where, GL_j is the j^{th} dimension of the global leader. Here, the positions of spider monkeys SM_i are updated based on a probabilities $prob_i$ which are calculated using their fitness, according to equation 1.4:

$$prob_i = 0.9 \times \frac{fit_i}{maxfit} + 0.1 \tag{1.4}$$

Where, fit_i is the fitness value of the i^{th} SM and $maxfit$ is the maximum fitness in the group. In this way a better candidate (which has the maximum fitness) will have a higher chance to make itself better (exploitation).
Algorithm 3 shows position update process in GLP.

Algorithm 3 Position update process in Global Leader Phase (GLP)

```
1: procedure GLOBAL-LEADER-PHASE
2:     for each $k \in \{1...NumGp\}$ do
3:         $count = 1$
4:         $GS = k^{th}$ group size
5:         while $count < GS$ do
6:             for each member $SM_i \in k^{th}$ group do
7:                 if $U(0,1) < prob_i$ then
8:                     $count = count + 1$
9:                     Randomly select $j \in \{1...D\}$
10:                    Randomly select $SM_r$ from $k^{th}$ group s.t. $r \neq i$.
11:                    $SM_{newij} = SM_{ij} + U(0,1) \times (GL_j - SM_{ij}) + U(-1,1) \times (SM_{rj} - SM_{ij})$
12:                end if
13:            end for
14:        end while
15:    end for
16: end procedure
```

d) Global Leader Learning (GLL) phase:

In this phase, the position of the global leader is updated by applying the greedy selection in the population i.e., the position of the SM having best fitness in the population is selected as the updated position of the global leader. After that, if the global leader is not updated (stagnated) then the *GlobalLimitCount* is incremented by 1.
The Algorithm 4 explains the process of this phase.

Algorithm 4 Global Leader Learning (GLL) phase

```
1: procedure GLOBAL-LEADER-LEARNING
2:     global leader position = position of SM which has best fitness.
3:     if global leader did not change its position then
4:         $GlobalLimitCount = GlobalLimitCount + 1$
5:     end if
6: end procedure
```

e) Local Leader Learning (LLL) phase:

In this phase, the position of the local leader is updated by applying the greedy selection in that group i.e., the position of the SM having best fitness in that group is selected as the updated position of the local leader. Next, if the local leader is not updated (stagnated) then the *LocalLimitCount* is incremented by 1.
The Algorithm 5 gives the process of this phase.

Algorithm 5 Local Leader Learning (LLL) phase

```
1: procedure LOCAL-LEADER-LEARNING
2:     for each $k \in \{1...NumGp\}$ do
3:         $k^{th}$ group leader position = position of member group which has best fitness.
4:         if $k^{th}$ group leader did not change its position then
5:             $LocalLimitCount = LocalLimitCount + 1$
6:         end if
7:     end for
8: end procedure
```

f) Local Leader Decision (LLD) phase:

If any local leader position is not updated up to a predetermined sill called *LocalLeaderLimit*, then based on pr all the members of that group update their positions either by random initialization or through Equation 1.5:

$$SM_{newij} = SM_{ij} + U(0,1) \times (GL_j - SM_{ij}) + U(0,1) \times (SM_{ij} - LL_{kj}) \qquad (1.5)$$

The process of Local Leader Decision (LLD) phase is described in Algorithm 6.

Algorithm 6 Local Leader Decision (LLD) phase

```
1: procedure LOCAL-LEADER-DECISION
2:     for each $k \in \{1...NumGp\}$ do
3:         if $LocalLimitCount_k > LocalLeaderLimit$ then
4:             $LocalLimitCount_k = 0$
5:             for each member $SM_i \in k^{th}$ group do
6:                 for each $j \in \{1...D\}$ do
7:                     if $U(0,1) \geq pr$ then
8:                         $SM_{ij} = SM_{minj} + U(0,1) \times (SM_{maxj} - SM_{minj})$
9:                     else
10:                        $SM_{newij} = SM_{ij} + U(0,1) \times (GL_j - SM_{ij}) + U(0,1) \times (SM_{ij} - LL_{kj})$
11:                    end if
12:                end for
13:            end for
14:        end if
15:    end for
16: end procedure
```

g) Global Leader Decision (GLD) phase:

In this phase, the position of global leader is monitored and if it is not updated up to a predetermined sill called *GlobalLeaderLimit*, then the global leader divides the population into smaller groups. Firstly, the population is divided into two groups and then three groups and so on till the maximum number of groups (MG), then the next time the global leader combine all the groups to form a single group. Thus the proposed algorithm is inspired from fusion–fission structure of SMs.

Algorithm 7 Global Leader Decision (GLD) phase

```
1: procedure GLOBAL-LEADER-DECISION
2:     if $GlobalLimitCount_k > GlobalLeaderLimit$ then
3:         $GlobalLimitCount_k = 0$
4:         if $NumGp < MG$ then
5:             Divide the swarm into smaller groups.
6:         else
7:             Combine all the groups to make a single group.
8:         end if
9:         Update Local Leaders position.
10:    end if
11: end procedure
```

1.3.3 Control parameters in SMO algorithm

(Jagdish Chand Bansal 2014) SMO algorithm has four control parameters: $LocalLeaderLimit$, $GlobalLeaderLimit$, the maximum number of groups (MG) and the perturbation rate (pr). The suggested parameter settings are given as follows:

- $MG = N/10$, it is chosen such that minimum number of SMs in a group should be 10.
- $GlobalLeaderLimit \in [N/2, N * 2]$
- $LocalLeaderLimit = D * N$
- $pr \in [0.1, 0.4]$, it will increment through iterations by: $0.3/NbrOfIteration$

N is the swarm size (which is between 40 and 160), and D is the number of dimensions in the objective function.

1.3.4 Performance Analysis of SMO

(Jagdish Chand Bansal 2019) Performance of SMO has been analyzed against three well-known meta-heuristics, Artificial Bee Colony (ABC), Differential Evolution (DE), and Particle Swarm Optimization (PSO) in (Jagdish Chand Bansal 2014). After testing on 25 benchmark problems and performing various statistical tests, it is concluded that the SMO is a competitive meta-heuristic for optimization. It has been shown that the SMO performed well for unimodal, multimodal, separable and non-separable optimization problems in (Jagdish Chand Bansal 2014). It was found that for continuous optimization problems SMO should be preferred over PSO, ABC or DE for better reliability.
Explaining terms:

- **Continuous optimization problems:** continuous optimization problems have continuous solution spaces.
- **Unimodal function:** is the function $f(x)$ who has a single extremum (minimum or maximum) in the range specified for x.
- **Multimodal function:** is the function who has more than one peaks in the search space (as shown in Figure 1.4).

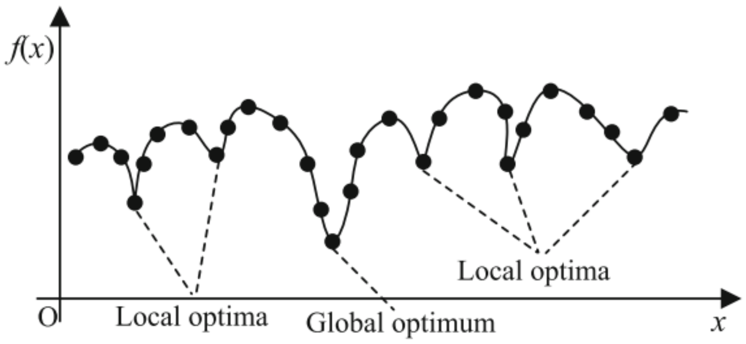


Figure 1.4: Illustration of global optimum and local optima

1.3.5 The main loop

Here, the precedent functions are exploited to construct the main loop of SMO algorithm.

Algorithm 8 Main Loop

```
1: procedure MAIN-LOOP($Iteration$)
2:     $Iteration = 0$
3:     while $Iteration < NbrOfIteration$ do
4:         Local-Leader-Phase()
5:         Global-Leader-Phase()
6:         Global-Leader-Learning()
7:         Local-Leader-Learning()
8:         Global-Leader-Decision()
9:         Local-Leader-Learning()
10:        Update $pr$
11:        $Iteration = Iteration + 1$
12:    end while
13: end procedure
```

1.3.6 Analyzing SMO

(Jagdish Chand Bansal 2014, 2019) SMO better balances between exploitation and exploration while search for the optima. So, let's explain that and see more details about the six steps of SMO algorithm as a general analyzing:

Firstly, Local Leader phase (LLP) is used to explore the search region as in this phase all the members of the groups update their positions with high perturbation in the dimensions. While the global leader phase (GLP) promotes the exploitation as in this phase, better candidates get more chance for updating their positions. This property makes SMO a better candidate among the search based optimization algorithms.

SMO algorithm also possesses an inbuilt mechanism for stagnation check. Local Leader Learning (LLL) phase and Global Leader Learning (GLL) phase, are used to check if the search process is stagnated or not. In case of stagnation (at local or global level) Local Leader and Global Leader Decision phases work.

The Local Leader Decision (LLD) phase creates an additional exploration, while in the Global Leader Decision (GLD) phase a decision about fission or fusion is taken depending on the number of groups in the population. Therefore, in SMO exploration and exploitation are better balanced while maintaining the convergence speed.

The algorithm 9 and the Figure 1.5 below represent the whole process of SMO algorithm.

Algorithm 9 Spider Monkey Optimization SMO algorithm

```
1: Initialize $GlobalLeaderLimit, LocalLeaderLimits$ , $pr$
2: $GlobalLimitCount = 0$ , $LocalLeaderLimits = 0$
3: Initialize $size$                 // The swarm size
4: Initialize $NbrOfIteration$
5: $MG = size/10$                    // Maximum Number of groups
6: $NumGp = 1$                       // Number of groups initial is one group
7: Initialization-Of-Population($size$)
8: Calculate fitness of each individual
9: Select global leader and local leaders applying greedy selection (see GLL phase and LLL
   phase in section 1.3.2).
10: Main-loop($Iteration$)
```

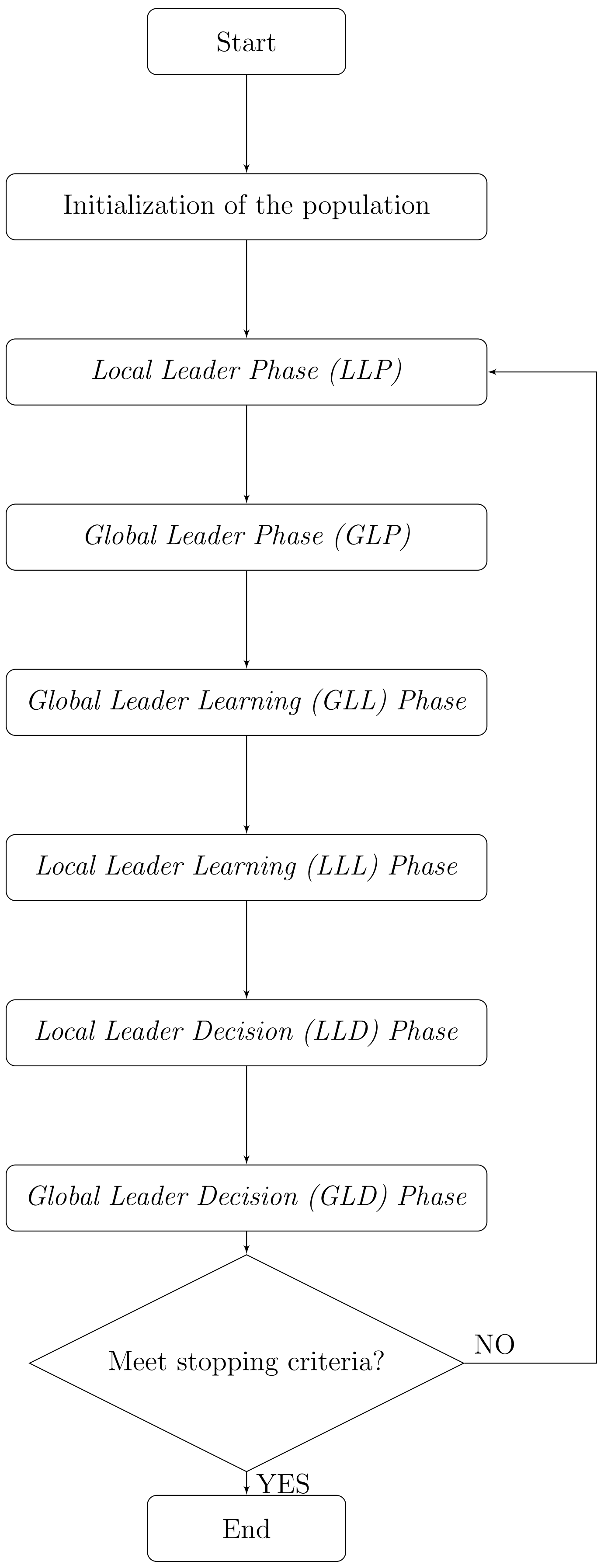

Start
Initialization of the population
Local Leader Phase (LLP)
Global Leader Phase (GLP)
Global Leader Learning (GLL) Phase
Local Leader Learning (LLL) Phase
Local Leader Decision (LLD) Phase
Global Leader Decision (GLD) Phase
Meet stopping criteria?
NO
YES
End

Figure 1.5: The process of Spider Monkey Optimization (SMO) algorithm

Conclusion

In this chapter, we have presented a recently created swarm intelligence based algorithm, which is the spider monkey optimization SMO algorithm for numerical optimization (). The inspiration of SMO process is from the social behavior of spider monkeys in foraging. Spider monkeys was the chosen, that is because they have been categorized as “fission-fusion” social structure based animals. This kind of animals split themselves from large to smaller groups and vice-versa based on the scarcity and the availability of food, and this is the important thing in SMO algorithm.

In order to explain this new approach, we divided this chapter into three parts. In the first part, we have given a definition of an optimization problem (OP). Then in the second part, we touched on the swarm intelligence (SI), a definition and the conditions that achieve obtaining intelligent swarming. Further, we presented a classification of all the algorithms that belong to the SI approach. Thus, the last part contained the reasons for the inspiration from spider monkeys and a detailed explaining of the SMO algorithm process, its parameters, its performance, and general analysis of it.

For better performance of the SMO algorithm, we proposed the first parallel version of the SMO algorithm in the literature. Therefore, the next chapter contains basic concepts of parallel computing and an overview of parallel computing architectures and parallel programming models.

Chapter 2

Parallel computing: basic concepts

Chapter 2

Parallel computing: basic concepts

Introduction

The serial computing means that the problem statement is broken into discrete instructions, those instructions are executed by a CPU one by one. Thus, at any moment only one instruction is executed. The last point was causing a huge problem in the computing industry, as only one instruction was getting executed at any moment of time. Further, this was a huge waste of hardware resources as only one part of the hardware will be running for particular instruction and of time. As problem statements (programs) were getting heavier and bulkier, so does the amount of execution time of those statements.

As a real-life example of this, would be people standing in a queue waiting for a movie ticket and there is only one cashier giving ticket one by one to the persons. The complexity of this situation increases when there are two queues and only one cashier. We could definitely say that complexity will decrease when there are two queues and two cashiers giving tickets to two persons simultaneously. This is the aim of using parallel computing, is to make the CPU treats and executes more than one instruction at any moment of time.

This chapter presents an overview on parallel computing. Firstly, we touch on the reasons for forwarding into parallel computing. Then, we present the scope of using parallel computing, also the general architecture of the computer (Von Neumann architecture). Finally, we identify the classification of parallel architectures and the different parallel programming models.

2.1 Motivating Parallelism

(Ananth Grama 2003, CHOUDHARY 1989)

2.1.1 The Computational Power Argument

It is possible to fabricate devices with very large transistor counts. How we use these transistors to achieve increasing rates of computation is the key architectural challenge. A logical recourse to this is to rely on parallelism.

On all, Parallel processing is the consensus approach to providing the necessary computational power for most computational intensive problems such as scientific, vision or any other.

2.1.2 The Memory/Disk Speed Argument

The overall speed of computation is determined not just by the speed of the processor, but also by the performance of the memory system which is determined by the fraction of the total memory requests that can be satisfied from the cache.

Parallel platforms typically yield better memory system performance because they provide a larger aggregate caches, and a higher aggregate bandwidth to the memory system. Furthermore, the principles that are at the heart of parallel algorithms (locality of data reference) lend themselves to cache-friendly serial algorithms.

2.1.3 The Data Communication Argument

As the networking infrastructure evolves, the vision of using the Internet as one large heterogeneous parallel/distributed computing environment has begun to take shape. In many applications there are constraints on the location of data and/or resources across the Internet. In such applications, even if the computing power is available to accomplish the required task without resorting to parallel computing, it is infeasible to collect the data at a central location.

In these cases, the motivation for parallelism comes not just from the need for computing resources but also from the infeasibility of alternate (centralized) approaches.

2.1.4 Time Argument

As discussed in the introduction of the chapter, the more important reason of heading towards parallel computing is that the last one provides time, and that is because of executing more than one instruction at one time. On the reverse of serial computing which will be impractical in case of complex programs.

2.2 Scope of Parallel Computing

(Ananth Grama 2003, CHOUDHARY 1989)

2.2.1 Applications in Engineering and Design

Parallel computing has traditionally been employed with great success in the design of airfoils (optimizing lift, drag, stability), internal combustion engines (optimizing charge distribution, burn), high-speed circuits (layouts for delays and capacitive and inductive effects), and structures (optimizing structural integrity, design parameters, cost, etc.), among others.

Parallel computers have been used to solve a variety of discrete and continuous **optimization problems**. Algorithms such as Simplex, Interior Point Method for linear optimization and Branch-and-bound, and Genetic programming for discrete optimization have been efficiently paralleled and are frequently used.

2.2.2 Scientific Applications

The past few years have seen a revolution in high performance scientific computing applications. The sequencing of the human genome by the International Human Genome Sequencing Consortium and Celera, Inc. has opened exciting new frontiers in bioinformatics.

Thus, analyzing biological sequences with a view to developing new drugs and cures for diseases and medical conditions requires innovative algorithms as well as large-scale computational power. Indeed, some of the newest parallel computing technologies are targeted specifically towards applications in bioinformatics.

Also, applications in astrophysics have explored the evolution of galaxies, thermonuclear processes, and the analysis of extremely large databases from telescopes. Weather modeling, mineral prospecting, flood prediction, etc., rely heavily on parallel computers and have very significant impact on day-to-day life.

2.2.3 Commercial Applications

Parallel platforms ranging from multiprocessors to Linux clusters are frequently used as web and database servers. For instance, large brokerage houses on Wall Street handle hundreds of thousands of simultaneous user sessions and millions of orders.

The availability of large-scale transaction data has also sparked considerable interest in data mining and analysis for optimizing business and marketing decisions. The sheer volume and geographically distributed nature of this data require the use of effective parallel algorithms for such problems as association rule mining, clustering, classification, and time-series analysis ...etc.

2.2.4 Applications in Computer Systems

As computer systems become more pervasive and computation spreads over the network, parallel processing issues become ingrained into a variety of applications.

In computer security, intrusion detection is an outstanding challenge. In this case, data is collected at distributed sites and must be analyzed rapidly for signaling intrusion, and this requires effective parallel and distributed algorithms. In the area of cryptography, some of the most spectacular applications of Internet-based parallel computing have focused on factoring extremely large integers.

Embedded systems such as in a modern automobile consists of tens of processors communicating to perform complex tasks for optimizing handling and performance. In such systems also, traditional parallel and distributed algorithms for leader selection, maximal independent set, etc., are frequently used.

2.3 Von Neumann Architecture

(Barney n.d.) Named after the Hungarian mathematician/genius John von Neumann. Also known as “stored-program computer”. Since then, virtually all computers have followed this basic design (Figure 2.1), which is comprised of four main components:

- **Memory:**
  Read/write, random access memory is used to store both program instructions and data.
- **Control Unit:**
  Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- **Arithmetic Logic Unit:**
  Arithmetic Unit performs basic arithmetic operations.
- **Input/Output:**
  Input/Output is the interface to the human operator.

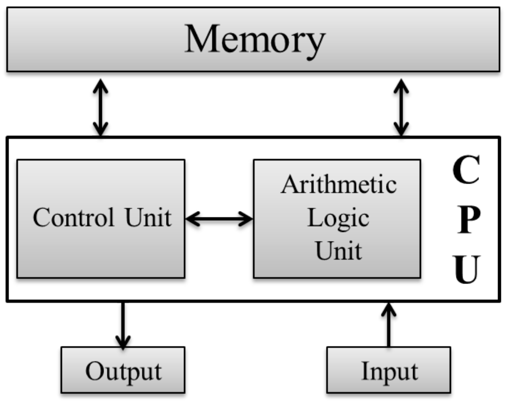


Figure 2.1: Von Neumann Architecture

Parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.

2.4 General Parallel Terminology

2.4.1 CPU/Processor/Core

CPU (Central Processing Unit) is a processor that carries out instructions sequentially. Higher frequency means faster calculations(Barney n.d.).

2.4.2 Task

A task is typically a program or program-like set of instructions that is executed by a processor(Barney n.d.).

2.4.3 Pipelining

Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line(Barney n.d.).
Pipelining is a type of parallel computing.

2.4.4 Communications

Parallel tasks typically need to exchange data. There are several ways to achieve this, such as through a shared memory bus or over a network...etc(Barney n.d.).

2.4.5 Synchronization

The coordination of parallel tasks in real time. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point(Barney n.d.).

2.4.6 Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication(Barney n.d.).

2.4.7 Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources(Barney n.d.).

2.5 Implicit and Explicit Parallelism

2.5.1 Implicit parallelism

(Joseph Awange 2018) Some of the programming languages (e.g. Matlab) are able to exploit multicore and multi-threading ability, automatically, partly or fully, without any special directives of the programming language. This characteristic of a programming language is called *implicit parallelism.*

The implicit parallelism may support only certain statements and may be efficient in case of large computational load. Mathematica also supports automatically the parallelization of some operations.

2.5.2 Explicit Parallelism

(Joseph Awange 2018) Parallelism provided by the code extension and controlled by the user, namely using or not using parallel execution is called *explicit parallelism.*

A feature of a programming language for a parallel processing system which allows or forces the programmer to annotate his program to indicate which parts should be executed as independent parallel tasks. This is obviously more work for the programmer than a system with implicit parallelism (where the system decides automatically which parts to run in parallel) but may allow higher performance.

2.6 Classification of Parallel Architectures

(Ruokamo 2018, Samir 2015) Among many proposed classifications of computer architectures and their data processing structure, "Flynn's taxonomy" classification which is presented by Michael Flynn in 1966 was the commonly used one, and it is still in use today in industry and many other fields. This classification is according to numbers of instruction streams and the number of data stream.

A stream is an ordered set of elements of the same nature (data or instructions). Each stream is independent of the others, and each element of a stream can consist of one (or more) objects or actions (ref: un algorithm génétic parallel sur....).

	Single Data stream	**Multiple Data stream**
Single Instruction stream	SISD	SIMD
Multiple Instruction stream	MISD	MIMD

Table 2.1: Flynn classification of computer architectures

The table 2.1 explains Flynn's classification as follow:

2.6.1 SISD (Single Instruction stream, Single Data stream)

As shown in Figure 2.2, *SISD* processor is a serial processor and it is capable of executing a single program instruction and a single data element at a time. Typically, *SISD* approach employs no parallelism at all and is not a choice for parallel computing platform.
*PE = Processing Element

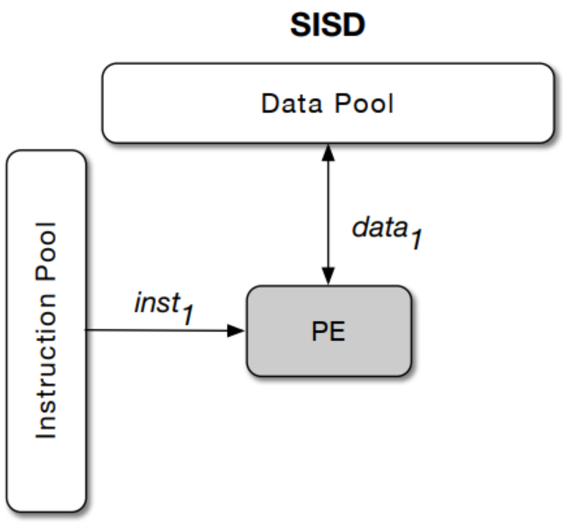


Figure 2.2: Single Instruction stream, Single Data stream Architecture

2.6.2 SIMD (Single Instruction stream, Multiple Data stream)

As shown in Figure 2.3, *SIMD* processor architecture promotes data processing parallelism. During the execution of one program instruction multiple data elements can be processed at a time.
*PE = Processing Element

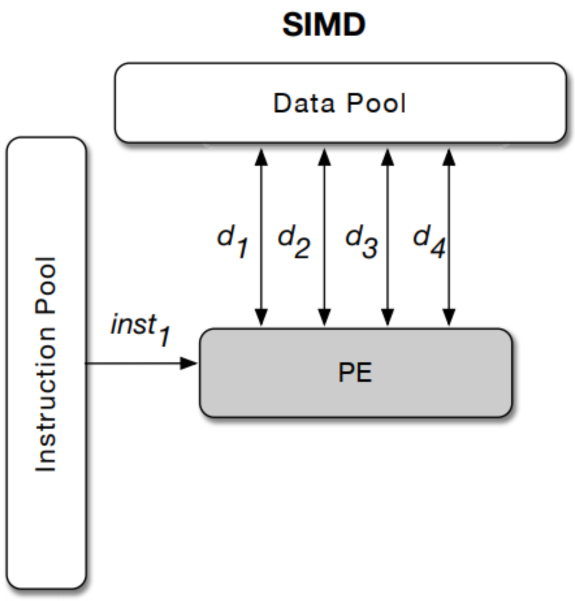


Figure 2.3: Single Instruction stream, Multiple Data stream Architecture

2.6.3 MISD (Multiple Instruction stream, Single Data stream)

Over one single data point, multiple instructions can be performed at one time. This is the work principle of MISD processor as shown in Figure 2.4.
*PE = Processing Element

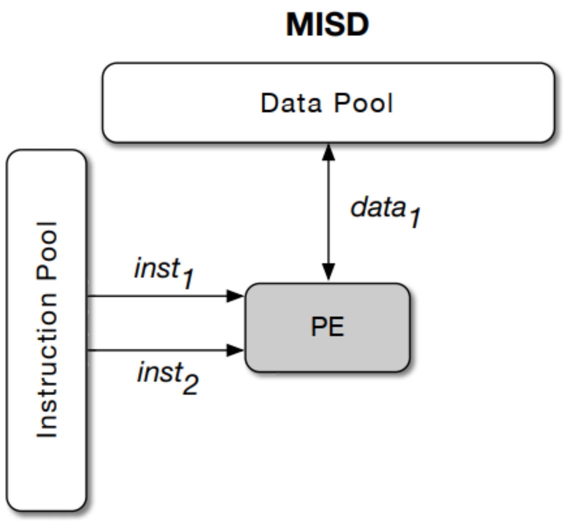


Figure 2.4: Multiple Instruction stream, Single Data stream Architecture

2.6.4 MIMD (Multiple Instruction stream, Multiple Data stream)

The most popular parallel computer architecture is *MIMD* architecture.

In *MIMD* architecture (Figure 2.5) multiple instructions can be processed over multiple data items at one time. In practice, this means that the processing involves multiple processing units i.e. processors are working in parallel.
*PE = Processing Element

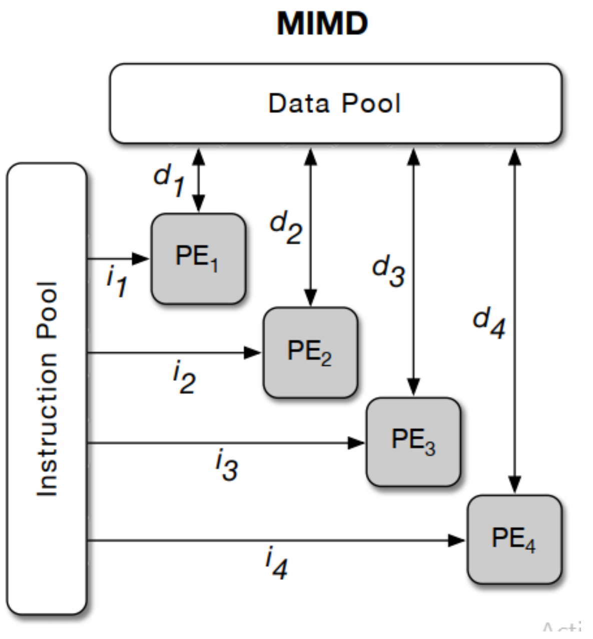


Figure 2.5: Multiple Instruction stream, Multiple Data stream Architecture

Further, in this architecture we distinguish two types of *MIMD* according to memory access:

a) Shared-memory *MIMD*:

In shared-memory *MIMD* machines (Figure 2.6), multiple processors can operate independently, but share the same memory resources (a global address space). While each change in a memory location made by one processor is visible to all other processors.

Synchronization between processors to get access to the memory can be done through:

- Semaphores: two operations P and V.
- Lock (mutex lock): binary semaphore used to protect a critical section.
- Monitors: high-level construction, implicit lock.

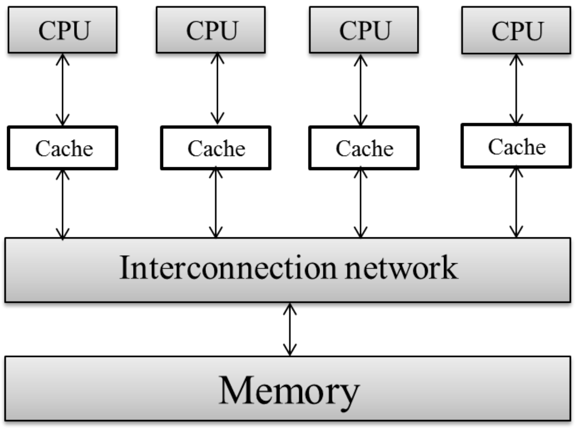


Figure 2.6: Shared-memory *MIMD*

b) Distributed-memory *MIMD* (Message-passing):

In Distributed-memory *MIMD* (Figure 2.7), each processor has its own private memory, and to connect processors memory a communication network is built, also the data exchange is through message passing.

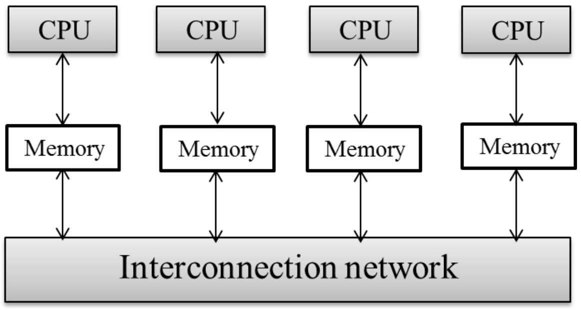


Figure 2.7: Shared-memory *MIMD*

2.7 Parallel Programming Models

(Barney n.d.) There are several parallel programming models in common use, let's explain some of them in brief:

2.7.1 Shared Memory (without threads)

In this programming model (Figure 2.8), processes/tasks share a common address space, which they read and write to asynchronously. Various mechanisms such as locks/semaphores are used to control access to the shared memory and to prevent deadlocks.

- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory.

- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality.

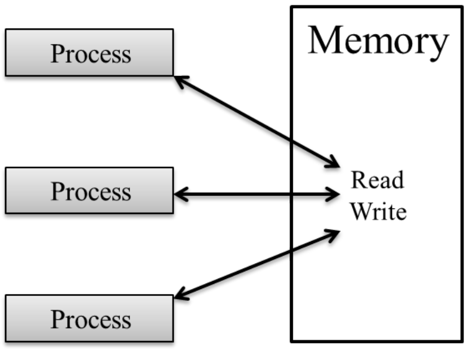


Figure 2.8: "Shared Memory" Parallel Programming Model

2.7.2 Threads

This programming model (Figure 2.9) is a type of shared memory programming. In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
For example:
- The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.
- **a.out** performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
- Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.

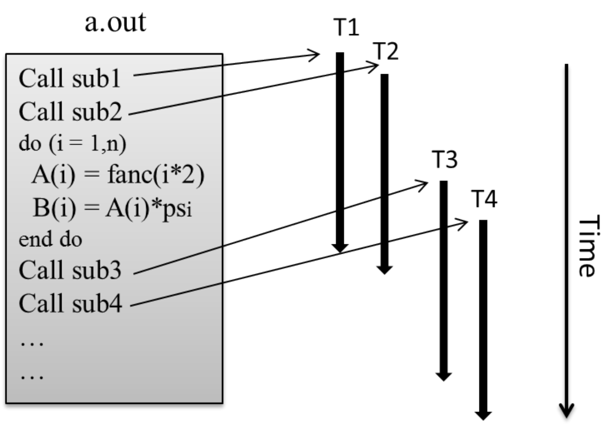


Figure 2.9: "Threads" Parallel Programming Model

2.7.3 Distributed Memory/Message Passing

In this model (Figure 2.10), a set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across number of machines. Exchanging data between tasks is through communications by sending and receiving messages.

Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

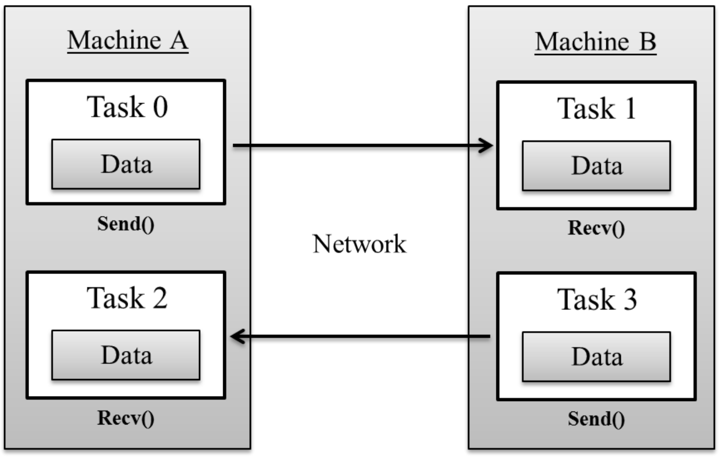


Figure 2.10: "Distributed Memory/Message Passing" Parallel Programming Model

2.7.4 Data Parallel

(Barney n.d.) In Data Parallel model the most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.

A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure. Those tasks perform the same operation on their partition of work, for example, add 4 to every array element.

2.7.5 SPMD and MPMD

a) Single Program Multiple Data (SPMD)

SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

- **SINGLE PROGRAM:** All tasks execute their copy of the same program simultaneously. This program can be threads, message passing or data parallel.
- **MULTIPLE DATA:** All tasks may use different data.

b) Multiple Program Multiple Data (MPMD)

Like SPMD, MPMD is actually a “high level” programming model that can be built upon any combination of the previously mentioned parallel programming models.

- **SINGLE PROGRAM:** Tasks may execute different programs simultaneously. The programs can be threads, message passing or data parallel.

- **MULTIPLE DATA:** All tasks may use different data.

Conclusion

This chapter introduced a general look of parallel computing. Whereas, at the end of this chapter the reader will be aware of the motivation of parallel computing and the scope of using it. Furthermore, the classification of computer architectures (SISD, SIMD, MISD, and MIMD). Finally, the different models of parallel programming.

Thus, we have completed the theoretical part of this report, which is totally represented in two chapters, the first one talks about the SMO algorithm and its process in detail, whereas the second one presents the concepts and architectures of parallel computing as discussed above. The next part of the report will discuss the practical results of the experiments made on the SMO algorithm with its two versions: the sequential and the proposed parallel one, and a comparison between them.

Part II

Contributions

Chapter 3

Analysis & Design

Chapter 3

Analysis & Design

Introduction

3.1 Analysis

Nowadays, every software developer or software company or computer science researchers seek to produce a very performed version of his product, and from the ways used to achieve that is Parallelism.

The application that we built we called it “ParSMO”. This application aim is showing the difference between the SMO sequential version and the proposed parallel SMO (as the first in the literature), by applying it on two test problems in terms of execution time, the near-optimum solution and the objective space density.

3.1.1 Description of the Application

“ParSMO” is an application implemented by Python with a view of applying the SMO algorithm on a single objective function; That’s because this algorithm destined to work on the single optimization problems. This application provides the user with two versions of the SMO algorithm.
The first is the sequential version, which needs the objective function, the swarm size and the number of iterations. This version applies the algorithm on step by step sequentially (Figure 3.1). The second one divides the SMO work between 2, 3 or 4 processes according to the user demand. Thus, this version will need 4 inputs, which are: the objective function, the number of processes, the swarm size and the number of iterations (Figure 3.2). Hence, the results which will appear to the user represented in a curve contains the initial population and the last one, also the execution time and the near-optimum solution.

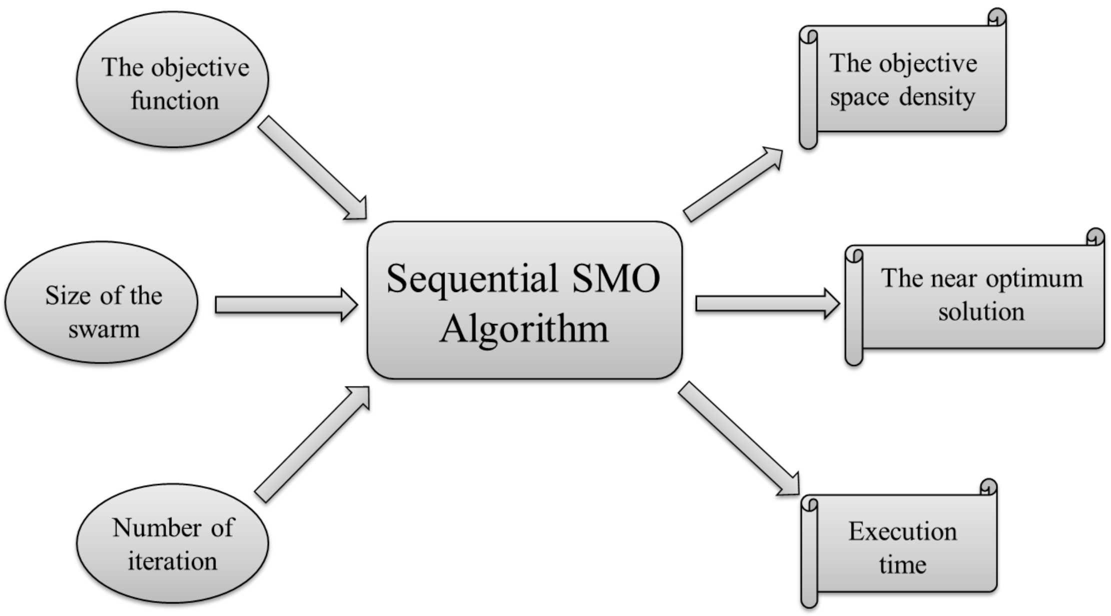


Figure 3.1: The Sequential version of SMO algorithm in "ParSMO"

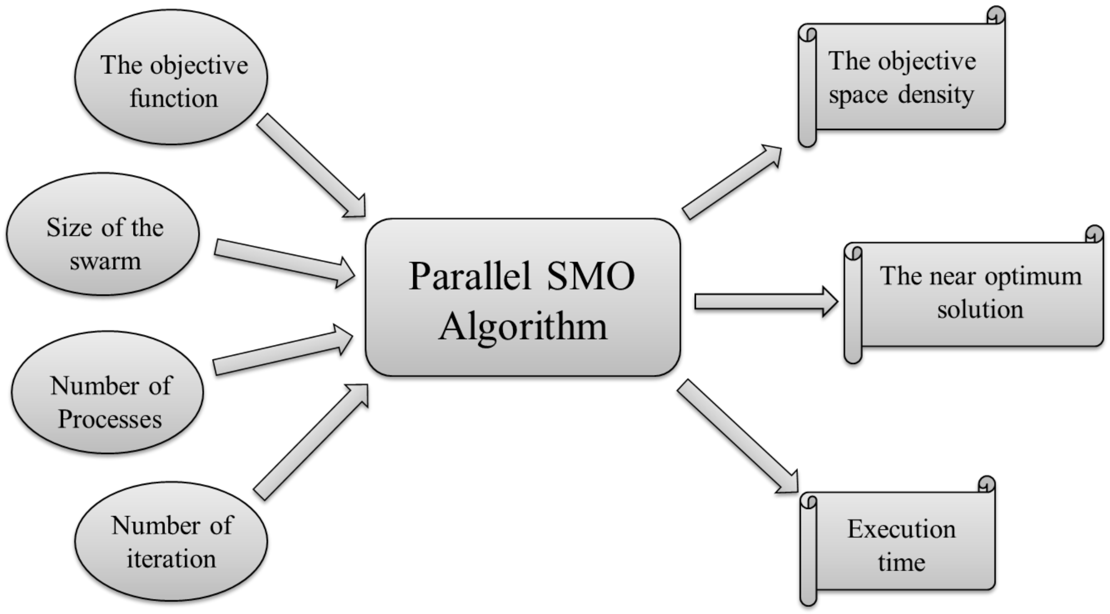


Figure 3.2: The Parallel version of SMO algorithm in "ParSMO"

The last part in "ParSMO" allows the user to make a comparison between the two ways in once (by a single click). The results here represented in three curves. The first for the sequential version, the second for the parallel one and the last contains the last population of sequential version and the last population of parallel one. Further, the execution time and the near-optimum solution of the two versions will be shown to the user. Thus, the user will be able to see the difference between the two versions in the application (Sequential & Parallel), in terms of time, objective space density and the near-optimum solution.

3.1.2 Project Development Cycle

To obtain the latest version of ParSMO application, we passed through several steps, explaining them below:

1. First, we have implemented SMO algorithm in a simple version using Python 3.7.
2. The second version of SMO algorithm is implemented using OOP paradigm.
3. And because that the aim of this project is making a parallel version of SMO algorithm as a contribution, we have chosen Multiprocessing package in Python to achieve that. Our choice is based on the lack of need for communication between processes.
4. Hence, we have implemented the first parallel SMO algorithm in the literature using Multiprocessing package.
5. As a step of experimentation, we have chosen from the problems presented on SMO paper (Jagdish Chand Bansal 2014) two test problems, which are:
   - Dekkers and Aarts (DA):
     - Formula:
     $$f(x) = 10^5 x_1^2 + x_2^2 - (x_1^2 + x_2^2)^2 + 10^{-5}(x_1^2 + x_2^2)^4 \tag{3.1}$$
     - Search range: [-20 , 20]
     - The number of inputs: 2
     - Optimum value: -24777
   - Six-hump Camelback (ShC):
     - Formula:
     $$f(x) = (4 - 2.1x_1^2 + x_1^4/3)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2 \tag{3.2}$$
     - Search range: [-5 , 5]
     - The number of inputs: 2
     - Optimum value: -1.0316

   Thus, the individuals will be presented as a two-dimensional vector (according to the number of inputs of the function, here 2).
6. The final step is creating ParSMO application which aims to test SMO algorithm on the previous test problems, by a sequential version of SMO or the parallel one, further the comparison between them in once.
   The comparison that the user can see will be in terms of execution time and near-optimum solution and the objective space density.

3.2 Design

After presenting the project aims, we have to identify the global architecture of ParSMO application using “Use Case”, “Class” and “Sequence” Diagrams.

3.2.1 Global Design

3.2.1.1 Use Case diagram

a) Definition:

Use case diagrams are used to gather a usage requirement of a system. They are used to identify functions and how roles interact with them (the primary purpose of use case diagrams), also for a high-level view of the system.

b) ParSMO use case diagram & description:

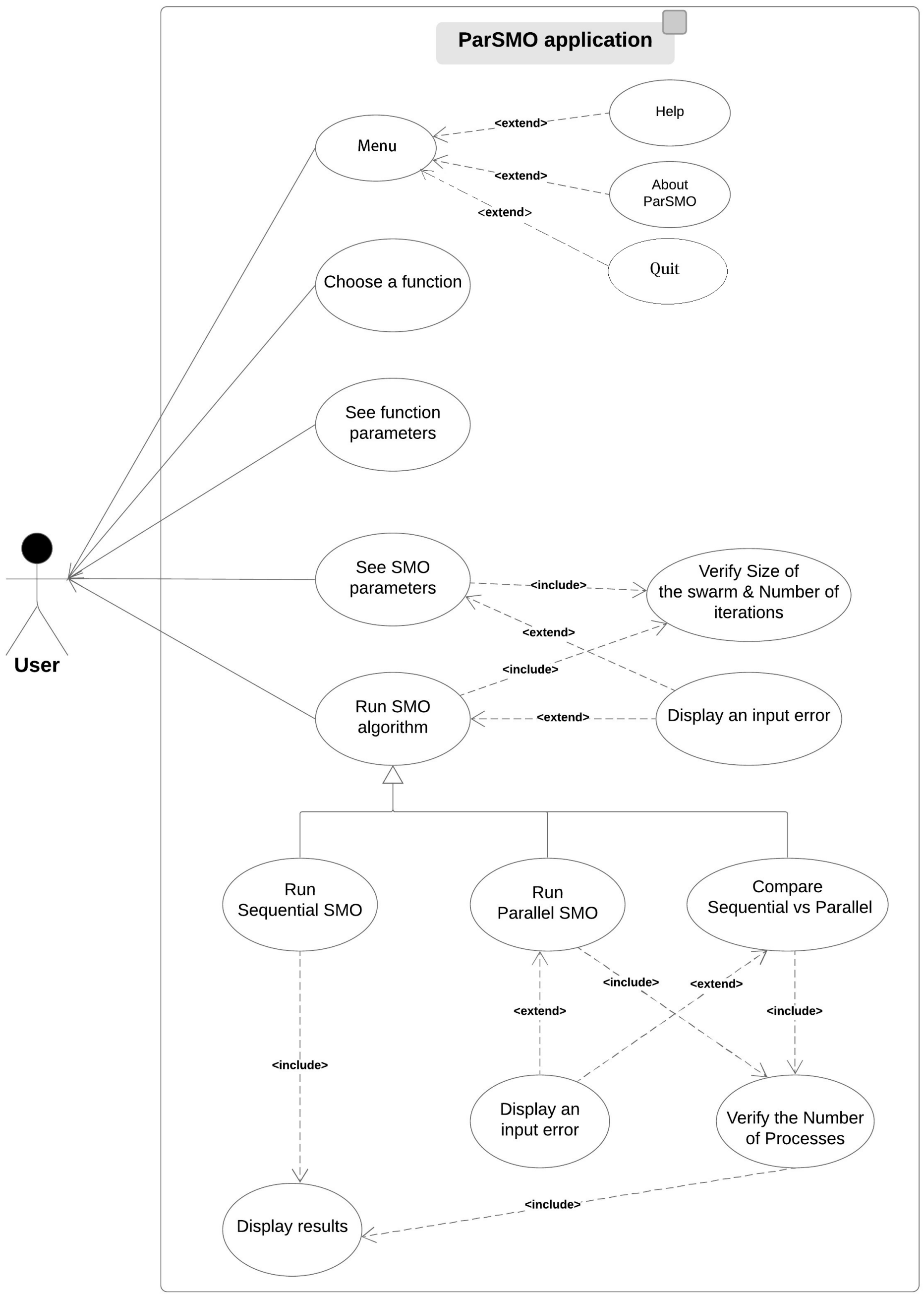


Figure 3.3: Use Case diagram of "ParSMO" application

The use case diagram of ParSMO (Figure 3.3) shows to us that the user of this application can do five acts, which are:

- Seeing "Menu", which is a menu bar in the home of ParSMO app, and it contains "Help", "About ParSMO" and "Quit".
- Choosing an objective function to test SMO algorithm on it.
- After choosing the function, the user can see the parameters of this function.

- Seeing SMO parameters, which are related to the swarm size and the number of iterations that the user entered.
- Running SMO algorithm, as a general use case. Whereas, the following use cases: sequential SMO, parallel SMO and both of them in once (comparison) are specialized use cases that the user can do each of them.

Two relationships between use cases are:

- <**include**>: is a relationship means that every time the base use case is executed, the use case included is executed as well.
- <**extend**>: is a relationship means that when the base use case is executed, the extend use case will be executed sometime but not every time.

All over, the relationships of this kind on our application are:

- The use cases “Help”, “About ParSMO” and “Quit” are extended from the main use case “Menu”, which means that when the user clicks on the menu bar “Menu”, it does not require a going to “Help”, “About ParSMO” or “Quit” necessarily, but they still a choices.
- The included use case “verify size & iteration” will be executed necessarily in each access of the user in the use cases “See SMO parameters” or “Run SMO” (with its three branches).
- Further, the use case “Display an input error” will not be done in each access on “See SMO parameters” or “Run SMO”, but just in case if the user made a mistake in the input process. Therefore, we put the relationship between them as an <extend> relationship.
- The same thing for the use cases “verify the number of processes” and “Display an input error”.
- Finally, when the user runs any of three specialized use cases of the general use case “Run SMO”, then “Display results” will be executed as well, as shown in (Figure 3.3).

3.2.1.2 Class diagram

a) Definition:

A class diagram is a UML diagram type that describes a system by visualizing the different types of objects within a system and the kinds of static relationships that exist among them. It also illustrates the operations and attributes of the classes. They are usually used to explore domain concepts, understand software requirements and describe detailed designs.

b) ParSMO class diagram & description:

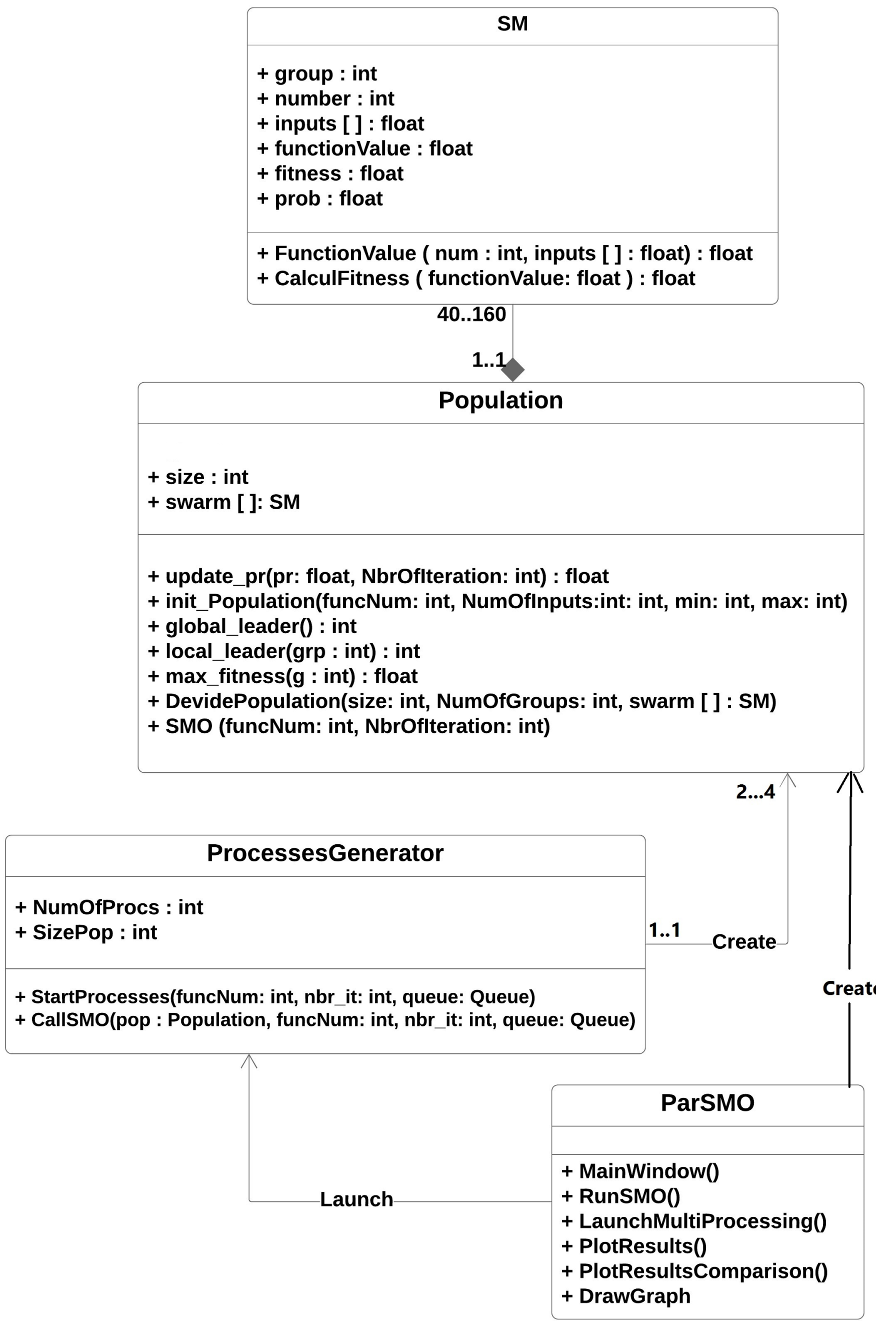

SM
+ group : int
+ number : int
+ inputs [ ] : float
+ functionValue : float
+ fitness : float
+ prob : float
+ FunctionValue (num : int, inputs [ ] : float) : float
+ CalculFitness (functionValue: float) : float
40..160
1..1
Population
+ size : int
+ swarm [ ]: SM
+ update_pr(pr: float, NbrOfIteration: int) : float
+ init_Population(funcNum: int, NumOfInputs:int: int, min: int, max: int)
+ global_leader() : int
+ local_leader(grp : int) : int
+ max_fitness(g : int) : float
+ DevidePopulation(size: int, NumOfGroups: int, swarm [ ] : SM)
+ SMO (funcNum: int, NbrOfIteration: int)
2...4
ProcessesGenerator
+ NumOfProcs : int
+ SizePop : int
+ StartProcesses(funcNum: int, nbr_it: int, queue: Queue)
+ CallSMO(pop : Population, funcNum: int, nbr_it: int, queue: Queue)
1..1
Create
Create
ParSMO
+ MainWindow()
+ RunSMO()
+ LaunchMultiProcessing()
+ PlotResults()
+ PlotResultsComparison()
+ DrawGraph
Launch

Figure 3.4: Class diagram of “ParSMO” application

As shown in the previous class diagram (Figure 3.4), ParSMO application which is implemented using Object-Oriented Programming (OOP) paradigm, contains four classes: SM, Population, ProcessesGenerator and ParSMO.

- **“ParSMO” class:**

It is the main class which contains the user interfaces, and also can create a population then run the sequential SMO, or launch "Processes Generator" to create processes in order to run SMO in parallel. Hence, it contains the methods that cover all the application work (running sequential SMO, parallel one, show results...etc), which are:
- MainWindow(): It represents the main user interface.
- RunSMO(): The operation that runs SMO sequentially after creating a population.
- LaunchMultiprocessing(): Launched when the user chooses to use the parallel version of SMO, after specifying the number of processes uses in implementation.
- PlotResults(): Is the operation that shows the result window after finishing the implementation of sequential or parallel SMO.
- PlotResultsComparision(): This operation like the previous one (showing results), but it's destined for the case of comparison between the sequential SMO and the parallel one, whereas the results window contains the results of the two versions and the comparison between them.
- DrawGraph(): An operation used to plot the objective space of a function.

- **"ProcessesGenerator" class:**

  "ProcessesGenerator" is the responsible of generating processes when the user chooses to work by the parallel SMO algorithm. "ProcessesGenerator" class has two attributes: the size of the population and the number of processes. Further, it has two operations: "StartProcesses" and "CallSMO".
  - StartProcesses(): This operation divides the population size into sub-sizes according to the number of processes given by the user, then it creates a population for each process. Finally, creating the processes and launch them to apply SMO severally.
  - CallSMO(): Is the operation that is taken by each process to run SMO and get the results after finishing.

- **"Population" class:**

  It is the class which contains the SMO algorithm as an operation to be applied on the population created. "Population" class has two attributes which are the size of the population and the population (swarm) itself. All the following operations are used by the main operation SMO(): update_pr, init_Population, global_leader, local_leader, max_fitness, and DevidePopulation.
  - update_pr(): The operation that updates pr[1].
  - init_Population(): This operation defines the first values of the population (the first generation) as SMs (individuals).
  - global_leader(): Determines the global leader of the swarm.
  - local_leader(): Determines the local leader of a group in the swarm.

[1]pr is an SMO algorithm parameter, see 1.3.3

- max_fitness(): Selects the number of the individual that has the best fitness.
- DevidePopulation(): This operation is responsible for dividing the population into smaller groups or combining them in a single group (explained in 1.3.2).

Furthermore, the main operation in the class and in the application at all is called SMO(), which contains the whole process of SMO algorithm, also it returns the result (the first and the last generation) after finishing the process.

- **"SM" class:**

  It is the class that represents an individual (a spider monkey) in the population. Each SM has a group, a number, inputs (which are his value), a function value, a fitness and probability (explained in section 1.3.2). This class has two operations, the first one is FunctionValue() which is the operation that calculate the function value of each individual, and the second one is CalculFitness() and it returns the fitness value of each individual according to his function value.

c) Relations between the classes:

- The population is a group of SMs (individuals), so a composition relation between the "Population" class and "SM" class. The population can contain 40 to 160 SMs, but each SM should be in a single population.
- A directed association relation between "ParSMO" and "ProcessesGenerator" classes, so "ParSMO" can launch the "ProcessesGenerator".
- "ParSMO" can create a population, so a directed association relation between "ParSMO" and "Population".
- "ProcessesGenerator" can create two, three or four populations, so also a directed association relation between "ProcessesGenerator" and "Population".

3.2.1.3 Sequence diagram

a) Definition:

A sequence diagram is a model of the interactions between objects in a single use case. It illustrates how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed. In simpler words, a sequence diagram shows the different parts of system behavior in a 'sequence' to get something done.

b) ParSMO sequence diagram & description:

The sequence diagrams (Figures 3.5, 3.6) show that the user of ParSMO application can do the following:

- Selecting a function (test problem) to run SMO algorithm on it.
- Seeing the parameters of the selected function.
- Seeing the SMO parameters, after ParSMO system receiving a "valid input" message from the verification system if the inputs are correct, and if it is not, the application system will show to the user an input error window as a warning and the user will not be able to see the SMO parameters until he enters a correct inputs (size number of iteration) as the conditions said.

However, the user can choose to test sequential SMO or parallel one or make a comparison between them.
The following part explains these three choices.

1. **Sequence diagram of sequential SMO:**

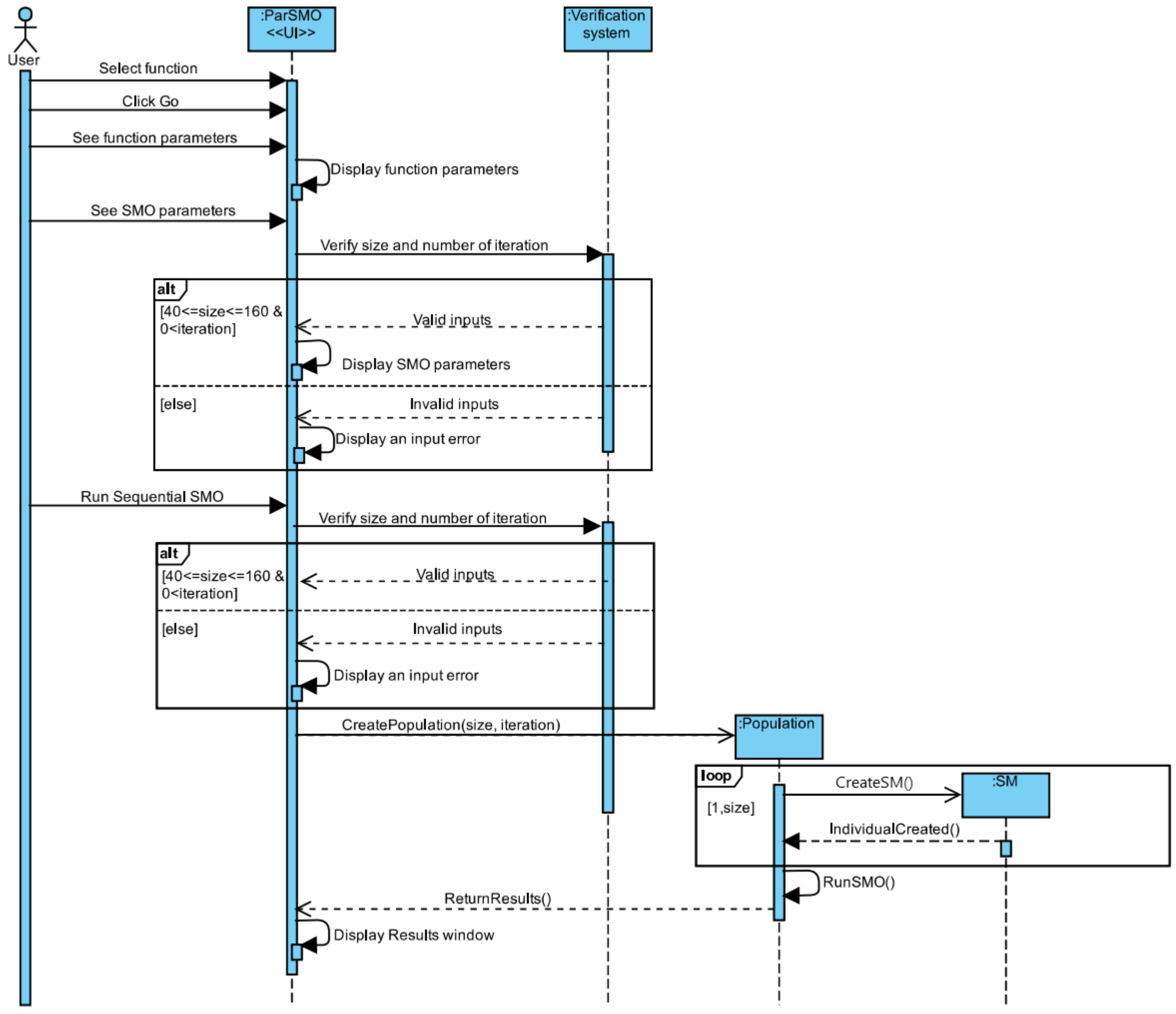


Figure 3.5: Sequence diagram of sequential SMO

The sequence diagram of the sequential object of the SMO algorithm as shown in (figure 3.5) above allows the user to run the sequential version of SMO on the function selected. The process followed by the sequential SMO object is:

- The ParSMO system verifies the size and the number of iteration entered by the user by sending a verification message to the verification system of ParSMO. If the inputs are wrong according to the conditions mentioned, the verification system returns an "invalid inputs" message to ParSMO, which in turn displays a warning (input error) to the user. When the user enters the valid both size and number of iterations, the ParSMO system sends a command: "Create Population", with the parameters: size and iterations, to the class Population.
- The last message makes a population created, in the size that sent with the message.
- To population be created, a loop of creating SMs are executed with the specified size.
- Each individual created by returning a message to the Population class.
- After finishing the creating of the population, the Population class runs the SMO algorithm on itself.
- Return the results to the ParSMO system after completing the implementation of the SMO algorithm, which in turn displays them to the user in a window.

2. **Sequence diagram of parallel SMO:**

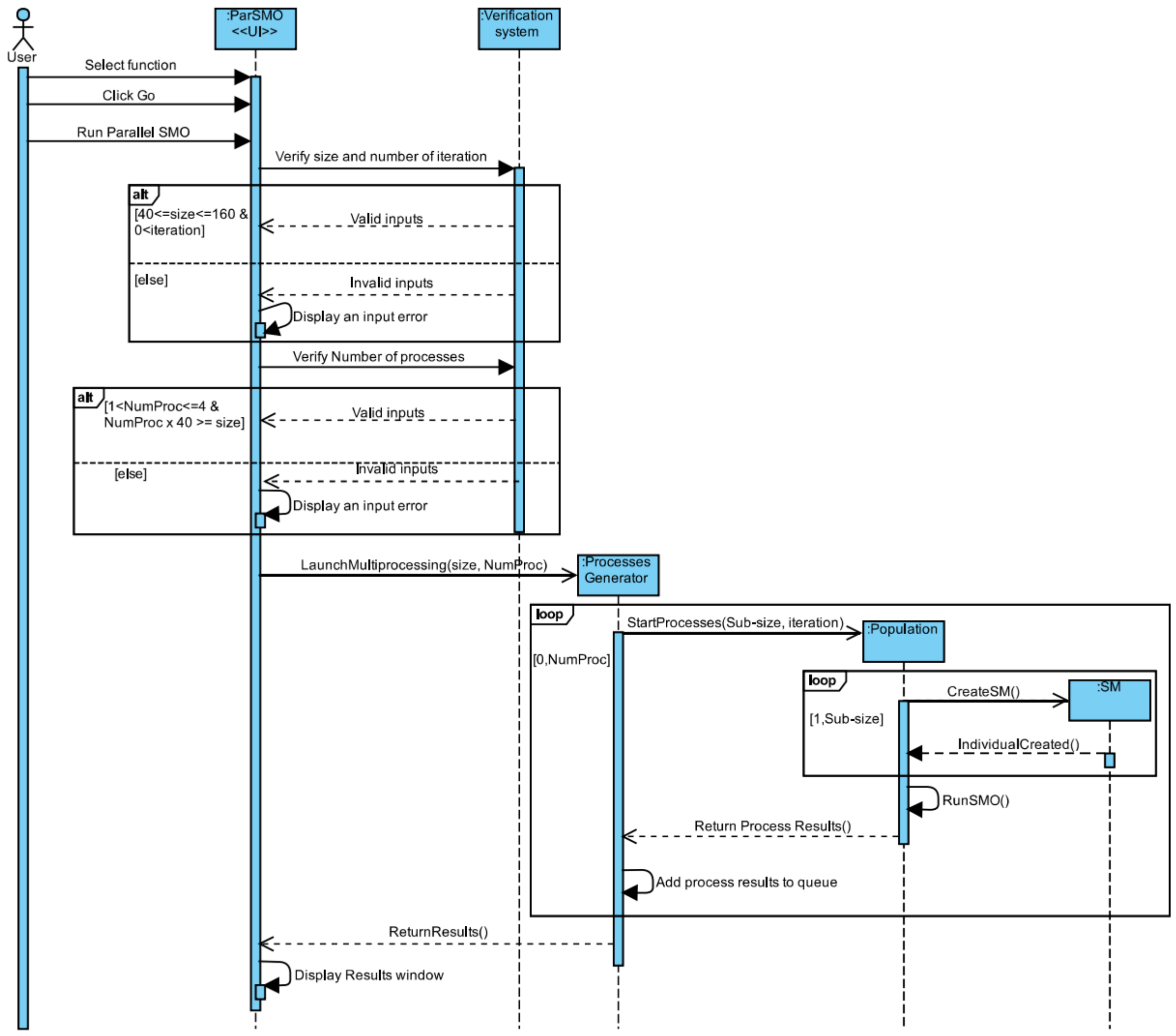


Figure 3.6: Sequence diagram of parallel SMO

The sequence diagram of the parallel object of the SMO algorithm as shown in (figure 3.6) above allows the user to run the parallel version of SMO on the function selected. The process followed by the parallel SMO object is:

- The ParSMO system verifies the size and the number of iteration entered by the user by sending a verification message to the verification system of ParSMO. If the inputs are wrong according to the conditions mentioned, the verification system returns an “invalid inputs” message to ParSMO, which in turn displays a warning (input error) to the user. When the user enters the valid both size and number of iterations, the ParSMO system sends a command: “Launch Multiprocessing”, with the parameters: size, iterations, and the number of processes to the class “ProcessesGenerator”.
- In a loop of the number of the processes, populations are created by the command

"Start Processes" for each one of them, with the sub-size [2] and the number of iterations as parameters. Thus, for each processes, a population with sub-size is created.

- Each process creates its own population by the loop that explained before (in the sequential object) and runs the SMO algorithm on its population.
- After a process completes the implementation of the SMO algorithm, it returns the results to the "Processes Generator", which in turn adds it to the queue of the results.
- "Processes Generator" ensures that all the processes finished their implementation of SMO and sent the results, then it sends the whole results to the ParSMO system, which in turn displays them to the user in a window.

3.2.2 Detailed Design

In order to illustrate ParSMO application deeply, let's explain the Python code of it and how we implemented all its methods.

- SM class:
  The name of this class means Spider Monkey, which represent an individual in optimization algorithms. The class SM has two methods:
  - FunctionValue()
  As shown in the next algorithm (Algorithm 11), the method FunctionValue() takes as inputs two parameters: Num which is the function number [3], and the second one is the inputs of the function specified, which are the value of the individual.

Algorithm 10 The method that calculate the function value

```
1:  function FunctionValue(Num, inputs[])
2:      if $Num == 1$ then
3:          $results = 10^5 * inputs[0]^2 + inputs[1]^2 - (inputs[0]^2 + inputs[1]^2)^2 + 10^{-5} * (inputs[0]^2 + inputs[1]^2)^4$
4:          //Here, result will contain the value of "Dekkers and Aarts" function using the inputs
5:           of the method as inputs of it.
6:      else
7:          //Num == 2
8:          $result = (4 - 2.1 * inputs[0]^2 + inputs[0]^4/3) * inputs[0]^2 + inputs[0] * inputs[1] + (-4 + 4 * inputs[1]^2) * inputs[1]^2$
9:          //Here, result will contain the value of "Six-hump camel back" function using the inputs
10:          of the method as inputs of it.
11:     return result
```

[2]sub-size is the size of the population of each process, where: sub-size = size / number of processes

[3]The function number is just an identification of the function, I put 1 for "Dekkers And Aarts" function, and 2 for the other "Six-hump camel back".

- CalculateFitness()

It is the method that calculates and returns the fitness value of each individual based on his function value (See explanation of fitness evaluation in section 1.2.2).

Algorithm 11 The method that calculate fitness

```
1: function CalculateFitness(funcVal)
2:     if $funcVal \geq 0$ then
3:         return $(1/(1 + funcVal))$
4:     else
5:         //Num == 2
6:         return $(1 + abs(funcVal))$
7:         // "fabs", it returns the absolute value of the passed parameter
```

Note: these methods are used by the other objects of other classes too.

- Population class:
  This is the class that contains the main method in this application, which is SMO algorithm. We had put it in Population class because the SMO algorithm is applied to objects of this class (a created populations).
  In Population class we defined the following methods:
  - update_pr()
  It is a method that updates the parameter "pr" through iterations by the equation presented in the pseudo-code of this method below (12).

Algorithm 12 The method of updating the parameter "pr"

```
1: function update_pr(pr, NbrOfIteration)
2:     $pr = pr + (0.4 - 0.1)/NbrOfIteration$
3:     // 0.4 and 0.1 are the bounds of "pr".
4:     return $pr$
```

- init_Population()

The first step in SMO algorithm process is initiating SMs (individuals) and this is the method which is responsible of creating individuals and giving them their first values (inputs, group, number, function value, and fitness).

The pseudo-code below (Algorithm 13) shows this process, where the group of all the individuals is 0 (one group), and the probability value of all of them are initiated by 0 (because we can not calculate it now), and their numbers are sequential. Finally, their inputs are initiated by the equation specified in the algorithm below (Algorithm 13).

Algorithm 13 The method of initialization of the population

```
1: function INIT_POPULATION(funcNum, NbrOfInputs, min, max)
2:     $group = 0$
3:     // All the population are in a single group 0 initially.
4:     $prob = 0$
5:     // All the individuals has an initial value in their probability 0.
6:     for $i = 0..size$ do
7:         inputs = []
8:         for $j = 0..NbrOfInputs$ do
9:             // NbrOfInputs is the number of inputs of the function which we will apply SMO on it.
10:            $a = min + U(0, 1) \times (max - min)$
11:            $inputs.add(a)$
12:        $swarm[i] = SM(funcNum, group, i, inputs, prob)$
13:        // i is the number of each SM.
14:        // “swarm” is the list of individuals (objects of SM class).
15:        $a = swarm[i].fitness$
16:        // This instruction added to calculate the function value, then the fitness of each individual.
```

- global_leader()

It is the method responsible for setting the global leader of the swarm in steps of SMO algorithm based on the fitness of the individuals, where the SM who has the best fitness will be the global leader.

Algorithm 14 The method of selecting the global leader

```
1: function GLOBAL_LEADER()
2:     $GL = 0$
3:     // set the SM number 0 in the swarm a global leader initially.
4:     for $i = 0..size$ do
5:         if $swarm[i].fitness > swarm[GL].fitness$ then
6:             $GL = i$
7:     return $GL$
```

- local_leader()

In the same way in the previous method, the local leader of a local group is selected when he has the best fitness value among his group. The following pseudo-code (Algorithm 15) represents this process.

Algorithm 15 The method of selecting the local leader of a local group

```
1: function LOCAL_LEADER(group)
2:     $groupMembers$ = members(groups)
3:     // "members" is a method takes the number of a group as input and returns the members numbers of this group in a list.
4:     $LL = groupMembers[0]$
5:     // set the first member of this local group a local leader initially.
6:     for $i$ in $groupMembers$ do
7:         if $swarm[i].fitness > swarm[LL].fitness$ then
8:             $LL = i$
9:     return $LL$
```

- max_fitness()

This method (Algorithm 16) returns the best fitness of a group in the population.

Algorithm 16 The method of getting the max fitness of a group

```
1: function MAX_FITNESS(group)
2:     $groupMembers$ = members(groups)
3:     $maxfitness = swarm[groupMembers[0]].fitness$
4:     // set the fitness of the first member of this local group a max fitness initially.
5:     for $i$ in $groupMembers$ do
6:         if $swarm[i].fitness > maxfitness$ then
7:             $maxfitness = swarm[i].fitness$
8:     return $maxfitness$
```

- DividePopulation()

In the last phase of SMO algorithm and under some necessary conditions, SMO divides the population into smaller groups and if the maximum number of groups in the swarm is reached, the groups are combined into a single group. This is the aim of this method, as explained below (Algorithm 17).

Algorithm 17 The method of dividing the population

```
1: function DividePopulation(NumOfGroups)
2:     $div, mod$ =divmod($size, NumOfGroups + 1$)
3:     // "divmod" is a method integrated into the Python language, which takes two integer numbers and returns their "div" and "mod".
4:     $start = 0$
5:     $end = div$
6:     for $i = 0...NumOfGroups + 1$ do
7:         if $i < mod$ then
8:             for $j = start...end + 1$ do
9:                 $swarm[j].group = i$
10:            $start = end + 1$
11:            $end = (end + 1) + div$
12:        else
13:            for $j = start...end$ do
14:                $swarm[j].group = i$
15:            $start = end$
16:            $end = end + div$
```

- SMO()

As we mentioned before, this is the main method in ParSMO application. SMO() method will use all the previous methods to complete the whole process of SMO algorithm. Firstly, and as a necessary act, creating a population by creating an object of the class Population, where that requires a specified size for this population created (between 40 and 160). Secondly, two parameters of the SMO method should be identified, which are the function number (function identifier) and the number of iterations. Thirdly, SMO method starts by initiating SMO parameters, similar to the different counters (iteration, number of groups, local limit count, global limit count) which all initiated by 0, then the maximum number of groups (MG), pr, local leader limit, and global leader limit. Furthermore, initiating the function parameters min, max and the number of function inputs, which are selected by a method return them according to the function identifier.

After initiating all the needs of SMO algorithm. The next step is initialization the values of all the individuals (SMs) in the population by the method init_population (Algorithm 13). Then, selecting the global leader (Algorithm 14), the local leader (Algorithm 15) and saving the initial generation of the population by a method called it getInputs() in order to plot it with the last one to see the progress of the objective space density.

In this method (Algorithm 18), the main loop of SMO algorithm contains the six steps of SMO algorithm, further updating "pr" and iteration number.

Algorithm 18 SMO method

```
1: function SMO(funcNum, NbrOfIteration)
2:     $iteration = 0$
3:     $MG = size//10$
4:     // MG: the maximum number of groups, the integer result of dividing size of the population on 10.
5:     $min, max, NumOfInputs =$ parameters($funcNum$)
6:     // Getting the function parameters
7:     $pr = 0.1$
8:     // pr: an SMO parameter which increases through iterations
9:     $LocalLimitCount[MG] = 0$
10:    $GlobalLimitCount = 0$
11:    // LocalLimitCount and GlobalLimitCount are SMO parameters (counters)
12:    $NumOfGroups = 1$
13:    // NumOfGroups: is the number of local groups in the population
14:    $GlobalLeaderLimit = size$
15:    // GlobalLeaderLimit: an SMO parameter identified in [size/2 , size*2], and had set equal to size.
16:    $LocalLeaderLimit = size * NumOfInputs$
17:    // LocalLeaderLimit: an SMO parameter identified as written
18:    init_Population($funcNum, NumOfInputs, min, max$)
19:    // Initialization of the population
20:    $GLeader =$ global_leader()
21:    // Selecting the global leader of the population
22:    $LLeaders[MG] = 0$
23:    // LLeaders: a list (of MG size) of local leader numbers
24:    $LLeaders[0] =$ local_leader(0)
25:    // Selecting the local leader of the group 0, where the population is in a single group (0) initially
26:    $init\_input1, init\_input2 =$ getInputs()
27:    // Getting the first generation
28:    while $iteration < NbrOfIteration$ do
29:        // The six phases of SMO algorithm, then updating pr and number of iterations
30:        LLP()
31:        GLP()
32:        GLL()
33:        LLL()
34:        LLD()
35:        GLD()
36:        update_pr($pr, NbrOfIteration$)
37:        $iteration = iteration + 1$
```

```
    $input1, input2$ = getInputs()
    // Getting the last generation
    $results$ = []
    // The list that we will store the results in
    for $i = 0...size$ do
        $individual$ = []
        // A list to save individual results in.
        $individual$.add($init\_input1$)
        $individual$.add($init\_input2$)
        $individual$.add($input1$)
        $individual$.add($input2$)
        $individual$.add(FunctionValue($funcNum$, [$input1, input2$]))
        $results$.add($individual$)
    return $results$
```

As we see in the previous pseudo-code of SMO method (Algorithm 18), and after finishing the main loop, the first generation and the last one are gathered to store in a list (called it results) in order to plot them as results that appear to the user at the end.

- ProcessesGenerator class:
  "ProcessesGenerator" is a class launched when the user wants to take the parallel way in applying SMO, where the this class has two methods:
  - StartProcesses()
  This is the method that be called by "LaunchMultiprocessing" method in the main class when the user wants to run parallel SMO. "StartProcesses" method has three parameters, which are the function number and number of iteration and a queue in order to store results in it.
  Firstly, the method creates a list to store objects and other for processes. Then, it divides the population size on the processes number, creates a population with this sub-size for each process, and these population objects will be stored in their list. Finally, two loops are implemented, the first one creates the processes, launches CallSMO method for each one of them, starts them and stores them in their list. The second one launches the "join" function for each process in order to wait for each other to finish.

Algorithm 19 The method of generating processes

```
function ProcessesGenerator(funcNum, nbr_itr, queue)
    $processes$ = []
    $objects$ = []
    $div, mod$ = divmod($SizePop, NumOfProc$)
    // “divmod” returns the “div” and “mod” of population size and number of processes.
    for $i = 0...NumOfproc$ do
        if $mod = 0$ then
            $SizePop = div$
        else
            $SizePop = div + 1$
            $mod = mod - 1$
        $pop$ = Population($SizePop$)
        $objects$.add($pop$)
    for $i = 0...NumOfproc$ do
        $proc$ = Process(target = CallSMO, args = ($objects[i], funcNum, nbr\_itr, queue$))
        // Launch CallSMO method for each process.
        $processes$.add($proc$)
        $proc$.start()
        // Start the process.
    for $process$ in $processes$ do
        $process$.join()
```

- CallSMO()

A small method launches SMO method on the population (of a process) given as a parameter and stores the results in the queue.

Algorithm 20 Call SMO method for a population

```
function CallSMO(Pop, funcNum, nbr_itr, queue)
    $result$ = Pop.SMO($funcNum, nbr\_it$)
    // Pop is a population object, and above we launch SMO method on Pop
    $queue$.put($results$)
    // The results saved in the queue
```

- ParSMO class (The main):
  It is the main class that contains the GUIs and calling of objects and methods of other classes, and all of that is explained above.

Conclusion

In this chapter, we have presented our application ParSMO in three main sections. Firstly, we have illustrated the global architecture of ParSMO application in terms of the way SMO work, where we presented the inputs and the output of sequential SMO and parallel one. Then, in the second section, we have given three UML diagrams of ParSMO application. Use case diagram, which shows the main actions of the user on the ParSMO system, including the necessary actions and the optional ones. Class diagram, which illustrates the four ParSMO classes with the relationships between them, and then a presentation of these classes with an explanation of their relationships. The last diagram is the sequence diagram, which is presented in two parts, the first one represents the object sequence when applying sequential SMO, and the other when applying the parallel one. Finally, the last part contains a detailed explanation of ParSMO body, where we have illustrated each method in each class of our project in order to the reader takes a general idea about the implementation of ParSMO.

In the next chapter, we will touch on the ParSMO graphical interface that the user can handle and will illustrate the usage of ParSMO by presenting all the GUIs with an explanation of them.

Chapter 4

Implementation & Test

Chapter 4

Implementation & Test

Introduction

After we had presented ParSMO application design, by UML diagrams with an explanation and a detailed presentation of the methods used in the code. This chapter aims to explain how we had implemented our application and a discussion of the results that we got after a test of all that ParSMO can do.

This chapter includes three sections. The first section introduces briefly the development tools and languages that we have learned and exploited them in the realization of our project. The second one presents the main implementation results of our final application. In the last section, we present and discuss some experimental results.

4.1 Development Tools and Languages

4.1.1 Python programming language

Python is an intelligent programming language that we have used in the implementation of our application. It is easy to learn, because it is flexible, and its syntax doesn't hard to learn. A Python program is short than other languages' programs, because of the availability of many implemented functions. Python is an open source and untyped programming language. It is available for all the operating systems (Windows, Linux, Mac OS).

4.1.2 PyCharm Programming Editor

PyCharm is an open-source Integrated Development Environment (IDE), used for Python programming. It is a powerful coding assistant, it can highlight errors and introduces quick fixes based on an integrated Python debugger. It is a suitable editor for writing and testing many lines of code and classes since it offers a structural project view and quick file navigation.

4.1.3 Tool Kit Interface “Tkinter” Package

Tool Kit Interface, in short “*Tkinter*” (*An Introduction to Tkinter* 2005), it is an open source Graphical User Interface (GUI) package. It is intended for Python programming language. We have preferred the Tkinter toolkit for developing GUIs of our application, because it is simple to learn it, and it is a powerful toolkit. It is available on both operating systems (Windows, Linux, and Mac OS).

4.1.4 Plotting Library “matplotlib”

matplotlib (*matplotlib* 2017) is an open source Python library. It is used for 2D and 3D plotting. With a short-code, one can generate plots, histograms, power spectra, bar charts, error charts, scatter plots, etc, and can produce quality figures for the generated plots in a variety of hard-copy formats. Line styles, font properties, and axes properties are controlled by simple lines of code. Since in our project we have 3D results that must be plotted, thus we have chosen matplotlib to plot them.

4.1.5 Multiprocessing

Multiprocessing (TechoPedia 2019) is an open source Python library, which refers to the ability of a system to support more than one processor at the same time. Applications in a multi-processing system are broken into smaller routines that run independently. The operating system allocates these threads to the processors improving the performance of the system. In our application, we decided to use this paradigm in order to parallelize the SMO algorithm.

4.1.6 Document Preparation System LATEX

LaTeX (Lamport 2019) is a powerful and flexible typesetting system for producing high quality technical and scientific papers. It based on the tags language. It follows the design philosophy of separating presentation from content, thus authors focus on what they are writing, not on what is displayed, because the appearance is handled by LaTeX. The appearance includes many aspects, document structure (part, chapter, section, ..etc), figures, cross-references and bibliographies. It is more familiar to a computer programmer, because it follows the code-compile-execute cycle.

4.1.7 Online LATEX editor (OVERLEAF)

Oerleaf (*Overleaf* 2019) is a free online editor for drafting papers, based on LaTeX system. It supports a powerful spell-checker, code auto-completion, and a *pdf* displayer. In addition, "*Overleaf*" gives us the ability to edit and work on our Latex project from anywhere and any computer. We have used "*Overleaf*" website to draft our report and make our presentation because it produces high-quality papers and talks.

4.1.8 Online UML diagrams editor (VP Online)

Visual-Paradigm Online (*VP Online* 2018) in short "*VP Online*", is an online diagrams editor that makes the user draw various types of diagrams, including the ArchiMate diagram, Flowchart, UML diagrams...etc, and it has more than 1300 templates and many tutorials of different kinds of diagrams for the user. "*VP Online*" has a free cloud repository that keeps the designs of the user in a work-space that can be accessed by a team added and manipulated by the user itself. During the conception of our application ParSMO, we have used "*VP Online*" editor to draw and plan all the UML diagrams that we have needed, which are: use case diagram, class diagram, and sequence diagrams.

4.2 Implementation & test

The studied algorithm in the previous chapter 3 are implemented using oriented object programming (OOP) paradigm and using a set of software and hardware which are summarized in the following table 4.1.

Software/Hardware	**Version**
OS	Microsoft Windows 8.1 Professional, 64bits
CPU	Intel(R) Core(TM) i5-4300M CPU @2.50GHz
RAM	8.00Go
Python Interpreter	3.7.3
PyCharm	2016.3.1
matplotlib	3.0.3
Tkinter	8.6

Table 4.1: Software/Hardware versions

Application Home

Figure (4.1) illustrates the basic GUI (Graphical User Interface) that allows the user to choose the problem (function) which will test SMO algorithm on it. This GUI contains a menu bar with three options (Help, About ParSMO and Quit for exit the application).

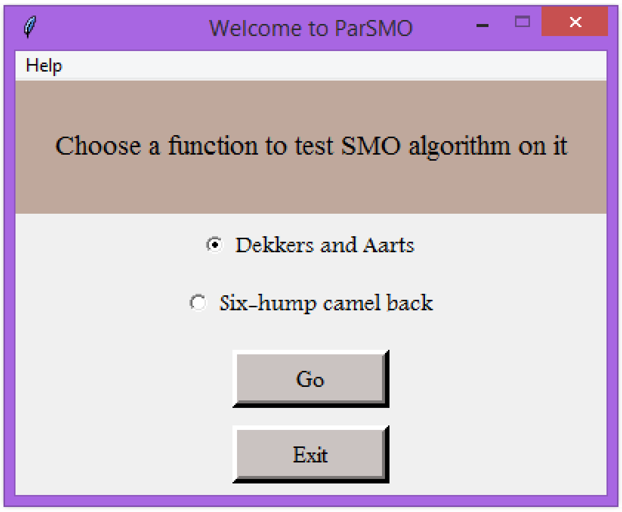


Figure 4.1: "ParSMO" Home page

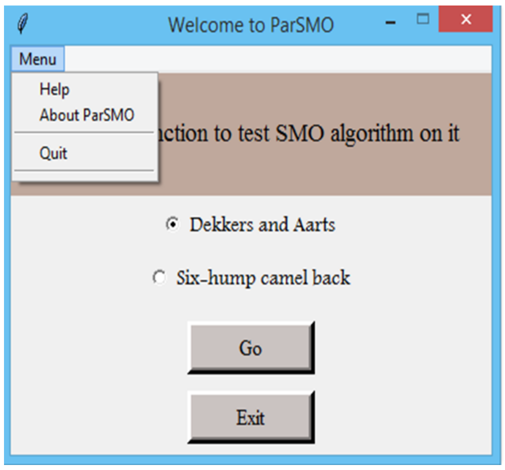
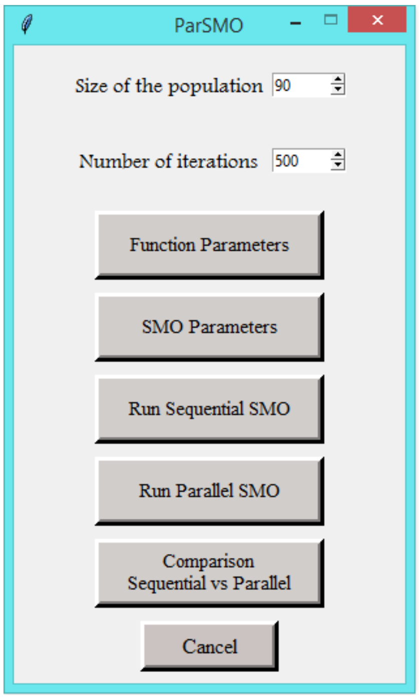


Figure 4.2: "ParSMO" Menu

In the menu bar "Menu" (Figure 4.2), we have three commands ("Help", "About ParSMO" and "Quit"). "Help" command gives to the user some instructions about using ParSMO application, as well as the formulas of the two test problems used in the application. "About ParSMO" command prints a list of the tools used for the implementation of the application and some information about the author of the dissertation and the programmer of the application. Finally, "Quit" command allows the user to exit the application.

After the selection of the function on the Home page, the button "Go" forwards to the main GUI in "ParSMO", where from this page the user can do all the actions that we have in our application. As shown in Figure 4.3, SMO inputs GUI contains two input boxes, the first is for the size of the swarm (population), and the second for the number of iterations, as well as it contains six buttons that the user can use.



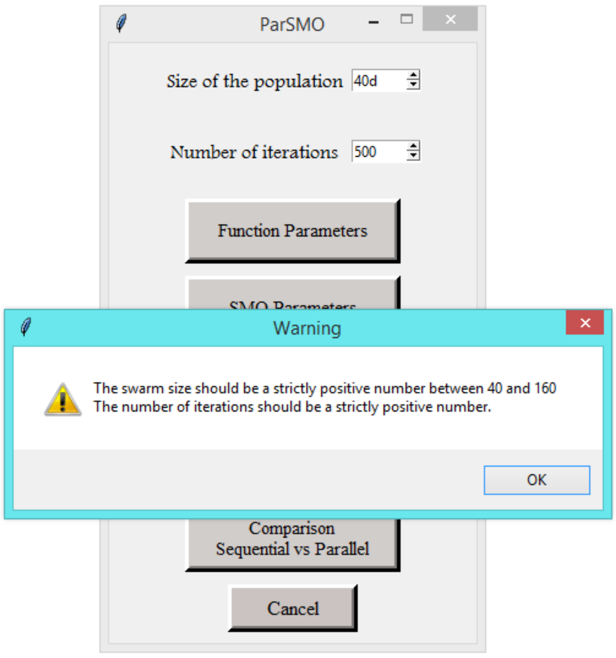
Size of the population 90

Number of iterations 500

Figure 4.3: SMO inputs GUI

If the user makes a mistake (among the following errors) in the input box of size or number of iterations, an input error window will be shown when trying to press a button:

1. **Case 1:** (Figure 4.4/4.5) If the user enters something other than a strictly positive number (e.g. a symbol or a character... etc):

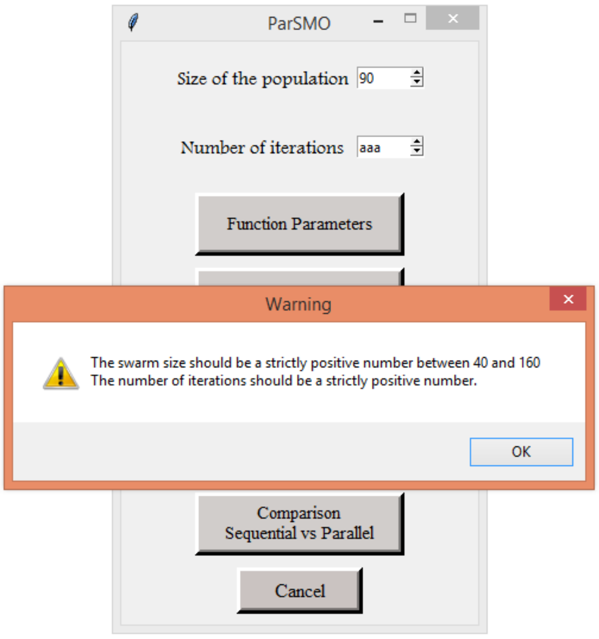


Size of the population 40d

Number of iterations 500

The swarm size should be a strictly positive number between 40 and 160
The number of iterations should be a strictly positive number.

Figure 4.4: Warning 1: Input error



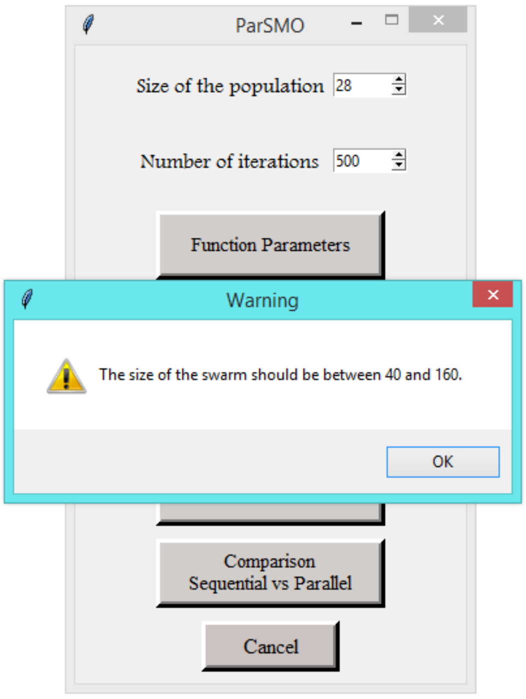
Size of the population 90

Number of iterations aaa

The swarm size should be a strictly positive number between 40 and 160
The number of iterations should be a strictly positive number.

Figure 4.5: Warning 1: Input error

2. **Case 2:** (Figure 4.6) In SMO algorithm, the swarm size (population size) should be between 40 and 160. Thus, if the user breaks this rule then the following window will be shown:



Size of the population: 28

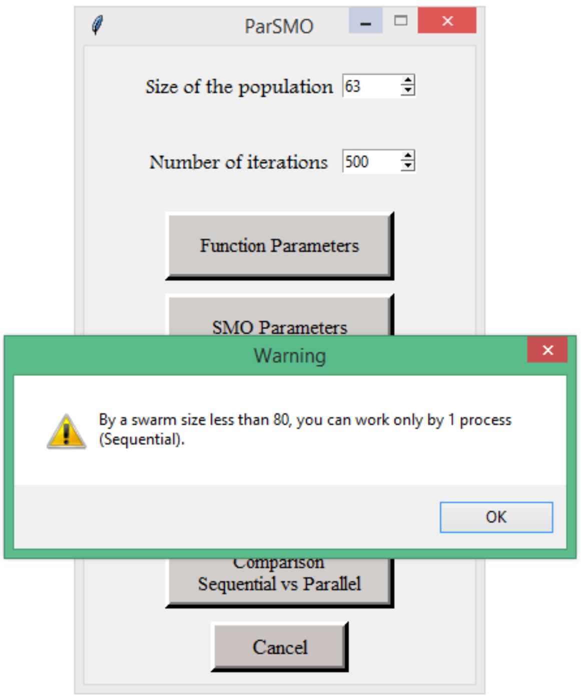
Number of iterations: 500

The size of the swarm should be between 40 and 160.

Figure 4.6: Warning 2: Input error

3. **Case 3:** (Figure 4.7) SMO algorithm requires a size greater than 80 to the user can run parallel SMO[1]:

[1]the size of a population should be greater than 40, so a size less than 80 could not be divided into two or more processes with a size equal or greater than 40 for each one.



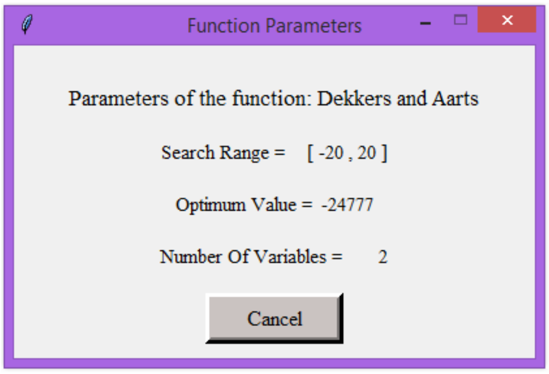
By a swarm size less than 80, you can work only by 1 process (Sequential).

Figure 4.7: Warning 3: Input error

Let's illustrate what the user see if he clicks on one of these buttons:

- **Function Parameters**

  This window (Figure 4.8) allows the user to know the parameters of the selected function as follow:



Parameters of the function: Dekkers and Aarts

Search Range = [-20 , 20]

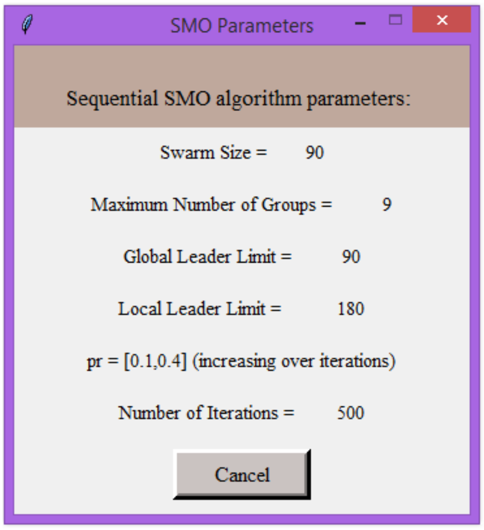
Optimum Value = -24777

Number Of Variables = 2

Figure 4.8: Function Parameters window

- **SMO Parameters**

  The following window (Figure 4.9) illustrates the parameters of SMO algorithm, which are identified according to the size and the number of iteration entered by the user:


SMO Parameters

Sequential SMO algorithm parameters:

Swarm Size = 90

Maximum Number of Groups = 9

Global Leader Limit = 90

Local Leader Limit = 180

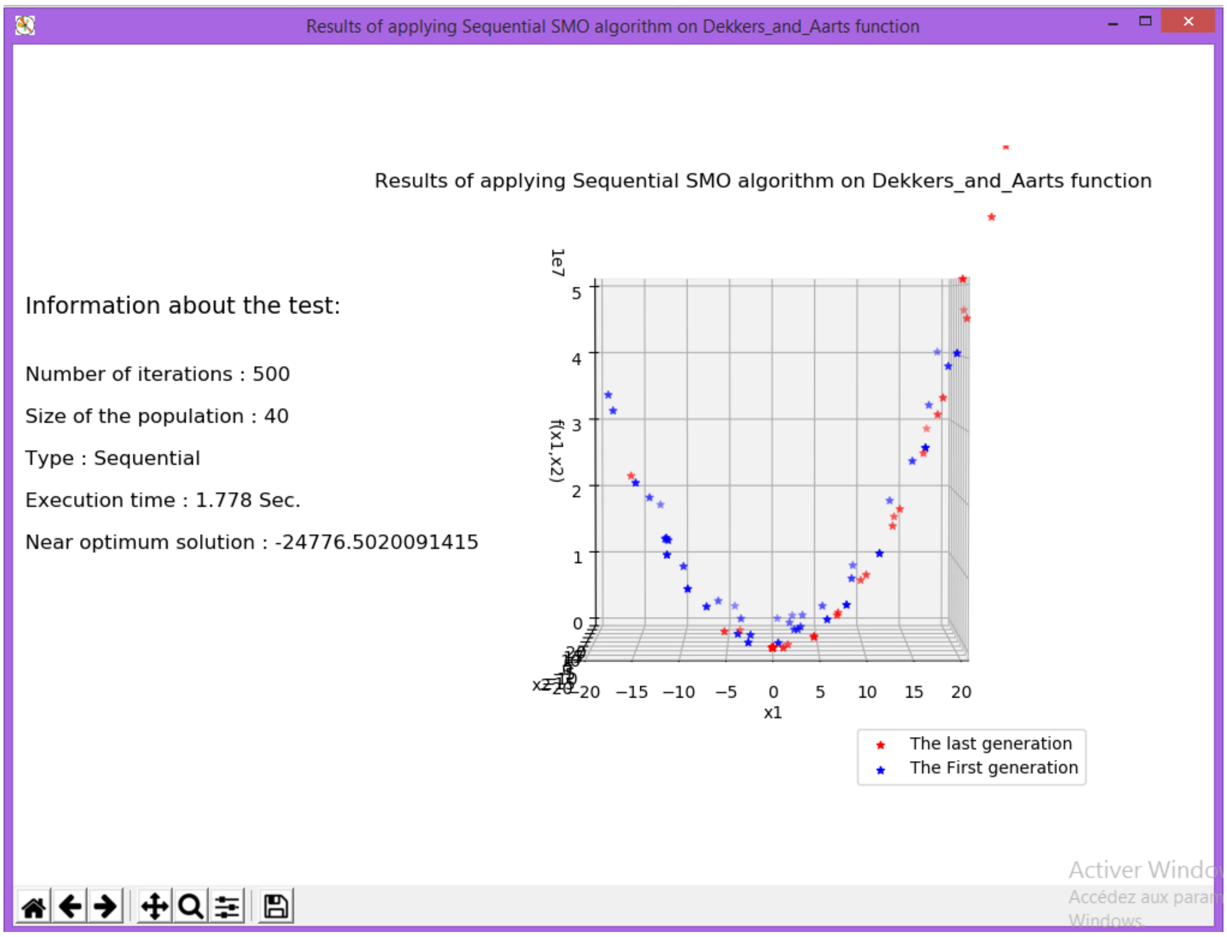
pr = [0.1,0.4] (increasing over iterations)

Number of Iterations = 500

Figure 4.9: SMO Parameters window

- **Run Sequential SMO**

The results of running the sequential SMO are printed in a window, as it is shown in Figure 4.10:


Results of applying Sequential SMO algorithm on Dekkers_and_Aarts function

Information about the test:

Number of iterations : 500

Size of the population : 40

Type : Sequential

Execution time : 1.778 Sec.

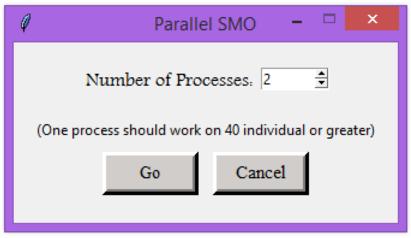
Near optimum solution : -24776.5020091415

Results of applying Sequential SMO algorithm on Dekkers_and_Aarts function
1e7
f(x1,x2)
x1
x2
The last generation
The First generation
Activer Windows
Accédez aux paramètres Windows

Figure 4.10: Results window of Sequential SMO

- **Run Parallel SMO**

If the user would like to work on the parallel SMO, an input window pops-up for entering the number of processes which will be used for running the parallel SMO (Figure 4.11):

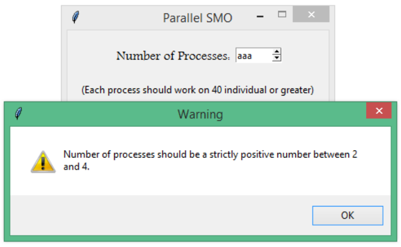

Parallel SMO

Number of Processes: 2

(One process should work on 40 individual or greater)

Figure 4.11: Number of processes

In the case where the user makes a mistake in the input box of the number of processes, one of the following warning message boxes (Figure 4.12, 4.14 or 4.15) are shown for the user (according to the error type):


Parallel SMO

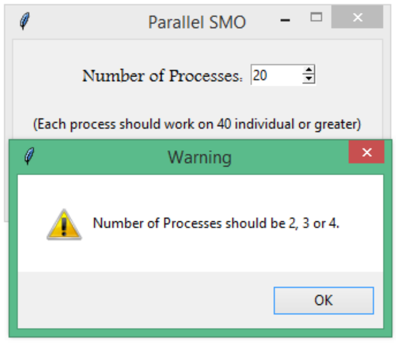
Number of Processes: aaa

(Each process should work on 40 individual or greater)

Warning

Number of processes should be a strictly positive number between 2 and 4.

Figure 4.12: Warning 4: Input error


Parallel SMO

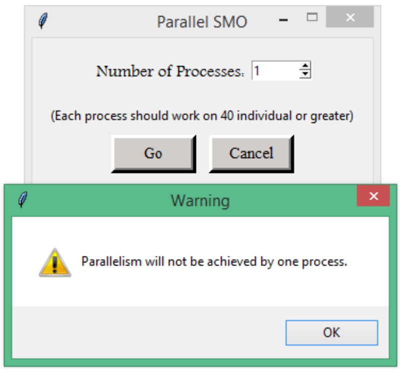
Number of Processes: 20

(Each process should work on 40 individual or greater)

Warning

Number of Processes should be 2, 3 or 4.

Figure 4.13: Warning 5: Input error


Parallel SMO

Number of Processes: 1

(Each process should work on 40 individual or greater)

Warning

Parallelism will not be achieved by one process.

Figure 4.14: Warning 6: Input error

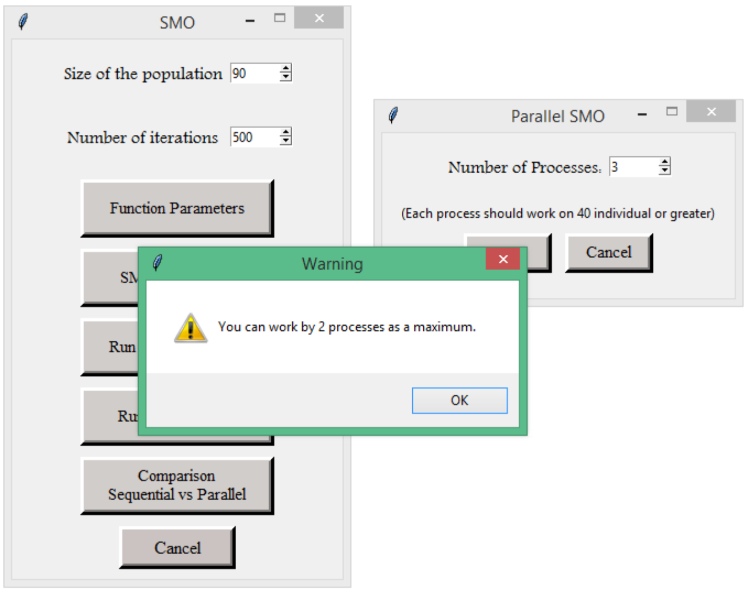
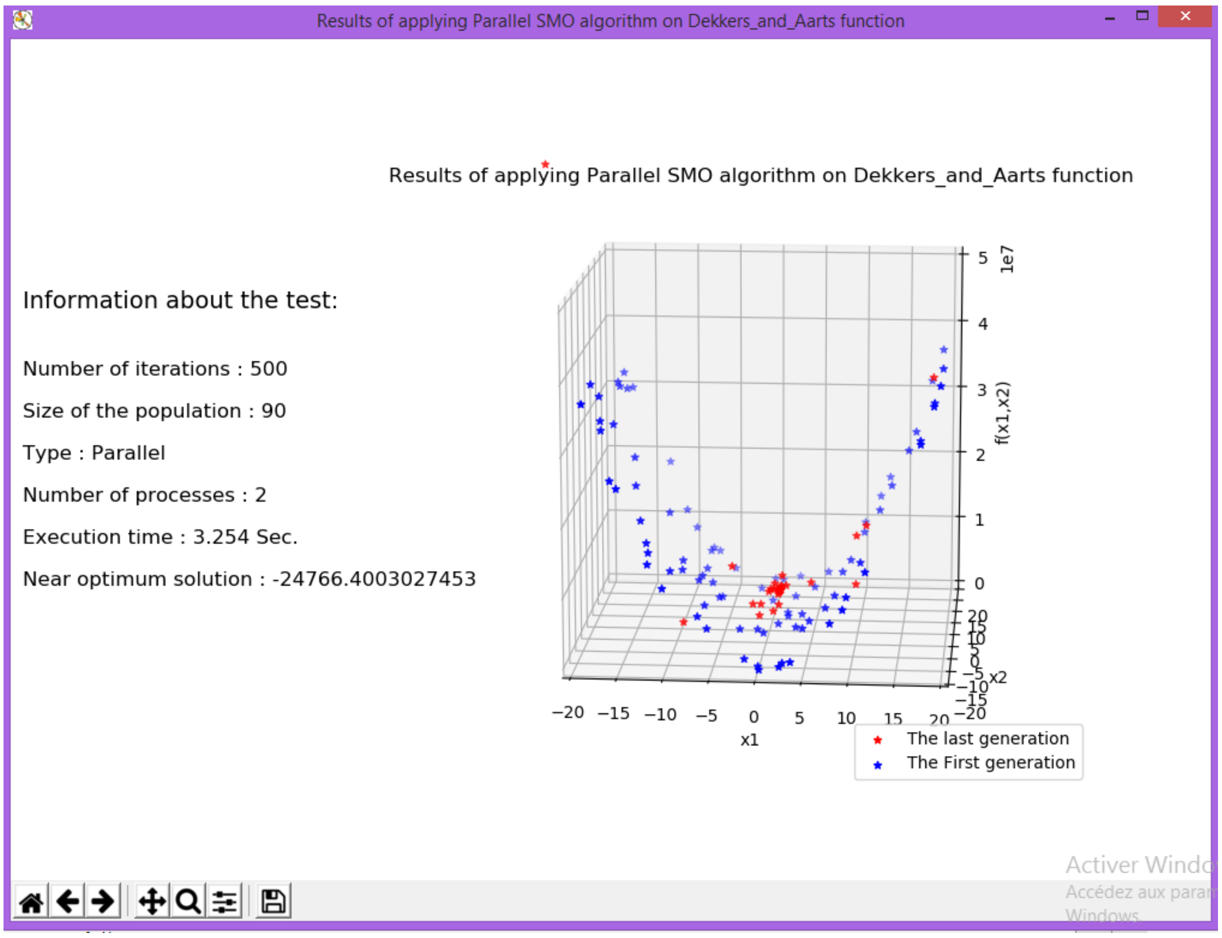

SMO
Size of the population 90
Number of iterations 500
Function Parameters
SM
Run
Ru
Comparison Sequential vs Parallel
Cancel
Parallel SMO
Number of Processes: 3
(Each process should work on 40 individual or greater)
Cancel
Warning
You can work by 2 processes as a maximum.
OK

Figure 4.15: Warning : Input error

Once the user entered the correct number of processes and pressed on “Ok”, the parallel SMO runs. After that, a results window is shown (Figure 4.16):


Results of applying Parallel SMO algorithm on Dekkers_and_Aarts function
Results of applying Parallel SMO algorithm on Dekkers_and_Aarts function
x1
x2
f(x1,x2)
1e7
★ The last generation
★ The First generation
Activer Windo
Accédez aux para
Windows

Information about the test:

Number of iterations : 500

Size of the population : 90

Type : Parallel

Number of processes : 2

Execution time : 3.254 Sec.

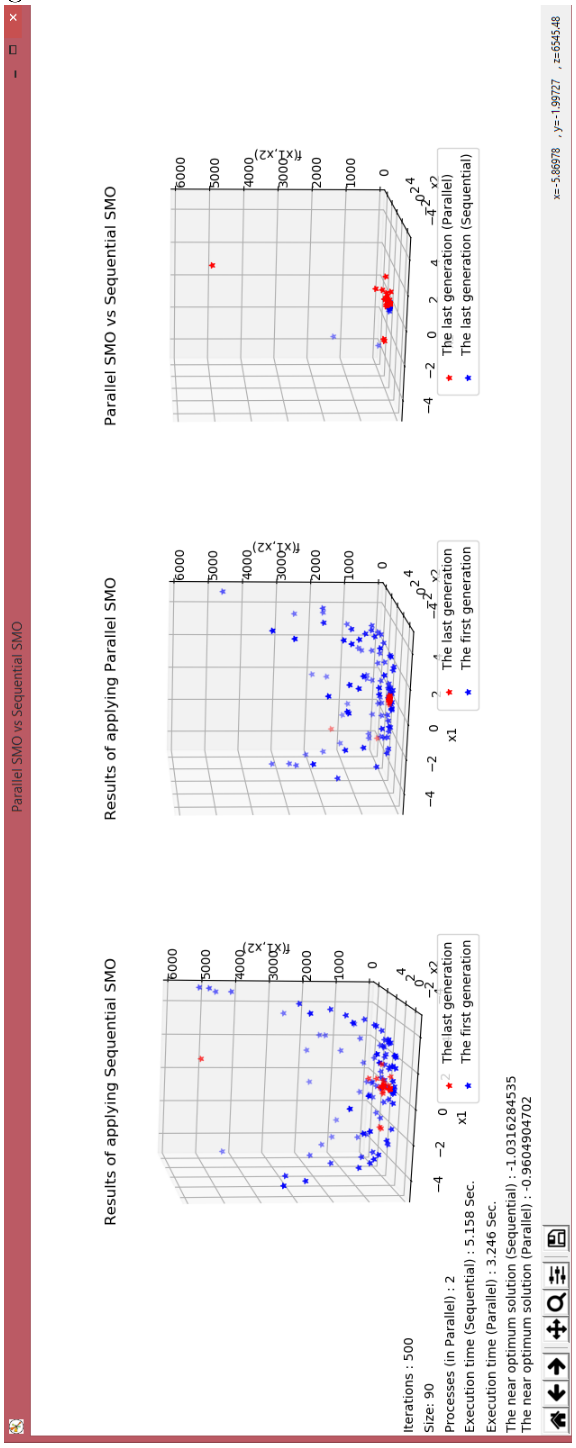
Near optimum solution : -24766.4003027453

Figure 4.16: Results window of Parallel SMO

- **Comparison: Sequential SMO vs Parallel SMO**

  As the parallel SMO, when the user press on comparison button an input window asks for the number of processes is shown, and after pressing "Ok" and finishing the run of sequential SMO and parallel SMO, the results window shows as follow (Figure 4.17):

Figure 4.17: Results window of Parallel SMO


Parallel SMO vs Sequential SMO
Results of applying Sequential SMO
Results of applying Parallel SMO
Parallel SMO vs Sequential SMO
f(x1,x2)
x1
x2
The last generation
The first generation
The last generation (Parallel)
The last generation (Sequential)
x=-5.86978 , y=-1.99727 , z=6545.48

Iterations : 500
Size: 90
Processes (in Parallel) : 2
Execution time (Sequential) : 5.158 Sec.
Execution time (Parallel) : 3.246 Sec.
The near optimum solution (Sequential) : -1.0316284535
The near optimum solution (Parallel) : -0.9604904702

- **Cancel button** Is a button for exiting this window.

4.3 Test & Experimental Study

In this part, we will test SMO algorithm using two state-of-the-art test problems which are: “Dekker and Aarts” and “Six-hump Camelback” (See 3.1.2). Our test will be in terms of execution time, objective space density and the near-optimum solution. Three sections are considered in our test, the first is for testing the sequential SMO algorithm, the second is for the parallel SMO, and the last is for comparing between them and discussion.

4.3.1 Sequential SMO algorithm

- The effect of “Swarm Size”:

In this part, we discuss the effect of population size on obtaining the near-optimum solution and the objective space density in the last generation.

Note: the number of iterations used in this test is 5000 iterations.

- The effect of “Swarm Size” on the near-optimum solution (NOS):
  The optimum value of the function “Dekkers and Aarts” (DA) is -24777, and the table below (Tabel 4.2) shows that the larger the size, the better near-optimum solution. The same for the second function “Six-hump Camelback” (ShC) in table 4.3.

Size	40	60	80	100
NOS	-23879.6155756556	-3803.7105749113	-24767.1785288615	-24776.5183423176
Size	100	120	140	160
NOS	-24776.5183423176	-24776.5183423176	-24776.5183423176	-24776.5183423176

Table 4.2: The effect of “Swarm Size” on the near-optimum solution (function DA)

Size	40	60	80	100
NOS	-1.03147562721598	-1.03162845172622	-1.03162735830379	-1.03162845348987
Size	100	120	140	160
NOS	-1.03162845348987	-1.03162845348987	-1.03162845348987	-1.03162845348987

Table 4.3: The effect of “Swarm Size” on the near-optimum solution (function ShC)

- The effect of “Swarm Size” on “The objective space density”:
  In contrast, for the two functions the objective space (OS) density of the last generation is better in the smaller sizes: 40, 60 and 80 as follow.

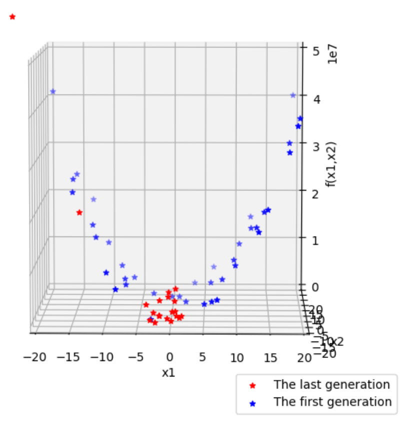


Figure 4.18: Objective space, size = 40

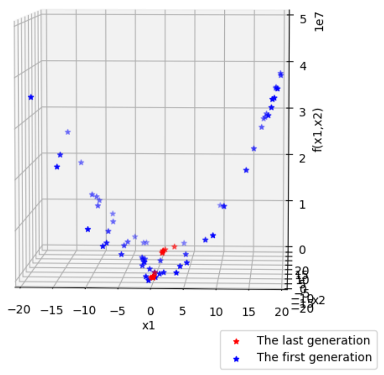


Figure 4.19: Objective space, size = 60

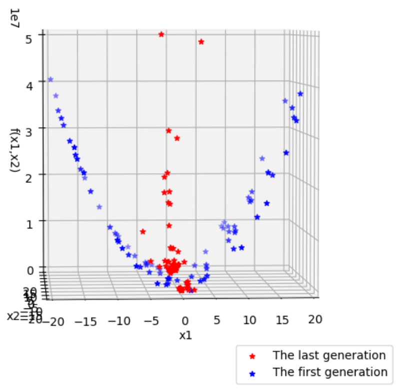


Figure 4.20: Objective space, size =80

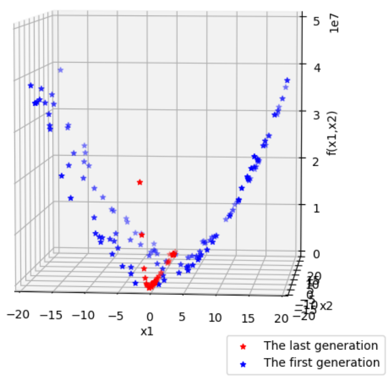


Figure 4.21: Objective space, size = 120

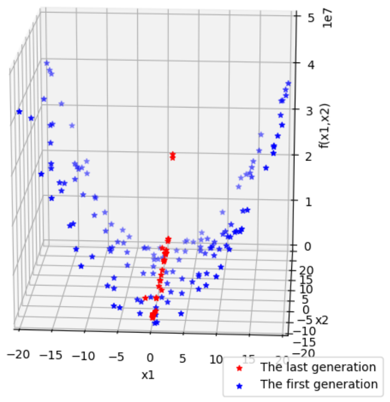


Figure 4.22: Objective space, size = 140

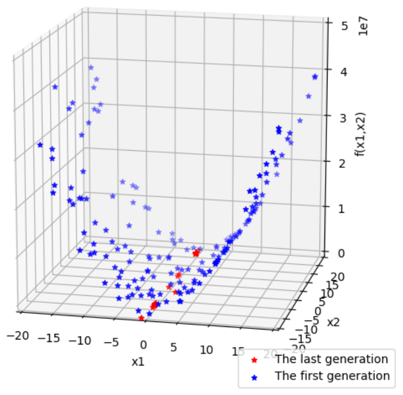


Figure 4.23: Objective space, size =160

The same thing for the second function:

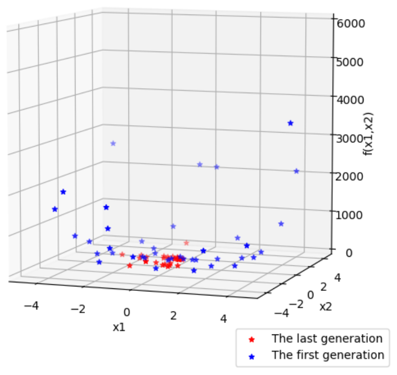


Figure 4.24: Objective space, size = 40

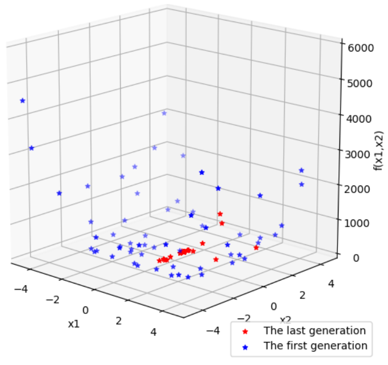


Figure 4.25: Objective space, size = 60

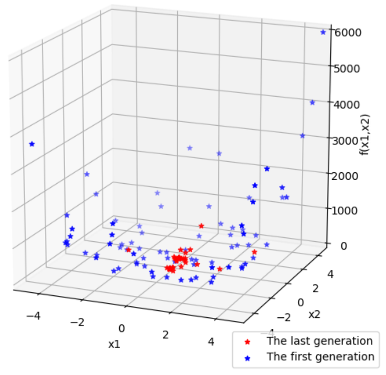


Figure 4.26: Objective space, size =80

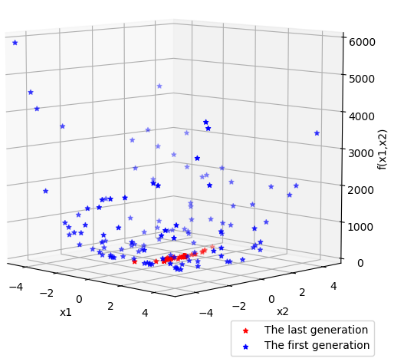


Figure 4.27: Objective space, size = 120

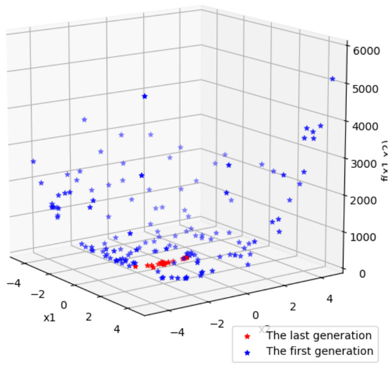


Figure 4.28: Objective space, size = 140

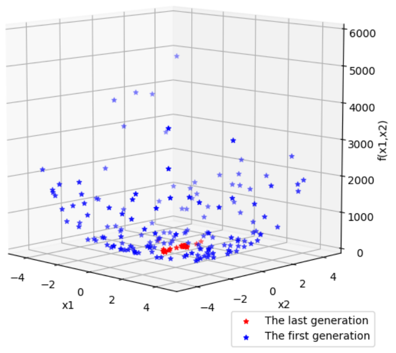


Figure 4.29: Objective space, size =160

4.3.2 Parallel SMO algorithm

- The effect of "Number of processes":

Over our test of the Parallel SMO, we noticed that the most important factor that affects on the results (execution time and objective space density) is the number of processes used. **Note:** the number of iteration used in this test is 5000 iterations.

- The effect of "Number of processes" on "Execution time":
  The tables below (4.4 and 4.5) show that for the two functions: the greater the number of processes, the shorter the time. Thus, SMO algorithm is faster in case of number of processes is greater.

Size	120		140		160		
Number of Processes	2	3	2	3	2	3	4
Execution Time (Sec)	48	37	78	46	90	61	50

Table 4.4: The effect of "Number of processes" on the execution time (function 1)

Size	120		140		160		
Number of Processes	2	3	2	3	2	3	4
Execution Time (Sec)	58	38	76	50	90	62	53

Table 4.5: The effect of "Number of processes" on the execution time (function 2)

- The effect of "Number of processes" on "The objective space density":
  Like the previous part, these test results illustrate that the greater the number of processes, the more objective space density of the last generation. We divided this part into three sub-parts according to the population size taken in the experimental study: 120, 140 and 160.

1. Populations with a size equal to 120:

   - For the first function:

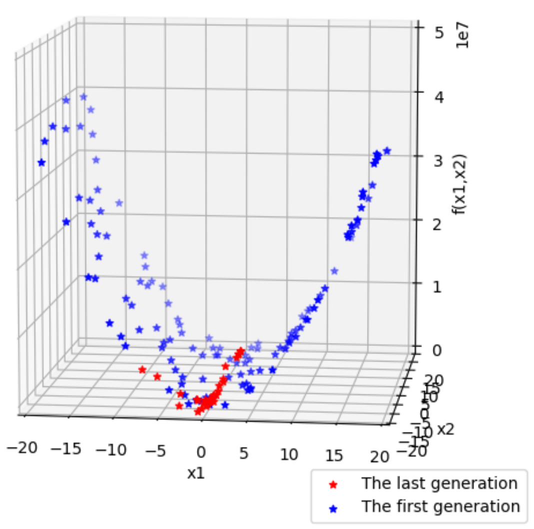


Figure 4.30: Objective space, using 2 processes

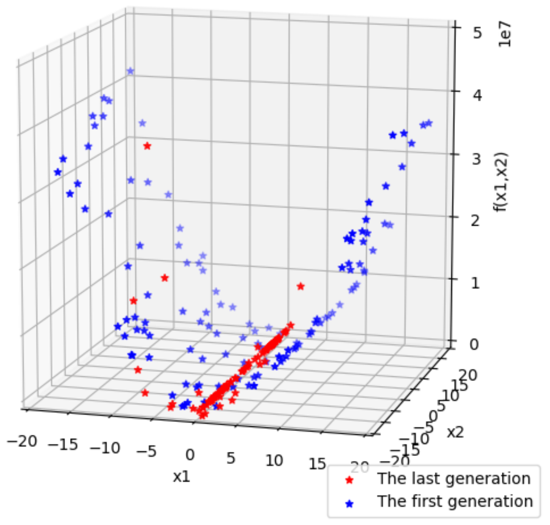


Figure 4.31: Objective space, using 3 processes

   - And for the second function:

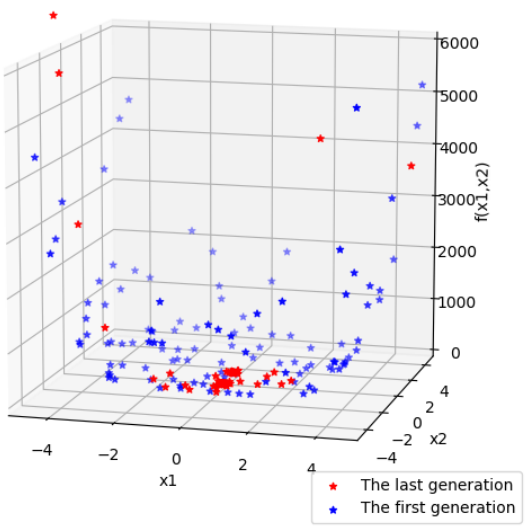


Figure 4.32: Objective space, using 2 processes

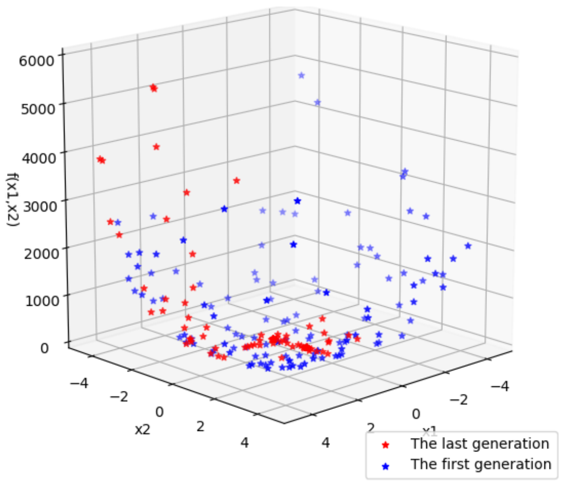


Figure 4.33: Objective space, using 3 processes

2. Populations with a size equal to 140:

   - For the first function:

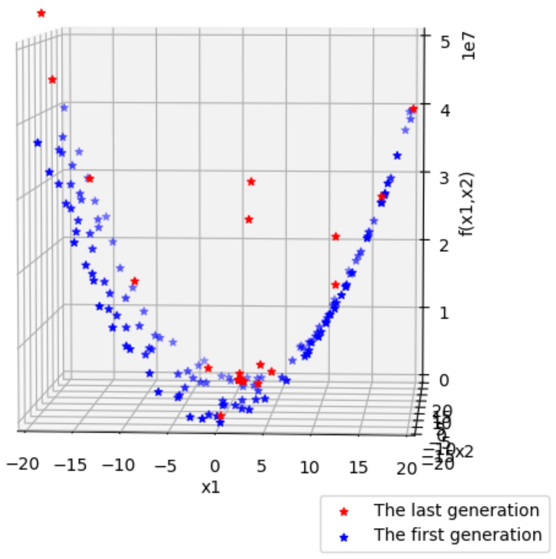


Figure 4.34: Objective space, using 2 processes

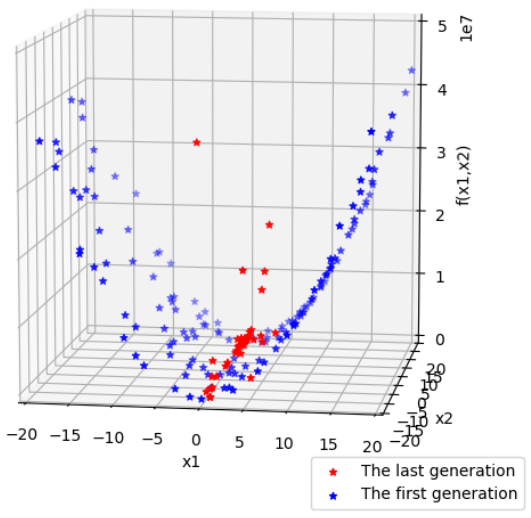


Figure 4.35: Objective space, using 3 processes

- And for the second function:

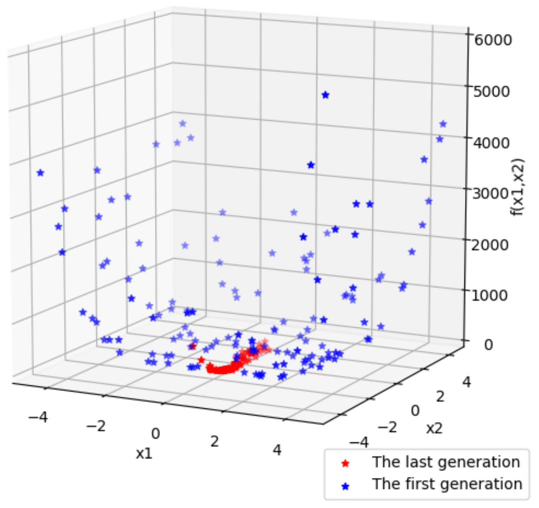


Figure 4.36: Objective space, using 2 processes

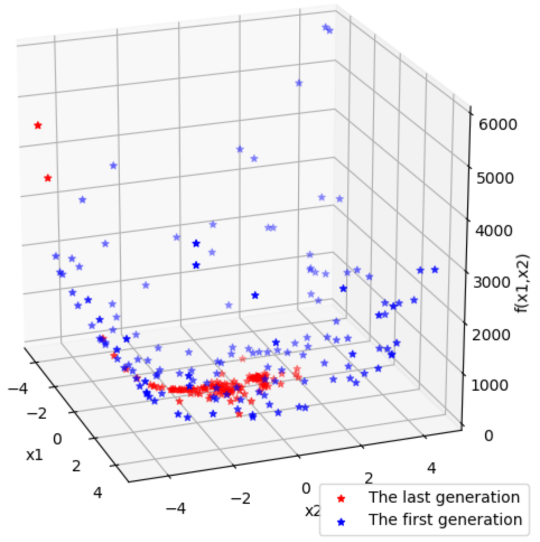


Figure 4.37: Objective space, using 3 processes

3. Populations with size equal to 160:

- For the first function:

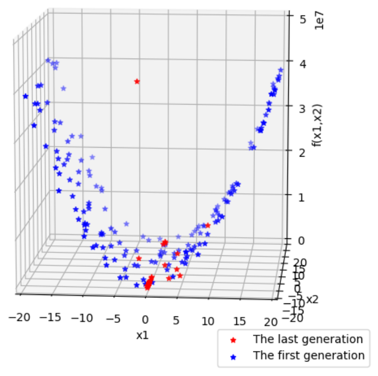


Figure 4.38: Objective space, using 2 processes

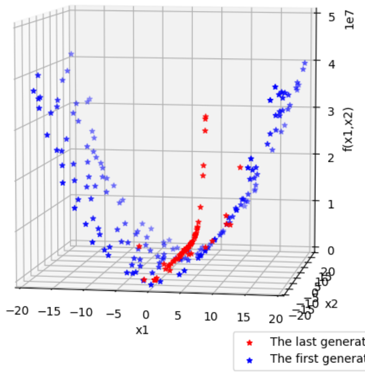


Figure 4.39: Objective space, using 3 processes

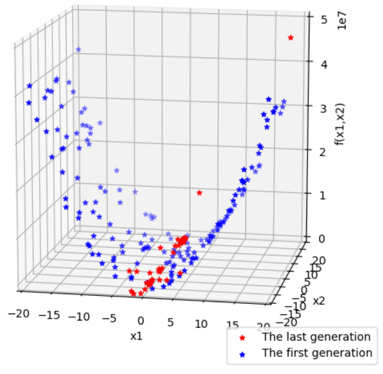


Figure 4.40: Objective space, using 4 processes

- And for the second function:

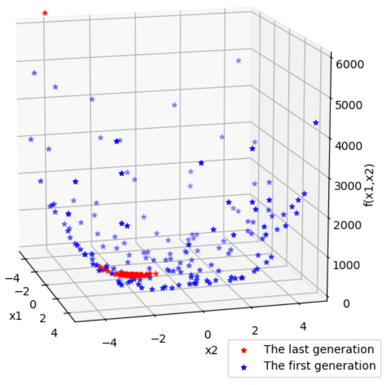


Figure 4.41: Objective space, using 2 processes

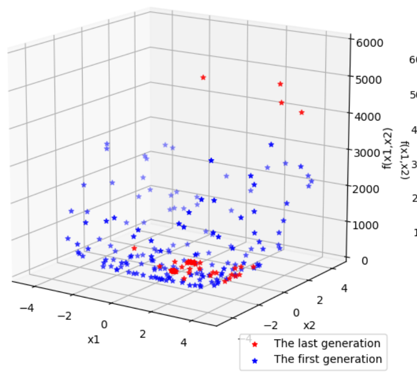


Figure 4.42: Objective space, using 3 processes

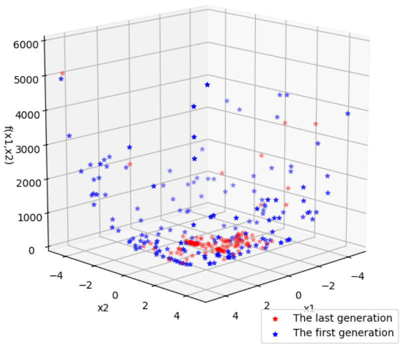


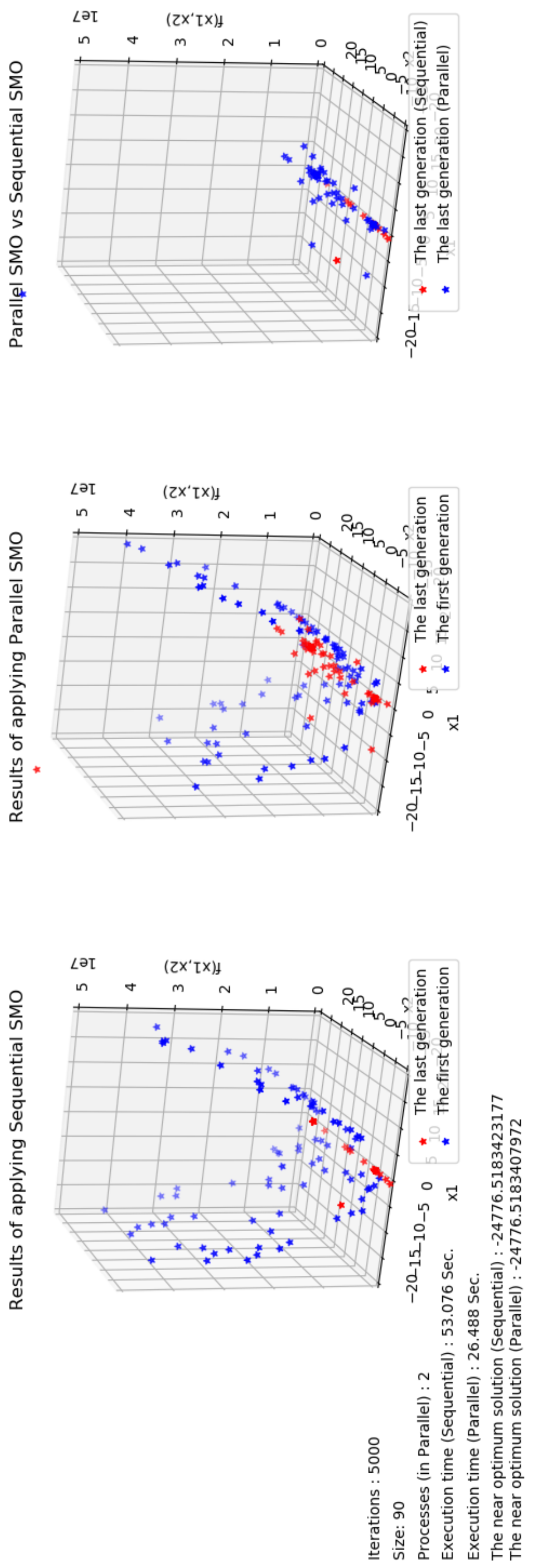
Figure 4.43: Objective space, using 4 processes

4.3.3 Comparison Sequential vs Parallel

After presenting the effect of the population size on the execution time and the near-optimum solution for the sequential SMO, as well the effect of the number of processes on the execution time and the objective space density in the parallel SMO, this part will be a comparison between the sequential SMO and the parallel SMO in terms of execution time, the near-optimum solution and the objective space density of the last generation. Hence, we test and display the results of the two functions separately. Then, a discussion of these results will be in a special part.

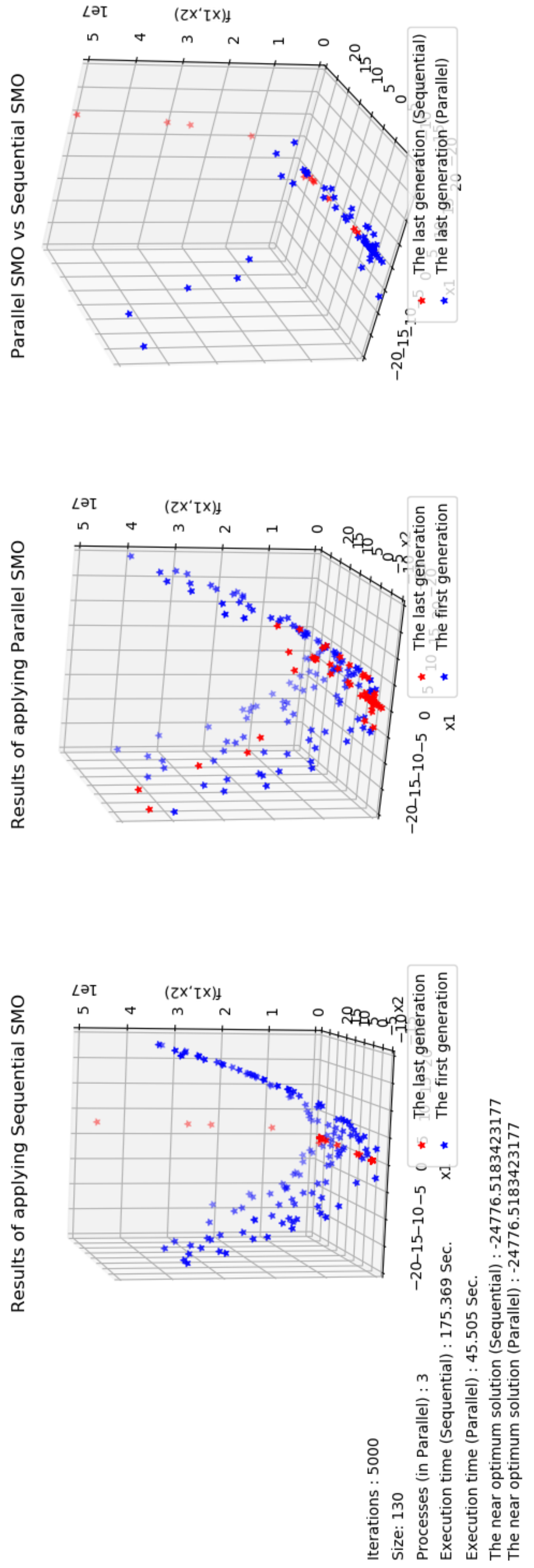
For each function, the number of iteration adopted is 5000 iterations, and the size and the number of processes in parallel SMO will be the variables.

1. “Dekkers and Aarts” function:



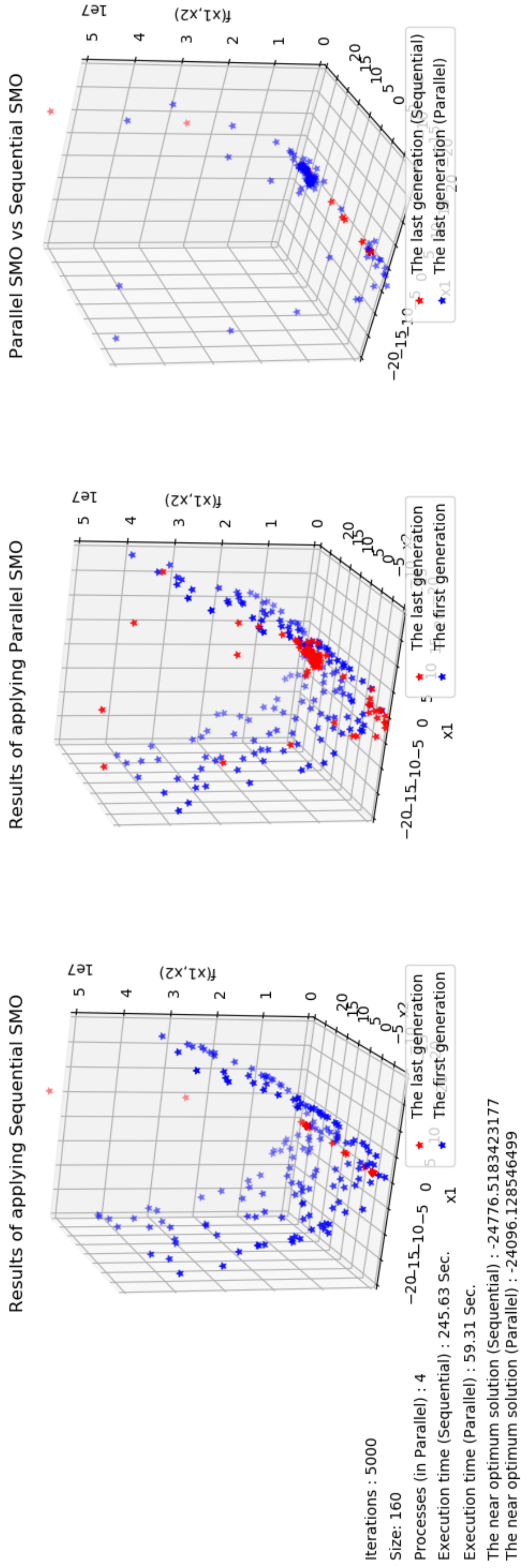
```
Iterations : 5000
Size: 90
Processes (in Parallel) : 2
Execution time (Sequential) : 53.076 Sec.
Execution time (Parallel) : 26.488 Sec.
The near optimum solution (Sequential) : -24776.5183423177
The near optimum solution (Parallel) : -24776.5183407972
```

Figure 4.44: Results of comparison Parallel SMO vs Sequential SMO (DA function)



Iterations : 5000
Size: 130
Processes (in Parallel) : 3
Execution time (Sequential) : 175.369 Sec.
Execution time (Parallel) : 45.505 Sec.
The near optimum solution (Sequential) : -24776.5183423177
The near optimum solution (Parallel) : -24776.5183423177

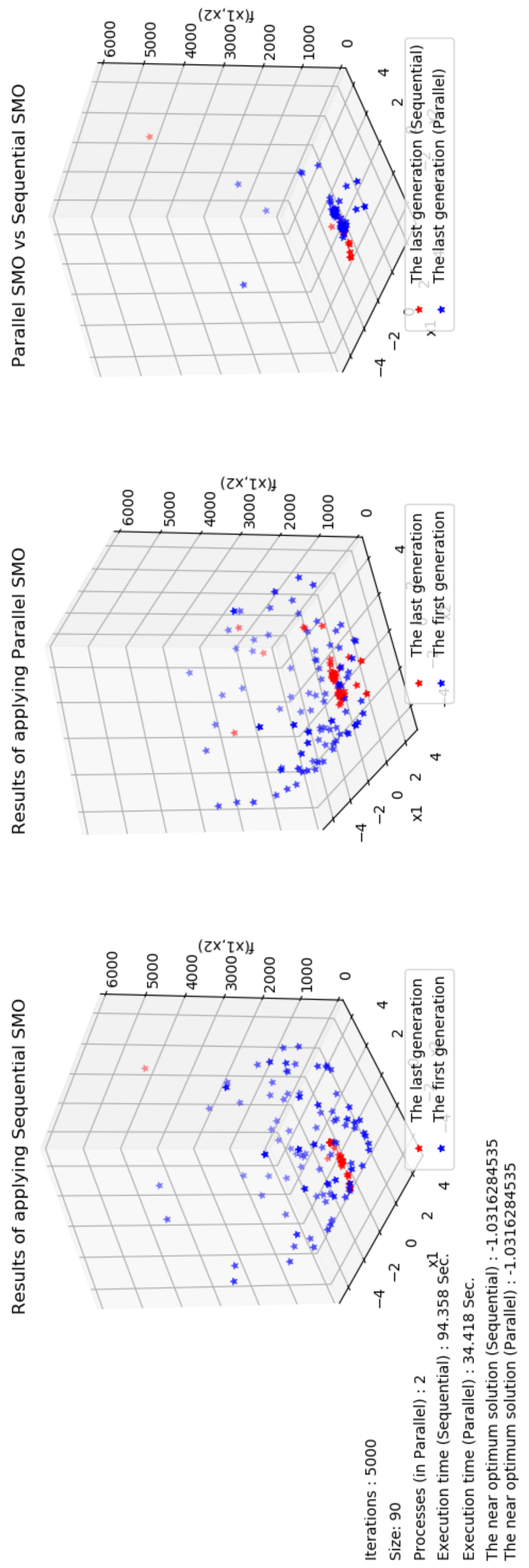
Figure 4.45: Results of comparison Parallel SMO vs Sequential SMO (DA function)



```
Iterations : 5000
Size: 160
Processes (in Parallel) : 4
Execution time (Sequential) : 245.63 Sec.
Execution time (Parallel) : 59.31 Sec.
The near optimum solution (Sequential) : -24776.5183423177
The near optimum solution (Parallel) : -24096.128546499
```

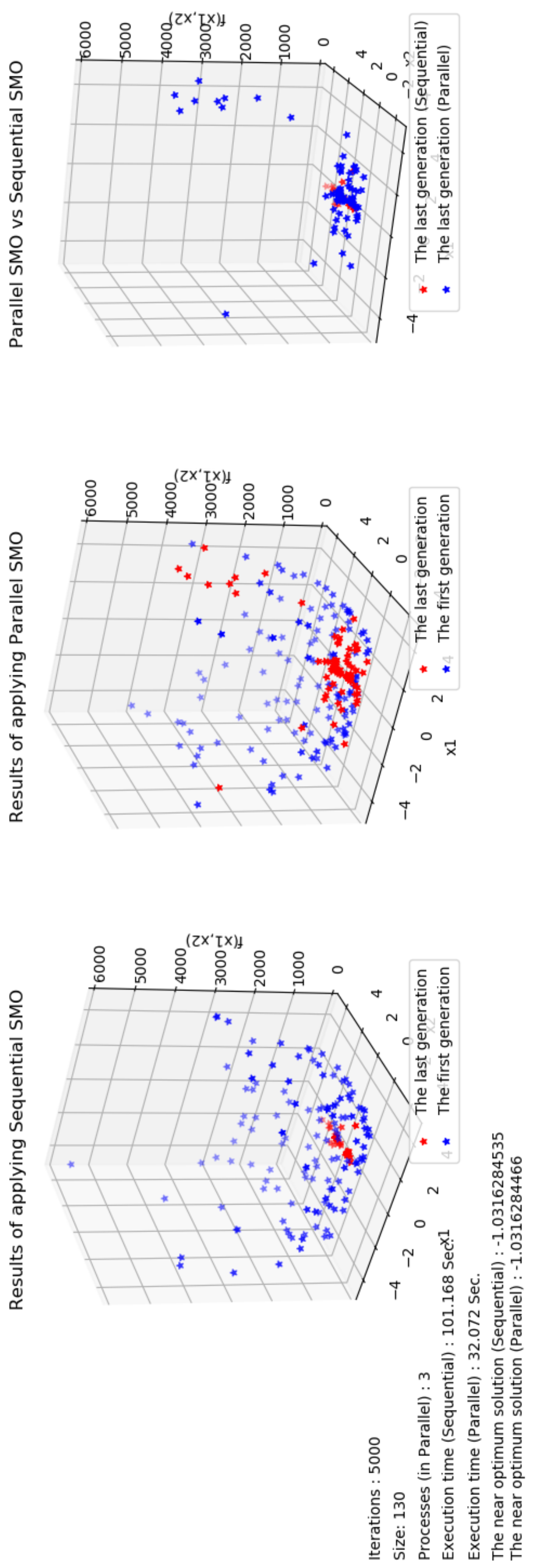
Figure 4.46: Results of comparison Parallel SMO vs Sequential SMO (DA function)

2. “Six-hump Camelback” function:



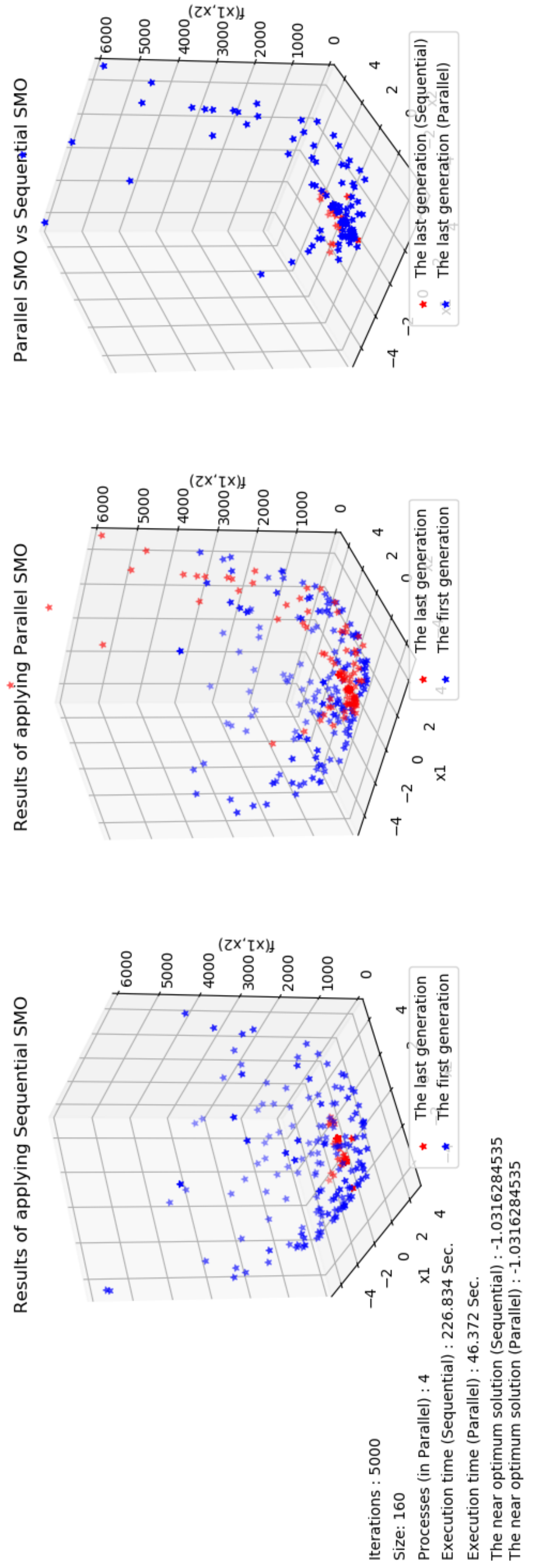
```
Iterations : 5000
Size: 90
Processes (in Parallel) : 2
Execution time (Sequential) : 94.358 Sec.
Execution time (Parallel) : 34.418 Sec.
The near optimum solution (Sequential) : -1.0316284535
The near optimum solution (Parallel) : -1.0316284535
```

Figure 4.47: Results of comparison Parallel SMO vs Sequential SMO (ShC function)



Iterations : 5000
Size: 130
Processes (in Parallel) : 3
Execution time (Sequential) : 101.168 Sec.
Execution time (Parallel) : 32.072 Sec.
The near optimum solution (Sequential) : -1.0316284535
The near optimum solution (Parallel) : -1.0316284466

Figure 4.48: Results of comparison Parallel SMO vs Sequential SMO (ShC function)



Iterations : 5000
Size: 160
Processes (in Parallel) : 4
Execution time (Sequential) : 226.834 Sec.
Execution time (Parallel) : 46.372 Sec.
The near optimum solution (Sequential) : -1.0316284535
The near optimum solution (Parallel) : -1.0316284535

Figure 4.49: Results of comparison Parallel SMO vs Sequential SMO (ShC function)

4.3.4 Discussion

In the previous part of the test, we displayed what we got from the comparison between sequential SMO and parallel SMO algorithm for both of the functions: “DA” and “ShC”. These results, in general, illustrate that the parallel SMO is better than the sequential one. However, let’s discuss each of the execution time and the near-optimum solution and the objective space density independently.

- The execution time:

Size	90	130	160
Number of processes in parallel	2	3	4
Execution Time (Seq) (second)	53	175	246
Execution Time (Par) (second)	26	45	59

Table 4.6: The execution time comparison (function 1)

Size	90	130	160
Number of processes in parallel	2	3	4
Execution Time (Seq) (second)	94	101	227
Execution Time (Par) (second)	34	32	46

Table 4.7: The execution time comparison (function 2)

The tables (4.6, 4.7) show that the execution time when we use the parallel version of SMO algorithm is shorter than the case when we use the sequential one. Thus, and as a result, **the parallel version of SMO algorithm is better than the sequential one in term of the execution time.**

- The near-optimum solution:

Size	90	130	160
Number of processes in parallel	2	3	4
The near-optimum (Seq)	-24776.518	-24776.518	-24776.518
The near-optimum (Par)	-24776.518	-24776.518	-24096.129

Table 4.8: The near-optimum solution comparison (function 1)

Size	90	130	160
Number of processes in parallel	2	3	4
The near-optimum (Seq)	-1.031628	-1.031628	-1.031628
The near-optimum (Par)	-1.031628	-1.031628	-1.031628

Table 4.9: The near-optimum solution comparison (function 2)

About the near-optimum solution, we noted from the results in the tables above (4.8, 4.9) that approximately both of the versions of SMO (sequential and parallel) have the same near-optimum solution in the last generation, even in changing the size of the swarm or the number of processes. Thus, **for the near-optimum solution, there is no version better than the other.**

- The objective space density:

For the objective space, it's so clear without any doubt that **the parallel version of SMO is better than the sequential one in term of the objective space of the last generation**, and the figures below 4.50, 4.51, 4.52, 4.53, 4.54, 4.55 illustrate that.

- For a population size = 90:

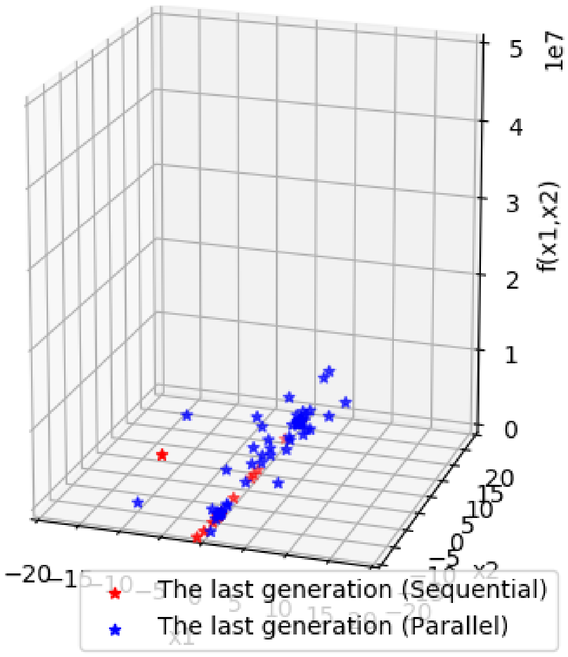


Figure 4.50: The last generation, sequential vs parallel

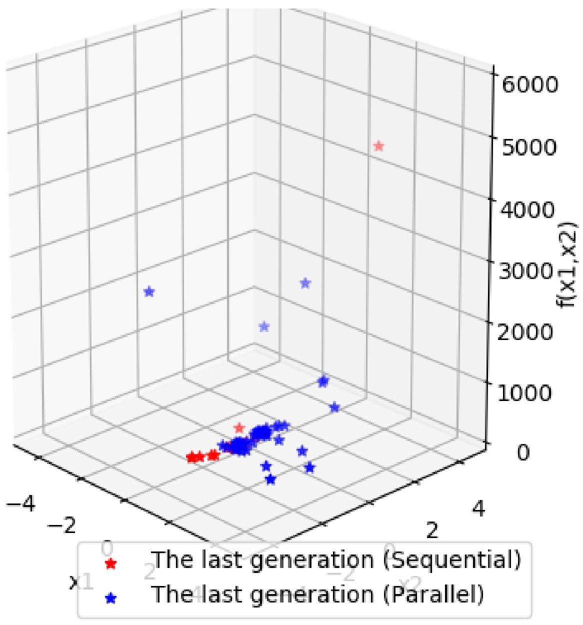


Figure 4.51: The last generation, sequential vs parallel

- For a population size = 130:

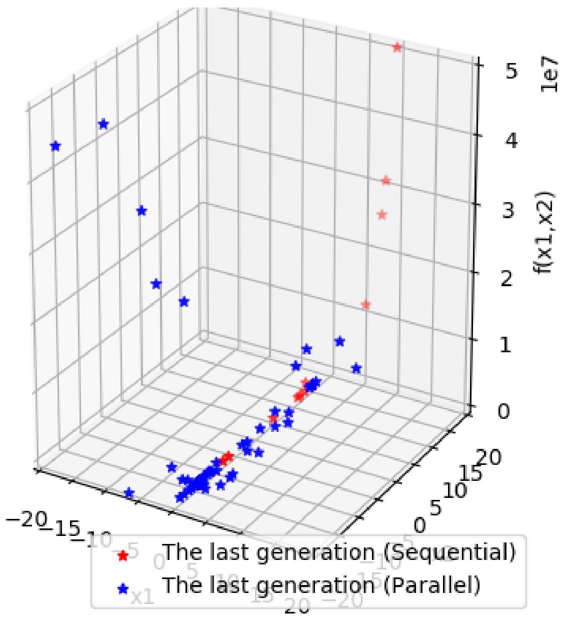


Figure 4.52: The last generation, sequential vs parallel

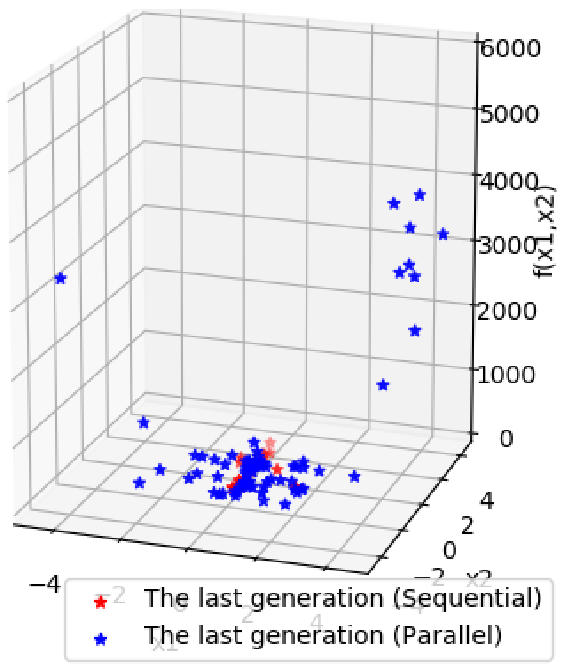


Figure 4.53: The last generation, sequential vs parallel

- For a population size = 160:

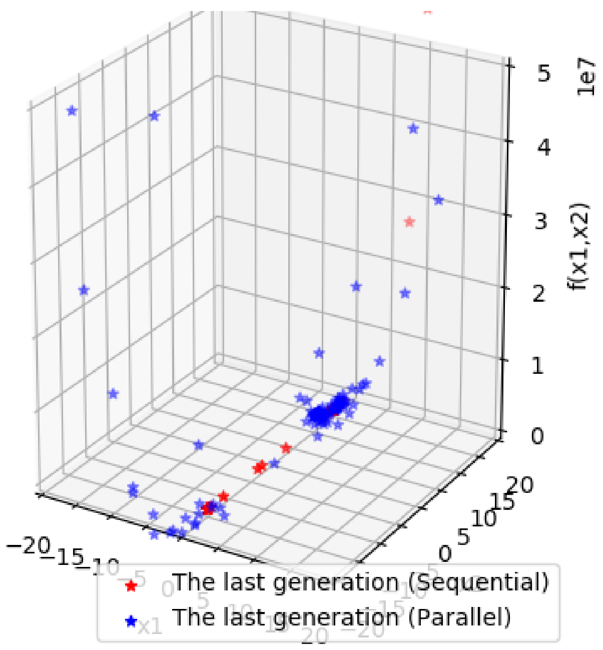


Figure 4.54: The last generation, sequential vs parallel

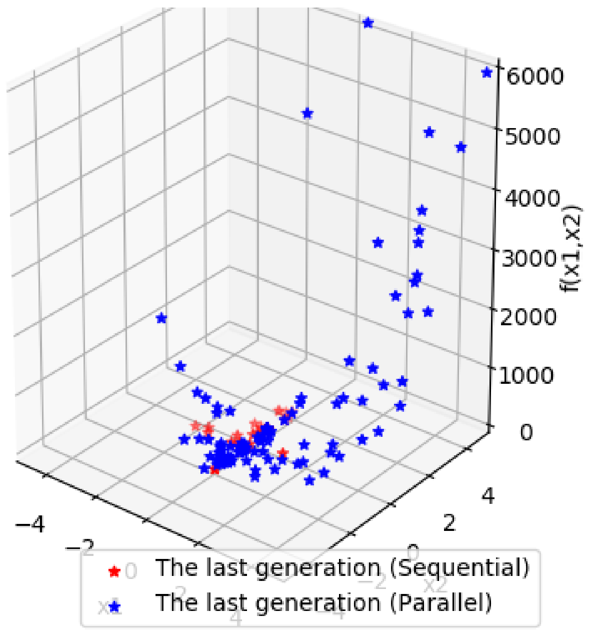


Figure 4.55: The last generation, sequential vs parallel

Conclusion

In this chapter, we have presented the tools that we have used in the implementation of ParSMO application. Then, we touched on presenting the GUIs of our application and how the user can use them. Finally, a global test of the sequential SMO version and the parallel one in the application, with a comparison between them and a discussion of the results we got, which take us to conclude that **the parallel version of SMO is better than the sequential one**, especially in the objective space and the execution time.

Conclusion

Almost all computations done during the early years of the history of computers could be called sequential (employs a single processor to solve a problem or task). In order to reduce time and cost and for better performance, parallel computing has received a rapidly increasing amount of attention by the researchers, being it a solution to that. The ParSMO application has been implemented to prove these aims of parallelism by applying it to an optimization algorithm.

In the first part of this thesis, we have taken a look at some basic terms which are related to our project theme:

- Optimization problem definition.
- Swarm intelligence definition and the conditions that achieve it (we touched on this term because of the algorithm that we implemented is based on this approach).
- The whole process (all the steps) of the SMO algorithm with a detailed explanation.
- Parallel computing: basic concepts and terminology, with an overview of the parallel architectures and the parallel programming models.

The second part of the thesis contains an analysis of our application ParSMO with a detailed design that explains all the application acts, which are represented in running the sequential version of the SMO algorithm, the parallel one, and the comparison between them.

ParSMO application was implemented by the programming language Python 3.7, using some specific packages like Multiprocessing (to achieve parallelism), Tkinter (to draw the GUIs), Matplotlib (for plotting) ... etc.

All the experiments that we have done during the ParSMO test and using both of the two functions “Dekker and Aarts” (Da) and “Six-hump camelback” (ShC) say that the parallel SMO outperforms the sequential SMO in terms of the execution time and the objective space density.

From the related work and the questions of our work that should be solved and carried out in future research:

- Implementing a parallel SMO algorithm using other parallel programming models, such as multi-core architecture, MPI, or OpenMP. In order to identify and discover the best parallel programming model for this kind of optimization problems.
- Implementing other swarm intelligence based algorithms, such as DE, PSO, ABC ...etc and implementing a parallel version of them, in order to compare the two versions and circulate or customize the results that we got.

Bibliography

Amrita Chakraborty, A. K. K. (n.d.), 'Nature-inspired computing and optimization'.

Ananth Grama, Anshul Gupta, G. K. V. K. (2003), *Introduction to Parallel Computing, Second Edition*, Addison Wesley.

An Introduction to Tkinter (2005), `http://effbot.org/tkinterbook/`. accessed on May 18, 2017.

Barney, B. (n.d.), 'Introduction to parallel computing'.
URL: *https://computing.llnl.gov/tutorials/parallel$_c$omp/*

Bonabeau E, Dorigo M, T. G. (n.d.).

CHOUDHARY, A. N. (1989), 'Parallel architectures and parallel algorithms for integrated vision systems'.

Jagdish Chand Bansal, Harish Sharma, S. S. J. M. C. (2014), 'Spider monkey optimization algorithm for numerical optimization', *Springer* .

Jagdish Chand Bansal, Pramod Kumar Singh, N. R. P. (2019), *Evolutionary and Swarm Intelligence Algorithms*, Springer International Publishing AG.

Joseph Awange, Bela Palancz, R. H. L. L. V. (2018), *Mathematical Geosciences: Hybrid Symbolic-Numeric Methods*, Springer International Publishing.

KARABOGA, D. (2005), 'An idea based on honey bee swarm for numerical optimization'.

Lamport, L. (2019), 'LaTeX', `https://en.wikipedia.org/wiki/LaTeX`. accessed on May 04, 2019.

Leila, B. (2017), Reconfiguration optimization in reconfigurable manufacturing systems, Master's thesis, University of Mohamed Khider Biskra.

matplotlib (2017), `http://matplotlib.org/`. accessed on May 18, 2017.

Overleaf (2019), `https://www.overleaf.com/`. accessed on June 01, 2019.

Rothlauf, F. (2011), *Design of Modern Heuristics Principles and Application*, Springer-Verlag Berlin Heidelberg.

Ruokamo, A. (2018), ‘Parallel computing and parallel programming models: application in digital image processing on mobile systems and personal mobile devices’, pp. 14–15.

Samir, H. (2015), Un algorithme génétique parallèle sur gpu pour le problème du flow shop de permutation, Master’s thesis, University of A-MIRA-BEJAIA.

TechoPedia (2019), ‘multiprocessing’, `https://www.techopedia.com/definition/3393/multi-processing`. accessed on June 1, 2019.

VP Online (2018), `https://online.visual-paradigm.com/`. accessed on June 20, 2019.

Yujun Zheng, Xueqin Lu, M. Z. S. C. (n.d.), *Biogeography-Based Optimization: Algorithms and Applications*, Springer Nature.