Ana I. Pereira · Andrej Košir ·
Florbela P. Fernandes · Maria F. Pacheco ·
João P. Teixeira · Rui P. Lopes (Eds.)

Communications in Computer and Information Science 1754

Optimization, Learning Algorithms and Applications

Second International Conference, OL2A 2022
Póvoa de Varzim, Portugal, October 24–25, 2022
Proceedings

Springer

OL2A INTERNATIONAL CONFERENCE

Communications in Computer and Information Science 1754

Ana I. Pereira · Andrej Košir ·
Florbela P. Fernandes · Maria F. Pacheco ·
João P. Teixeira · Rui P. Lopes (Eds.)

Optimization, Learning Algorithms and Applications

Second International Conference, OL2A 2022
Póvoa de Varzim, Portugal, October 24–25, 2022
Proceedings

Springer

Editors
Ana I. Pereira
Instituto Politécnico de Bragança
Bragança, Portugal

Florbela P. Fernandes
Instituto Politécnico de Bragança
Bragança, Portugal

João P. Teixeira
Instituto Politécnico de Bragança
Bragança, Portugal

Andrej Košir
University of Ljubljana
Ljubljana, Slovenia

Maria F. Pacheco
Instituto Politécnico de Bragança
Bragança, Portugal

Rui P. Lopes
Instituto Politécnico de Bragança
Bragança, Portugal

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-031-23235-0 ISBN 978-3-031-23236-7 (eBook)
https://doi.org/10.1007/978-3-031-23236-7

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

This CCIS volume 1754 contains the refereed proceedings of the Second International Conference on Optimization, Learning Algorithms and Applications (OL2A 2022), a hybrid event held during October 24–25, 2022.

OL2A 2022 provided a space for the research community on optimization and learning to get together and share the latest developments, trends, and techniques, as well as to develop new paths and collaborations. The conference had more than three hundred participants in an online and face-to-face environment throughout two days, discussing topics associated with optimization and learning, such as state-of-the-art applications related to multi-objective optimization, optimization for machine learning, robotics, health informatics, data analysis, optimization and learning under uncertainty, and Industry 4.0.

Five special sessions were organized under the following topics: Trends in Engineering Education, Optimization in Control Systems Design, Measurements with the Internet of Things, Advances and Optimization in Cyber-Physical Systems, and Computer Vision Based on Learning Algorithms. The OL2A 2022 program included presentations of 56 accepted papers. All papers were carefully reviewed and selected from 145 submissions in an single-blind process. All the reviews were carefully carried out by a scientific committee of 102 qualified researchers from 21 countries, with each submission receiving at least 3 reviews.

We would like to thank everyone who helped to make OL2A 2022 a success and hope that you enjoy reading this volume.

October 2022

Ana I. Pereira
Andrej Košir
Florbela P. Fernandes
Maria F. Pacheco
João P. Teixeira
Rui P. Lopes

Organization

General Chairs

Ana I. Pereira	Polytechnic Institute of Bragança, Portugal
Andrej Košir	University of Ljubljana, Slovenia

Program Committee Chairs

Florbela P. Fernandes	Polytechnic Institute of Bragança, Portugal
Maria F. Pacheco	Polytechnic Institute of Bragança, Portugal
João P. Teixeira	Polytechnic Institute of Bragança, Portugal
Rui P. Lopes	Polytechnic Institute of Bragança, Portugal

Special Session Chairs

João P. Coelho	Polytechnic Institute of Bragança, Portugal
Luca Oneto	University of Genoa, Italy

Technology Chairs

Alexandre Douplik	Ryerson University, Canada
Paulo Alves	Polytechnic Institute of Bragança, Portugal

Local Organizing Chairs

José Lima	Polytechnic Institute of Bragança, Portugal

Program Committee

Ana I. Pereira	Polytechnic Institute of Bragança, Portugal
Abeer Alsadoon	Charles Sturt University, Australia
Ala' Khalifeh	German Jordanian University, Jordan
Alexandre Douplik	Ryerson University, Canada
Ana Maria A. C. Rocha	University of Minho, Portugal
Ana Paula Teixeira	University of Trás-os-Montes and Alto Douro, Portugal
André Pinz Borges	Federal University of Technology – Paraná, Brazil
André R. da Cruz	Federal Center for Technological Education of Minas Gerais, Brazil

Andrej Košir	University of Ljubljana, Slovenia
Arnaldo Cândido Júnior	Federal University of Technology – Paraná, Brazil
António J. Sánchez-Salmerón	Universitat Politécnica de Valencia, Spain
António Valente	Universidade de Trás-Os-Montes e Alto Douro, Portugal
Bilal Ahmad	University of Warwick, UK
Bruno Bispo	Federal University of Santa Catarina, Brazil
Carlos Henrique Alves	CEFET/RJ, Brazil
Carmen Galé	University of Zaragoza, Spain
B. Rajesh Kanna	Vellore Institute of Technology, India
Carolina Gil Marcelino	Federal University of Rio de Janeiro, Brazil
Christopher E. Izquierdo	University of Laguna, Spain
C. Sweetlin Hemalatha	Vellore Institute of Technology, India
Damir Vrančić	Jozef Stefan Institute, Slovenia
Daiva Petkeviciute	Kaunas University of Technology, Lithuania
Dhiah Abou-Tair	German Jordanian University, Jordan
Diego Brandão	CEFET/RJ, Brazil
Dimitris Glotsos	University of West Attica, Greece
Diamantino Silva Freitas	University of Porto, Portugal
Eduardo Vinicius Kuhn	Federal University of Technology – Paraná, Brazil
Elizabeth Fialho Wanner	Federal Center for Technological Education of Minas Gerais, Brazil
Elaine Mosconi	Université de Sherbrooke, Canada
Esteban Clua	Federal Fluminense University, Brazil
Eric Rogers	University of Southampton, UK
Felipe N. Martins	Hanze University of Applied Sciences, Netherlands
Florbela P. Fernandes	Polytechnic Institute of Bragança, Portugal
Florentino F. Riverola	University of Vigo, Spain
Gaukhar Muratova	Dulaty University, Kazakhstan
Gediminas Daukšys	Kauno Technikos Kolegija, Lithuania
Glaucia Maria Bressan	Federal University of Technology – Paraná, Brazil
Glotsos Dimitris	University of West Attica, Greece
Humberto Rocha	University of Coimbra, Portugal
J. Marcos Moreno Veja	University of Laguna, Spain
João Paulo Carmo	University of São Paulo, Brazil
João P. Teixeira	Polytechnic Institute of Bragança, Portugal
Jorge Igual	Universidad Politécnica de Valencia, Spain
José Boaventura-Cunha	University of Trás-os-Montes and Alto Douro, Portugal
José Lima	Polytechnic Institute of Bragança, Portugal

Joseane Pontes	Federal University of Technology – Ponta Grossa, Brazil
Juan Alberto G. Esteban	Universidad de Salamanca, Spain
Juan A. Méndez Pérez	University of Laguna, Spain
Juani Lopéz Redondo	University of Almeria, Spain
Julio Cesar Nievola	Pontifícia Universidade Católica do Paraná, Brazil
João Paulo Coelho	Polytechnic Institute of Bragança, Portugal
Jorge Ribeiro	Polytechnic Institute of Viana do Castelo, Portugal
José Ramos	NOVA University Lisbon, Portugal
Kristina Sutiene	Kaunas University of Technology, Lithuania
Lidia Sánchez	University of León, Spain
Lino Costa	University of Minho, Portugal
Luis A. De Santa-Eulalia	Université de Sherbrooke, Canada
Luís Coelho	Polytecnhic Institute of Porto, Portugal
Luca Oneto	University of Genoa, Italy
Luca Spalazzi	Marche Polytechnical University, Italy
Maria F. Pacheco	Polytechnic Institute of Bragança, Portugal
Manuel Castejón Limas	University of León, Spain
Marc Jungers	Université de Lorraine, France
Marco A. S. Teixeira	Universidade Tecnológica Federal do Paraná, Brazil
Maria do R. de Pinho	University of Porto, Portugal
Marco A. Wehrmeister	Federal University of Technology – Paraná, Brazil
Markus Vincze	TU Wien, Austria
Martin Hering-Bertram	Hochschule Bremen, Germany
Mikulas Huba	Slovak University of Technology in Bratislava, Slovakia
Michał Podpora	Opole University of Technology, Poland
Miguel Ángel Prada	University of León, Spain
Nicolae Cleju	Technical University of Iasi, Romania
Paulo Lopes dos Santos	University of Porto, Portugal
Paulo Alves	Polytechnic Institute of Bragança, Portugal
Paulo Leitão	Polytechnic Institute of Bragança, Portugal
Paulo Moura Oliveira	University of Trás-os-Montes and Alto Douro, Portugal
Pavel Pakshin	Nizhny Novgorod State Technical University, Russia
Pedro Luiz de P. Filho	Federal University – Paraná, Brazil
Pedro Miguel Rodrigues	Catholic University of Portugal, Portugal
Pedro Morais	Polytechnic Institute of Cávado e Ave, Portugal
Pedro Pinto	Polytechnic Institute of Viana do Castelo, Portugal

Roberto M. de Souza	Federal University of Technology – Paraná, Brazil
Rui P. Lopes	Polytechnic Institute of Bragança, Portugal
Sabrina Šuman	Polytechnic of Rijeka, Croatia
Sani Rutz da Silva	Federal University of Technology – Paraná, Brazil
Santiago Torres Álvarez	University of Laguna, Spain
Sara Paiva	Polytechnic Institute of Viana do Castelo, Portugal
Shridhar Devamane	Global Academy of Technology, India
Sofia Rodrigues	Polytechnic Institute of Viana do Castelo, Portugal
Sławomir Stępień	Poznan University of Technology, Poland
Teresa P. Perdicoulis	University of Trás-os-Montes and Alto Douro, Portugal
Toma Roncevic	University of Split, Croatia
Uta Bohnebeck	Hochschule Bremen, Germany
Valeriana Naranjo-Ornedo	Universidad Politécnica de Valencia, Spain
Vivian Cremer Kalempa	Universidade Estadual de Santa Catarina, Brazil
Vitor Duarte dos Santos	NOVA University Lisbon, Portugal
Wynand Alkema	Hanze University of Applied Sciences, Netherlands
Wojciech Paszke	University of Zielona Gora, Poland
Wojciech Giernacki	Poznan University of Technology, Poland
Wolfgang Kastner	TU Wien, Austria

Contents

Optimization

An Improved Multi-Threaded Implementation of the MCSFilter Optimization Algorithm

Luís Monteiro[1], José Rufino[1(✉)], Andrey Romanenko[2], and Florbela P. Fernandes[1]

[1] Research Centre in Digitalization and Intelligent Robotics (CeDRI), Instituto Politécnico de Bragança, 5300-252 Bragança, Portugal
a36788@alunos.ipb.pt, {rufino,fflor}@ipb.pt

[2] MORE – Laboratório Colaborativo Montanhas de Investigação, Av. Cidade de Léon 506, 5300-358 Bragança, Portugal
aromanenko@morecolab.pt

Abstract. The Multistart Coordinate Search Filter (MCSFilter) is an optimization method suitable to find all minimizers of a nonconvex problem, with any type of constraints. When used in industrial contexts, execution time may be critical, in order to keep production processes within safe and expected bounds. One way to increase performance is through parallelization. In this work, a second parallel version of the MCSFilter method is presented, aiming at faster execution times than a previous parallel implementation. The new solver was tested with a set of fourteen problems, with different characteristics and behavior. The results obtained represent an improvement of the execution times over all previous MCSFilter implementations (sequential and parallel). They also allowed to identify bottlenecks to be lifted in future parallel versions.

Keywords: Optimization · MCSFilter Method · Parallelization

1 Introduction

The main goal of Multilocal Programming is to compute all the minimizers (global and local) of constrained nonlinear optimization problems [9,17], i.e.

$$\min_{x \in \Omega} f(x) \tag{1}$$

where the feasible region is given by $\Omega = \{x \in \mathbb{R}^n : l \leq x \leq u\,,\ g_i(x) \leq 0,\ i = 1, \cdots, k_1,\ h_j = 0,\ j = 1, \cdots, k_2\}$, l and u are, respectively, the lower and the upper bounds, and g_i and h_j are the constraint functions.

These problems may be simple bound problems, problems with more specific constraints, or even problems with different types of variables. For instance, problems can be nonlinear, nonconvex, with equality (or inequality) constraints and with integer, continuous, or mixed integer variables [1,2,6].

A. I. Pereira et al. (Eds.): OL2A 2022, CCIS 1754, pp. 229–245, 2022.
https://doi.org/10.1007/978-3-031-23236-7_17

Multilocal programming has a wide range of applications. Particularly, this kind of problems are recurrent in some industrial contexts (e.g., Chemical Engineering related), typically with the aim of obtaining global minimizers. Obtaining these solutions in the shortest amount of time may be of utmost importance, specially if there are time-sensitive industrial processes at play. Efficiently solving these problems can also generate economical savings for companies.

There is a considerable number of techniques and methods to solve those problems. Nevertheless, there are some problem features that should be considered when choosing the solving method. For instance, a large number of problems exists, with practical applications, where the derivatives are not known, or the function is not continuous. In this case, the correct choice of the method to solve the problem is really important [7,11].

The method addressed in this paper is MCSFilter, a derivative-free method able to obtain the minimizers (in particular, the global minimizer) of a non-convex, constrained, nonlinear programming problem. This method is based in a multistart strategy, coupled with a coordinate search filter procedure to treat the constraints. MCSFilter has already shown promising results [1,7], including with real-world applications [3,4,14,16].

MCSFilter has also been implemented in several platforms and languages, starting with a prototype in MATLAB [7] from which a JAVA implementation was derived and used to solve process engineering problems [4]. More recently, a C-based implementation was developed, also with base on the MATLAB code, which proved to be faster than both the MATLAB and JAVA versions [5].

However, all these implementations are sequential, and thus cannot take advantage of the parallel computing capabilities of modern multi-core architectures, which basically became pervasive. Indeed, even industrial-level Systems-on-Chips/embedded systems have already these capabilities in place, and so it becomes imperative to re-architect applications to fully take advantage of them. Having this in mind, a preliminary effort was already undertaken to produce a first parallel version of the C-based MCSFilter implementation [15].

This paper gives continuity to that effort. We present an enhanced parallel version of MCSFilter (ParMCSFilter2) that, like the predecessor (ParMCSFilter1), targets Linux-based multi-core shared memory systems and builds on the Posix Threads programming model. The two versions are compared for a common set of reference problems, with the new version showing a considerable performance improvement, while keeping or improving the amount of minimizers found. In addition, the new version is put to the test with a set of medium/large complexity problems, with the goal of assessing its performance and scalability. This study allowed to identify some bottlenecks to be solved in future work.

The rest of this paper is organized as follows: in Sect. 2, the original MCSFilter algorithm is briefly revised; in Sect. 3, the parallelization approach is presented; in Sect. 4, the computational experiments conducted and its results are discussed; finally, Sect. 5 concludes and addresses future work.

2 The MCSFilter Optimization Method

The MCSFilter algorithm was originally introduced in [8]. Its main purpose is to find multiple minimizers of nonlinear and nonconvex constrained optimization problems, without the use of derivatives. The approach used involves two complementary tools: the exploration feature relies on a multistart strategy; the exploitation of promising regions is based on the CSFilter algorithm.

The CSFilter procedure combines the coordinate search method with the filter methodology to treat the constraints. Using the coordinate search has the disadvantage of a slow convergence; nevertheless, it is easy to implement and suitable for problems without derivatives [11].

In the original version, the MCSFilter algorithm sequentially calls, inside the multistart part, the local procedure CSFilter. To avoid redundant local procedure calls, the MCSFilter method takes into consideration the regions of attraction of the minimizers already found. Thus, if an initial point, randomly generated, is inside one of those regions, the local procedure may not be called for that point.

The stop condition of MCSFilter is related to the fraction of the unsearched space, meaning that the algorithm stops if the following condition is met:

$$P_{min} = \frac{m(m+1)}{t(t-1)} \leq \epsilon \tag{2}$$

where m is the number of different minimizers found, t is the number of the local procedure calls and $\epsilon \in (0, 1]$. Further details can be found in [7,8].

To illustrate the application of this algorithm, the problem P_3 later presented in Sect. 4.1 is used, but only with simple bounds. The representation of the objective function can be seen in Fig. 1, where it is possible to observe the existence of one global minimum and three local minima.

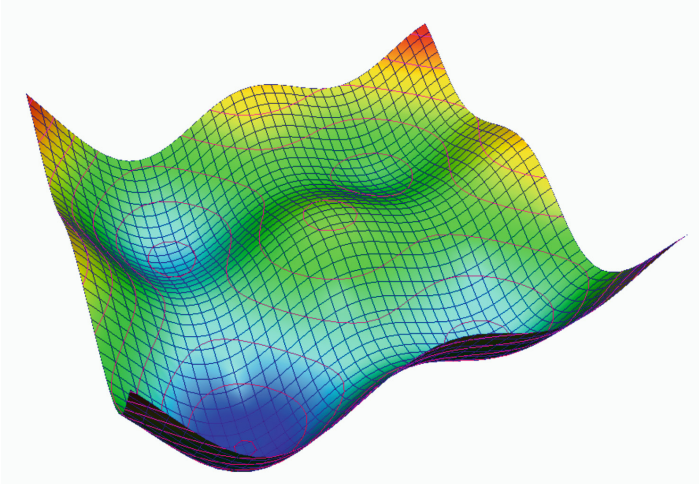
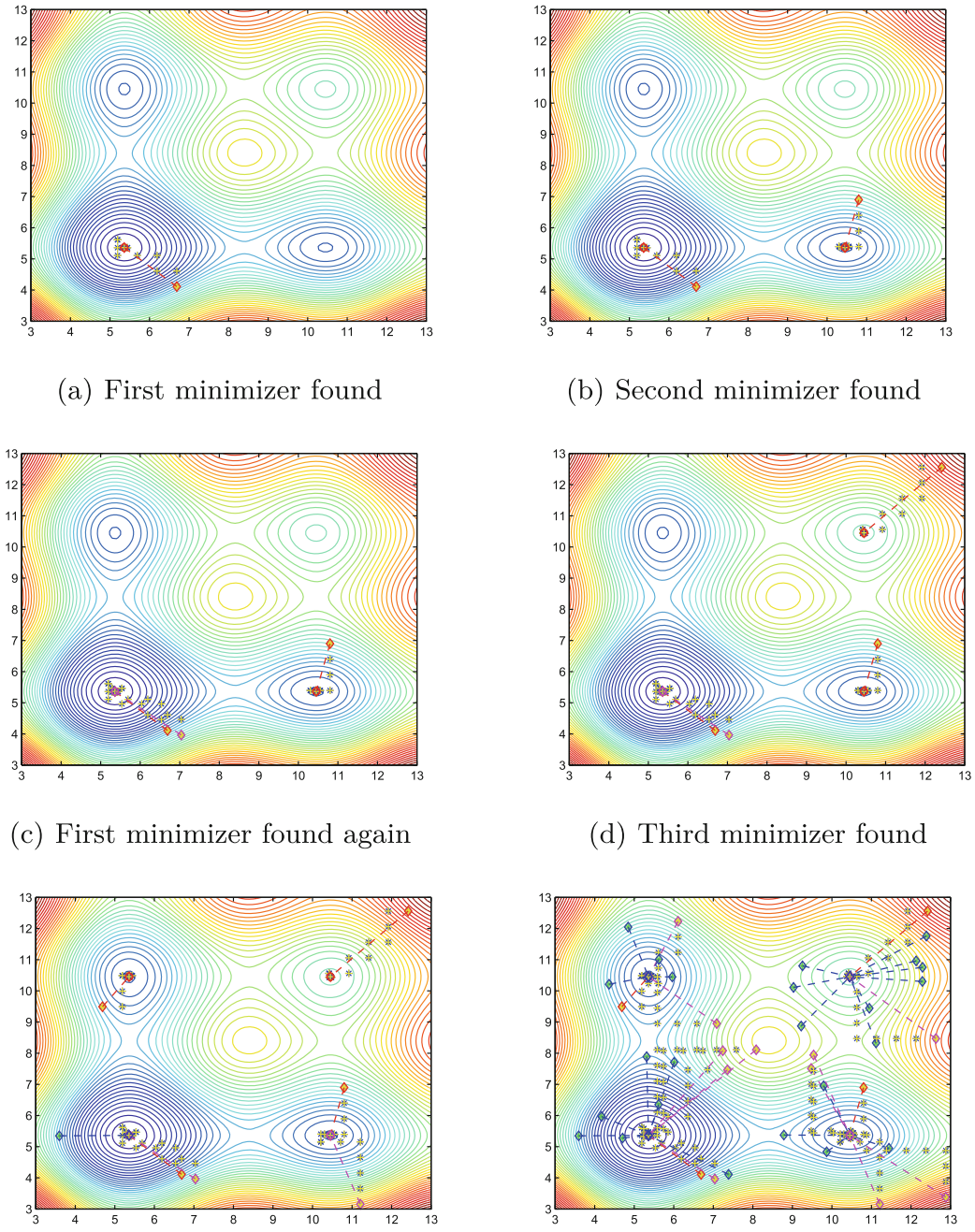


Fig. 1. Representation of the objective function of problem P_3.

Figure 2 shows, step by step, the MCSFilter algorithm when applied to obtain the global and local solutions of the above problem. The red lines represent the



(a) First minimizer found

(b) Second minimizer found

(c) First minimizer found again

(d) Third minimizer found

(e) Fourth minimizer found

(f) At the final stage

Fig. 2. Step by step illustration of MCSFilter [7].

first time the solution was found, the pink lines represent other calls that find minimizers previously obtained, and the blue lines represent calls that weren't executed taking into account the regions of attraction.

3 Parallel Approach

Given the previous description of the MCSFilter algorithm, the easiest path to its parallelization is to have multiple threads working in parallel, with each

one conducting successive local searches until it meets the stop condition. This approach is formally represented in Algorithm 1, where the main loop (lines 5 to 30) is executed in parallel by a set of worker threads, spawned by a master thread; then, the master thread basically awaits (line 32) for the workers to finish and consolidates and outputs the findings (line 33). This corresponds to the classical master-workers model, which perfectly fits this use case.

In this approach, the worker threads share some data in order to prevent repeating local searches already performed by others, and to evaluate condition 2 in a consistent way. This means there is a unique set of minimizers found, shared by all threads, and they all must evaluate condition 2 based on globally shared variables representing m and t. These shared resources must be adequately protected during concurrent access, by means of mutual exclusion mechanisms.

The reasoning behind expecting a performance gain from this approach is as follows: multiple local searches in parallel, departing from different random starting points, increase the chance of finding minimizers and finding them sooner; moreover, with more searches performed, and potentially more minimizers found, the stop condition 2 will be met sooner and so the overall MCSFilter algorithm will execute faster. It should be noted, however, that the parallel algorithm does not exclude the possibility of the same minimizer being found multiple times by different threads (but even then P_{min} decreases, as t increases).

Both parallel versions discussed in this paper share these general principles. However, the more recent version introduced several optimizations. These differences are next clarified on the sub-sections dedicated to each version.

3.1 First Version (ParMCSFilter1)

The first parallel version of MCSFilter [15] closely follows Algorithm 1, except for the steps in blue. Its implementation was derived from a first C-based implementation [5] of MCSFilter which, in turn, corresponded to a direct translation of the MATLAB primordial code, with the main focus on the correctness of the implementation and not so much on the code optimization.

As such, the data structure used to store the minimizers found was a dynamic array, that only grew (via `realloc`) one cell at a time, mimicking the MATLAB code; discarding cells was implemented by marking them with a special flag, thus avoiding shrinking the array, but incurring in some waste of memory.

The parallel variant was built on the POSIX Threads programming model, and included the necessary mutual exclusion mechanisms to protect the shared state between the threads (basically, PThreads mutex locks were used).

3.2 Second Version (ParMCSFilter2)

The second parallel version of MCSFilter (ParMCSFilter2) also follows Algorithm 1, but now includes the steps in blue, as well as several other enhancements. This version was derived from a new sequential C-based version [12],

Algorithm 1. MCSFilter algorithm – Parallel Version

Require: $M^* = \emptyset$, $k = 1$, $t = 1$, $n_{threads} > 1$;
Randomly generate $x \in [l, u]$; compute $B_{\min} = \min_{i=1,\ldots,n}\{u_i - l_i\}$;
Compute $m_1 = \textbf{CSFilter}(x)$, $R_1 = \|x - m_1\|$; Set $r_1 = 1$, $M^* = M^* \cup m_1$;
Main thread creates $n_{threads}$ worker threads
if Current thread = Worker thread **then**
  while $P_{min} > \epsilon$ **do**
    Randomly generate $x \in [l, u]$;
    Set $o = \arg\min_{j=1,\ldots,k} d_j \equiv \|x - m_j\|$;
    if $d_o < R_o$ **then**
      if $P_{min} \leq \epsilon$ **then** break **endif** // check stop condition
      if the direction from x to y_o is ascent **then**
        Set $prob = 1$;
      else
        Compute $prob = \varrho\,\phi(\frac{d_o}{R_o}, r_o)$;
      end if
    else
      Set $prob = 1$;
    end if
    if $\zeta^{\ddagger} < prob$ **then**
      if $P_{min} \leq \epsilon$ **then** break **endif** // check stop condition
      Compute $m = \textbf{CSFilter}(x)$; set $t = t + 1$;
      if $P_{min} \leq \epsilon$ **then** break **endif** // check stop condition
      if $\|m - m_j\| > \gamma^* B_{\min}$, for all $j = 1, \ldots, k$ **then**
        Set $k = k+1$, $m_k = m$, $r_k = 1$, $M^* = M^* \cup m_k$; compute $R_k = \|x - m_k\|$;
      else
        Set $R_l = \max\{R_l, \|x - m_l\|\}$; $r_l = r_l + 1$;
      end if
    else
      Set $R_o = \max\{R_o, \|x - m_o\|\}$; $r_o = r_o + 1$;
    end if
  end while
else
  Main thread waits for all Worker threads to finish
  Main thread consolidates and outputs results
end if

that used as reference the JAVA implementation of the MCSFilter method. This JAVA version already included some improvements and refinements at the algorithmic level. Moreover, being much more close to the C syntax than the MATLAB version, its conversion to C was much more straightforward. Also, the main data structure, used to store the minimizers found, was redesigned: it still is based on a growing dynamic array, but the array grows by a chunk of cells at a time, making its use much more efficient, as shown in [12]. The decision to stick with an array as the main data structure is due to it being a cache-friendly data structure. This decision, however, introduced unintended consequences for the performance of the new parallel version, as later found (see Sect. 3.2).

Similarly to the first parallel version, the new implementation was also produced by making use of the POSIX Threads programming model, that targets shared-memory multi-core systems. Besides being derived from a different sequential version, several differences and enhancements deserve to be mentioned.

As already stated, the new version includes the steps in blue from Algorithm 1. These correspond to new evaluations of the stop condition, in addition to the one performed at the head of the main loop (step 5), with the goal of allowing worker threads to end as soon as possible, provided the stop condition is met.

Additionally, several shared resources (e.g., statistical counters, etc.) that were protected by a single mutex lock, in the old version, were unfolded into separate instances, per thread, allowing independent and simultaneous access, without contention. The only downturn was the need for the master thread to sequentially post-process those resources, but the additional time spent in this activity was largely offset by the time spared by the worker threads.

Lastly in order to further improve performance and tackle a bottleneck identified in the access to shared structures, specially to the array of minimizers, all PThreads mutex lock objects were replaced by PThreads read-write locks, allowing full parallel read-access, while enforcing exclusive write access.

Together, all these changes translated in tangible performance gains, as shown in the comparison performed between the two solvers – see next section. However, as often happens when optimizing an application, removing a bottleneck may expose new ones in other areas. During the discussion of the performance evaluation results the bottleneck identified in this new implementation will be detailed and a possible solution to it will be presented.

4 Computational Experiments

This section is devoted to the experimental evaluation of both parallel solvers, with the selected set of benchmark problems. The definition of each problem is given below, along with the experimental evaluation conditions, as well as the obtained results (both numerical and performance-related).

4.1 Benchmark Problems

A total of 14 benchmark problems were chosen from [7,13] (and the references therein). The problems were selected so that different characteristics were addressed: they are multimodal problems with more than one minimizer; they can have just one global minimizer or more than one global minimizer; the dimension of the problems varies between 2 and 310. These are the selected problems:

—Problem (P_1)

dimension: $n = 10$; known global minimum: $f^* = -391.6581$

$$\min f(x) = \frac{1}{2} \sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i$$

$$\text{s.t. } -5 \leq x_i \leq 5, i = 1, ..., n$$

—Problem (P_2)
dimension: $n = 2$; known global minimum: $f^* = 0.75$

$$\min f(x) = x_1^2 + (x_2 - 1)^2$$

$$\text{s.t. } x_2 - x_1^2 = 0$$

$$-1 \leq x_i \leq 1, i = 1, 2$$

—Problem (P_3)
dimension: $n = 2$; known global minimum: $f^* = -2.4305$

$$\min f(x) = \sum_{i=1}^{n} \sin(x_i) + \sin\left(\frac{2x_i}{3}\right)$$

$$\text{s.t. } -2x_1 - 3x_2 + 27 \leq 0$$

$$3 \leq x_i \leq 13, i = 1, 2$$

—Problem (P_4)
dimension: $n = 2$; known global minimum: $f^* = -64.1956$

$$\min f(x) = \frac{1}{2} \sum_{i=1}^{2} x_i^4 - 16x_i^2 + 5x_i$$

$$\text{s.t. } (x_1 + 5)^2 + (x_2 - 5)^2 - 100 \leq 0$$

$$-x_1 - x_2 - 3 \leq 0$$

$$-5 \leq x_i \leq 5, i = 1, 2$$

—Problem (P_5)
Problem P_1 with dimension: $n = 14$; known global minimum: $f^* = -548.3261$

—Problem (P_6)
dimension: $n = 300$; known global minimum: $f^* = 0$

$$\min f(x) = \sum_{i=1}^{n-1} \left[x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7\right]$$

$$\text{s.t. } -15 \leq x_i \leq 15, i = 1, ..., n$$

—Problem (P_7)
dimension: $n = 5$; known global minimum: $f^* = 0$

$$\min f(x) = \begin{cases} \sum_{i=1}^{n} x_i^6 \left[2 + \sin\left(\frac{1}{x_i}\right)\right], \prod_{i=1}^{n} x_i \neq 0 \\ 0 \qquad\qquad\qquad\qquad\quad , \prod_{i=1}^{n} x_I = 0 \end{cases}$$

$$\text{s.t. } -1 \leq x_i \leq 1, i = 1, ..., n$$

—Problem (P_8)
dimension: $n = 5$; known global minimum: $f^* = -1$

$$\min f(x) = -exp\left(-0.5 \sum_{i=1}^{n} x_i^2\right)$$

$$\text{s.t. } -1 \leq x_i \leq 1, i = 1, ..., n$$

—Problem (P_9)
dimension: $n = 50$; known global minimum: $f^* = 1$

$$\min f(x) = \prod_{i=0}^{n-1} \left[1 + (i+1) \sum_{k=1}^{32} \left\lfloor 2^k x_i \right\rfloor 2^{-k}\right]$$

$$\text{s.t. } 0 \leq x_i \leq 100, i = 1, ..., n$$

—Problem (P_{10})
dimension: $n = 310$; known global minimum: $f^* = 2$

$$\min f(x) = (1 + x_n)^{x_n} \; ; \; x_n = n - \sum_{i=1}^{n-1} x_i$$

$$\text{s.t. } 0 \leq x_i \leq 1, i = 1, ..., n$$

—Problem (P_{11})
dimension: $n = 150$; known global minimum: $f^* = 0$

$$\min f(x) = \sum_{i=1}^{n-1} \left[100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\right]$$

$$\text{s.t. } -5 \leq x_i \leq 10, i = 1, ..., n$$

—Problem (P_{12})
dimension: $n = 5$; known global minimum: $f^* = 0$

$$\min f(x) = 1 - \cos\left(2\pi \sqrt{\sum_{i=1}^{n} x_i^2}\right) + 0.1 \sqrt{\sum_{i=1}^{n} x_i^2}$$

$$\text{s.t. } -100 \leq x_i \leq 100, i = 1, ..., n$$

—Problem (P_{13})
dimension: $n = 300$; known global minimum: $f^* = 0$

$$\min f(x) = \left(\sum_{i=1}^{n} x_i^2\right)^{\sqrt{\pi}}$$

$$\text{s.t. } -100 \leq x_i \leq 100, i = 1, ..., n$$

—Problem (P_{14})
dimension: $n = 5$; known global minimum: $f^* = 0$

$$\min f(x) = \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i) + 10\right]$$

$$\text{s.t. } -5.12 \leq x_i \leq 5.12, i = 1, ..., n$$

Problems P_1 to P_4 were already considered in the preliminary evaluation of the old parallel solver [15] (ParMCSFilter1) and are now considered again to allow for a comparison with the new parallel solver (ParMCSFilter2). The 10 newly introduced problems, P_5 to P_{14}, are to be all solved only by the new solver. They are all multidimensional and the definition of the dimension on each one was made to ensure an execution time from a few seconds to up to around 1 min (small, or very small execution times are usually not appropriate for a scalability study). The visualization of some of these problems, with $n = 2$, can be seen in https://www.al-roomi.org/benchmarks, where it is possible to observe the hard problems that were chosen to be solved.

4.2 Experimental Conditions

The problems were evaluated in the same computational system used in previous related works [5,15] – a virtual machine in the CeDRI cluster with 16 cores from an Intel Xeon W-2195 CPU, 32 GB of RAM, Linux Ubuntu 20.04 Operating System and GNU C Compiler (gcc) version 9.3.0 (optimization level -O2 used).

All problems were solved 100 times, each time with a different random number generator seed per each thread used. Such a number of executions, all with different seeds, prevents any possible biasing in the results. The execution times presented are an average of the 100 executions, ignoring the first one. The benchmark process was fully automated, with the help of BASH scripts.

4.3 Solvers Comparison

In this section, the previous (ParMCSFilter1) [15] and the new (ParMCSFilter2) multithreaded implementations of MCSFilter are compared for problems P_1 to P_4. Again, these problems were selected due to being the same used for the validation of the preliminary ParMCSFilter1 solver [15]. For this comparison, both implementations shared the following common parameters:

- Fraction of the unsearched space (MCSFilter stop condition): $\epsilon = 10^{-1}$ (Eq. 2);
- Newly found minimizers (line 22 of the algorithm): $\gamma^* = 10^{-1}$;
- Local search CSFilter stop condition ($\alpha < \alpha_{min}$): $\alpha_{min} = 10^{-5}$.

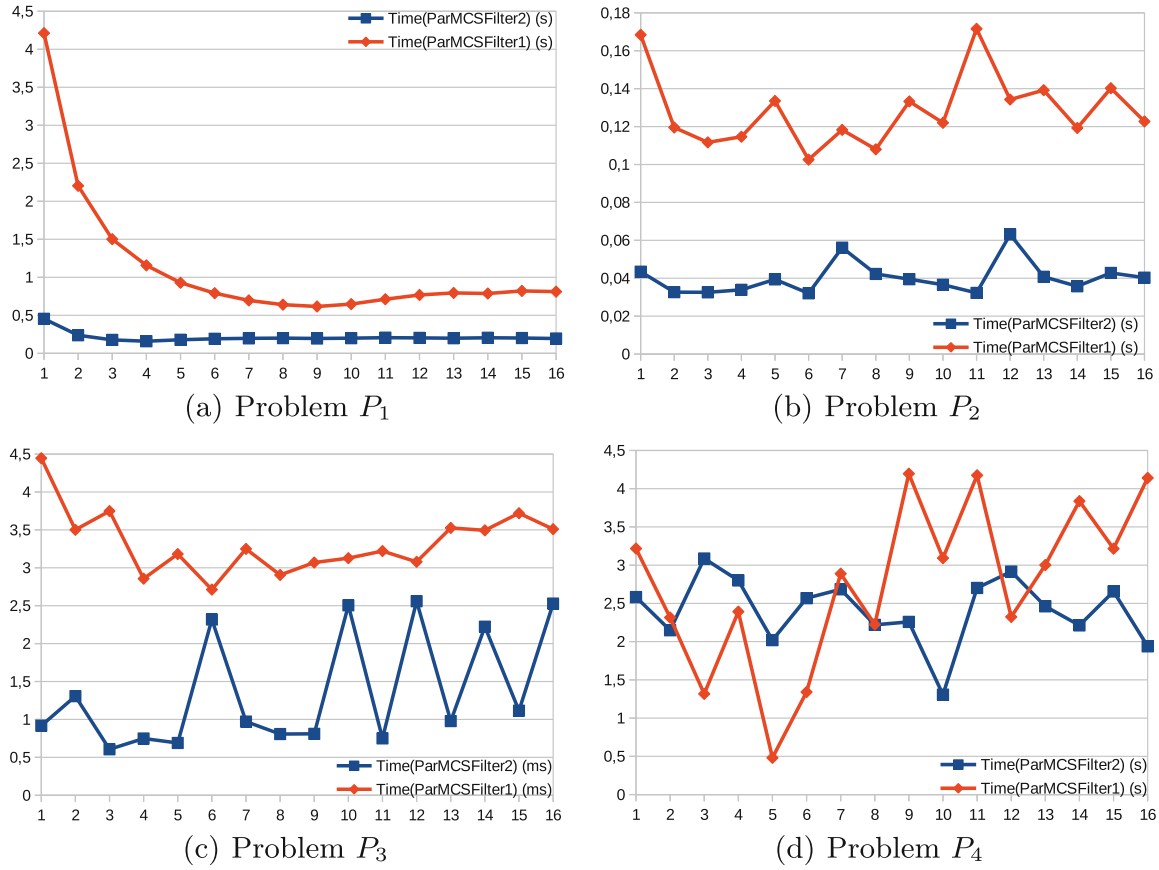


Fig. 3. ParMCSFilter1 vs ParMCSFilter2: execution time for problems P_1–P_4

Table 1. Speedups of the new MCSFilter parallel solver against the old solver.

Threads (n)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Prob P_1	**9.3**	**9.3**	8.5	7.3	5.2	4.1	3.5	**3.2**	**3.2**	**3.2**	3.5	3.8	4.0	3.8	4.1	4.2
Prob P_2	**3.9**	3.7	3.4	3.4	3.4	3.2	**2.1**	2.6	3.4	3.3	5.3	**2.1**	3.4	3.3	3.3	3.0
Prob P_3	4.8	2.7	**6.2**	3.8	4.6	**1.2**	3.3	3.6	3.8	**1.2**	4.3	**1.2**	3.6	1.6	3.3	1.4
Prob P_4	1.2	1.1	0.4	0.9	**0.2**	0.5	1.1	1.0	1.9	**2.4**	1.5	0.8	1.2	1.7	1.2	2.1

Figure 3 shows the execution time of both solvers, for the common set of problems, with a different number of threads. All times plotted are measured in seconds, except for problem P_3, for which times are represented in milliseconds.

Except for problem P_4, the performance gains introduced by the new parallel solver are very noticeable. Table 1 shows the speedups involved, corresponding to the ratio Time(ParMCSFilter1)/Time(ParMCSFilter2). These speedups vary, depending on the problem and number of threads. Problem P_1 is the one where the new solver introduces higher speedups (between 3.2 and 9.3), while in problem P_4 it may be marginally faster or visibly slower (speedup of 2.4 vs 0.2).

The execution times exhibit a different evolutionary pattern as the number of threads increase. For problem P_1, that pattern is smooth and matches the expectations. But for the other problems it becomes very irregular. The irregularities for problems P_2, P_3 and P_4 can be explained due to these being problems

with equality (P_2) and inequality (P_3 and P_4) constraints in addition to simple bounds. More specifically, for problem P_4 (the one with more extreme variations), due to the complexity introduced by the constraints and the randomness introduced by the algorithm, we observed individual execution times between 0.001157 and 104.779081 s using the old solver, while in the new solver that range was considerably tighter (between 0.000747 and 40.638619 s). It should be noted that for both solvers these times are not directly related to the number of threads used, meaning there could be extremely large or small execution times with any number of threads. With this in mind, for problem P_4, even though the new solver does not translate in a significant performance improvement, the multithreaded behaviour is nevertheless more stable.

It is also fair to say that the optimizations introduced in the code of the new solver make it more efficient to the extent that it became harder to collect additional benefits from parallelization. A significant amount of the execution time in each thread is now spent in the CSFilter local search (line 20 of Algorithm 1), that is intrinsically sequential (non-parallelizable), and in changing the shared set M^* of minimizers (line 23), which can only be done by one thread at a time.

Table 2. Number of minimizers found by each solver vs number of known minimizers.

			Threads (n)							
Problem	Known Mins	Solver	1	2	3	4	5	6	7	8
P_1	**1024**	**Old**	884.8	884.5	881.5	880.0	877.1	878.8	878.5	875.3
		New	1002.9	1003.1	998.0	995.2	994.3	997.7	998.5	993.8
P_2	**2**	**Old**	4.1	4.2	4.2	4.4	4.3	4.5	4.6	4.8
		New	2.4	2.4	2.3	2.3	2.3	2.3	2.3	2.5
P_3	**4**	**Old**	3.9	4.0	3.9	4.1	4.1	4.1	4.1	4.1
		New	3.99	4.0	3.98	4.0	3.99	3.99	4.01	4.0
P_4	**5**	**Old**	4.3	4.4	4.5	4.6	4.6	4.7	4.8	4.7
		New	4.8	4.8	4.8	4.7	4.7	4.7	4.8	4.6

(a) Number of threads: n=1...8

		Threads (n)								
Problem	Solver	9	10	11	12	13	14	15	16	Average Mins
P_1	**Old**	875.4	871.2	868.1	873.3	876.1	872.6	873.9	876.2	876.7 (85.6%)
	New	992.4	999.4	998.6	999.1	999.9	999.8	1001.7	1002.0	998.5 (97.5%)
P_2	**Old**	4.7	4.9	4.9	4.8	5.0	4.9	5.0	4.6	4.6 (230.4%)
	New	2.3	2.4	2.4	2.4	2.6	2.3	2.4	2.5	2.4 (118.4%)
P_3	**Old**	4.1	4.1	4.1	4.2	4.2	4.1	4.2	4.2	4.1 (102.3%)
	New	4.01	4.0	3.98	4.01	4.01	4.01	3.99	4.0	4.0 (100%)
P_4	**Old**	4.7	4.9	4.9	4.9	4.9	5.0	4.9	5.0	4.8 (95%)
	New	4.7	4.7	4.6	4.6	4.7	4.5	4.7	4.5	4.7 (93.8%)

(b) Number of threads: n=9...16

To finish the comparison between the solvers, Table 2 shows the average number of minimizers found by each one, for each problem, with a different thread number. The table also contains the number of known minimizers (Known Mins)

and a global average (Average Mins) of the specific averages of each number of threads; this global average is both presented as an absolute value and also as a relative (%) value against the Known Mins. It can be concluded, for both solvers, that the number of threads used doesn't have a systematic influence on the amount of minimizers found (e.g., that amount may grow or shrink with the number of threads). However, the new solver seems to better avoid overestimating the number of minimizers found (problems P_2 and P_3), while improving (problem P_1) or keeping the same findings as the old solver (problem P_4).

4.4 Performance of the New Solver with Demanding Problems

In this section the performance of the new implementation is assessed when solving problems P_5 to P_{14}. As previously stated, these are multidimensional problems that take some time to solve and so are deemed more appropriate to be handled by a parallel solver, once it should be able to generate more sensible performance gains than with problems that are not so demanding.

Figure 4 and Fig. 5 present the execution time of problems P_5 to P_{14}, with a number of threads ranging from 1 to 16. The results are split in two different charts, to allow a better visualization and reflect the type of evolution (regular/irregular) of execution times as the number of threads used grows.

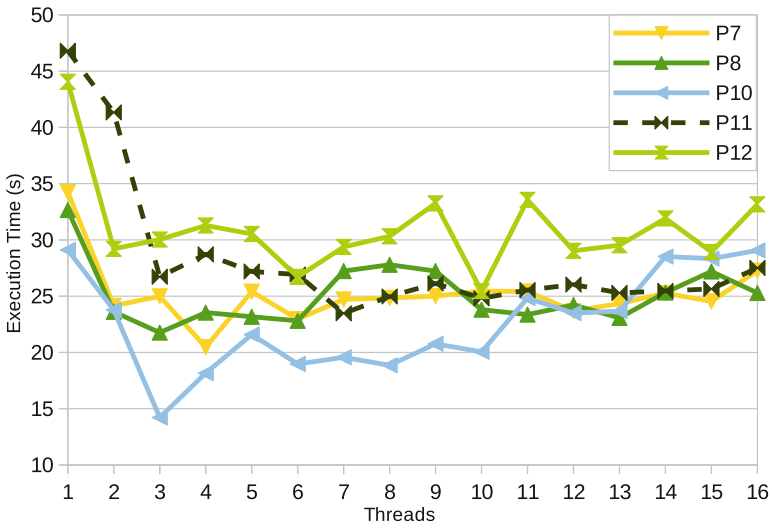


Fig. 4. ParMCSFilter2: execution time for problems P_7, P_8, P_{10}, P_{11}, P_{12}

As may be observed in both figures, the execution time (T_n) decreases with the number of threads used (n), in general up to 3 threads. But further increasing the number of threads does not translate in lower execution times, with very few exceptions (P_7 with 4 threads, P_9 with 5 threads, P_{11} with 7 threads, and P_{12} with 10 threads) that, however, benefit very little from the extra number of threads. These exceptions may be easily identified in Table 3, corresponding to problems with very poor parallel efficiency ($< 50\%$). The table shows the maximum speedups ($S_n = T_1/T_n$) achieved for each problem, and the corresponding

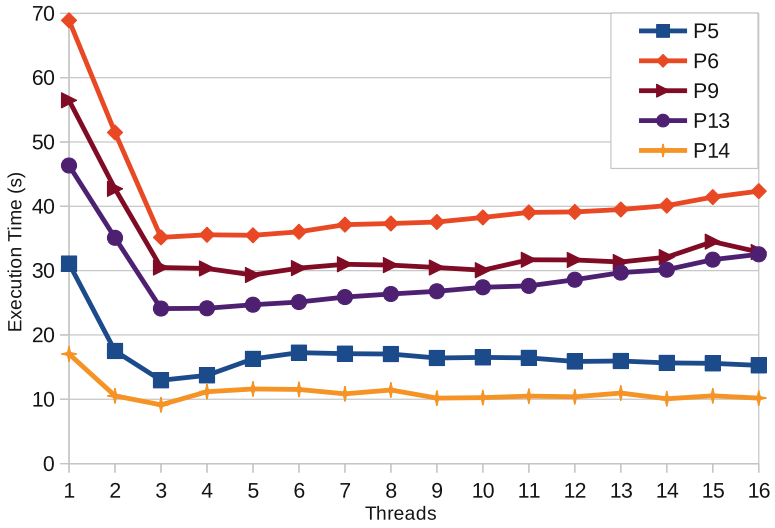


Fig. 5. ParMCSFilter2: execution time for problems P_5, P_6, P_9, P_{13}, P_{14}

number of threads (n) and parallel Efficiency ($E_n = S_n/n$). Only problems for which the best speedup is reached with 3 threads are able to ensure a parallel efficiency of at least 50%. Moreover, the speedups achieved for these problems are very modest, ranging from 1.5 to 2.4. Clearly, a serious bottleneck is at play.

Table 3. ParMCSFilter2: Best Speedups achieved and respective number of threads.

	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}
Number of threads (n)	3	3	4	3	5	3	7	10	3	3
Maximum speedup (S_n)	2.4	1.96	1.67	1.5	1.93	2.05	1.99	1.73	1.92	1.87
Parallel efficiency (E_n)	80%	65%	42%	50%	39%	68%	28%	17%	64%	62%

Through some investigation it was possible to identify the major bottleneck in the new solver, explaining the sudden loss in performance with more than 3 worker threads. The bottleneck derives from the decision brought up in Sect. 3.2, to keep using a growing dynamic array as the main data structure to store the minimizers found (albeit with some modifications). As it was found, the main problem identified with this this approach is the high contention suffered by worker threads when trying to access the shared array, despite replacing the mutex lock of the old solver with a read-write lock in the new one.

There are several reasons for this. Firstly, it's an unsorted array and the algorithm needs to compare every single member with a new minimizer candidate, having to perform a write either once it finds a match or adding it to the end of the array once it's done iterating it. This $O(n)$ search may end up to be very costly, specially for problems with many possible minimizers. Secondly, the competition between threads for read/write access rights to the array becomes more intense as the number of threads grows, which further aggravates the problem.

Thus, even though the local search method is executed in parallel by the worker threads, execution times are strongly penalized by the need for exclusive access to shared data. A possible solution to this problem is to replace the dynamic array by a lock-free sorted data structure, like lock-free Binary Search Trees [10]. Although less cache-friendly, such data structure brings with it at least two benefits which fit our current problems: the binary search represents a much better time complexity compromise $O(log(n))$; it solves (or minimizes) the read/write access contention problem. This will be pursued in future work.

Finally, it was also found that the parallel algorithm may have a limitation regarding the unaccounted redundant searches introduced by the parallelism to local searches, as mentioned in Sect. 3. The reason for this is that there may be multiple threads executing a local search for the same minimizer simultaneously, without before taking advantage of the attraction radius, as it is only updated at the end of each local search. Therefore, as more threads are employed, more redundant searches will occur, resulting in more total time spent processing each local search. A solution for this could be the introduction of a new mechanism in the algorithm to reduce currently unaccounted redundant searches.

5 Conclusions and Future Work

Due to the nature of the local procedure (a pattern search method), the MCSFilter algorithm may take considerable time to converge to the solution. With this in mind, the original MATLAB implementation was re-coded in JAVA and later in C, with every new implementations showing increased performance over the previous one. In the quest for more performance, parallelization was the next logical step. After a first attempt laid out the foundation of the parallel approach to be taken [15], this paper presented a refinement of the approach, which exhibits better performance than the first parallel solver (that already surpassed all sequential versions), while keeping or improving the number of minimizers found. Moreover, the new parallel solver was shown to solve specially demanding problems in half the time of their sequential execution. And, even though some scalability constraints were found with more than 3 worker threads, the current parallel solver is certainly able to take advantage of low core-count computing systems, like those usually found in industrial-level/embedded systems.

In the future, aiming to overcome the scalability bottlenecks identified, the shared data structure used to store the minimizers will be replaced with a lock-free alternative. It is expected that such change will provide the desirable performance and scalability levels for a parallel multilocal programming solver.

Other possible parallelization approaches will also be investigated, namely based on the domain decomposition of the search space. From a performance perspective, these are usually very promising. However, this would result in the finding of possible minimizers in each sub-domain that could be false positives and would need further validation (their exclusion could be done by checking their compliance with the problem restrictions within the admissible region).

We will also revisit the MCSFilter algorithm in order to assess the possibility of a parallel implementation using many-core co-processors, namely GP-GPUs.

Acknowledgements. This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: UIDB/05757/2020.

References

1. Abhishek, K., Leyffer, S., Linderoth, J.: Filmint: an outer-approximation-based solver for convex mixed-integer nonlinear programs. Inf. J. Comput. **22**, 555–567 (2010)
2. Abramson, M., Audet, C., Chrissis, J., Walston, J.: Mesh adaptive direct search algorithms for mixed variable optimization. Optim. Lett. **3**, 35–47 (2009)
3. Amador, A., Fernandes, F.P., Santos, L.O., Romanenko, A., Rocha, A.M.A.C.: Parameter estimation of the kinetic α-Pinene isomerization model using the MCS-Filter algorithm. In: Gervasi, O., et al. (eds.) ICCSA 2018. LNCS, vol. 10961, pp. 624–636. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95165-2_44
4. Amador, A., Fernandes, F.P., Santos, L.O., Romanenko, A.: Application of mcs-filter to estimate stiction control valve parameters. AIP Confer. Proc. **1863**(1), 2700051–2700054 (2017)
5. Araújo, L., Pacheco, M.F., Rufino, J., Fernandes, F.P.: Towards a high-performance implementation of the mcsfilter optimization algorithm. In: Pereira, A.I., et al. (eds.) Optimization, Learning Algorithms and Applications, pp. 15–30. Springer International Publishing, Cham (2021)
6. Eronen, V.-P., Westerlund, T., Mäkelä, M.M.: On mixed integer nonsmooth optimization. In: Bagirov, A.M., Gaudioso, M., Karmitsa, N., Mäkelä, M.M., Taheri, S. (eds.) Numerical Nonsmooth Optimization, pp. 549–578. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-34910-3_16
7. Fernandes, F.P.: Programação não linear inteira mista e não convexa sem derivadas. Ph.D. thesis, Univ. of Minho, Braga, Portugal (2014)
8. Fernandes, F.P., Costa, M.F.P., Fernandes, E.M.G.P.: Multilocal programming: a derivative-free filter multistart algorithm. In: Murgante, B., et al. (eds.) ICCSA 2013. LNCS, vol. 7971, pp. 333–346. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39637-3_27
9. Floudas, C.: Recent advances in global optimization for process synthesis, design and control: enclosure of all solutions. Comput. Chem. Eng. **963**, 963–973 (1999)
10. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: Blelloch, G.E., Herlihy, M. (eds.) 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, Pittsburgh, PA, USA, June 25–27, 2012, pp. 161–171. ACM (2012)
11. Kolda, T., Lewis, R., Torczon, V.: Optimization by direct search: new perspectives on some classical and modern methods. SIAM Rev. **45**, 85–482 (2003)
12. Monteiro, L., Rufino, J., Romanenko, A., Fernandes, F.P.: MCSFilter performance: a comparison study, pp. 3–5. Book of Abstracts of SASYR 2022, F. P. Fernandes, P. Morais and P. Pinto (eds.) (2022)
13. Murray, W., Ng, K.M.: Handbook of global optimization. In: Pardalos, P.M., Romeijn, H.E., (eds.) Algorithms for Global Optimization and Discrete Problems based on Methods for Local Optimization, vol. 2, pp. 87–114. (2002)
14. Romanenko, A., Fernandes, F.P., Fernandes, N.C.P.: PID controllers tuning with mcsfilter. In: AIP Conference Proceedings, vol. 2116, p. 220003 (2019)
15. Rufino, J., Araújo, L., Pacheco, M.F., Fernandes, F.P.: A multi-threaded parallel implementation of the mcsfilter optimization algorithm. In: AIP Conference Proceedings. in press (2022)

16. Seiça, J.C., Romanenko, A., Fernandes, F.P., Santos, L.O., Fernandes, N.C.P.: Parameter estimation of a pulp digester model with derivative-free optimization strategies. In: AIP Conference Proceedings, vol. 1863, no. 1, p. 270006 (2017)
17. Yang, X.S.: Optimization Techniques and Applications with Examples. Wiley (2018)