NOVA
NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

NELSON ALEXANDRE CHARRÉU SANTOS

Bachelor of Computer Science and Engineering

CONTRIBUTIONS IN GLOBAL DERIVATIVE-FREE OPTIMIZATION TO THE DEVELOPMENT OF AN INTEGRATED TOOLBOX OF SOLVERS

COMPUTER SCIENCE

NOVA University of Lisbon
September, 2021

NOVA
NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

CONTRIBUTIONS IN GLOBAL DERIVATIVE-FREE OPTIMIZATION TO THE DEVELOPMENT OF AN INTEGRATED TOOLBOX OF SOLVERS

NELSON ALEXANDRE CHARRÉU SANTOS

Bachelor of Computer Science and Engineering

Adviser: Pedro Abílio Duarte Medeiros
Associate Professor, NOVA University of Lisbon

Co-advisers: Ana Luísa Custódio
Associate Professor, NOVA University of Lisbon
Vítor Manuel Alves Duarte
Assistant Professor, NOVA University of Lisbon

COMPUTER SCIENCE

NOVA University of Lisbon
September, 2021

Contributions in Global Derivative-free Optimization to the development of an integrated toolbox of solvers

Copyright © Nelson Alexandre Charréu Santos, NOVA School of Science and Technology, NOVA University of Lisbon.
The NOVA School of Science and Technology and the NOVA University of Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

Ao Tiago, Pedro, Joana e Maria.

Acknowledgements

First of all, I would like to thank the Portuguese Army for the opportunity and for selecting me to accomplish this mission: "Given mission, accomplished mission."

Thanks to Professors Ana Luísa Custódio and Pedro Medeiros for the constant advice, availability, comprehension, and patience throughout the elaboration of this dissertation; the order above reflects the real relevance of each adviser in this work. Their professionalism, enthusiasm, dedication to different areas, and human side will be something I will take as a model for the rest of my life.

Thanks to Professor Vítor Duarte, for the a crucial role in the elaboration of the preparation of the dissertation.

To my university colleagues for constant help and sharing. A special thanks to Filipe for sharing the last three years of the course. He is an example of dedication, commitment, and not exhausting yourself in the search for knowledge.

To my friends who, even not being present, always supported me. Especially Bruno, who, despite the distance, was always present. To Nuno, thanks for having awakened in me the interest in increasing the horizons of knowledge. Though always very occupied, I know he is there when I need him.

A final special thanks to my family. They are my support. Although absent in many moments, Tiago, Pedro and Joana, I never forgot you. To my wife Maria, for her continuous support as mother and father in many situations. Without her, it would not have been possible to complete this hard stage of our lives.

Thank you all.

Abstract

This dissertation is framed in the research project "BoostDFO: Improving the performance and moving to newer directions in Derivative-Free Optimization", funded by Fundação para a Ciência e Tecnologia, whose objective is to develop efficient and robust algorithms for Global and Multiobjective Derivative-Free Optimization. The demand for algorithms belonging to this class appears in different application areas, such as robotics, electrical engineering, aeronautics or oceanography, where, for several reasons, it is not possible to use derivatives.

Global and Local Optimization using Direct Search (GLODS) is an algorithm belonging to this class of optimization methods, which aims at identifying the global minimum of a given problem by computing all the local minima. However, identifying points as global minimizers is always a hard task, being of increased complexity when the function to optimize is computational expensive and time-consuming. Typically, the higher the dimension and complexity of the problem, the more computational effort and time will be required to run the algorithm.

The current version of GLODS is developed sequentially, and it is not fully optimized. The present work will analyze in detail the execution and behavior of GLODS, and will propose and evaluate the numerical performance of different parallelization strategies, implemented using the MATLAB Parallel Computing Toolbox (PCT). The parallelization structure will have a primary purpose of allowing to distribute objective function evaluations among different processors of the hardware platforms, both host locally or in public clouds.

Three strategies for GLODS parallelization were designed and implemented centered on the poll phase of the algorithm. The first two - parallelization of the poll step with one and two poll centers - were successful and gave good results in terms of solution quality and execution time reduction. The third level of parallelization introduced the possibility of dynamically selecting the number of poll centers, according with the number of processors available, a functionality that can be particularly useful in future work in the area.

Keywords: Derivative-free Optimization, Directional Direct Search, GLODS algorithm, Parallel computing, Cloud computing, MATLAB.

Resumo

Esta dissertação enquadra-se no projeto de investigação *"BoostDFO: Aumentando o desempenho e abordando novas dimensões em optimização sem recurso a derivadas"*, financiado pela Fundação para a Ciência e a Tecnologia, cujo objetivo é desenvolver algoritmos eficientes e robustos para Otimização Global e/ou Multiobjectivo sem recurso a derivadas. A procura e utilização de algoritmos pertencentes a esta classe ocorre em diferentes áreas, como a robótica, a engenharia eletrotécnica, a aeronáutica ou a oceanografia, onde, por diversas razões, não é possível a utilização de derivadas.

Global and Local Optimization using Direct Search (GLODS) é um algoritmo pertencente a esta classe de métodos de otimização, que visa identificar o mínimo global de um certo problema através do cálculo dos diferentes mínimos locais. No entanto, identificar pontos como minimizantes globais é sempre uma tarefa difícil, sendo de maior complexidade quando a função a otimizar apresenta elevados custos computacionais e tempo de execução. Normalmente, quanto maior a dimensão e a complexidade de um problema, mais recursos e tempo computacional serão necessários para a execução do algoritmo.

A versão atual do algoritmo *GLODS* encontra-se implementada sequencialmente e não está totalmente otimizada. O presente trabalho analisará em detalhe a execução e o comportamento deste algortimo, propondo e avaliando o desempenho númerico de diferentes estruturas de paralelização para o mesmo, implementadas recorrendo às ferramentas de paralelismo do MATLAB. A estrutura de paralelização terá como objetivo principal a distribuição das avaliações da função objetivo entre diferentes processadores, num ambiente local ou na *cloud* pública.

Delinearam-se e implementaram-se três estratégias para a paralelização do GLODS, centradas na fase de sondagem do algoritmo. As primeiras duas - paralelização do passo de sondagem com um e com dois centros de sondagem - foram bem sucedidas, permitindo bons resultados em termos de qualidade da solução e também uma redução no tempo de execução do algoritmo. O terceiro nível de paralelização introduziu a possibilidade de selecionar o número de centros de sondagem de foma dinâmica, considerando o número de processadores disponíveis, uma funcionalidade que será particularmente útil no trabalho futuro a desenvolver na área.

Palavras-chave: Optimização sem recurso a derivadas, Procura direta dirececional, Algoritmo *GLODS*, Computação paralela, Computação na *cloud*, MATLAB.

Contents

List of Figures xii

List of Tables xv

Acronyms xvi

1 Introduction 1

1.1 Context and motivation . . . 1

1.2 Challenges to address . . . 2

1.3 Expected contributions . . . 2

1.4 Document structure . . . 3

2 State of the art and related work 4

2.1 Derivative-free Optimization . . . 4

2.1.1 Some Derivative-free Optimization applications . . . 5

2.1.2 Directional Direct Search . . . 6

2.1.3 Global Direct Search Optimization . . . 7

2.2 Parallelism . . . 8

2.2.1 Parallel hardware . . . 9

2.2.2 Shared memory multiprocessors . . . 10

2.2.3 Design of parallel algorithms . . . 11

2.2.4 Matlab solutions . . . 13

2.2.5 Metrics . . . 18

2.3 Alternatives for using parallel hardware . . . 19

2.3.1 Interactive use . . . 19

2.3.2 Batch Systems . . . 19

2.3.3 Cloud computing . . . 20

2.3.4 Serverless computing . . . 21

2.3.5 How parallel hardware was used in this thesis . . . 22

2.4 Application of parallelism in Directional Direct Search . . . 22

3 GLODS Algorithm 24
3.1 GLODS general framework 24
3.1.1 Algorithm initialization 24
3.1.2 Search step 25
3.1.3 Poll step 25
3.2 GLODS differentiation from others DDS methods 26
3.3 Potential for parallelization 27

4 GLODS Parallelization 30
4.1 Benchmarking 30
4.1.1 Data Profiles and Performance Profiles 30
4.1.2 Hardware used for benchmarking 32
4.1.3 Testing PCT 32
4.2 Calibration 33
4.3 First level of parallelization 36
4.3.1 Parallelization Structures 36
4.3.2 Implementation using PCT 37
4.3.3 Numerical results 39
4.4 Second level of parallelization 42
4.4.1 Parallelization Structures 43
4.4.2 Implementation using PCT 45
4.4.3 Numerical results 45
4.5 Third level of parallelization 50
4.5.1 Parallelization Structures 50
4.5.2 Implementation using PCT 52
4.5.3 Numerical results 52

5 Conclusions and future work 56

Bibliography 58

Appendices

A Appendix 1: Problem set 62

Annexes

List of Figures

2.1 An example of application of DIRECT optimization algorithm. 8
2.2 Multiprocessor organization. 11
2.3 PCAM: A design methodology for parallel programs or parallel algorithms. 12
2.4 Parallel pool overview. 15
2.5 Example of the use of PCT parfeval with an opportunistic strategy. Results are gathered in order. 17
2.6 Amdahl's law. 19

3.1 Spread of 500 points generated by a Halton Sequence (left) and a Sobol Sequence (right). 25
3.2 Four levels of the 2^n-Centers strategy. 26

4.1 Data profile for the frequency of unsuccessful iterations considered for initializing new searches. 34
4.2 Data profile when initialization is based on the number of active points in the list, not yet identified as local minimizers. 34
4.3 Data profile comparing new initializations based on the frequency of unsuccessful iterations and on the number of active points in the list. 35
4.4 Data profile comparing two different strategies for generating new points in the search step. 35
4.5 Data profile comparing opportunistic and complete polling. 35
4.6 Code snippet with the parallelization implemented at the algorithm initialization and at the search step. 38
4.7 Code snippet with the parallelization implemented at the poll step. 38
4.8 Data profile comparing versions: sequential, parallel ordered and parallel unordered. 39
4.9 Data profile comparing versions: sequential, parallel ordered and unordered (mean, low/high interval). 40
4.10 Data profile comparing versions: sequential, parallel ordered and unordered (mean, low/high confidence interval), now with level of accuracy $\varphi = 10^{-3}$. 40

4.11 Performance profile comparing computational time, with 0,03 seconds delay for function evaluation (average results). 40
4.12 Performance profile comparing computational time, with 0,04 seconds delay, for function evaluation (average results). 40
4.13 Performance profile comparing computational time, with 0,04 seconds delay for function evaluation (low extreme of average confidence interval). 41
4.14 Performance profile comparing computational time, with 0,04 seconds delay for function evaluations (high extreme of confidence interval). 41
4.15 Code snippet with the parallelization implemented at poll step in second phase. 45
4.16 Data profile with all parallel versions of phase two and the best parallel version of phase one. 46
4.17 Performance profile comparing computational time, for the two best strategies of phase two, considering a 0.04 seconds delay in function evaluation. . . . 46
4.18 Performance profile comparing the number of local minima identified for the best strategies of phase two. 46
4.19 Performance profile comparing the best objective function value computed, for the two best strategies of phase two. 47
4.20 Performance profile comparing the number of function evaluations required until the global minimum is identified, for the best two strategies of phase two. 47
4.21 Performance profile comparing computational time, for the best versions of parallelization phases one and two, considering a 0.04 second delay for function evaluation. 47
4.22 Performance profile comparing the number of local minima identified, for the best versions of parallelization phases one and two. 47
4.23 Performance profile comparing the best objective function value computed, for the best versions of parallelization phase one and two. 48
4.24 Performance profile comparing the number of function evaluations required until the global minimum is identified, for the best versions of parallelization phase one and two. 48
4.25 Code snippet with the parallelization implemented at poll step in third phase. 52
4.26 Data profile with all versions of phase 3 and the best parallel version of phase 2. 53
4.27 Performance profile comparing computational time, for all strategies of phase three and the strategy of phase two. 54
4.28 Performance profile comparing the number of functions evaluations required until the global minimum is identified, for all strategies of phase three and the strategy of phase two. 54
4.29 Performance profile comparing the number of local minima identified, for all strategies of phase three and the strategy of phase two. 54

4.30 Performance profile comparing the best objective function value computed, for all strategies of phase three and the strategy of phase two. 54

List of Tables

2.1 Parallelizing tasks inside MATLAB. 17

4.1 Real computational time vs estimated time. 33
4.2 Computational times (sec.) required for solving problems of different dimensions (4 workers to 19 workers). 33
4.3 Computational times (sec.) required for solving problems of different dimensions (20 workers to 64 workers). 33
4.4 Speedup and Efficiency for phase 1. 42
4.5 Speedup and efficiency for phase 2. 49
4.6 Analysis of individual computational times. 55

A.1 A description of the test set (first part). The variable n represents the dimension of the problem, l and u, lower and upper bounds, respectively, on the problem variables, loc and $glob$, the number of local and global minimizers, respectively, reported in the literature. 63
A.2 A description of the test set (second part). The variable n represents the dimension of the problem, l and u, lower and upper bounds, respectively, on the problem variables, loc and $glob$, the number of local and global minimizers, respectively, reported in the literature. 64

Acronyms

APPS Asynchronous Parallel Pattern Search 22

BaaS Backend as a Service 21

CMA Centre of Mathematics and Applications 32
CS Coordinate Search 6, 7

DDS Directional Direct Search 1, 2, 3, 4, 6, 7, 8, 22, 24, 26, 27
DFO Derivative-free Optimization 1, 3, 4, 5, 6, 8, 22, 30, 56, 57
DLP Data-level Parallelism 10
DMP Distributed-memory multiprocessors 10, 11
DS Direct Search 1, 27

FaaS Function as a Service 21

GLODS Global and Local Optimization using Direct Search vi, vii, viii, ix, 1, 2, 3, 6, 8, 9, 11, 13, 16, 19, 22, 24, 25, 26, 27, 30, 31, 33, 36, 52, 56, 57
GPS Generalized Pattern Search 6
GPUs Graphic Processor Units 10, 14

IaaS Infrastructure as a Service 20, 21
INCD National Distributed Computing Infrastructure 22, 32

MADS Mesh Adaptive Direct Search 6

NIST National Institute of Standarts and Technology 20

PaaS Platform as a Service 20

PCT Parallel Computing Toolbox vi, xii, 2, 9, 13, 14, 15, 16, 17, 32, 36, 56
PMD Personal Mobile Devices 9
PSD-MADS Parallel Space Decomposition-Mesh Adaptive Direct Search 22

SaaS Software as a Service 20
SMP Shared-memory Multiprocessors 10, 11, 19, 22
SPMD Single Program Multiple Data 16
SSH Secure Shell 19

TLP Task-Level Parallelism 10

VMs Virtual Machines 21, 22

WSCs Warehouses-Scale Computers 10

1

Introduction

1.1 Context and motivation

This dissertation is framed in the research project: "BoostDFO: Improving the performance and moving to newer directions in Derivative-Free Optimization (PTDC/MAT-APL/28400-/2017)", funded by Fundação para a Ciência e Tecnologia, whose main objective is to develop efficient and robust algorithms for Global and Multiobjective Derivative-free Optimization.

Optimization problems are widespread in all areas of knowledge, emerging in science, economy, engineering or health. Developments in mathematical programming and coupling of mathematical knowledge with computational capabilities, allowed the development of optimization models and methods, impossible to consider until recently, due to the associated computational cost [22].

In the scientific domain of Mathematics, optimization focuses on the study of a mathematical problem in which it is intended to minimize or maximize a function f, possibly subject to a set of constraints [47]. Derivatives are an important tool in this optimization process, since they can be used to identify and classify optimal points. However, it is not always possible to use derivatives to solve a mathematical optimization problem. There are cases in which a mathematical expression is not available to be used and estimation, using numerical techniques or automatic differentiation, is unviable. Furthermore, the function to optimize could be nonsmooth, thus it is impossible to consider derivatives for it (in its most traditional form). This way, a particular class of methods was developed to address these challenges, namely the one of Derivative-free Optimization (DFO).

In the resolution of DFO problems, several methods can be considered, and the choice of the most proper one depends on the characteristics of the problem being analyzed [28]. The present work will focus on deterministic algorithms, belonging to the Direct Search (DS) class, which do not attempt to model the objective function. The choice is motivated by the fact that GLODS [13], the algorithm to be optimized and parallelized, is also a DS method, in particular a Directional Direct Search (DDS) method.This class of algorithms usually proceeds by alternating between two main procedures: the search

and poll steps. The search step has as main goal the identification of regions with the potential to be explored, and the poll step will explore locally these potential regions previously identified. The main goal of GLODS is to compute all local minima of a given problem (assuming a finite and reasonably small number for it), wherefrom the global minimum would be easily identified. However, identifying points as global minimizers is a hard task, as it is computationally expensive and time-consuming, combined with the difficulties mentioned above [13].

Therefore, parallel computing emerges as a tool that can be useful for the resolution or mitigation of the above mentioned difficulties. The main goal of parallel computing is to design programs that execute tasks on multiple processors, or where processors cooperate to solve a single task [26]. With parallel computing, a problem or a program could execute more efficiently, saving computational time and improving performance. In most cases of efficiency gains and added resources, parallel computing scales with problem size, allowing the resolution of problems of higher complexity and/or dimension [34].

Another possibility for increasing computing capabilities is resourcing to cloud computing, which offers elastic and scalable computing, where only the resources used are charged. With cloud computing, an user has a powerful and cost-effective computing capacity [29]. Parallel computing combined with cloud computing, are options to consider when an user wants to solve a complex optimization problem.

As it will be described in more detail in Section 2.4, the use of parallel computing in DDS methods allows to increase the numerical performance of this algorithmic class.

1.2 Challenges to address

As previously mentioned, being a DDS method, GLODS proceeds by alternating between a search step and a poll step, which will be more detailed in Chapter 3.

Both steps are not optimized. The search step, which is not performed at every iteration, initializes new lines of search by sampling new points. Currently, previous information gathered about objective function is not taken into account for placing these new points. Also the evaluation of these new points is done sequentially.

Regarding the poll step, a local search is done around a selected point, the poll center, again by sequentially evaluating a finite set of points. Additionally, only one poll center is selected at the time, without taking advantage of the characteristics of the hardware, more specifically, the existence of different processors.

Another challenge is related to the GLODS implementation language: MATLAB. The whole optimization and parallelization structure will have to be developed using the toolboxes available in MATLAB, namely the PCT.

1.3 Expected contributions

The expected contributions of the present work are:

- Study and analysis of the existing literature, with a focus on four significant domains: DFO, parallelism, cloud computing and application of parallelism in DDS methods;
- Analysis of the algorithmic structure of GLODS;
- Optimization of the sequential version of GLODS;
- Development and numerical evaluation of parallelization strategies for GLODS.

1.4 Document structure

The document is organized as follows. Chapter 2 reports the state of the art and other related work, consisting particularly of four areas of analysis: DFO; parallelism; cloud computing; and parallelism application in DDS. Chapter 3 details the GLODS algorithm, analyzing its algorithm structure and proposing strategies to optimize it. Numerical results are reported in Chapter 4, including the different parallelization strategies considered for performance improvement. Finally, Chapter 5 is devoted to some conclusions and future work.

2

State of the art and related work

This chapter aims to present the state of the art and other work related to the subject of this thesis. Section 2.1 introduces the concept of DFO and provides motivation for the use of this class of optimization methods. Section 2.2 revises parallelism, presenting the corresponding basic concepts, and also solutions for its use provided in MATLAB. Section 2.3 revises cloud computing concepts. Finally, Section 2.4 reports some uses of parallelism in DDS.

2.1 Derivative-free Optimization

Mathematical optimization is a scientific domain which addresses the problem of minimizing or maximizing a function. Derivatives are an important tool for this task. However, there are situation where derivatives are not available or are not reliable to be used, motivating the class of DFO methods.

According to [10, p.1] "we consider optimization without derivatives one of the most important, open, and challenging areas in computational science and engineering, and one with enormous practical potential". The constant evolution of computing capabilities, coupled with the increasing complexity of the mathematical models considered, and the existence of a large amount of legacy codes, justifies the continued evolution and analysis of DFO techniques.

In the early days of nonlinear optimization methods, their use was essentially due to a lack of computational power, and to the simplicity of the corresponding algorithms. With the increase in the complexity of mathematical models, the evolution and implementation of techniques based on the use of derivatives has become fundamental. The development of optimization methods based on derivatives allowed the efficient solution of complex and large scale problems, but required exact information about derivatives or, when it is not possible, to consider accurate estimates of them, using finite differences or automatic differentiation tools.

As an example, when the objective function results from legacy codes, which are

programs that were developed long ago in the past and have not been maintained, derivatives are typically not available. The possibility of computing derivatives is dependent on how a company makes its source code available. For example, if binary files are used, reverse engineering methods will be required to make the binary files legible for using automatic differentiation tools for computing derivatives. However, even if the source code is available, there are cases where the objective function is nonsmooth. Thus, any attempt of estimating derivatives will be unsuccessful [10].

The numerical estimation of derivatives, using finite-differences, can also present some inconveniences, for instance when the function is expensive to evaluate or when in presence of numerical noise. The first situation typically appears in simulation based optimization, when each function evaluation can take from several seconds to even minutes of computational time. In the second situation, noise will jeopardize the accuracy of the finite-differences estimates [10].

From what has been said, we see that there is a need for DFO algorithms. Additionally, the problems requiring optimization without derivatives can be very diversified: continuous or discrete; convex or nonconvex; smooth or nonsmooth. In the case of smooth problems, there could be situations where all functions defining objective and constraints are differentiable, but even so derivatives are not available [10].

2.1.1 Some Derivative-free Optimization applications

Once the concept of DFO has been defined, it becomes important to show that indeed there are real application problems which require this type of optimization techniques.

The first example is related to the positioning of buoys for tsunami detection [42]. Tsunamis are essentially large waves that are caused by oscillations originating on the ocean floor. One of the most deadly tsunamis occurred in 2004 in Indonesia, in the Sumatra island, leading to about 250.000 deaths. After this catastrophe, it was decided to place sea buoys in the Pacific Ocean, each one being connected to a vibration sensor, placed at the bottom of the ocean. Thus, when abnormal vibrations occur on the ocean floor, each sensor, connected to a buoy, communicates via satellite to a command and control center that analyses the data received. The related optimization problem consists in determining the correct location of the buoys, x and y coordinates, so that the warning time, in case of danger, is maximized. To get a sense of the specificity of these problems, the optimization of the placement of only 6 buoys generates 12 variables, and each simulation, corresponding to one function evaluation, may take up to 30 seconds to be executed. Therefore, if for 6 buoys the necessary time for each simulation is 30 seconds, let's imagine the same problem, but open to a larger number of buoys, and consequently more variables, as well as increasing the simulation time. We are then reaching the importance of having good quality DFO algorithms, for the reduction of computing times, allowing to tackle harder problems, possibly with a large number of variables.

Another example of the use of DFO is also related to catastrophes, namely earthquakes,

which occur every day around the world, leading densely populated and urbanized urban areas to be severely affected [7]. Therefore, one of the tools used to minimize the effects of earthquakes is the installation of dampers in adjacent buildings, avoiding buildings collision and increasing the corresponding structural stability. The costs related to the effects that an earthquake can cause, as well as to the installation of dampers, led researchers to develop mathematical methods in order to minimize the amount of dampers needed, but at the same time to be able to withstand the maximum possible oscillations.

A final example is related to the vibration transmitted by the rotor of a helicopter [8, 9, 40]. The objective of the optimization problem is the design of a rotor that transmits as few vibrations as possible in operation, therefore a minimization problem. There are several factors that may affect the vibration caused by the rotor, such as atmospheric conditions (e.g. wind), or even inherent vibrations transmitted by the engine. In this problem there are several constraints that need to be satisfied, such as the aerodynamics necessary for the stability of the helicopter and the resistance of materials. Taking into account all the variables and constraints of this type of problems, the simulation may take from minutes to days of computational time.

2.1.2 Directional Direct Search

Direct Search algorithms emerged as an evolution and correction of limitations to the early DFO methods. Their designation arises because these methods interact directly with the objective function, by sampling it, not making any attempt to model it neither its derivatives (when they exist).

Coordinate Search (CS) is one of the first and simplest algorithms belonging to this class. CS constitutes a starting point for the so called DDS methods, such as Generalized Pattern Search (GPS) [3], Mesh Adaptive Direct Search (MADS) [4], or GLODS algorithm [13], which will be further analyzed in Chapter 3.

A typical iteration of a DDS method generates a finite set of points in a neighborhood of the point corresponding to the current iterate, x_k. This set of points, named as poll set, is generated by considering a set of directions, the poll directions, which justifies the terminology DDS. Each new point is computed by adding to the current iterate, x_k, the value of $\alpha_k d$, where α_k represents a positive step length, and d is an element of the finite set of directions D_k, considered [6]. These poll directions need to satisfy some geometrical properties, in order to ensure convergence of the algorithms. Typically, positive spanning set are considered [14]. A positive spanning set for $\mathbb{R}^n$ is a set of vectors that generates $\mathbb{R}^n$ through non negative linear combinations of its elements.

The objective function f is then evaluated at the poll points, and x_{k+1} is selected as a poll point that improves the function value over the previous x_k, if possible. If successful, the step length could be increased before the algorithm proceeds. If no poll point was found that improves the objective function value, then $x_{k+1} = x_k$ and the step length is

decreased. In both cases, the set D_k of directions can be modified to get D_{k+1} [6].

Algorithm 1 presents the pseudo-code of a typical algorithm of DDS. Additionally to the previously described poll step, there could be a search step, where the function f is evaluated at finite set of points Y_k. If the search step is successful, then the poll step can be skipped. The search step can be omitted, since the convergence properties of the algorithms relies on the poll step. However, the search step is one of the places where one can invest, so that gains can occur in runtime or in the quality of the local minimizer computed. Concerning computational time, notice that the poll step has a natural structure for parallelization, which can improve, the numerical performance of the algorithm, and will be an important part of the present work [28].

Algorithm 1: A Directional Direct Search method

```
1  Set parameters 0 < γ_dec < 1 ≤ γ_inc
2  Choose an initial point x_0 and step size α_0 > 0
3  for k = 0,1,2,... do
       // (search step)
4      Choose and order the finite set Y_k ⊂ ℝ^n
5      x_k^+ ← test_descent(f, x_k, Y_k)
6      if x_k^+ = x_k then
           // (poll step)
7          Choose and order the poll directions D_k ⊂ ℝ^n
8          x_k^+ ← test_descent(f, x_k, {x_k + α_k d_i : d_i ∈ D_k}
9      end
10     if x_k^+ = x_k then
11         α_{k+1} ← γ_dec α_k
12     else
13         α_{k+1} ← γ_inc α_k
14     end
15     x_{k+1} ← x_k^+
16 end
```

Different DDS methods correspond to different search steps, respecting to the use of different sets Y_k, or to different sets of poll directions, D_k . In the case of CS, $Y_k = \emptyset$ and D_k is defined by $D_k = [I - I]$, where I represents the identity matrix [28]. Polling can be opportunistic, in which the evaluation of the poll points is stopped once that a success is found, or complete, where all poll points are always evaluated.

2.1.3 Global Direct Search Optimization

The identification of a global minimum is always a difficult task, even when derivatives are available, unless we have additional information regarding the objective function (e.g. convexity). Difficulties increase in the absence of derivatives. A simple approach considered by users is to run a local optimization algorithm from different initializations,

a technique known as multistart. However, when in presence of expensive function evaluation, this approach is often unrealistic. Specific algorithms need to be developed for this class of problems and DIRECT [25] is one of them.

DIRECT, from dividing rectangles, builds on Lipschitz optimization and was developed for bound constrained DFO optimization. It proceeds by dividing the feasible region into subregions, using the central point of each rectangle as representative of that subregion (see Figure 2.1, extracted from [15]).

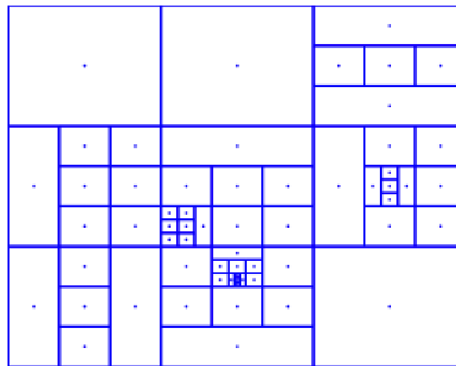

Figure 2.1: An example of application of DIRECT optimization algorithm.

One of the fundamental aspects of a global DFO algorithm is to ensure that no region with potential is overlooked, thus avoiding the loss of points that can be considered global minimums. Another important aspect is efficiency. The global algorithms previous to DIRECT typically started with a global search emphasis, and later exchange for local search. DIRECT algorithm balances global and local search, performing a little bit of the two at every iteration, by subdividing the most promising subrectangles [25].

As mentioned before, simple multistart approaches are not suited for global DFO, since they can lead to inefficiencies, where several local searches can converge to the same local minimizer. To overcome this inefficiency, and to be further detailed in Chapter 3, GLODS algorithm allows the launch of multiple instances of DDS methods, to explore the feasible region, but these lines of search are merged when they start to be closed to each other [13].

The search step of DDS methods can also be used to confer some global behavior to the algorithms. That was the case of PSwarm [44], which implemented a particle swarm algorithm at the search step. Whenever the search step fails, the poll step is applied to the particle corresponding to the best function value.

2.2 Parallelism

Before proceeding to characterize the concept of parallelism, it is essential to identify the problem under analysis, in order to understand the structure of this section. The

main objective of the optimization to be performed comprises the launching of multiple instances of functions evaluations, to be executed in parallel.

GLODS algorithm is implemented in MATLAB, which is characterized by implicit parallelism [21], being limited and not highly configurable. To use explicit parallelism, the PCT can be used, ensuring higher freedom and customization for a specific problem, which will be detailed in Subsection 2.2.4. Subsection 2.2.3 describes a methodology that will help to organize the solution.

Regarding parallelism, an important milestone not only for computer design, but also to achieve greater gains in computing time, was the possibility of computers possessing several processors. Having several processing elements allows us to partition the computations, conducting in most cases to shorter execution times [12].

Nowadays, any electronic device is composed by hardware that allows parallelism, either through the existence of multiple processors, multithreads or multicore processors. It is imperative to use parallel programming exploiting the full capabilities of the underlying hardware, better fulfilling the requirements of the end user [12].

2.2.1 Parallel hardware

There are five classes/types of computers according to [19]: Personal Mobile Devices (PMD), desktop computers, servers, clusters-warehouses-scale computers, and embedded computers. Due to the scope of the present work, PMD and embedded computers will not be discussed.

Desktop computers represent the largest segment, and currently generate the highest sales volume. This class is materialized from portable computers to workstations. A concept associated with this class is price-performance. This concept translates into the fact that the final value of the computer depends on the desired performance, where a higher performance is usually associated to a more elevated cost [19].

Servers have the main goal of supporting all the needs of the previous classes, but there is a fundamental concept that characterizes servers: availability [19]. For example, a system that demands the existence of availability for its operation is the ATM network, becoming necessary to have a set of servers that guarantees availability at all times. Availability is usually ensured through server replication. Therefore, in case of server failure of a given server, it is promptly replaced, ensuring the continued operation of the system. Another fundamental aspect for servers is scalability [19], which is the capacity that a system has to do operations in an effective way in case there is a significant increase of users [11]. As challenges for this class of computers, we can mention that servers are used in such a way that they have a high yield in ensuring availability and scalability. In the context of this work, the availability dimension is not relevant.

Clusters are sometimes made up of servers or desktop computers connected to each other, usually running on a network, their behavior being materialized as if there was only one computer, providing the illusion of the existence of a supercomputer. The term

Warehouses-Scale Computers (WSCs) appears to describe large clusters. Clusters can be made up of tens, or thousands of computers, and the total cost of ownership of these type of systems is higher than the one of classes described above [19].

Regarding the types of parallelism present in the software, there are two models: Data-level Parallelism (DLP) and Task-Level Parallelism (TLP). DLP means that the data in the systems can be accessed in a parallel way, e.g. applying the same operation simultaneously to a set of data elements. TLP refers to the possibility of creating tasks that can be executed in parallel. Therefore, DLP is more data oriented and TLP is more focused on tasks that can create parallelism [19]. To operate these models, the hardware can be used in several ways, namely:

- Instruction-Level Parallelism: uses the model DLP with the help of the compiler for organizing the processes, for example, in Pipeline;
- Vector Architectures and Graphic Processor Units (GPUs): also explore the model DLP, and aim only to apply parallelism to a data set;
- Thread-Level Parallelism: can be used with both DLP and TLP models and goes through the use of threads in a parallel way;
- Request-Level Parallelism: consists of the use of tasks, performed on request by an user or the operating system [19].

The different types of hardware available to programmers allow the construction of parallel programs, providing the time needed to produce results. Choosing the type of hardware to use, and the type of parallelism, is part of a decision process that a programmer will have to perform, with the goal of obtaining the highest earnings.

2.2.2 Shared memory multiprocessors

A multiprocessor is built interconnected with P processors and M memory modules [43]. There are two main multiprocessor organizations (see Figure 2.2):

Shared-memory Multiprocessors (SMP) Each P processor can access all the M memory modules directly and in constant time. This organization makes very high demands over the network that connects processors and memory modules, thus limiting the P number.

Distributed-memory multiprocessors (DMP) Here, P is equal to M as each memory module is only connected to one processor. Each set of a processor plus a memory module is called a *node*; all the nodes have hardware that allows a fast connection to the other nodes, and some (or all) the nodes have other input/output devices. This causes less requirements over the interconnection network, allowing a much greater number of processors.

Today's largest machines are DMP where each node is itself an SMP.

Programming a SMP is easier than programming a DMP, as the programmer does not have to deal with the explicit transfer of data between distinct memory modules.

In this work, the programming model used to develop the parallel implementations assumes a SMP. In section 3.3 we discuses alternatives for the deployment of our implementations of GLODS.

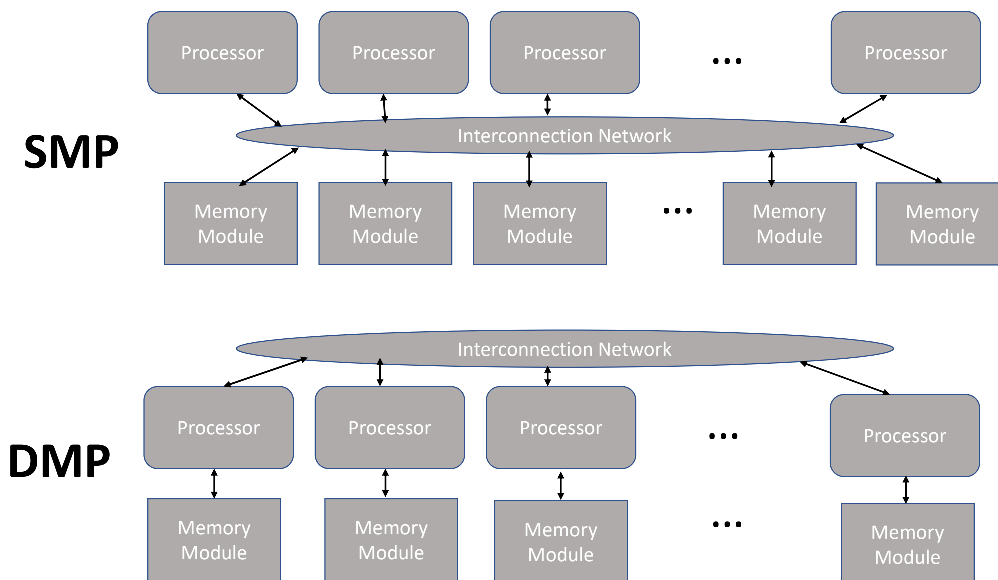

Figure 2.2: Multiprocessor organization.

2.2.3 Design of parallel algorithms

Not all algorithms with sequential characteristics can be optimized with the application of parallelism. There are situations where this transformation is not beneficial. Depending on the problem under analysis, it could be more advantageous to use a sequential architecture instead of a parallel one.

Additionally, if there is an advantage in parallelizing a specific algorithm, it may not be an easy task, due to the diversity of ways that can be used to achieve it. Thus, in order to support the implementation of parallel solutions, a design methodology has been established that allows: to maximize the whole range of parallelization possibilities; mechanisms for evaluating alternatives; and reducing costs in case of bad decisions about parallelization. This methodology is structured in four phases: Partitioning, Communication, Agglomeration and Mapping (see Figure 2.3, extracted from [35]).

The first two phases consist of examining implementations that support concurrency and scalability, and the remaining phases are aimed at analyzing and maximizing performance [18]. Describing each phase, and according to [18]:

Partitioning Intends to expose opportunities for applying parallel solutions. The hardware characteristics are ignored at this stage, such as the number of processors.

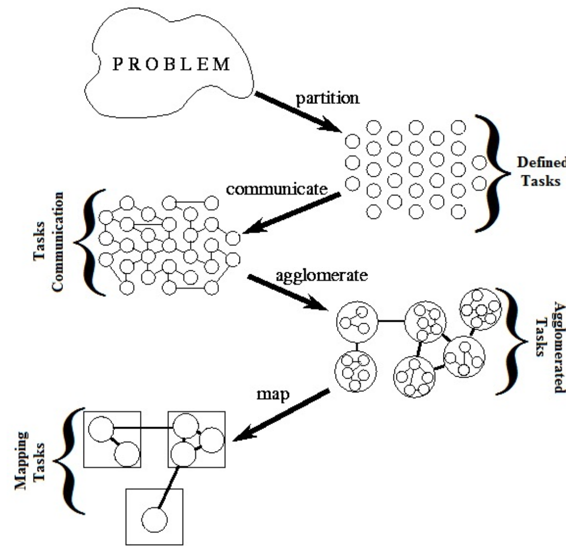

Figure 2.3: PCAM: A design methodology for parallel programs or parallel algorithms.

In order to analyze the potential for parallelization, it is essential to define a large number of small tasks that materialize the fine-grained decomposition of a problem. When executed correctly, a good partition phase combines not only the computation associated with the problem, but also the data used by it. Sometimes programmers first focus on problem data, then partition the computations and finally combine both. This partition technique is called domain decomposition. On the other hand, the functional decomposition primarily focuses on the decomposition of computations and then on the problem data. These two approaches are complementary and should be used in order to obtain several parallel algorithms.

Communication The tasks that are partitioned in the first phase are intended to be executed simultaneously, but not independently, since there will be sharing of data or resources between them. Therefore, the flow of information between tasks is specified in this design phase. If decomposition was of the domain type, the communication requirements are more challenging to achieve. Many operations require the data they share to remain for use by other tasks, and structuring this communication in an efficient way is a challenge. If we are in the presence of functional decomposition, the communication requirements are simpler to solve: it corresponds to the data flow between tasks. In summary, the objective of this phase comprises the analysis of the communication between tasks, reducing the communications flux between them.

Agglomeration In the first phase, tasks were divided into a large number of small tasks, to materialize fine-grained decomposition. This division is somewhat abstract and it is not an optimal solution, because, for example, it creates a large number of tasks

compared to the total number of processors, and computers were not designed to perform such small tasks. So, this phase is intended to move from an abstract state to a real state, by combining tasks, with the primary purpose of obtaining an adequate granularity.

Mapping The objective of this phase is the placement and distribution of processes among the different processors, with the main goal of minimizing the execution time: minimize communications by aggregating processes that frequently communicate, increasing locality; maximize concurrency by separating independent processes; uniforming the distribution of processes by different processors. During the implementation of the aforementioned objectives, conflicts or limitations may occur. A trade-off will have to be assumed, according to the objectives of the problem.

Performing an approximation to the methodology for the problem under analysis, a task resulting from the partition phase would be an unitary evaluation of the objective function. Regarding the second phase, communication between tasks is not expected. As the evaluation of a function may take some time to execute, no agglomeration will be executed. In the mapping phase, the work will target different hardware platforms (symmetric multiprocessors and clusters), hosted locally or in public clouds.

2.2.4 Matlab solutions

The purpose of this Subsection is to describe the tools already available in MATLAB for parallelization. The justification for this analysis is the fact that the target algorithm, GLODS, which we intend to parallelize to increase the corresponding numerical performance, is implemented in MATLAB. According to [46], MATLAB is a language which comprises a development environment with the objective of constructing numerical analysis, graphs or algorithms, transversal to several scientific areas such as Mathematics, Physics, Biology, among others [41].

MATLAB is developed by MathWorks, its first release took place in 1984, being used all over the world by engineers and scientists. MATLAB offers routines that extend its base version, called toolboxes, allowing developers to choose and use the specific functions they need for addressing a given problem. One of the toolboxes provided by MATLAB is the PCT, which will be the subject of a more detailed analysis in the current Subsection and will be used in the parallel implementation of GLODS.

MATLAB language is characterized by implicit parallelism, applying it automatically, without the need of any code modifications. However, this automatic process is not configurable, not allowing to evaluate the efficiency of the use of parallel hardware. We intend to use explicit parallelism, in order to maximize the capabilities of the hardware. Explicit parallelism is possible, for example, using PCT.

The main function of PCT is to assist the programmer to address the question of parallelism, achieving the maximum use of the resources that the hardware can provide,

such as the existence of multicore or multiprocessor computers, or the access to a cluster [37].

According to [37], PCT can be used in different contexts to improve the latency of a program:

- For applications with segments of repeated code, allowing the use of a parallel for-loop statement, which will permit the code to be executed in a parallel way;
- For applications in which there is no dependency between the different tasks, enabling the distribution to different workers;
- For workers whose tasks are divided among them;
- Use of clusters or GPUs to improve computing capabilities.

According to [38], to improve the performance of a program, there is a sequence of steps that can serve as a reference for building programs in a methodical and stable manner, particularly when using MATLAB:

1. Initially, it is necessary to measure the performance of the program, in order to decide whether or not it is necessary to improve it;
2. If improvement is required, a tool called profiler should be used, which will identify application hotspots. We can define hotspots as places in the code that will be critical and directly influence performance;
3. Finally, modifications are introduced, tested and, if necessary, we will return to the beginning of this sequence of steps.

Before introducing parallelism in the algorithm, it is important to detail the profiler tool, which was mentioned in the second point of the previous list and which is fundamental for the analysis of the program behavior. According to [1, p. 26], "The MATLAB Profiler is the single most important tool in our arsenal when we need to improve MATLAB application performance." In fact, profiling is the set of actions that aim to use the profiler in order to analyze the latency spend in all operations executed, being therefore a fundamental process when one intends to analyze the performance, highlighting the following adage: "whatever cannot be monitored and measured, cannot really be improved" [1, p. 25].

Thus, the use of the profiler in programs implemented in MATLAB can help any developer with the: identification of existing bottlenecks; identification of code that is not executed and can be removed; identification of data that can be cached, so that it can be accessed more quickly; or the identification of segments that are called unnecessarily and that may contribute to the increase of latency. MATLAB incorporates the following tools that allow profiling: profiler tool and tic/toc functions. The profiler tool is used

to execute profiling, while the tic/toc functions allow the measurement of performance using a stopwatch timer [1].

If parallelization can be used to improve performance a given code, MATLAB PCT provides three constructions, `parfor`, `spmd`, and `parfeval`, to accomplish it. These three tools will allow the execution of phase four referred to in Subsection 2.2.3: mapping.

`Parfor` is used in the `for` loops body and has the task of executing the corresponding statements in parallel [36], using multiple processors. In each statement where `parfor` is invoked, the necessary computations to be executed in parallel are sent by the MATLAB client to a group or set of workers, where the computations will be carried out, and the results will be sent back to the client [1]. The group of workers used by MATLAB, belonging to a computer or cluster, as shown in Figure 2.4 [36], is designated by parallel pool and should have been previously assigned. In Figure 2.4, a MATLAB client only requests three workers in a set of eight workers. The poll size can be customized, using the commands `parpol` and `parcluster` [36].

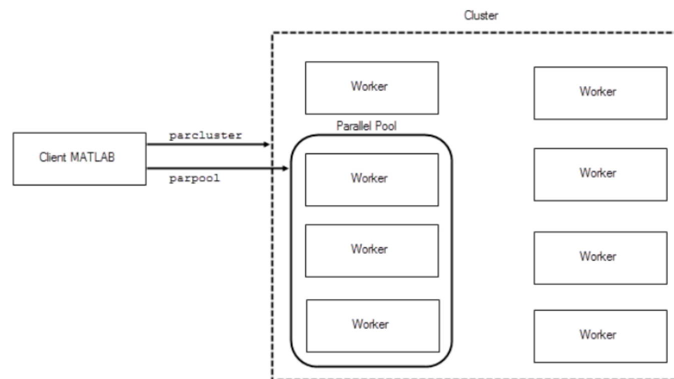

Figure 2.4: Parallel pool overview.

Each execution performed by the set of statements covered by `parfor` is an iteration. The pool of workers, responsible for the parallel execution of the computations, evaluates and executes the iterations without any specific order and independently. In case of independence between the operations, the result will be deterministic. However, if situations occur in which there is dependence between the different executions, the program may present non deterministic results. If the number of iterations is equal to the number of workers available or selected, each worker performs an iteration. If there are more or less iterations to perform than the number of workers, there will be workers who execute more or less iterations, and this management is performed internally by PCT [1].

The tool or feature `parfor`, if well used, will allow the developer to obtain gains or decrease latency when compared to sequential versions. However, there are some situations where it is not advisable to use it. More specifically, as mentioned before, if there is a dependency between the iterations, it may lead to a non deterministic behavior. Furthermore, it is not advisable in situations where the computations to be executed are not very complex, not leading to a decrease in latency. This is the result of some overheads

that influence the final execution time, namely the one related to parallel pool creation, the one inherent to the loading of the libraries, or even the communication overhead that occurs between the client and the workers.

Another statement that developers have at disposable is Single Program Multiple Data (SPMD). `spmd` is very similar to the statement used in loops. However, it provides the developer with greater control over parallelism and distribution of iterations among the available workers [1], allowing "seamless interleaving of serial and parallel programming" [36, p. 3-2].

`spmd` allows a block of code to run on multiple workers, unlike `parfor`, which can only be used in a loop. It works as follows: a given program is executed on the client, and all the blocks within the statement `spmd` are handed over to the workers to execute them. When they finish, the program continues to execute on the client. The variables inside the code block inserted in the statement `spmd` are accessed and changed by reference [36].

Like `parfor`, the number of workers can be customized. According to [36], we should use `spmd` in two situations: in programs where the latency is very high, enabling the different workers to perform the tasks simultaneously; and in programs that use large data sets, resulting in the data being distributed among the available workers.

The feature `parfeval`, allows the function execution on a parallel poll of workers, without waiting for the end of the iteration. This feature may be useful because it allows to interrupt the cycle earlier, if the results are already good enough, which may be the case with the GLODS poll step with opportunistic strategy [36]. Figure 2.5 demonstrates the working flow of `parfeval` where 10 tasks are distributed by 3 workers. The first 3 tasks are completed and there is no improvement in evaluated points. Workers are assigned with new tasks and a successful evaluation (task 4) is obtained by worker 1, and the pending computations are aborted.

A fundamental aspect to consider when choosing `parfor`, `spmd` or `parfeval` is that `parfor` requires all iterations to be executed, not allowing the cycle to be interrupted, unlike `spmd` or `parfeval`, which make possible the aborting of current non-relevant computations.

A set of preliminary tests was conducted on the Stratus system, whose characteristics are described in Subsection 4.1.2, to confirm and better understand the behaviour of parallelization structures inside MATLAB PCT. The results, presented below, complement another set of tests presented in Subsection 4.1.3.

As already mentioned, in real applications, function evaluation can take seconds or even minutes of computational time. Considering the large number of tests needed to be executed (e.g., varying the number of processors, problem size, number of runs), it is not feasible to include artificial delays in the evaluation of the academic functions to mimic the behaviour of real applications.

In all numerical experiments with GLODS algorithm, computational time is estimated by counting the number of parallel cycles as $\frac{\#evaluations}{\#workers}$ (rounded up) and then multiplying it by the considered time delay, simulating the high costs mentioned above. This value is

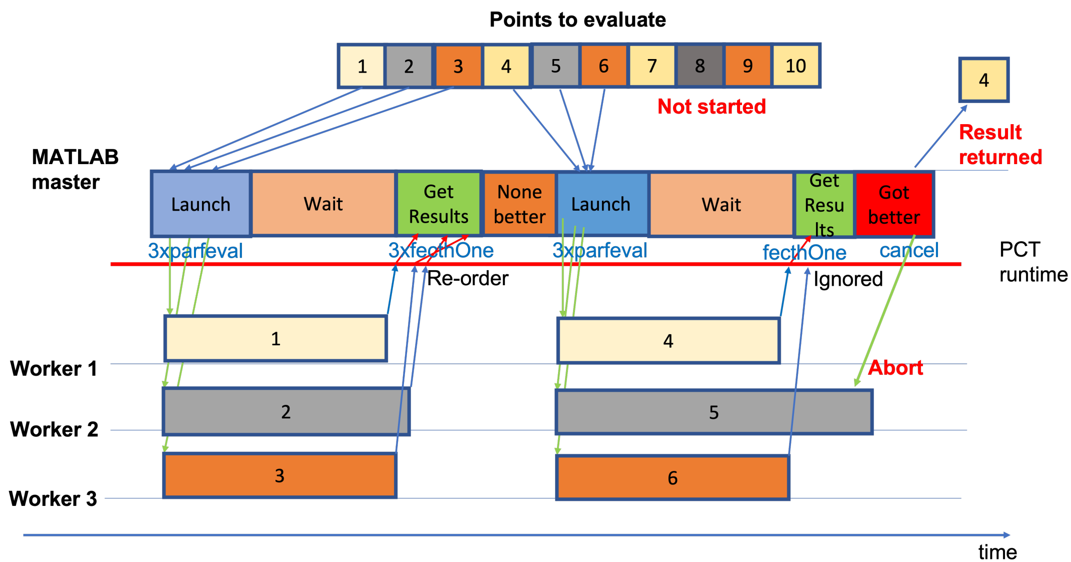

Figure 2.5: Example of the use of PCT parfeval with an opportunistic strategy. Results are gathered in order.

finally added to the real computational time. The first question that arises is if this is an acceptable strategy for estimating the computational time. The results on Table 2.1 and on Section 4.1.3 address the question.

The first test conducted, reported in Table 2.1, attempts to grasp the way as `parfor` and `parfeval` distribute the tasks among the different workers available. The assumption is that a task is only assigned to a worker, once a worker is free. Since we are considering the same computation time for all functions evaluations, doubling the number of workers would result in halving the required computational time.

The MATLAB pause function was used to include an artificial delay of 3 seconds, every time that a simple operation was performed inside a loop of 64 operations. `Parfor` and `parfeval` were used to parallelize this loop. Table 2.1 reports the average real execution time for 30 runs, corroborating the paradigm assumed.

Table 2.1: Parallelizing tasks inside MATLAB.

Tasks	CPU'S	Parallel cycles	Time PARFOR (sec.)	Time PARFEVAL (sec.)
64	8	8	24,14	24,20
	12	6	18,14	18,28
	16	4	12,13	12,22
	24	3	9,14	9,14
	32	2	6,11	6,23
	48	2	6,13	6,16
	56	2	6,17	6,10
	64	1	3,18	3,31

2.2.5 Metrics

In order to evaluate the gains with parallelism, metrics are necessary and they depend on some concepts such as latency and throughput.

The time a task takes to execute is designated as latency[1], being materialized in time units. The higher the latency, the longer a program takes to perform a particular task, and therefore a low latency is better than a higher one. On the other hand, throughput is materialized by units of work per unit of time, and aims to measure the set of tasks that can be executed at a given time. A higher throughput value is, therefore, better than a lower one [30].

Regarding metrics that help us to better understand the behavior and performance of parallel software development, speedup and efficiency are the most commonly used. Speedup is the measurement unit that has the objective of quantifying latency improvement. When applied to the same computational task, it relates the latency of the execution with one hardware unit (or worker), with the one related to P workers, materialized in the following expression: $speedup = S_P = \frac{T_1}{T_P}$, where $T_i, i \in \{1, P\}$ is the latency of a program executed with i workers [30].

On the other hand, efficiency is given by the expression $efficiency = \frac{S_P}{P}$, S_P being the value of speedup and P the total number of workers available. Efficiency aims at measuring the return on the hardware investment. As closer to 100%, as efficient our program is. If efficiency is 100%, we are in presence of linear speedup, meaning that our program makes full use of added workers/resources [30].

As mentioned above, linear speedup means that a given program has a latency P times lower with P processors, by comparison to the sequential version of the same computational program. Nevertheless, linear speedup is a very rare phenomenon, because the use of parallelism introduces the existence of overheads related to the creation and management of the tools used for it, such as threads, or other aspects related with libraries loading, and communications [30].

An important concept of speedup limit, called Amdahl's Law (see Figure 2.6 [30, p. 59]) has emerged in 1967. For Amdahl, the latency of a program constitutes the sum of the time spent in performing the sequential work, and the work performed in parallel: $T_1 = W_{ser} + W_{par}$, $T_p \geqslant W_{ser} + \frac{W_{par}}{P}$, where W_{ser} represents serial or sequential work, and W_{par} the work corresponding to parallel tasks. Thus, the imposed limit of T_P guarantees that there will be no superlinear speedup (efficiency higher than 100% if the parallelization of a sequential program is perfectly parallel) [30].

Conjugating all these concepts, we have the expression referring to the speedup according to the definitions introduced by Amdahl: $S_p \leqslant \frac{W_{ser} + W_{par}}{W_{ser} + W_{par}/P}$ [30].

The Amdahl's law demonstrates that the sequential component of an algorithm influences speedup. Thus, it must be ensured that these sequential executions consume the

[1] Latency means execution time and not the startup time involved in sending a message between two entities.

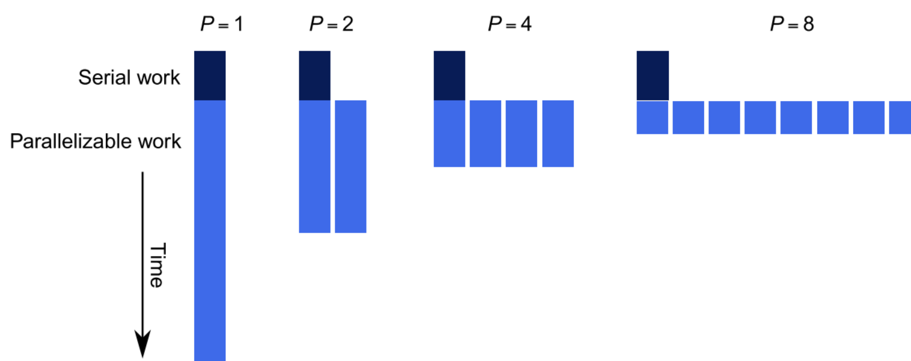

Figure 2.6: Amdahl's law.

lowest possible execution time. This aspect motivates the sequential optimization of GLODS, reported in Section 4.2.

The use of metrics is fundamental because it allows to obtain a method for comparison of solvers that compete for the same objective. These metrics, combined with the other performance assessment tools, referred in Subsection 4.1.1, namely data and performance profiles, will allow the comparison of results between different solvers, validating the optimization developed.

2.3 Alternatives for using parallel hardware

In the following subsections we discuss alternatives for running the GLODS parallel implementations over SMP.

2.3.1 Interactive use

The developer accesses a machine managed by some organization through a login procedure similar to the one used in a desktop system, which requires a login and a password. The login is performed over a network connection and is managed by the Secure Shell (SSH) cryptography network protocol. SSH supports an interactive command-line interpreter, allowing the input of commands in a local machine executed in the remote multiprocessor system. The multi-terminal *screen* supports multiple sessions and the execution of programs without being logged.

2.3.2 Batch Systems

The resources of large multiprocessors - processors, memory, input/output devices - are managed by *resource managers* [23] that support the exploitation of the machine by many users that run, at some time, many applications with distinct requirements,

Users submit *jobs* specified by *job scripts* that indicate the executable file to be run, the input and output files, and the resources (mainly the number of nodes) required to

run the application. The job is placed in a queue and will be executed when the required resources are available. The user has commands to monitor the job execution and to retrieve the results when it finishes. There is no interactivity in this type of system, and the turnaround time can be in the order of minutes or hours.

2.3.3 Cloud computing

According to the National Institute of Standarts and Technology (NIST) [33], cloud computing presents a model characterized by ubiquity, allowing for the access to a set of configurable computing resources, easily accessible, and that can be made available and launched almost instantly without much effort from the service provider.

The cloud model is identified by five essential characteristics and three different service models.

According to [33], the five essential characteristics, are:

- On-demand-service: A consumer should get only the resources required without human interaction with the service provider;
- Broad network access: The capacities and resources of cloud services should be made available over the standard network;
- Resource pooling: The provider must make its resources available in such a way that it can serve multiple consumers, with the objective that virtual and physical resources are allocated according to the consumer requirements;
- Rapid elasticity: The resources will have to be made available in a way that they can be quickly requested, or released, sometimes automatically, by a consumer;
- Measured service: The resources used must be measurable to have control, and establishing a relationship of transparency between consumer and provider.

Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) are the three service models related to cloud computing facilities [33]. According to [2, p.50], "the line between low-level infrastructure and a higher-level platform is not crisp". Although the definition of the service models is clear, there may be situations in which some services are located at the border of the models, or which belong to two models.

SaaS model consists of the ability given to the consumer to run applications in the cloud infrastructure, which is a collection of hardware and software resources aggregating the five characteristics mentioned above. The consumer cannot control resources (e.g. storage or operating system) with the limited possibility of changing specific settings. Dropbox and Gmail are examples of this model.

PaaS is a model allowing for a consumer to deploy their application in the cloud infrastructure, supported by a provider. The consumer is unable to control the resources

of the cloud infrastructure's underlying layer. Google App Engine and Heroku are examples of this model.

IaaS represents the model that gives a consumer the ability to provide fundamental computing resources. The consumer is unable to control the resources of the underlying layer of the cloud infrastructure, but can control the upper layers (e.g. Virtual Machines (VMs), operating systems, storage, firewalls). Amazon Web Services its an example of this model.

According to the needs of a consumer application, there are four deployment models that can be considered: private, community, public and hybrid cloud [33].

Private cloud is a deployment model where the cloud infrastructure is made available exclusively by a single organization, allowing access to multiple consumers. It can be managed by the provider that owns a specific application; a third provider can also manage it, or a combination of both.

In community cloud, the cloud infrastructure is provisioned for the exclusive use of a community of consumers who share interests (e.g. security requirements or policy). It should be managed by one or more community organizations, or a combination of these.

In a public cloud, the infrastructure is open for use by the general public. It is controlled by government, academic, business organizations, or a combination of these.

A hybrid cloud consists of a composition about two or more deployment models mentioned above: private, community or public. This composition allows them to appear to be unique entities.

2.3.4 Serverless computing

There is yet another emerging concept being considered by Microsoft as a fourth model, serverless computing [45].

In a serverless service, a consumer will only have to write a cloud function in a high-level language, schedule the event that will activate the execution of the function (e.g. change a column in a database, or upload a photo), and give to the provider the capacity to control and manage all resources (e.g. auto-scaling, auto instance selection) [24].

Therefore, a serverless service should scale automatically, without explicit provisioning, and only the execution time would be paid. Serverless computing is a combination of two models: Function as a Service (FaaS) and Backend as a Service (BaaS) [24].

FaaS, also called cloud functions, is the core of serverless computing. It constitutes the consumer's ability to produce a piece of code, and the cloud provider performs the remaining provisioning tasks and resources to execute it. BaaS represents the capacity of a service to respond to specific consumer applications [24].

There are three significant differences between serverless and serverfull computing, according to [24]:

- Decoupled computing and storage: Storage and computation are scaled and priced independently. The computation is stateless, and another cloud service provides

the storage;

- Executing code without managing resource allocation: Instead of a consumer requesting services or capabilities from a provider, he only has to produce one function, being the resources for the execution automatically provided by the cloud;
- Paying in proportion to resources used instead of for resources allocated: A new dimension appears related to the payment of services. Specifically, execution instead of quantity. A consumer will only pay for the execution time of his code, instead of paying for the quantity of resources (e.g. number of instances, or VMs).

Cloud computing offers the possibility of using resources, with the features and advantages presented above, which enhances the capabilities of parallel computing, allowing to obtain significant computational gains at a low and fair cost.

2.3.5 How parallel hardware was used in this thesis

The work described in this dissertation used two of the alternatives described above:

Interactive use SMP *Markov* belonging to Department of Mathematics of FCT NOVA (see hardware characteristics in Subsection 4.1.2);

Cloud computing SMP *Stratus* belonging to the National Distributed Computing Infrastructure (INCD) (see hardware characteristics in Subsection 4.1.2). Stratus is managed by the OpenStack Cloud Management System [23].

Both parallel configurations allowed an interactive interaction, thus accelerating the development process of the GLODS parallel implementations.

2.4 Application of parallelism in Directional Direct Search

Parallelism was already used as an effective tool to improve the numerical performance of some DDS algorithms. The first mention to the use of parallel implementations in Direct Search appears in [16], where the cost of synchronization is said to be minimal and the speed-up is showed to be almost linear with the addition of more processors.

An asynchronous parallel implementation of a DDS method was proposed in [20], namely Asynchronous Parallel Pattern Search (APPS). Instead of a methodical and continuous execution, the algorithm initiates actions in response to events, allowing an effective balance of the computational load across all available processors.

Decomposition techniques were also used in parallel approaches, like is the case of Parallel Space Decomposition-Mesh Adaptive Direct Search (PSD-MADS) algorithm [5]. Problems are transformed into smaller sub-problems, relative to the number of variables, and solved in an asynchronous way, using parallelism. This technique allowed to solve problems up to 500 variables, which is clearly large-scale DFO.

The PSwarm algorithm [44], already mentioned in Section 2.1, also presents a natural parallel version. Function evaluations are distributed to the different processors, both at the search and the poll steps.

3

GLODS Algorithm

This chapter aims to provide a detailed description of GLODS algorithm. Sections 3.1 and 3.2 analyze its sequential implementation, detailing it and highlighting its distinguishing features. Section 3.3 identifies the main parts of the algorithmic structure to be parallelized and the methodology planned to accomplish it.

3.1 GLODS general framework

GLODS is a global DDS search method, used in global optimization of derivative-free problems. This algorithm aims at computing all local minima of a given function (assuming a finite number for it), which would allow the identification of the global minimum [13]. As in any other DDS method, the algorithmic structure of GLODS consists of two main processes, the search step and the poll step, which will be detailed in Subsections 3.1.2 and 3.1.3, respectively.

During the optimization process, a list, L_k, is maintained, which keeps a record of previously evaluated points, and from which poll centers are selected. Points generated both in the search (S_k) and poll (P_k) steps can be added to the list L_k. These points are stored in tuples, $(x;\alpha;r;i)$, where: x is the new point to be stored; α represents the corresponding step size parameter; r is a comparison radius; and i denotes a index which refers to whether the point is active or inactive, presenting a value of 1 or 0, respectively. A point is considered active when it has the best objective function value in its proximity. This proximity criterion will be defined by the comparison radius, r. Only active points will be selected as poll centers, and a point can change its status from active to inactive but never the reverse [13].

3.1.1 Algorithm initialization

The initialization step of the algorithm respects to the selection of initial values for the different algorithmic parameters: step size, α_0; the set of positive spanning sets, $\mathcal{D}$, used through the iterations, which should have bounded directions; and r_0, the initial comparison radius. This last one should be large enough to always allow the comparison

between the poll center and the poll points ($r_0 \geq \alpha_0 d_{max}$, where d_{max} is the maximum norm of a poll direction).

3.1.2 Search step

The search step is responsible for identifying promising regions in terms of objective function value, that have potential to be explored, by sampling and evaluating sets of points. This exploration is carried out later by the poll step, that guarantees the convergence of the method. For the execution of the search step, GLODS is versatile, allowing the use of different sampling techniques, such as random sampling [39], Latin hypercube sampling [31], Sobol or Halton sequences [27] (see Figure 3.1). In [13], the authors proposed a new strategy for the sampling procedure to be conducted at search step of GLODS, namely the 2^n-Centers strategy (see Figure 3.2). However, for large problems, it was not efficient. So, the default of the current version of GLODS is the use of Sobol sequences [27].

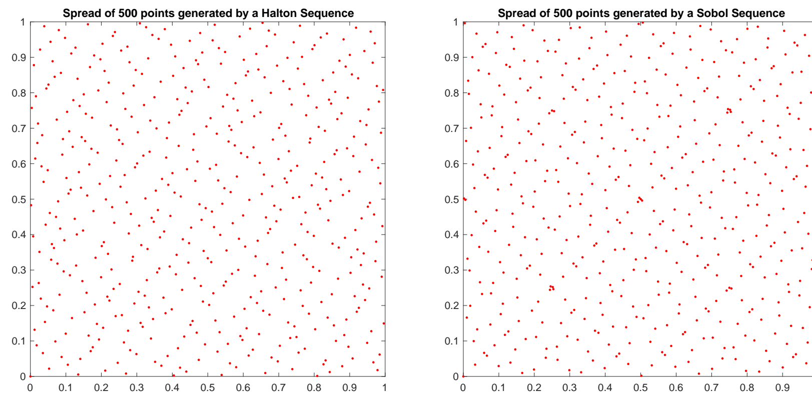

Figure 3.1: Spread of 500 points generated by a Halton Sequence (left) and a Sobol Sequence (right).

The points evaluated at the search step can be added to the list as active or inactive points. Since this is a distinguishing feature of GLODS algorithm, we will detail it in Section 3.2.

3.1.3 Poll step

The poll step starts by sorting all points that are active in the list, L_k, and selects an active point $(x;\alpha_x;r_x;1)$ as a new poll center. In the current implementation of GLODS, the one corresponding to the lowest objective function value. Then, a local exploration will take place around this poll center, by testing directions belonging to a positive spanning set [14]. A set of poll points $P_k = \{(x_k+\alpha_k d;\alpha_k;\alpha_k\|d_k\|;0) : d \in D_k \wedge f_\Omega(x_k+\alpha_k d) < +\infty\}$, will be computed, considering the poll directions scaled by the step size parameter, α_k. If there is a necessity to add new points from P_k to L_k, the procedure $L_{k+1} = \text{add}\ (L_k, P_k)$, detailed in Section 3.2, is executed.

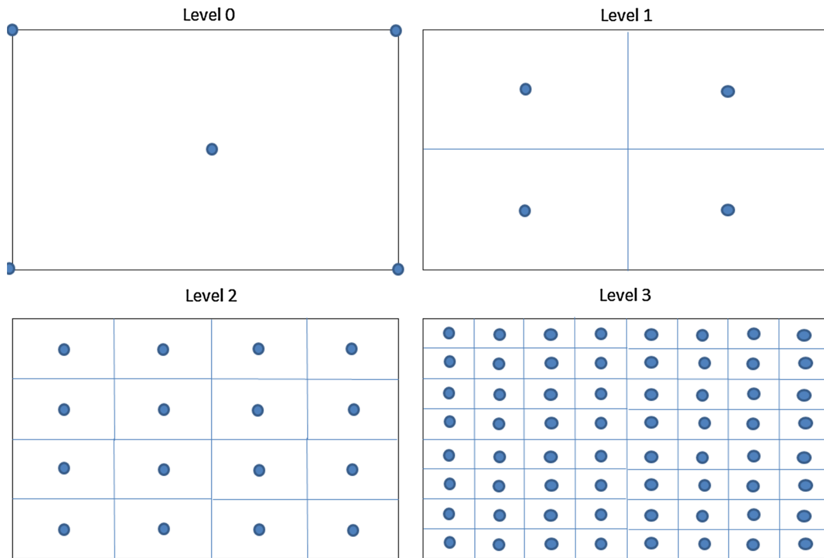

Figure 3.2: Four levels of the 2^n-Centers strategy.

In the polling procedure, two strategies can be adopted: complete or opportunistic [13]. Opportunistic polling means that not all poll points need to be evaluated. The polling procedure is interrupted once that a success is found, meaning that a new active point was added to the list L_k. In the complete strategy, all poll points are evaluated and, if adequate, added to the list L_k.

3.2 GLODS differentiation from others DDS methods

A distinguishing feature of GLODS, when compared with others DDS methods is the list of points. Algorithm 2 details the procedure for adding the set of points L_2 to L_1.

A point is added to the list as active under one of the two conditions:

- when the distance to any point present in the list is greater than the comparison radius, meaning that the new point belongs to a region that has not yet been explored;
- when the new point, x_{new}, is comparable to at least some active point, y, in the list, $\|x_{new} - y\| \leq r_y$, presenting a better objective function value ($f(x_{new}) < f(y)$), and no other point, comparable with x_{new}, presents a better function value than $f(x_{new})$.

If two points are comparable, the one that presents the worst function value will be inactive. This means that inactive points can also be added to the list, when they are better than an active point in list but there is another point, comparable with them, that presents a better function value. An active point in the list can change its status to inactive, but never the opposite.

Whenever a point is added to the list, the algorithm defines its step size parameter and comparison radius. If the point is in a part of the feasible region not yet explored, meaning that it is at a distance greater than the comparison radius of any point in the list, the parameters (step size and comparison radius) to be considered are equal to the algorithm initialization. Otherwise, the new point inherits the parameters of the point with largest step size to which it was comparable, and for which it presented a better function value. This comparison strategy distinguishes GLODS from DDS simply coupled with multistart [13].

In GLODS, if two distinct points are within a distance less or equal than the comparison radius, only one of them will remain active, thus allowing merging searches resulting from different initializations when they are relatively close [13].

Iterations are then classified as: successful, when at least one new active point is added to the list; unsuccessful if no points are added to the list; and merging if only inactive points have been added to the list [13].

The step size is reduced at unsuccessful iterations, is kept constant at merging iterations and is also kept constant or can increase at successful iterations. Algorithm 3 [13] details the structure of GLODS.

3.3 Potential for parallelization

The analysis of GLODS, in Sections 3.1 and 3.2, allowed the identification of procedures that have the potential to be optimized:

- Search step: The search step corresponds to the evaluation of the objective function at a sample set, that will provide new DS initializations. The evaluation procedure is performed sequentially;

- Poll step: The evaluation of the objective function is performed sequentially at the poll points;

- Poll step: At each iteration several active points could be available in the list, but only one is selected as poll center.

For the first two items, it is clear that the capabilities of the hardware are not being fully explored, specifically the total number of existing processors. Therefore, these two procedures can be defined as the first level of optimization, allowing to distribute function evaluations among different processors.

The third item constitutes the second level of optimization. In this case, selecting more than one poll center at each iteration and performing the polling procedure using parallelization can allow gains not only in computational time but also in the quality of the solution computed. Strategies need to be defined for the selection of the active points to be considered as poll centers.

Algorithm 2: $L_1 = \text{add}(L_1, L_2)$

Procedure for adding points in L_2 to L_1.

1 **forall** $(x;\alpha_x;r_x;0) \in L_2$ **do**
3   **if** $\min_{y \in L_1}(\|x-y\| - r_y) > 0$ **then**
5         $L_1 = L_1 \cup \{(x;\alpha_0;r_0;1)\}$
6   **else**
7   **if** $x \notin L_1$ **then**
8   set $\alpha_a = 0$, $r_a = 0$, $i_{dom} = 0$, $p_{dom} = 0$ and $i_{comp} = 0$
9   **forall** $(y;\alpha_y;r_y;i_y) \in L_1$ **do**
11   **if** $\|x-y\| - r_y \leq 0$ **then**
13   **if** $f(x) < f(y)$ **then**
14                         $i_{comp} = 1$
15                         $i_{dom} = i_{dom} + i_y$
16                         $i_y = 0$
17   **if** $\alpha_y > \alpha_a$ **then**
18                             $\alpha_a = \alpha_y$
19                             $r_a = r_y$
20   **end**
21   **else**
23   **if** $f(y) \leq f(x)$ **then**
24                             $p_{dom} = 1$
25   **end**
26   **end**
27   **end**
28   **end**
29   **if** $p_{dom} = 0$ **then**
30                 $i_x = 1$
31   **end**
33   **if** $i_{dom} > 0 \vee (p_{dom} = 0 \wedge i_{comp} = 1)$ **then**
34   **if** $\alpha_x = 0$ **then**
36                     $L_1 = L_1 \cup \{(x;\alpha_a;r_a;i_x)\}$
37   **else**
39                     $L_1 = L_1 \cup \{(x;\alpha_x;r_x;i_x)\}$
40   **end**
41   **end**
42   **end**
43   **end**
44 **end**

Algorithm 3: L_1= add(L_1,L_2)
Procedure for adding points in L_2 to L_1.

Initialization

Let $\mathcal{D}$ be a (possibly infinite) set of positive spanning sets, such that $\forall d \in D \in \mathcal{D}, 0 < d_{min} \leq \|d\| \leq d_{max}$. Choose $r_0 \geq d_{max}\alpha_0 > 0$, $0 < \beta_1 \leq \beta_2 < 1$, and $\gamma \geq 1$. Set $L_0 = \emptyset$.

For $k = 0, 1, 2, \ldots$

1. **Search step:** Compute a finite set of distinct points $A_k = \{(x_j; 0; 0; 0) : f_\Omega(x_j) < +\infty\}$. Call L_k = add(L_k,A_k) to possibly add some new points in A_k to L_k. If $k = 0$, go to the poll step. Otherwise, if there is a new active point in L_k then set $L_{k+1} = L_k$, declare the iteration (and the search step) as successful and skip the poll step.

2. **Poll step:** Order the list L_k and select an active point $(x; \alpha_x; r_x; 1) \in L_k$ as the current iterate, corresponding step size parameter, and comparison radius (thus setting $(x_k; \alpha_k; r_k; i_k) = (x; \alpha_x; r_x; 1)$).

   Choose a positive spanning set D_k from the set $\mathcal{D}$. Compute the set of poll points $P_k = \{(x_k + \alpha_k d; \alpha_k; \alpha_k \|d_k\|; 0) : d \in D_k \wedge f_\Omega(x_k + \alpha_k d) < +\infty\}$. Call L_{k+1} = add(L_k,P_k) to possibly add some new points in P_k to L_k. If there is a new active point in L_{k+1} declare the iteration (and the poll step) as successful. If no new point was added to L_k declare the iteration (and the poll step) as unsuccessful. Otherwise declare the iteration (and the poll step) as merging.

3. **Step size parameter and radius update:** If the iteration was successful then maintain or increase the corresponding step size parameters: $\alpha_{new} \in [\alpha, \gamma\alpha]$ and replace all the new active points $(x; \alpha_x; r_x; 1)$ in L_{k+1} by $(x; \alpha_{new}; d_{max}\alpha_{new}; 1)$, if $d_{max}\alpha_{new} > r_x$, or by $(x; \alpha_{new}; r_x; 1)$, when $d_{max}\alpha_{new} \leq r_x$.
   If the iteration was unsuccessful then decrease the corresponding step size parameter: $\alpha_{new} \in [\beta_1\alpha_k, \beta_2\alpha_k]$ and replace the poll point $(x_k; \alpha_k; r_k; 1)$ in L_{k+1} by $(x_k; \alpha_{new}; r_k; 1)$.

4

GLODS Parallelization

This chapter covers the GLODS parallelization process. Section 4.1 presents all the necessary concepts and tools used to assess the validity of the code defaults and the results of parallelization. Section 4.2 presents the results related to the calibration of some parameters in the sequential version of GLODS. Sections 4.3, 4.4, and 4.5 respect to the results obtained with GLODS parallelization, corresponding to three different approaches.

4.1 Benchmarking

Benchmarking is an important phase of software development because it allows the validation and assessment of performance of a given solver by comparison with other solvers suited for the same purpose. It can be also used to evaluate the value of different algorithmic strategies, here perceived as corresponding to different solvers. Benchmarking is only possible with the use of metrics that enable quantitative measures of performance. In this section, all the tools used for benchmarking will be presented, specifically, data and performance profiles, detailing the different metrics particularly used in the later, and the characteristics of the set of problems and hardware used in test execution.

4.1.1 Data Profiles and Performance Profiles

Data profiles are a benchmarking tool specifically developed for DFO [32], that allows to visualize the percentage of problems solved by a given algorithm inside a fixed computational budget. The computational budget is expressed in sets of n_p+1 functions evaluations, avoiding the dependence on the problem dimension n_p. Let us define $h_{p,s}$ as the total number of function evaluations required by an algorithm $s \in S$ to solve a problem $p \in P$, within a certain accuracy. The cumulative expression of the data profile is given by $d_s(\sigma) = \frac{1}{|P|} \left| \{p \in P : \frac{h_{p,s}}{n_p+1} \leq \sigma\} \right|$. A problem is considered solved to a given level of accuracy φ when the decrease obtained relatively to the initialization is at least $(1-\varphi)$ of the decrease obtained by the best solver on the same problem: $f(x_0) - f(x) \geq (1-\varphi)(f(x_0) - f_{best})$. In all

data profiles reported, an accuracy level of 10^{-5} has been considered, with the exception of the data profile reported in Figure 4.10, where the accuracy level was 10^{-3}.

Since data profiles only take into account functions evaluations, they are not adequate for assessing the value of parallelization strategies, where computational time is a key indicator. Thus, we introduce performance profiles as another benchmarking tool [17]. In this case, performance is evaluated through the ratio $r_{p,s} = \frac{t_{p,s}}{min\{t_{p,s}:s \in S\}}$, where S is the set of solvers, $p \in P$ is a given problem in the collection, and $t_{p,s}$ is the value of a given performance measure obtained by solver s on problem p. It is assumed that low values of the performance measure represents a better performance and this values are always positive.

To assess the value of the parallelization schemes, the following performance measures were considered:

- Computational time;
- The best objective function value computed;
- The total number of local minima identified;
- The total number of function evaluations required until the global minimum is identified.

Speedup and efficiency (see Subsection 2.2.5) were also computed, but not combined with performance profiles.

Since the final objective function value can take null or negative values, we used the approach proposed in [44] computing $t_{p,s} = \frac{\overline{f}_{p,s} - f_p^*}{f_{p,w} - f_p^*}$. In the case of the number of local minimizers identified, a high value represents a better performance, and not the opposite. So, to compute performance profiles, we consider the numerical inverse of it.

Additionally, some of these metrics, like is the case of computational time, are not deterministic. Thus, everytime that a metric has a nondeterministic behavior for one of the solvers compared, a total of 30 runs are performed and 95% confidence intervals are computed for the average value of the metric, using the expression $\overline{X} \pm \frac{\sigma \times 1.96}{\sqrt{n}}$. Performance profiles are then reported not only for the average value of the metric, but also for the lower and upper bounds of the confidence interval.

Three different variants of the algorithm were compared: the sequential version of the existing GLODS; an unordered parallel version, where the order of reception of the points evaluated in parallel is not respected; and an ordered parallel version, which respects the order of reception of the evaluated points, as in the sequential version.

As test set, we considered 61 bound constrained minimization problems, collected from the global optimization literature, reported in Tables A.1 and A.2, with dimension between $n = 2$ and $n = 10$.

4.1.2 Hardware used for benchmarking

The different versions of the code were run in two different systems:

- For the calibration of the sequential algorithm (see Subsection 4.2), a system belonging to the Centre of Mathematics and Applications (CMA) of FCT NOVA, denoted by Markov, was used, with the following characteristics: 2 Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz (20 cores, 40 threads total), 200GB of RAM, and Linux operating system;
- The remaining tests (reported in sections Subsections 2.2.4, 4.1.3, 4.3, 4.4 and 4.5) were performed on a cloud computing system, namely the INCD Stratus, with the following characteristics: 2x AMD EPYC 7501 @2.6GHz (64 cores), 64GB of RAM, and Linux operating system.

Markov is a system shared by other users, not allowing a reliable evaluation of computational time. However, it could be used in the initial phase of calibration, which only intended to validate the defaults of the code, being focused in the quality of the solution computed and not in the computational time.

As we will see later, the ideal number of workers to solve a problem of dimension n is $2*n$. However, this would require resetting the parallel poll of workers in PCT very often, causing additional delays. The average size of the problems belonging to the collection is $n = 5$, and 12 was chosen as reference value for the number of workers to be used, which is also the default of PCT. Exceptions occur in the numerical results reported in Tables 4.1, 4.2, 4.3 and Section 4.5, where the number of workers considered is specified.

4.1.3 Testing PCT

A set of tests was conducted to complement the one reported in Subsection 2.2.4. Again, the goal is to assess if the procedure to estimate computational time, by counting the number of parallel cycles and including artificial time delays, is adequate.

A subset of 14 problems, which allow different dimensions, was selected from the test set.

The problems were solved twice, considering 16 workers. In one of the times, computational time was estimated using the number of parallel cycles, as described is Subsection 2.2.5, considering a time delay of 2 seconds for function evaluation. In the second time, the delay was directly included in the code, at the time of function evaluation, and the real computational time is measured (no estimation is performed). The similarity between the two approaches, evidenced in last column of Table 4.1, validates the adopted procedure for estimation of the computational time.

To determine if the ideal number of workers needed to solve a problem changes with dimension, the 14 problems were ran for different dimensions ($n = 5, 10, 20, 40, 80$), considering different numbers of workers. Tables 4.2 and 4.3 report average results from

Table 4.1: Real computational time vs estimated time.

Problem size	Real time (sec.)	Estimated time (sec.)	Difference
n=5	16551,19	16562,24	0%
n=10	35064,81	35299,46	1%
n=20	44643,56	43664,95	2%
n=40	39328,08	38477,08	2%
n=80	37495,40	37543,44	0%

30 runs, where computational time was estimated again considering a delay of 2 seconds for function evaluation.

We can conclude that the ideal number of workers to solve a given problem is $2 * n$. For dimensions n= 5, n=10, and n=20, the gains in computational time were explicitly obtained from 10 workers, 20 workers and 40 workers, respectively.

Table 4.2: Computational times (sec.) required for solving problems of different dimensions (4 workers to 19 workers).

Size	4w	8w	9w	10w	11w	16w	19w
n=5	37525,55	25868,75	25488,29	16860,45	16850,71	16562,24	16851,98
n=10	83660,65	51213,60	50328,12	37373,95	36943,01	35299,46	34668,60
n=20	136729,08	71255,64	69369,93	57985,16	56833,37	43664,96	41915,99
n=40	143233,94	73457,91	66211,38	59536,48	58092,76	38477,08	37200,50
n=80	142535,42	72518,46	65204,40	58493,20	54450,31	37545,44	33306,18

Table 4.3: Computational times (sec.) required for solving problems of different dimensions (20 workers to 64 workers).

Size	20w	21w	32w	39w	40w	41w	64w
n=5	16861,67	16836,08	16874,53	16858,39	16870,73	16840,33	16854,90
n=10	22985,04	23000,18	22996,70	22983,95	22999,55	23005,17	22993,85
n=20	31824,18	31514,11	29637,54	28585,20	19010,59	18995,54	19014,31
n=40	31602,78	31202,20	23855,08	22987,29	17735,12	17516,32	16466,49
n=80	30558,70	29982,12	19916,75	19017,91	16589,06	16332,26	12440,18

4.2 Calibration

Before initializing the parallelization work, it was important to establish the best version of the sequential algorithm, given all the options available for the solver to be executed. Data profiles were used to analyze the results of this calibration.

The search step, which is responsible for initializing new direct searches, intends to explore the feasible region, not being executed at all iterations. GLODS allows to choose the frequency criterion for the execution of the search step: if it is based on the frequency

of consecutive unsuccessful iterations; or if the execution depends on the number of current active points not yet identified as local minimizers. A point is identified as local minimizer when it remains active and the corresponding step size parameter is below 10^{-8}.

Regarding the first criterion, the cases of new initializations were considered after 1,2 or 3 consecutive unsuccessful iterations. Figure 4.1 reports the corresponding data profiles.

In this case, the results are very close, with a slight advantage of the version that initializes new searches after every unsuccessful iteration, for larger budgets of function evaluations. However, this algorithmic variant is the one that less explores the algorithmic structure of the poll step.

Regarding the second criterion, the number of current active points in the list not yet identified as local minimizers, new initializations were considered when only 1, 2, or 3 active points remain in the list. Figure 4.2 reports the corresponding results, which are again very similar, with a slight advantage of the variants that considered 2 or 3 active points remaining in the list, before considering new initializations. The version with 2 active points was selected, reducing the number of initializations, potentiating the exploration of new points through poll step.

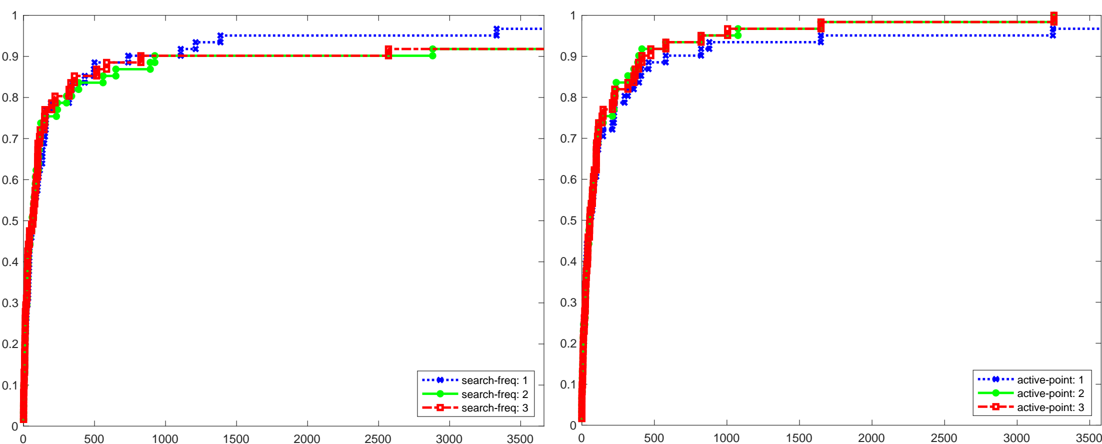

Figure 4.1: Data profile for the frequency of unsuccessful iterations considered for initializing new searches.

Figure 4.2: Data profile when initialization is based on the number of active points in the list, not yet identified as local minimizers.

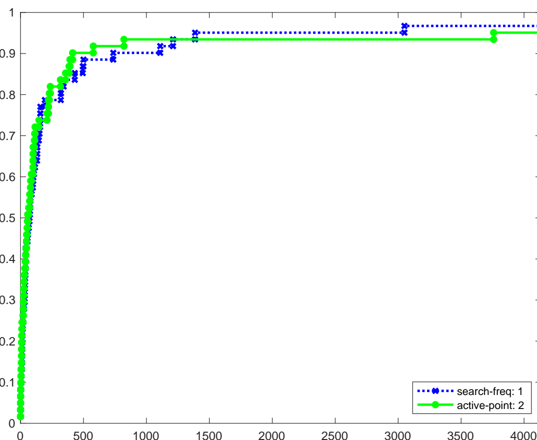

Figure 4.3: Data profile comparing new initializations based on the frequency of unsuccessful iterations and on the number of active points in the list.

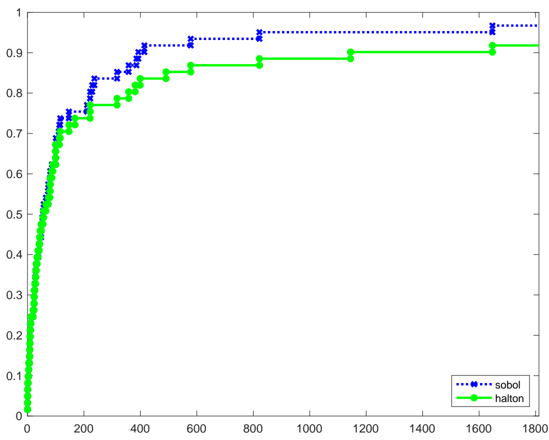

Figure 4.4: Data profile comparing two different strategies for generating new points in the search step.

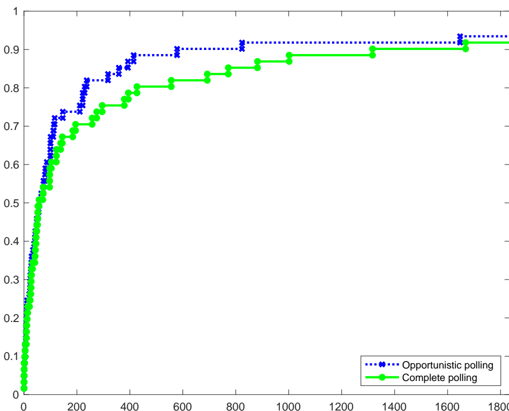

Figure 4.5: Data profile comparing opportunistic and complete polling.

Figure 4.3 compares the best versions for launching new initializations based on the frequency of unsuccessful iterations and based on the number of active points in the list. Again, the results are very similar. The version that launches new searches based on the number of active points in the list was selected, since it presents a better performance for smaller budgets of functions evaluations and allows a better exploration of the new points generated.

Having chosen the strategy for the frequency related to the execution of the search step, the strategies considered for generating new points were compared, namely Sobol and Halton sequences. Sobol sequences have a clear advantage over Halton sequences (see Figure 4.4).

With the two previous choices, related to the search step, namely perform new initializations with Sobol sequences every time that 2 active points, not yet identified as local

minimizers, remain in the list, the behavior of the algorithm was tested considering opportunistic or complete polling strategies. Figure 4.5 reports the results obtained, where the advantage of opportunistic polling is clear. This is considered the default version of the code, over which parallel versions will be developed.

4.3 First level of parallelization

The analysis of GLODS in Chapter 3 allowed the identification of procedures that have the potential to be parallelized, namely, at initialization, search step and poll step, where the evaluation of the objective function is performed sequentially. The capabilities of the hardware are not fully explored, more precisely the total number of existing processors. Therefore, allowing the distribution of function evaluations among the different processors will be considered the first parallelization level.

4.3.1 Parallelization Structures

Algorithms 4 and 5 correspond to the pseudocode concerning the parallelization strategies implemented in the initialization, search and poll steps.

The difference between the two algorithms is the possibility of interrupting the parallel loop, if a success is found at the poll step. It is always unnecessary to interrupt the parallel loop at the initialization and at the search step, since all points will be evaluated. However, in the poll step (see Algorithm 5), the opportunistic strategy stops function evaluation when a new active point is added to the list, as shown in lines 26-29. The algorithm interrupts the pending function evaluations and continues for a new iteration. Different constructs from MATLAB PCT were used in the implementation, which will be detailed in the following subsection.

Algorithm 4: Pseudocode: Parallel approach implemented in phase one at the initialization and the search step.

```
   Data: list (points to evaluate), cache, problem
 1 forall point ∈ list in parallel do
 2 |  feasible ← check_feasible(point)
 3 |  if feasible then
 4 |  |  if cache then
 5 |  |  |  match,ftemp ← search_cache(point)
 6 |  |  end
 7 |  |  if ¬ match then
 8 |  |  |  ftemp ← eval_point(problem, point)
 9 |  |  end
10 |  end
11 end
   // Update lists, counters, cache if available and stopping criteria
```

Algorithm 5: Pseudocode: Parallel approach implemented in phase one at the poll step.

```
   Data: list (points to evaluate), cache, problem
 1 futures_list ← {}
 2 forall point ∈ list in parallel do
 3     match ← 0
 4     feasible ← check_feasible(point)
 5     if cache then
 6         match,ftemp ← search_cache(point)
 7     else
 8         if feasible then
 9             ftemp = eval_point(problem, point)
10         end
11     end
12     futures_list ← futures_list ∪ {ftemp,match,point,feasible}
13 end
14 for k ← 1 to list.lenght do
15     index, point, match, ftemp, feasible ← futures_list(k)
16     finite ← is_finite(ftemp)
17     while index = k do
18         if feasible then
19             if ¬match ∧ cache then
20                 update_cache(point,ftemp)
21             end
22             if finite then
23                 success ← add_point(ftemp)
24             end
25         end
26         if success then
27             cancel(futures_list)
28             break;
29         end
30     end
31 end
```

4.3.2 Implementation using PCT

Both at the algorithm initialization and the search step, all points will be evaluated, and it is not necessary to interrupt the parallel loop. All function evaluations verify if there is a cache hit to avoid the realization of unnecessary computations. Thus, the parallel

implementation considered `parfor` (line 2 of Figure 4.6).

```
1 % For each poin check: cache hit if it feasible
2 parfor i = 1: pointsToEvaluate
3     %calculate X_initial
4     if feasible
5         if cache
6             if ~isempty(cache)
7                 [match,~,ftemp] = match_point(x_initial);
8                 %save ftemp to array
9             end
10        end
11        if ~match
12            ftemp = feval(func_f, x_initial);
13            %save ftemp to array
14        end
15    end
16 end
17 % Update lists, counters and cache if available
```

Figure 4.6: Code snippet with the parallelization implemented at the algorithm initialization and at the search step.

```
1 % For each point check: cache hit and if it is feasible
2 % Inital loop to launch function evaluations in parallel
3 for i=1:pointsToEval
4     F(i) = parfeval(@eval_point,3,xtemp,cache,func_f);
5 end
6 % Loop to collect functions evaluations launched before
7 for ii = 1:pointsToEval
8     [completedIdx, match,xtemp,ftemp] = fetchNext(F);
9     while completedIdx == (lastIndex)
10        % While loop to guarantee the evaluation order while completedIdx == lastIndex
11        if feasible
12            if isFinite
13                [success] = merge(ftemp...)
14                    if cache
15                    % update cache
16                    end
17            end
18            if success
19                endcycle = 1;
20                break;
21            end
22            % check if next point is already ready to evaluate
23        end
24    % Break cycle if success and abort pending computations
25    end
26    if endcycle
27        cancel(F);
28        break;
29    end
30 end
```

Figure 4.7: Code snippet with the parallelization implemented at the poll step.

Regarding the poll step, to guarantee the possibility of interrupting the parallel loop, in order to implement the opportunistic strategy, `parfeval` was used. The use of `parfeval` demands a strategy distinct from the one used in `parfor`. Instead of a single loop, one needs two loops: an initial loop (line 3 of Figure 4.7) for launching the points evaluations and a collecting loop (line 7 of Figure 4.7) to receive all computed evaluations. Another difference is that `parfeval` uses Future objects, which are functions

to be executed on the parallel pool of workers. When a specific worker ends a job, it delivers the FevalFuture object to the collection loop. In addition, the FevalFuture object also returns an index value that represents the position of the concluded task in the initial queue of jobs to be performed, necessary to implement the ordered versions.

In these ordered versions, the parallel evaluation of the function respects the order of the sequential implementation (see line 9 of Figure 4.7). The unordered versions do not respect this order, causing a random behavior in the algorithm.

In line 27 of Figure 4.7 all pending computations are aborted. At this time, the computations have the follow state: concluded; not started; in course, already completed but not processed. This described strategy – ordered, opportunistic and with aborting of pending computations – will be used in the following parallelization phases.

4.3.3 Numerical results

As previously mentioned, opportunistic polling was used due to its better performance over the complete polling strategy in the calibration phase (see Figure 4.5).

Analyzing the data profile of Figure 4.8, the difference between the ordered and unordered versions is slight, with some advantage of the former. Notice that the sequential and the ordered versions curves are overlapping, confirming the determinism between both version.

Figures 4.9 and 4.10 include data profiles corresponding to the lower and upper limits of the confidence interval corresponding to the average results obtained with the unordered version for to different accuracy levels ($\varphi = 10^{-5}$ and $\varphi = 10^{-3}$, respectively).

Regarding computational time, a function evaluation delay of 0.04 second, already brings some advantage for the ordered versions, according to the performance profiles of Figures 4.11 - 4.14.

For the remaining metrics (number of local minima identified, total number of functions evaluations before identifying the global minimum and best objective function value found), all versions present a similar performance.

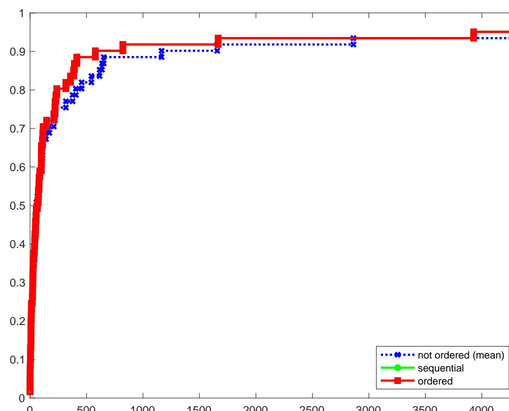

Figure 4.8: Data profile comparing versions: sequential, parallel ordered and parallel unordered.

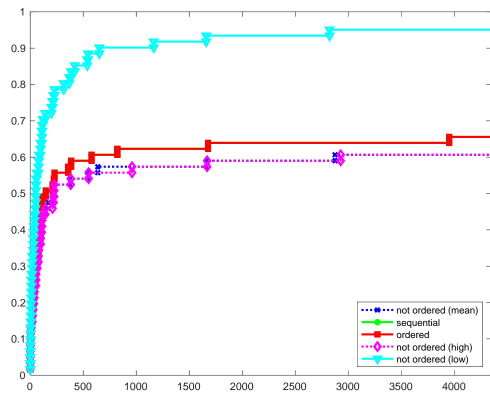

Figure 4.9: Data profile comparing versions: sequential, parallel ordered and unordered (mean, low/high interval).

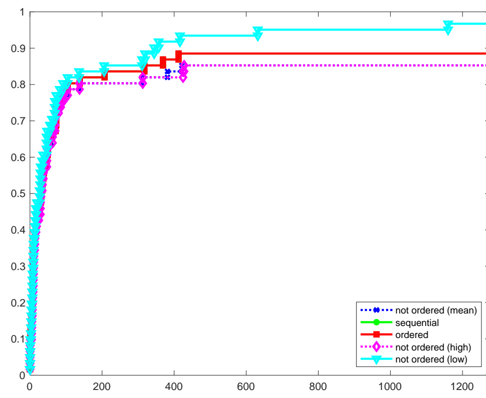

Figure 4.10: Data profile comparing versions: sequential, parallel ordered and unordered (mean, low/high confidence interval), now with level of accuracy $\varphi = 10^{-3}$.

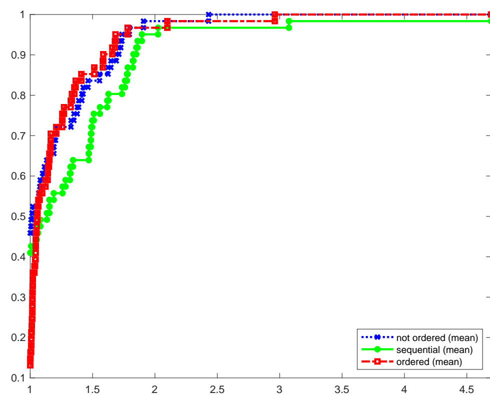

Figure 4.11: Performance profile comparing computational time, with 0,03 seconds delay for function evaluation (average results).

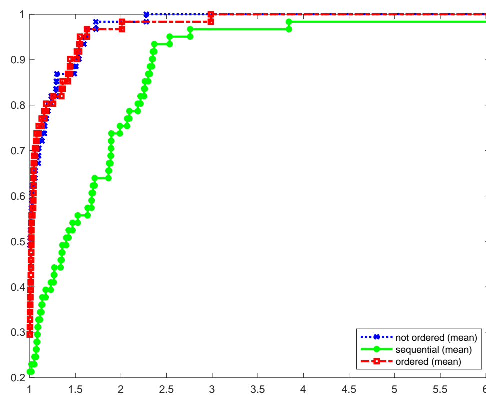

Figure 4.12: Performance profile comparing computational time, with 0,04 seconds delay, for function evaluation (average results).

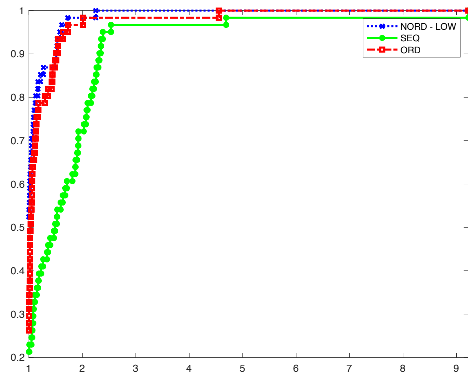

Figure 4.13: Performance profile comparing computational time, with 0,04 seconds delay for function evaluation (low extreme of average confidence interval).

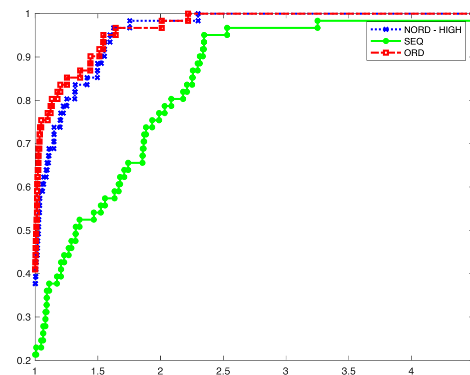

Figure 4.14: Performance profile comparing computational time, with 0,04 seconds delay for function evaluations (high extreme of confidence interval).

Table 4.4 reports the results regarding speedup and efficiency, ordered by problem dimension, comparing the sequential version and the best version of this phase. Two efficiencies are represented: one considering 12 workers (EFF 12w) and another considering $2n$ workers (EFF 2*N w). The low values of speedup and efficiency are justified by the small artificial delay considered for function evaluation (0,04 seconds), which makes the parallelization overhead relevant when compared to the function calculation time.

Table 4.4: Speedup and Efficiency for phase 1.

#	Dimension	Sequential (sec.)	1st Phase (sec.)	Speedup Sequential/1st Phase	EFF 12w	EFF 2*N w
2	2	7,55	6,66	1,133	9%	28%
3	2	10,50	10,16	1,033	9%	26%
4	2	6,65	6,80	0,978	8%	24%
5	2	12,26	10,25	1,196	10%	30%
8	2	3,18	5,60	0,567	5%	14%
10	2	11,88	10,83	1,097	9%	27%
13	2	14,69	18,93	0,776	6%	19%
15	2	6,36	4,48	1,421	12%	36%
20	2	14,50	17,49	0,829	7%	21%
26	2	9,12	11,00	0,829	7%	21%
29	2	13,35	12,64	1,056	9%	26%
31	2	11,69	15,50	0,754	6%	19%
35	2	55,60	50,49	1,101	9%	28%
36	2	11,82	9,09	1,300	11%	33%
39	2	610,29	463,89	1,316	11%	33%
42	2	176,41	121,69	1,450	12%	36%
43	2	357,91	293,97	1,218	10%	30%
50	2	45,35	40,21	1,128	9%	28%
52	2	18,22	25,75	0,707	6%	18%
55	2	5,71	4,16	1,373	11%	34%
59	2	6,00	4,23	1,420	12%	35%
23	3	38,26	36,85	1,038	9%	17%
24	3	10,55	11,76	0,897	7%	15%
53	3	20,33	12,94	1,571	13%	26%
6	4	21,80	19,87	1,097	9%	14%
9	4	6,10	6,10	0,999	8%	12%
14	4	31,08	27,36	1,136	9%	14%
16	4	12,42	7,79	1,595	13%	20%
27	4	454,06	418,12	1,086	9%	14%
30	4	607,16	365,05	1,663	14%	21%
32	4	21,76	22,81	0,954	8%	12%
37	4	605,46	320,05	1,892	16%	24%
45	4	97,06	52,78	1,839	15%	23%
46	4	114,45	62,03	1,845	15%	23%
47	4	120,26	65,13	1,846	15%	23%
56	4	10,58	6,23	1,698	14%	21%
61	4	605,63	335,88	1,803	15%	23%
11	5	84,11	62,22	1,352	11%	14%
22	5	60,23	41,82	1,440	12%	14%
40	5	113,92	71,93	1,584	13%	16%
48	5	122,49	59,33	2,065	17%	21%
17	6	61,62	29,54	2,086	17%	17%
25	6	60,16	46,34	1,298	11%	11%
57	6	17,49	9,47	1,846	15%	15%
18	8	106,86	49,23	2,171	18%	18%
58	8	139,77	63,52	2,200	18%	18%
54	9	609,95	267,06	2,284	19%	19%
60	9	603,58	295,86	2,040	17%	17%
1	10	40,91	21,13	1,936	16%	16%
7	10	260,99	155,78	1,675	14%	14%
12	10	605,07	263,09	2,300	19%	19%
19	10	121,23	52,86	2,293	19%	19%
21	10	180,55	109,43	1,650	14%	14%
28	10	604,48	202,34	2,987	25%	25%
33	10	604,48	260,84	2,317	19%	19%
34	10	27,01	20,28	1,332	11%	11%
38	10	162,98	71,84	2,269	19%	19%
41	10	427,96	189,29	2,261	19%	19%
44	10	603,48	240,01	2,514	21%	21%
49	10	144,67	71,81	2,015	17%	17%
51	10	604,85	258,44	2,340	20%	20%

4.4 Second level of parallelization

This optimization phase aims at defining new parallelization schemes for the poll step, namely by choosing two poll centers at each iteration. In this case, selecting more than one poll center can allow gains in computational time and the in quality of the solution computed.

Before the poll center selection, the list of active points corresponding to the lower objective function values will be moved to the top. The first poll center will be the point at the top of the list, that presents the best objective function value. The second point is selected based on the step size parameter, α, namely the active point with the highest value of it. High step size parameters represent regions that have not yet been adequately explored and/or are more promising.

During the selection of the point corresponding to the active point with the highest

step size, it is very common to have points with the same α value. The following criteria were considered to break ties:

- criterion 1: choosing the point with the best objective function value, among those with the highest α;
- criterion 2: calculate the average of the objective function values for the active points with the highest α, and choose the point with the objective function value closer to this average value;
- criterion 3: choosing the point with the highest objective function value, among those with the highest α.

When selecting the first poll center, if there is a draw between points with the best objective function value, the point with highest α is selected.

4.4.1 Parallelization Structures

The parallelization structures required for the implementation of this phase are very similar to the ones required for the previous one. The main difference rely in the number of points that need to be evaluated, resulting from selection of the two poll centers and on the implementation of criteria for this selection.

Algorithm 6 provides the corresponding pseudocode. Additionally to the evaluation of a large number of points, there is a need to register the success or unsuccess associated to each point, because this distinction is necessary later, when updating the parameter α.

Algorithm 6: Pseudocode: optimization implemented at poll step, in second phase.

```
   Data: list (points to evaluate), cache, problem
1  futures_list ← {}
2  success_points ← 0 0
3  current_point ← 1
4  endcycle ← 0
5  forall point ∈ list in parallel do
6      match ← 0
7      feasible ← check_feasible(point)
8      if cache then
9          match,ftemp ← search_cache(point)
10     else
11         if feasible then
12             ftemp = eval_point(problem, point)
13         end
14     end
15     futures_list ← futures_list ∪ {ftemp,match,point,feasible}
16 end
17 for k ← 1 to list.lenght do
18     index, point, match, ftemp, feasible ← futures_list(k)
19     finite ← is_finite(ftemp)
20     while index = k do
21         if feasible then
22             if ¬match ∧ cache then
23                 update_cache(point,ftemp)
24             end
25             if finite then
26                 success ← add_point(ftemp)
27             end
28         end
29         if success then
30             success_points(current_point) ← 1
31             if sum(success_points) ≡ poll_centers or current_point ≡ 2 then
32                 endcycle ← 1
33                 break
34             end
35             current_point ← current_point + 1
36             k ← index(current_point)
37         end
38     end
39     if endcycle then
40         cancel(futures_list)
41         break
42     end
43 end
```

4.4.2 Implementation using PCT

Like in the implementation of the first phase, `parfeval` was used, to guarantee the possibility of interrupting a parallel loop, in order to implement the opportunistic strategy. The stopping criterion is different, as in this situation there are have two poll centers. In the first phase, when a success occurred, the parallel cycle was interrupted. In this implementation, it is necessary to distinguish which point is under analysis. If any of the associated with the first point corresponds to a success, the cycle will not stop, being necessary to evaluate the remaining directions referring to the second poll point.

```
 1 % For each point: check cache hit and if is it feasible
 2 % Inital Loop to launch function evaluations in parallel
 3 for i=1:pointsToEval
 4     F(i) = parfeval(@eval_point,3,xtemp,cache,func_f);
 5 end
 6 success_points = [0 0];
 7 current_point = 1;
 8 poll_centers = 2;
 9 endcycle = 0;
10 % Loop to collect functions evaluations launched before
11 for ii = 1:pointsToEval
12     [completedIdx, match,xtemp,ftemp] = fetchNext(F);
13     while completedIdx == (lastIndex)
14         % While loop to guarantee the evaluation order while completedIdx == lastIndex
15         if feasible
16             if isFinite
17                 [success] = merge(ftemp...)
18                     if cache
19                     % update cache
20                     end
21             end
22             if success
23                 success_points(current_point) = 1;
24                 if sum(success_points) == poll_centers | current_point == 2
25                     endcycle = 1;
26                     break
27                 end
28                 current_point = current_point + 1;
29                 % Update completedIdx to current_point index
30             end
31             % check if next point already ready to evaluate
32         end
33          % Break cycle if success and abort pending computations
34         if endcycle
35             cancel(F);
36             break
37         end
38     end
39 end
```

Figure 4.15: Code snippet with the parallelization implemented at poll step in second phase.

4.4.3 Numerical results

Ordered and unordered parallel versions were tested, in combination with the three criteria for the second poll center selection, described in Subsection 4.4.

Figure 4.16 reports the corresponding data profile, where the best version of the first parallelization phase (1 ORD) is also represented.

The advantage of ordered versions over unordered ones is clear. Inside the ordered versions, two present a slight advantage over the best version of the phase one, namely the ones that for the selection of the second poll center, break ties in value of the step size by being more greedy, selecting points with better objective function value (first and second criteria for draw).

Analyzing the performance profiles of Figures 4.17-4.20, now only elaborated for the two best versions, we can conclude that the solver referring to the first draw criterion presents better results, namely in terms of computational time. There is also a tiny

advantage in what respects to the number of local minima identified and the best objective function value computed.

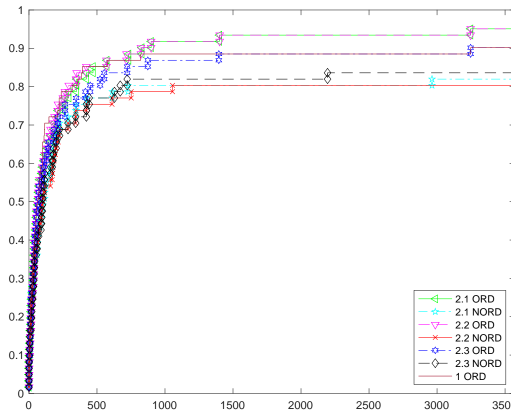

Figure 4.16: Data profile with all parallel versions of phase two and the best parallel version of phase one.

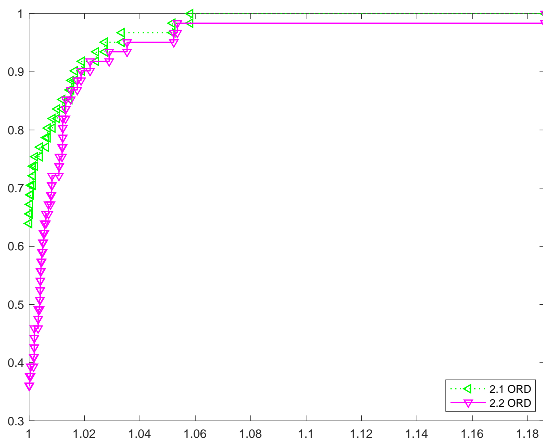

Figure 4.17: Performance profile comparing computational time, for the two best strategies of phase two, considering a 0.04 seconds delay in function evaluation.

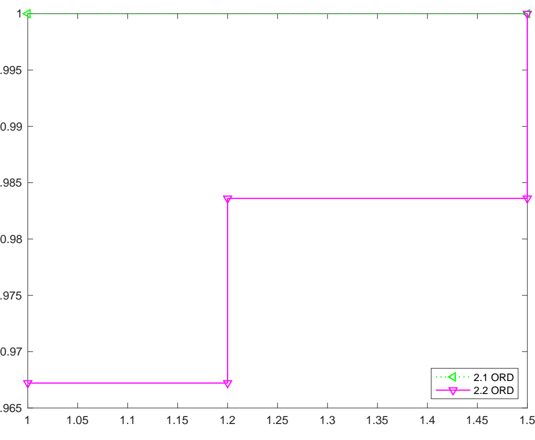

Figure 4.18: Performance profile comparing the number of local minima identified for the best strategies of phase two.

Finally, a comparison is made between the best solvers of phases one and two. Figures 4.21-4.24 report the corresponding performance profiles. Although there is a lower performance in terms of number of functions evaluations required before identifying the global minimum (see Figure 4.24), due to the exploration of two poll centers, there are notable gains in computational time (see Figure 4.21) and a slight advantage in terms of quality of the best point found (see Figure 4.23), obtained by the solver that results from this parallelization phase.

Table 4.5 reports the results regarding speedup and efficiency, ordered by problem dimension, comparing the best version of the previous parallelization phase and the best

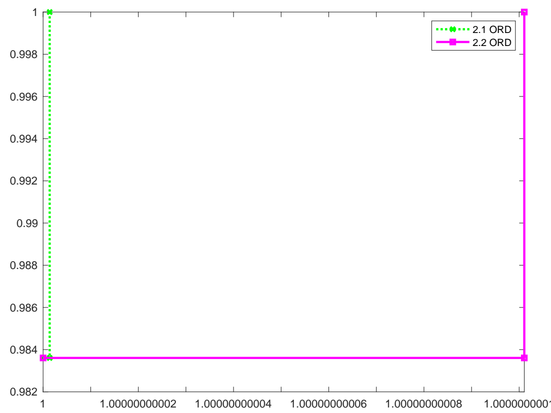

Figure 4.19: Performance profile comparing the best objective function value computed, for the two best strategies of phase two.

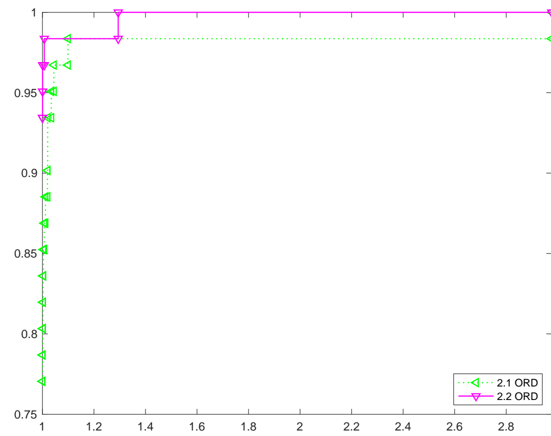

Figure 4.20: Performance profile comparing the number of function evaluations required until the global minimum is identified, for the best two strategies of phase two.

version of this phase. Again the artificial delay used for function evaluation is small (0,04 seconds), which makes the parallelization overhead relevant, when compared to function evaluation time, traducing in low speedups and efficiencies.

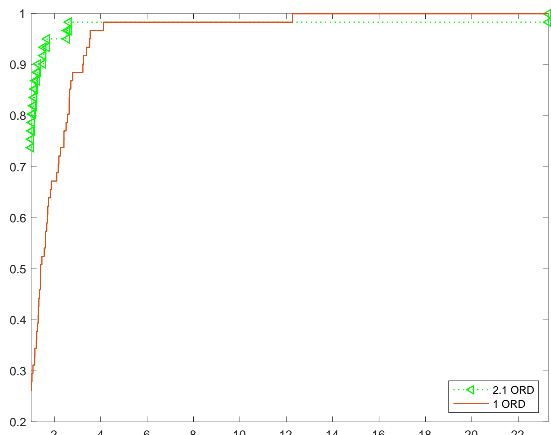

Figure 4.21: Performance profile comparing computational time, for the best versions of parallelization phases one and two, considering a 0.04 second delay for function evaluation.

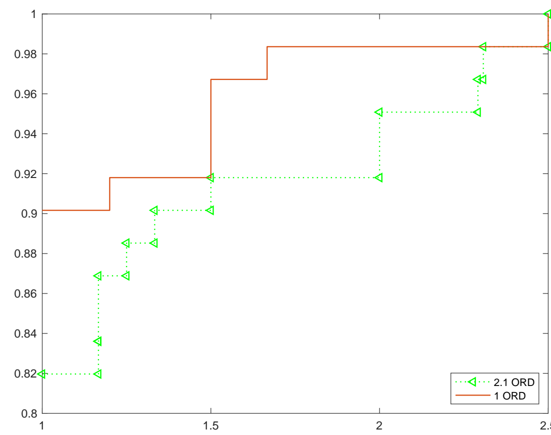

Figure 4.22: Performance profile comparing the number of local minima identified, for the best versions of parallelization phases one and two.

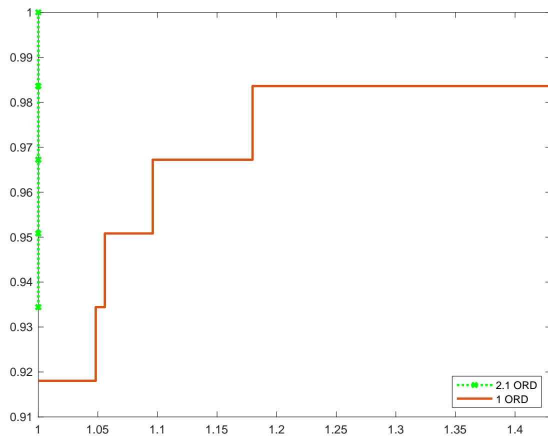

Figure 4.23: Performance profile comparing the best objective function value computed, for the best versions of parallelization phase one and two.

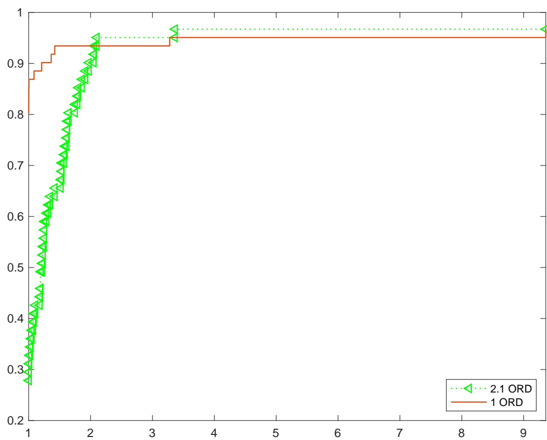

Figure 4.24: Performance profile comparing the number of function evaluations required until the global minimum is identified, for the best versions of parallelization phase one and two.

Table 4.5: Speedup and efficiency for phase 2.

#	Dimension	V1 (sec.)	V21 (sec.)	Speedup V1/V21	EFF 12w	EFF 2*N w
2	2	6,66	4,71	1,413	12%	35%
3	2	10,16	8,17	1,244	10%	31%
4	2	6,80	3,23	2,107	18%	53%
5	2	10,25	12,77	0,802	7%	20%
8	2	5,60	2,10	2,662	22%	67%
10	2	10,83	5,93	1,827	15%	46%
13	2	18,93	6,78	2,793	23%	70%
15	2	4,48	4,32	1,038	9%	26%
20	2	17,49	6,44	2,714	23%	68%
26	2	11,00	4,27	2,573	21%	64%
29	2	12,64	7,34	1,722	14%	43%
31	2	15,50	5,87	2,640	22%	66%
35	2	50,49	56,44	0,894	7%	22%
36	2	9,09	7,11	1,278	11%	32%
39	2	463,89	266,07	1,743	15%	44%
42	2	121,69	34,36	3,541	30%	89%
43	2	293,97	83,35	3,527	29%	88%
50	2	40,21	11,33	3,548	30%	89%
52	2	25,75	7,91	3,257	27%	81%
55	2	4,16	2,84	1,467	12%	37%
59	2	4,23	3,00	1,411	12%	35%
23	3	36,85	37,93	0,971	8%	16%
24	3	11,76	4,69	2,508	21%	42%
53	3	12,94	9,92	1,303	11%	22%
6	4	19,87	12,28	1,618	13%	20%
9	4	6,10	2,81	2,168	18%	27%
14	4	27,36	11,33	2,415	20%	30%
16	4	7,79	19,83	0,392	3%	5%
27	4	418,12	172,99	2,417	20%	30%
30	4	365,05	259,99	1,404	12%	18%
32	4	22,81	10,06	2,267	19%	28%
37	4	320,05	274,98	1,163	10%	15%
45	4	52,78	39,51	1,336	11%	17%
46	4	62,03	39,66	1,564	13%	20%
47	4	65,13	65,66	0,992	8%	12%
56	4	6,23	9,54	0,652	5%	8%
61	4	335,88	179,58	1,870	16%	23%
11	5	62,22	37,12	1,676	14%	17%
22	5	41,82	24,57	1,701	14%	17%
40	5	71,93	27,23	2,642	22%	26%
48	5	59,33	17,50	3,390	28%	34%
17	6	29,54	36,95	0,799	7%	7%
25	6	46,34	21,08	2,198	18%	18%
57	6	9,47	9,86	0,960	8%	8%
18	8	49,23	53,34	0,922	8%	8%
58	8	63,52	166,60	0,381	3%	3%
54	9	267,06	314,97	0,847	7%	7%
60	9	295,86	91,50	3,233	27%	27%
1	10	21,13	21,26	0,993	8%	8%
7	10	155,78	134,64	1,157	10%	10%
12	10	263,09	215,29	1,222	10%	10%
19	10	52,86	80,60	0,655	5%	5%
21	10	109,43	80,56	1,358	11%	11%
28	10	202,34	18,12	11,164	93%	93%
33	10	260,84	298,14	0,874	7%	7%
34	10	20,28	15,47	1,311	11%	11%
38	10	71,84	93,01	0,772	6%	6%
41	10	189,29	116,47	1,625	14%	14%
44	10	240,01	222,09	1,080	9%	9%
49	10	71,81	121,08	0,593	5%	5%
51	10	258,44	252,44	1,023	9%	9%

4.5 Third level of parallelization

In the two previous phases, the parallelization strategies developed considered a fixed number of poll centers: one (see Section 4.3) or two (see Section 4.4). However, depending on problem dimension, the full capacity of hardware resources may not be entirely exploited.

This parallelization phase aims to dynamically select the number of poll centers to be explored at each poll step to maximize the use of hardware capacity. Although it is expected that this implementation will spend the budget of function evaluations more quickly than the previous parallelization approaches, due to the exploration of more points in each iteration, selecting more than two poll centers can allow gains in the quality of the computed solution and in the other metrics presented in Subsections 2.2.5 and 4.1.1.

The number of poll centers to be selected at each iteration is determined as:

$$floor\left(\frac{\#workers}{\#polldirections}\right)$$

The selection criteria are the same as those referred in the Subsection 4.4, with the difference that if more than two poll centers need to be chosen, the selection will continue to be based on the largest step size.

4.5.1 Parallelization Structures

Algorithm 7: Parallel approach implemented in phase three, at the poll step.

```
   Data: list (points to evaluate), cache, problem, poll_centers
 1 futures_list ← {}
 2 success_points ← zeros(poll_centers)
 3 current_point ← 1
 4 last_poll_center ← size(poll_centers)
 5 forall point ∈ list in parallel do
 6     match ← 0
 7     feasible ← check_feasible(point)
 8     if cache then
 9         match,ftemp ← search_cache(point)
10     else
11         if feasible then
12             ftemp = eval_point(problem, point)
13         end
14     end
15     futures_list ← futures_list ∪ {ftemp,match,point,feasible}
16 end
17 for k ← 1 to list.lenght do
18     index, point, match, ftemp, feasible ← futures_list(k)
19     finite ← is_finite(ftemp)
20     while index = k do
21         if feasible then
22             if ¬match ∧ cache then
23                 update_cache(point,ftemp)
24             end
25             if finite then
26                 success ← add_point(ftemp)
27             end
28         end
29         if success then
30             success_points(current_point) ← 1
31             if sum(success_points) ≡ poll_centers or current_point ≡ last_poll_center then
32                 endcycle ← 1
33                 break
34             end
35             current_point ← current_point + 1
36             k ← index(current_point)
37         end
38     end
39     if endcycle then
40         cancel(futures_list)
41         break
42     end
43 end
```

The parallelization structures required for the implementation of this phase continue to be very similar to the previous ones. The main differences rely on the number of points that need to be evaluated, resulting from the selection of several poll centers and the implementation of the criteria for this selection. Algorithm 7 provides the corresponding pseudocode. Like in the previous parallelization phase, there is a need to register the success or unsuccess associated with each poll center, which is required later to update the parameter α.

4.5.2 Implementation using PCT

Like in the cases of the first and second phases, `parfeval` was used to guarantee the possibility of interrupting a parallel loop to implement the opportunistic strategy. In this situation, the stopping criterion is similar to the one of second phase, but now with several poll centers. Even if a given direction, associated to a poll center, corresponds to a success, the cycle will not stop, being necessary to evaluate the remaining directions referring to the next poll center (see Figure 4.25).

```
 1 % For each point: check cache hit and if is it feasible
 2 % Inital cycle to launch function evaluations in parallel
 3 for i=1:pointsToEval
 4     F(i) = parfeval(@eval_point,3,xtemp,cache,func_f);
 5 end
 6 success_points = zeros(size(poll_points_array,2),1);
 7 current_point = 1;
 8 poll_centers = size(poll_points_array,2);
 9 endcycle = 0;
10 last_poll_center = size(poll_points_array,2);
11 % Cycle to collect functions evaluations launched before
12 for ii = 1:pointsToEval
13     [completedIdx, match,xtemp,ftemp] = fetchNext(F);
14     while completedIdx == (lastIndex)
15         % While loop to guarantee the evaluation order while completedIdx == lastIndex
16         if feasible
17             if isFinite
18                 [success] = merge(ftemp...)
19                     if cache
20                     % update cache
21                     end
22             end
23             if success
24                 success_points(current_point) = 1;
25                 if sum(success_points) == poll_centers | current_point == last_poll_center
26                     endcycle = 1;
27                     break
28                 end
29                 current_point = current_point + 1;
30                 % Update completedIdx to current_point index
31             end
32             % check if next point already ready to evaluate
33         end
34     end
35      % Break cycle if success and abort pending computations
36     if endcycle
37         break;
38     end
39 end
```

Figure 4.25: Code snippet with the parallelization implemented at poll step in third phase.

4.5.3 Numerical results

GLODS algorithm has two default stopping criteria: the step size parameter of all active points (lowest value allowed 10^{-8}) and the number of function evaluations (maximum of 20000).

The current parallelization scheme, predictably, will consume the function evaluations budget and will and terminate its execution more quickly, not reaching a good quality in approximation to the global minimum, being necessary to adjust the stopping criteria of

the algorithm to assess all the potential of the parallelization. Thus, three versions were considered, corresponding to different stopping criteria:

- Version 3.1 ORD free, corresponding to third parallelization phase with stopping criteria based on the step size parameter and function evaluation budget;
- Version 3.2 ORD time 2.1, corresponding to third parallelization phase with stopping criteria based on the step size parameter and on the computational time of the best version of the second optimization phase (2.1 ORD);
- Version 3.2 ORD time seq, corresponding to third parallelization phase with stopping criteria based on the step size parameter and on the computational time of the sequential version.

Analyzing the data profile of Figure 4.26, the advantage of the versions that use several poll centers (3.1 ORD free and 3.2 ORD free 2.1) over the best version that uses only two poll centers (2.1 ORD) is clear. However, given the larger computational time associated with the sequential version, it would be expected that the curve referring to version 3.2 ORD time seq would be closer to ones of versions that can be observed.

Analyzing the performance profiles, the version that presents better performance in terms of computational time (see Figure 4.27) is version 3.1 ORD time 2.1, which is surprising, considering the performance off version 3.1 ORD free for the same metric.

Regarding the analysis number of functions evaluations required until the global minimum is identified (see Figure 4.28), the version with the worst performance is 2.1 ORD, when it would be expected to be the version with the best performance.

Relatively the number of local minimal identified, the current parallelization phase allows better performance over the best version of the second phase (see Figure 4.29).

Considering now Figure 4.30, version 2.1 ORD is the most efficient in terms of approximation on to the global minimum, which seems to contradict the results reported on Figure 4.26, where it presented worst performance.

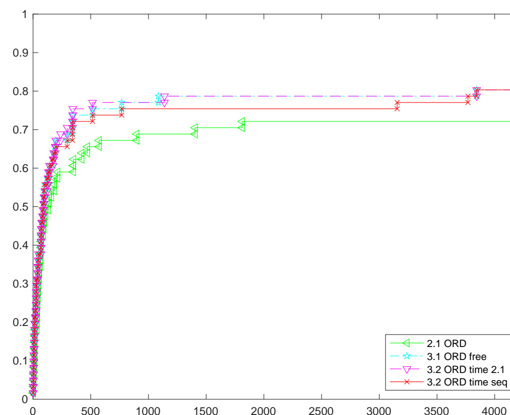

Figure 4.26: Data profile with all versions of phase 3 and the best parallel version of phase 2.

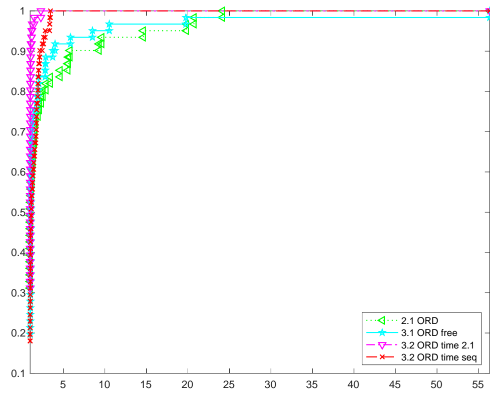

Figure 4.27: Performance profile comparing computational time, for all strategies of phase three and the strategy of phase two.

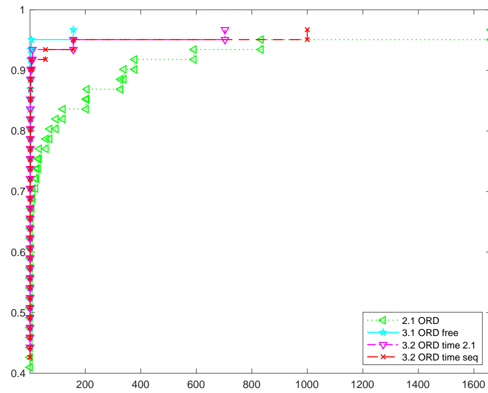

Figure 4.28: Performance profile comparing the number of functions evaluations required until the global minimum is identified, for all strategies of phase three and the strategy of phase two.

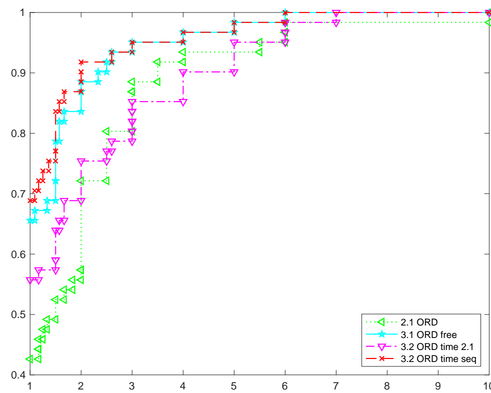

Figure 4.29: Performance profile comparing the number of local minima identified, for all strategies of phase three and the strategy of phase two.

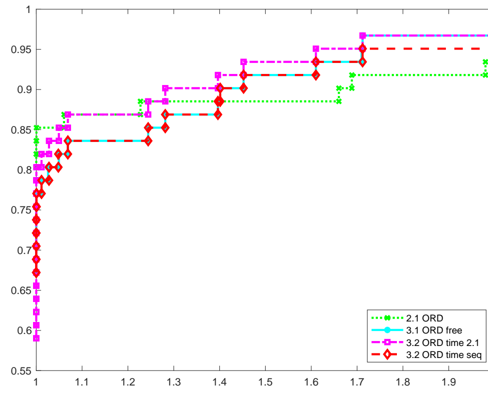

Figure 4.30: Performance profile comparing the best objective function value computed, for all strategies of phase three and the strategy of phase two.

In order to understand the inconsistent results, the computational times of all versions tested in this parallelization phase were analyzed.

Table 4.6 reports four cases: two that present a normal behaviour; and two anomalous ones.

The first column represents the problem number; the second column represents the problem dimension; the third column, represents the computational time of the sequential version; the remaining columns represent the computational times of parallel versions.

In problem 2, the version 3.2 ORD time seq (last column) stops based on the time of the sequential version. In problem 3, although version 3.2 ORD time seq does not use the

time of the sequential version to stop, it uses a stopping criterion based on the step size, like the other versions of third phase.

The first anomalous case, referring to problem 28, demonstrates that the computational time of 3.2 ORD time seq is not coherent. The stopping criterion is based on the step size parameter, and thus, not reaching the reference time of the sequential version, it should present a computational time similar to the version 3.1 ORD free.

In the second anomalous case, referring to problem 35, the computational time of version 3.2 ORD time seq does not match the stopping criterion of the sequential time, and it is higher than the one of 3.1 ORD free version.

Table 4.6: Analysis of individual computational times.

#	Dim	Seq	2.1	3.1 ORD free	3.2 ORD time 2.1	3.2 ORD time seq
2	5	60,23	24,57	70,86	24,62	60,34
3	2	10,50	8,16	4,18	4,31	4,35
28	10	604,48	18,12	154,21	18,27	23,49
35	2	55,60	56,44	10,08	10,22	34,77

5 Conclusions and future work

The present work proposed and analyzed parallelization strategies to improve the numerical performance of GLODS, a global DFO solver, implemented in MATLAB. During the process, auxiliary experiments were conducted that allowed a better understanding of the parallelization tools provided by the MATLAB PCT.

In real applications of DFO, function evaluation can take seconds or even minutes of computational time. Considering the large number of tests needed to be conducted, when assessing the value of new numerical strategies, it is not feasible to simulate this large computational time by including artificial delays in code execution. The numerical experiments with the MATLAB PCT allowed to establish and validate a formula that estimates the total computational time using the number of parallel cycles performed.

Regarding the solver GLODS, a first assessment of the default version of the sequential implementation of the algorithm was performed, namely in what respects the frequency and the strategy to be used for new initializations in the search step or performing complete or opportunistic polling. As consequence, the criterion for new initializations was changed from having a single active point in the list, not yet identified as local minimizer, to having two.

Parallelization strategies were then defined and tested for the three relevant components of the algorithm, namely the initialization, the search step, and the poll step. In the first parallelization level, function evaluation was simply distributed among different workers, requiring different parallelization approaches, due to the opportunistic strategy of the poll step. Benefits in computational time are expected, once than function evaluation takes and average time of 0.04 seconds or higher.

In an attempt of improving not only the computational time but also the quality of the best point found, as approximation to the global minimizer of the problem under analysis, a second parallelization level was considered, where at each poll step, two poll centers are selected: one corresponding to the best point found and another corresponding to the largest step size parameter, representing regions that have not yet been explored or are promising. Three different strategies were considered and tested to break ties, when selecting the second point. The one with best performance corresponded to a

greedy approach, again selecting the point with the lowest objective function value, from the ones with the largest step size. This second parallelization phase allowed not only an improvement in computational time, but also an increase in the quality of the approximation to the global minimizer.

Finally, a third parallelization level was addressed, in an attempt of exploring all the computational power available. The number of poll centers to be considered at each poll step would depend on the problem dimension and on the number of workers available in the hardware. This strategy may present the disadvantage of quickly exhausting the function evaluation budget, since it typically would perform more function evaluations at each iteration. However, it can potentiate improvements in the quality of the final solution if computational times like the ones of the previous parallel versions are allowed. Unfortunately, the first numerical data obtained seems inconsistent, requiring a complete check of the implementation and new testing. This version would be particularly useful in future work, where we intend to define a search step for GLODS based on the minimization of radial basis functions and use the local minimizers of these models (which often will be higher than two) as poll centers.

As future work, additionally to the tasks previously mentioned regarding the definition of a search step based on the minimization of radial basis functions and the phase three of the parallelization, the code needs to be cleaned, properly commented, and integrated in the toolbox BoostDFO. This toolbox, already developed, would be an important tool for the academic and the industrial communities, allowing to run different solvers sequentially or in parallel, for solving local/global, single/multi objective DFO problems.

Bibliography

[1] Y. M. Altman. *Accelerating MATLAB Performance: 1001 Tips to Speed Up MATLAB Programs*. CRC Press, 2014 (cit. on pp. 14–16).

[2] M. Armbrust, A. Fox, G. Rean, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, R. Ariel, I. Stoica, and M. Zaharia. "A view of cloud computing". In: *International Journal of Networked and Distributed Computing* 1.1 (2013), pp. 2–8. ISSN: 22117946. DOI: 10.2991/ijndc.2013.1.1.2 (cit. on p. 20).

[3] C. Audet and J. E. Dennis Jr. "Analysis of generalized pattern searches". In: *SIAM Journal on Optimization* 13.3 (2002), pp. 889–903 (cit. on p. 6).

[4] C. Audet and J. E. Dennis Jr. "Mesh adaptive direct search algorithms for constrained optimization". In: *SIAM Journal on Optimization* 17.1 (2006), pp. 188–217 (cit. on p. 6).

[5] C. Audet, J. E. Dennis Jr, and S. L. Digabel. "Parallel Space Decomposition of the Mesh Adaptive Direct Search Algorithm". In: *SIAM Journal on Optimization* 19.3 (2008), pp. 1150–1170 (cit. on p. 22).

[6] C. Audet and W. Hare. *Derivative-free and Blackbox Optimization*. Springer, 2017 (cit. on pp. 6, 7).

[7] K. Bigdeli, W. Hare, and S. Tesfamariam. "Configuration optimization of dampers for adjacent buildings under seismic excitations". In: *Engineering Optimization* 44.12 (2012), pp. 1491–1509 (cit. on p. 6).

[8] A. Booker, P. Frank, J Dennis Jr, D. Moore, and D. Serafini. "Managing surrogate objectives to optimize a helicopter rotor design-further experiments". In: *7th Symposium on Multidisciplinary Analysis and Optimization*. 1998, p. 4717 (cit. on p. 6).

[9] A. J. Booker, J. Dennis, P. D. Frank, D. B. Serafini, and V. Torczon. "Optimization using surrogate objectives on a helicopter test example". In: *Computational Methods for Optimal Design and Control*. Springer, 1998, pp. 49–58 (cit. on p. 6).

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-free Optimization*. Vol. 8. SIAM, 2009 (cit. on pp. 4, 5).

[11] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Ed. by Addison-Wesley. 5th ed. Addison-Wesley, 2012, p. 1048. ISBN: 0-13-214301-1 (cit. on p. 9).

[12] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing, 1999 (cit. on p. 9).

[13] A. L. Custódio and J. F. A. Madeira. "GLODS: Global and Local Optimization using Direct Search". In: *Journal of Global Optimization* 62.1 (2015), pp. 1–28 (cit. on pp. 1, 2, 6, 8, 24–27).

[14] C. Davis. "Theory of positive linear dependence". In: *American Journal of Mathematics* 76.4 (1954), pp. 733–746 (cit. on pp. 6, 25).

[15] G. Deng and M. C. Ferris. "Extension of the DIRECT optimization algorithm for noisy functions". In: *2007 Winter Simulation Conference*. IEEE. 2007, pp. 497–504 (cit. on p. 8).

[16] J. E. Dennis and V. Torczon. "Direct search methods on parallel machines". In: *SIAM Journal on Optimization* 1.4 (1991), pp. 448–474 (cit. on p. 22).

[17] E. D. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical Programming* 91.2 (2002), pp. 201–213 (cit. on p. 31).

[18] I. Foster. *Designing and building parallel programs: concepts and tools for parallel software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 11).

[19] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011 (cit. on pp. 9, 10).

[20] P. D. Hough, T. G. Kolda, and V. J. Torczon. "Asynchronous parallel pattern search for nonlinear optimization". In: *SIAM Journal on Scientific Computing* 23.1 (2001), pp. 134–156 (cit. on p. 22).

[21] *Implicit Parallelism (Multithreading) : TechWeb : Boston University*. URL: https://www.bu.edu/tech/support/research/training-consulting/online-tutorials/matlab-pct/implicit-parallelism/ (visited on 07/24/2020) (cit. on p. 9).

[22] M. D. Intriligator. *Mathematical Optimization and Economic Theory*. SIAM, 2002 (cit. on p. 1).

[23] K. Jackson, C. Bunch, E. Sigler, and J. Denton. *OpenStack Cloud Computing Cookbook, Fourth Edition*. Packt Publishing, 2018 (cit. on pp. 19, 22).

[24] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, and N. Yadwadkar. “Cloud programming simplified: A Berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019) (cit. on p. 21).

[25] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. “Lipschitzian optimization without the Lipschitz constant”. In: *Journal of Optimization Theory and Applications* 79.1 (1993), pp. 157–181 (cit. on p. 8).

[26] A. Kaminsky. “Big CPU, Big Data”. In: *Rochester Institute of Technology* (2015) (cit. on p. 2).

[27] L. Kocis and W. J. Whiten. “Computational investigations of low-discrepancy sequences”. In: *ACM Transactions on Mathematical Software* 23.2 (1997), pp. 266–294 (cit. on p. 25).

[28] J. Larson, M. Menickelly, and S. M. Wild. “Derivative-free Optimization Methods”. In: *Acta Numerica* 28 (2019), pp. 287–404 (cit. on pp. 1, 7).

[29] D. C. Marinescu. *Cloud computing: Theory and Practice*. Morgan Kaufmann, 2017 (cit. on p. 2).

[30] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012 (cit. on p. 18).

[31] M. D. McKay, R. J. Beckman, and W. J. Conover. “A comparison of three methods for selecting values of input variables in the analysis of output from a computer code”. In: *Technometrics* 42.1 (2000), pp. 55–61 (cit. on p. 25).

[32] J. J. Moré and S. M. Wild. “Benchmarking Derivative-free Optimization algorithms”. In: *SIAM Journal on Optimization* 20.1 (2009), pp. 172–191 (cit. on p. 30).

[33] NIST. *The NIST Definition of Cloud Computing*. Tech. rep. 2016, pp. 1–8. DOI: 10.6028/NIST.SP.800-145. URL: https://www.nist.gov/publications/nist-definition-cloud-computing (cit. on pp. 20, 21).

[34] J. L. Ortega-Arjona. *Patterns for parallel software design*. Vol. 21. John Wiley & Sons, 2010 (cit. on p. 2).

[35] *Parallel algorithm design approach | PCAM | Engineer's Portal*. URL: https://er.yuvayana.org/parallel-algorithm-design-approach-pcam/ (visited on 07/25/2020) (cit. on p. 11).

[36] *Parallel Computing Toolbox™ User's Guide R2020a*. Tech. rep. 2020. URL: www.mathworks.com (cit. on pp. 15, 16).

[37] N. Ploskas and N. Samaras. *GPU programming in MATLAB*. Morgan Kaufmann, 2016 (cit. on p. 14).

[38] J. Reese and S. Zaranek. “GPU programming in Matlab”. In: *MathWorks News&Notes. Natick, MA: The MathWorks Inc* (2012) (cit. on p. 14).

[39] T. J. Santner, B. J. Williams, W. I. Notz, and B. J. Williams. *The Design and Analysis of Computer Experiments*. Vol. 1. Springer, 2003 (cit. on p. 25).

[40] D. B. Serafini. “A framework for managing models in optimization of computationally expensive functions”. PhD thesis. Department of Computational and Applied Mathematics, 1998 (cit. on p. 6).

[41] G. Sharma and J. Martin. “MATLAB®: a language for parallel computing”. In: *International Journal of Parallel Programming* 37.1 (2009), pp. 3–36 (cit. on p. 13).

[42] M. Spillane, E Gica, and V. Titov. “Tsunameter Network Design for the US DART® Array”. In: *AGUFM* 2009 (2009), OS43A–1368 (cit. on p. 5).

[43] T. Sterling, M. Anderson, and M. Brodowicz. *High Performance Computing: Modern Systems and Practices*. Elsevier, 2018 (cit. on p. 10).

[44] A. I. F. Vaz and L. N. Vicente. “PSwarm: A hybrid solver for linearly constrained global derivative-free optimization”. In: *Optimization Methods & Software* 24.4-5 (2009), pp. 669–685 (cit. on pp. 8, 23, 31).

[45] *What Is Cloud Computing? A Beginner’s Guide | Microsoft Azure*. URL: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/{\#}cloud-deployment-types (visited on 07/21/2020) (cit. on p. 21).

[46] *What Is MATLAB?* URL: https://www.mathworks.com/videos/matlab-overview-61923.html (visited on 06/04/2020) (cit. on p. 13).

[47] S. Wright and J. Nocedal. “Numerical Optimization”. In: *Springer Science* 35.67-68 (1999), p. 7 (cit. on p. 1).

A

Appendix 1: Problem set

Problem	n	l	u	loc	$glob$
ackley	10	-30	30	–	1
aluffi_pentini	2	-10	10	2	1
becker_lago	2	-10	10	4	4
bohachevsky	2	-50	50	–	1
branin_hoo	2	$[-5\ 0]^\top$	$[10\ 15]^\top$	3	3
cauchy	4	3	17	–	–
cauchy	10	2	26	–	–
cosine_mixture	2	-1	1	–	–
cosine_mixture	4	-1	1	–	–
dekkers_aarts	2	-20	20	3	2
epistatic_michalewicz	5	0	π	–	1
epistatic_michalewicz	10	0	π	–	1
exponencial	2	-1	1	–	1
exponencial	4	-1	1	–	1
fifteenn_local_minima	2	-10	10	15^2	1
fifteenn_local_minima	4	-10	10	15^4	1
fifteenn_local_minima	6	-10	10	15^6	1
fifteenn_local_minima	8	-10	10	15^8	1
fifteenn_local_minima	10	-10	10	15^{10}	1
goldstein_price	2	-2	2	4	1
griewank	5	-600	600	–	1
griewank	10	-400	400	–	1
gulf	3	$[0.1\ 0\ 0]^\top$	$[100\ 25.6\ 5]^\top$	–	1
hartman_4	3	0	1	4	1
hartman_4	6	0	1	4	1
hosaki	2	$[0\ 0]^\top$	$[5\ 6]^\top$	2	1
kowalik	4	0	0.42	–	1
langerman	10	0	10	–	1
mccormick	2	$[-1.5\ -3]^\top$	$[4\ 3]^\top$	2	1
miele_cantrell	4	-10	10	–	1

Table A.1: A description of the test set (first part). The variable n represents the dimension of the problem, l and u, lower and upper bounds, respectively, on the problem variables, loc and $glob$, the number of local and global minimizers, respectively, reported in the literature.

Problem	n	l	u	loc	$glob$
multi_gaussian	2	-2	2	5	1
neumaier2	4	0	4	–	1
neumaier3	10	-100	100	–	1
paviani	10	2.001	9.999	–	1
periodic	2	-10	10	50	1
poissonian	2	$[1\ 1]^\top$	$[21\ 8]^\top$	–	–
powell	4	-10	10	1	1
rastrigin	10	-5.12	5.12	–	1
rosenbrock	2	-5.12	5.12	1	1
salomon	5	-100	100	–	1
salomon	10	-100	100	–	1
schaffer1	2	-100	100	–	1
schaffer2	2	-100	100	–	1
schwefel	10	-500	500	–	1
shekel_45	4	0	10	5	1
shekel_47	4	0	10	7	1
shekel_410	4	0	10	10	1
shekel_foxholes	5	0	10	–	1
shekel_foxholes	10	0	10	–	1
shubert	2	-10	10	760	18
sinusoidal	10	0	180	–	1
sixhumpcamel	2	$[-3\ -2]^\top$	$[3\ 2]^\top$	6	2
sphere	3	-5.12	5.12	1	1
storn tchebychev	9	-128	128	–	1
tenn_local_minima	2	-10	10	10^2	1
tenn_local_minima	4	-10	10	10^4	1
tenn_local_minima	6	-10	10	10^6	1
tenn_local_minima	8	-10	10	10^8	1
three_hump_camel	2	-5	5	3	1
transistor	9	-10	10	–	1
wood	4	-10	10	–	1

Table A.2: A description of the test set (second part). The variable n represents the dimension of the problem, l and u, lower and upper bounds, respectively, on the problem variables, loc and $glob$, the number of local and global minimizers, respectively, reported in the literature.