Scaling the SOO Global Blackbox Optimizer on a 128-core Architecture

David Redon, Pierre Fortin, Bilel Derbel

▶ To cite this version:

David Redon, Pierre Fortin, Bilel Derbel. Scaling the SOO Global Blackbox Optimizer on a 128-core Architecture. HiPC 2022 - IEEE 29th International Conference on High Performance Computing, Data, and Analytics, Dec 2022, Bangalore, India. pp.203-213, ⟨10.1109/HiPC56025.2022.00037⟩. ⟨hal-03927251⟩

HAL Id: hal-03927251

https://hal.science/hal-03927251v1

Submitted on 4 Oct 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Scaling the SOO Global Blackbox Optimizer on a 128-core Architecture

David Redon*, Bilel Derbel* and Pierre Fortin†
* Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
† Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
{ david.redon , bilel.derbel , pierre.fortin } @univ-lille.fr

***Abstract*—Blackbox optimization refers to the situation where no analytical knowledge about the problem is available beforehand, which is the case in a number of application fields, e.g., multi-disciplinary design, simulation optimization. In this context, the so-called Simultaneous Optimistic Optimization (SOO) algorithm is a deterministic tree-based global optimizer exposing theoretically provable performance guarantees under mild conditions. In this paper, we consider the efficient shared-memory parallelization of SOO on a high-end HPC architecture with dozens of CPU cores. We thereby propose different strategies based on eliciting the possible levels of parallelism underlying the SOO algorithm. We show that the naive approach, performing multiple evaluations of the blackbox function in parallel, does not scale with the number of cores. By contrast, we show that a parallel design based on the SOO-tree traversal is able to provide substantial improvements in terms of scalability and performance. We validate our strategies with a detailed performance analysis on a compute server with two 64-core processors, using a number of diverse benchmark functions with both increasing dimensions and number of cores.**

***Index Terms*—Parallel Optimization, Blackbox Optimization, Global Optimization, Multi-Threading, Many-Core CPU**

I. Introduction

Optimization problems are ubiquitous to countless modern engineering and scientific applications and imply increasingly sophisticated and compute intensive dedicated algorithms. They can be classified into different categories depending on different criteria such as their domain, their computational complexity, the amount of information that can be made available for a given solver, etc. Generally speaking, in this work, we are interested in tackling blackbox continuous optimization problems while taking benefits from the increasingly available parallel and HPC compute facilities.

More precisely, we assume given a function of n real-valued variables, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, modeling the optimization problem. The goal is to compute a solution $\mathbf{x} \in \mathbb{R}^n$, such that $f(\mathbf{x})$ is minimized (or maximized). The function f is assumed to be blackbox, that is, nothing about the function f can be assumed before hand, such as, its derivatives, or any mathematical or structural information that might be used a priori by a search process, such as, smoothness, convexity, etc. In such a setting, a blackbox optimization algorithm can only probe the function f for some input $\mathbf{x}$, receives a response $f(\mathbf{x})$ which is the fitness value of solution $\mathbf{x}$, and advances the search accordingly. Blackbox optimization problems have been extensively studied since they are of special interest in a number of application fields, where no clear information about the problem can be available. This is indeed the case in engineering problems that undergo some complex models, or some numerical simulation. For example, in aeronautic where a simulation of the air flow around a vehicle to optimize aerodynamics is mandatory, in nuclear physics with the diffusion of heat or particles in a vat to find optimal dimensions, in public transportation where the traffic in a simulated city depends on the behavior of stoplights, etc. For such problems, traditional numerical gradient-based optimization is not applicable, and different classes of blackbox optimization algorithms, also referred to as derivative-free algorithms, were developed [1], [2].

In this paper, we are specifically interested in the so-called Simultaneous Optimistic Optimization (SOO) algorithm [3]. Our interest in the SOO algorithm stems from two facts. Firstly, SOO is a global optimizer having well-established theoretical background from the machine learning community, and exposing the unique property of providing strong quality guarantees under very mild conditions. Secondly, it is a relatively simple algorithm which, like any 'exact' search algorithm, can benefit from the increasingly available HPC compute resources. However, to the best of our knowledge, there are no investigations on the effective parallelization of SOO on high-end HPC servers, which is precisely the purpose of this paper. Before going into further considerations, let us first briefly discuss the working principles of SOO and the benefits of leveraging it with high-performance computing.

The theoretical background of SOO was introduced in [3], [4], and its practical relevancy studied in a number of works, e.g., [5], [6]. SOO is a deterministic tree-based global optimizer. In contrast to a local optimizer, which may be trapped in locally optimum solutions, a global optimizer is able to search for global (exact) optima. In particular, the SOO algorithm provides the guarantee that the global minimum (or maximum) can be deterministically approached to an arbitrarily small constant. SOO is based on partitioning the problem domain (or the search space) into sub-domains of smaller size, called cells. As it will be detailed later, the different cells are maintained using a tree, so that at each iteration, the algorithm consists in traversing the tree and deciding which cells to divide. The tree maintained by SOO has a global nature in the sense that it progressively refines the knowledge about the whole search space, and gradually explore all interesting regions (cells).

The design of SOO is inspired by the Bandit theory from reinforcement learning, since the main question is to efficiently decide which cells to expand in order to approach a high-quality solution efficiently, and at the same time, to avoid being trapped into locally optimum cells. This is actually known as the exploration / exploitation dilemma. Due to its global search ability, SOO has an arguably analytically proven performance if the function is smooth around its global optimum. Having a proven convergence rate under such weak assumptions is unique, which is why SOO was leveraged in a number of applications and problems, e.g. [7]–[12].

Interestingly, the global nature of SOO makes it particularly appealing to be deployed and studied in a parallel compute environment. In fact, maintaining the SOO tree may benefit from the availability of high-end HPC servers, and the underlying search could be substantially accelerated. However, designing efficient parallel strategies for dealing with the computational flow of SOO has not been addressed in the past. It is worth-noticing that a lot of techniques have been derived to parallelize tree-based search algorithms coming from different fields. For instance, Branch-&-Bound tree search for combinatorial problems have been extensively parallelized on different computing systems [13]. Besides, in tree-based machine learning related techniques, such as Gradient Boosting (GBM) [14], there are a number of efforts for a better parallel scalability [15], [16]. However, such techniques cannot be transferred to the case of blackbox numerical optimization. For instance, GBM trees are decision trees and not partitions of a search space. Similarly, B&B has different specific tree traversal policies. As such, parallelization methods cannot be replicated from GBM or B&B to SOO. Moreover, it is not fully clear how to derive an efficient parallel variant of SOO due to the interdependence between the tree traversal and the cell partition policy. This can hence raise difficult challenges especially when considering high-end HPC servers with increasing number of CPU cores. Indeed, since the mid-2000s, the number of cores per processor has kept increasing. Nowadays, high-end CPUs can have several dozens of cores. As examples, the ARM A64FX processor, which equips Fugaku (among the most powerful supercomputers in the latest TOP500 list [17]) has 48 cores (plus 4 assistant cores), the Intel Xeon Platinum 9282 processor has 56 cores, and the AMD Rome and Milan EPYC 77XX processors, as well as the ARM Graviton3 processor, offer 64 cores. The new Ampere Altra Max M128-30 even has 128 cores. Using multiple sockets, a high-end HPC server can thus offer more than one hundred CPU cores. It is however much more challenging for a parallel algorithm to scale on a hundred CPU cores than on a few ones. Thus, the main objective of this paper is to design a parallel SOO algorithm that scales on dozens of CPU cores, using shared-memory parallelism within one HPC server. As such, we describe the following contributions.

- We first introduce a naive parallel strategy for SOO, where multiple evaluations of the blackbox function are performed in parallel among the threads (PE – Parallel Evaluations). This strategy is also finely improved and optimized with respect to other issues, dealing with parallel memory allocations and parallel heap operations.
- We then leverage a second, less obvious, parallelism level to fully parallelize the computational flow of SOO based on its tree traversal (PT – Parallel Traversals). This strategy is further improved by allowing the swap of traversals, eventually reducing the threads waiting times.
- Finally, we present the results of a detailed experimental study in order to fairly compare the relative performance of the different designed parallel SOO variants. Using the BBOB (Black-Box Optimization Benchmarking) [18] benchmark and varying the dimensions, as well as the number of threads, we analyze the parallel speedups delivered by each parallel SOO variant (using C programming with OpenMP) on top of a 128-core architecture. In particular, we show that the PT versions outperform the PE ones, and are able to scale up to 128 CPU cores.

The rest of this paper is organized as follows. In Section II, we give some background on SOO. In Section III, we describe the proposed parallel strategies. In Section IV, we report our experimental analysis. In Section V, we conclude the paper.

II. SOO in a Nutshell

The SOO (Simultaneous Optimistic Optimization) algorithm was introduced in [3]. SOO is a global blackbox optimizer. Global optimizers are to be contrasted with local optimizers. Roughly speaking, local optimizers iteratively improve on a solution by choosing among a set of neighboring ones. A global optimizer keeps a more global information about the whole search space and progressively refines it. In this context, SOO enjoys an analytically proven performance for functions that are locally smooth around their global optimum. That is, functions that do not vary too much around at least one of their global optima. More precisely, after a budget of N calls to the function evaluation f, the difference $f(\mathbf{x}^*) - f^*$ between the value of the (unknown) global optimum $f^* = \min_{\mathbf{x} \in X} f(\mathbf{x})$ and the value of the best solution $\mathbf{x}^*$ returned by SOO, is decreasing in $O(N^{-1/d})$, where d is the so-called near-optimality dimension of f, or even in an exponential rate $e^{-O(N)}$ for $d = 0$ (see [3] for details). Besides, the knowledge of d is *not* required. It is only used for the purpose of the theoretical complexity analysis. Such a provable complexity bound, under such a weak assumption on the blackbox function f, is a unique feature of SOO.

Let us now describe the working principles of SOO as summarized in the high-level pseudo-code of Algorithm 1. First, SOO is to be viewed as a divide-and-conquer algorithm that handles the search space as *a tree*. Each node of the tree corresponds to some region of the search space, which is called a cell. A *leaf* cell can be split (or expanded) into smaller cells covering the same original space. These new cells are then added to the tree as the children of the split node, hence expanding the tree with new leaves. The split node becomes an internal node of the tree, and is not processed anymore in subsequent iterations. Hence, the tree is organized in different

Algorithm 1: Pseudocode of SOO

1 $H_0 \leftarrow \{$ root cell representing the whole domain $\}$
2 value(root) $\leftarrow f$(center of root)
3 **while** budget not exhausted **do**
4 $\quad \mathcal{S} \leftarrow \varnothing$
5 $\quad v_{\min} \leftarrow +\infty$
6 $\quad$ **for** $i \in \{0, \ldots, \textit{tree height}\}$ **do**
7 $\quad\quad \ell^{\star} \leftarrow best(H_i)$
8 $\quad\quad$ **if** $value(\ell^{\star}) \leqslant v_{\min}$ **then**
9 $\quad\quad\quad v_{\min} \leftarrow value(\ell^{\star})$
10 $\quad\quad\quad \mathcal{S} \leftarrow \mathcal{S} \cup \{i\}$
11 $\quad$ **for** $i \in \mathcal{S}$ *in decreasing order* **do**
12 $\quad\quad \ell \leftarrow$ extract $best(H_i)$
13 $\quad\quad$ split ℓ into $\ell 1, \ell 2, \ell 3$
14 $\quad\quad value(\ell 1) \leftarrow f$(center of $\ell 1$)
15 $\quad\quad value(\ell 2) \leftarrow value(l)$
16 $\quad\quad value(\ell 3) \leftarrow f$(center of $\ell 3$)
17 $\quad\quad$ insert $\ell 1, \ell 2, \ell 3$ in heap H_{i+1}

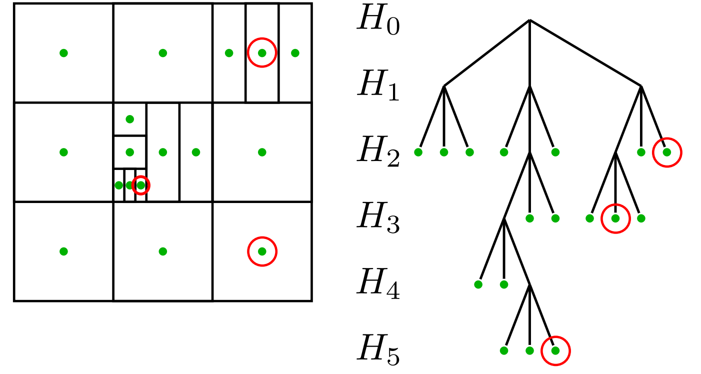

Fig. 1: Example of the partitioning into leaves of a search space (left) and the associated tree (right):

- leaf, position of its center in the space,
- chosen leaf in a given traversal.

levels; where each level may contain either internal nodes or leaf nodes. Given that the root of the tree is the whole search space, the leaves of the tree represent a partition of the space. An example of such a partition of the search space, and the associated tree, is shown in Figure 1.

In more details, SOO extends the underlying tree in successive iterations by probing the blackbox function f. The center of a new cell is evaluated using the f function, and this evaluation is stored within the cell. Implementation-wise, a cell is split into three equal-sized sub-cells (see leaves $\ell 1$, $\ell 2$ and $\ell 3$ at line 13 of Algorithm 1) by slicing along one dimension of the search space. This is actually a common choice in practice [5], [10], with the advantage of avoiding to re-evaluate the center of the sub-cell being in the middle of the original cell. The dimension a cell is cut along depends on its level in the tree. Given an arbitrary ordering of the space dimensions, SOO iterates cyclically over the dimensions to split along a new dimension at each new level. Having this in mind, a main design component underlying the computational flow of SOO is the choice of the leaf cells to split in each iteration. This is a critical aspect allowing to decide on how much effort should be devoted to the different regions of the search space, given an overall budget. Here, the budget is a given number of times the (blackbox) objective function f is called over all SOO execution. On the one hand, splitting all leaf cells in each iteration is obviously not an effective choice; since some regions might not contain any interesting solutions. On the other hand, focusing only on high-quality cells can be misleading, since some regions could hide promising solutions that were not yet discovered. Consequently, SOO adopts a specific strategy allowing a good balance between these two extreme situations. This is discussed in the following.

At each iteration, SOO traverses the tree starting from the root. For *each* level of the tree, it considers the leaf (if any) having the best center f-value. This best leaf is selected *if and only if* it has a better f-value than the leaves that have been chosen previously (i.e. in the levels being closer to the root). In other words, SOO selects the leaves with a better f-value than all leaves previously encountered in the traversal (which is encoded in Algorithm 1 using variable $v_{\min}$). Once the tree is traversed, each cell corresponding to a chosen leaf is split into new sub-cells (namely, three), and the tree is updated. A new iteration is then performed using the newly obtained tree, and so on. When the budget is exhausted, SOO returns the center of the cell with the best f-value.

From the previous description, it should be clear that SOO aims at providing a balance between exploration and exploitation. Choosing a large cell (relatively near the root) in the hope of finding interesting regions can be viewed as an exploration component; while choosing a cell with a good central point (deeper in the tree) can be viewed as an exploitation component. The fact that SOO does not select the cells having worst center values than other cells closer to the root, enforces the balance between these two components. In Figure 1, we show an illustrative example of SOO selection mechanism. At each level, at most one leaf is chosen: the one with the best value among all leaves at this level, which also has better f-values than all the leaves in previous levels. Hence, the chosen leaves, in a decreasing order of f-values, are the ones selected at levels 2, 3 and 5, respectively. Notice that no leaf is selected at level 4, which means that none of the two leaves there-in had a better f-value than the best leaf in the previous two levels.

Since the leaves are the only cells required by SOO, we do not keep track of internal nodes in our tree implementation, as shown in Algorithm 1. Moreover, the only required information needed about the position of cells is their levels in the tree. The tree is hence represented as a list of levels. Each level i is encoded as a heap of leaves (H_i), since this allows us to retrieve the best leaf in constant time, and to remove or to insert leaves in logarithmic time (in the level size). The function *best* (line 7 of Algorithm 1) returns the leaf with best value from a heap. The set $\mathcal{S}$ (lines 4 and 10), represents

the levels (or the depths) of chosen leaves in each iteration. It is implemented as an array whose size is doubled whenever full. The selected levels are thus inserted in the order they are selected, namely, in increasing order. One then has to browse this array in reverse order, processing the chosen leaves in the order of their decreasing depth, in order not to insert in a heap before extracting from it. Indeed, in the case an inserted leaf had a better value than the one to be extracted, the former would otherwise take the first place of the latter in the heap.

Although being relatively simple, it is not clear how to parallelize the SOO algorithm efficiently. In fact, SOO exposes two important dependencies. First, the tree traversal has to be performed in an inherently sequential manner starting from the root. Second, the tree is updated at each iteration, leading to a strong dependency between iterations as well. In the remaining of this paper, we provide a sound treatment to these dependencies leading to an efficient parallelization of SOO.

III. The parallel strategies

In this section, we describe the proposed parallel strategies for SOO, focusing first on the function evaluation step.

A. Parallel evaluations

1) A first naive strategy: Let us call *division* the process of splitting a leaf followed by the evaluation of the children centers using the function f. The selected leaves (variable $\mathcal{S}$ in Algorithm 1) are divided after the traversal of the tree (in lines 13 to 16). Notice that there is no dependency between the different evaluations, since this only requires the knowledge of a cell center. There is no dependency neither between the splitting of leaves, since we only need to know the center of a parent and its level, to determine the corresponding children. Hence, all divisions can be processed independently in parallel. Consequently, a simple way to start deploying SOO on multiple threads is to execute a parallel loop allowing to divide and to evaluate the selected leaves. After a tree traversal, the divisions are then spread among K threads, each one dealing with one slice of $\mathcal{S}$ according to a 1D bloc distribution with a static load balancing (since each evaluation of f is assumed to be deterministic and requires the same computation load). In more details, each i index in a slice of $\mathcal{S}$ is used to retrieve the best leave of the ith level and to divide it. In order to avoid any conflict on the heap structures (see Figure 2), the extraction of the leaves (line 12) and the insertion of the new leaves (line 17) are performed after the parallel evaluations, in one extra sequential loop. For our multi-thread implementation, we use OpenMP [19]. Since divisions and removals/insertions are separated in two distinct loops, we just need to add one OpenMP directive which makes the parallel implementation straightforward. We call this first version “Parallel Evaluations” (PE).

Notice that we also considered to use OpenMP tasks to improve this first multi-threaded version. Having each division being performed in an independent task, we could start executing tasks in parallel during the tree traversal, and not after as in the PE multi-threaded version. However, such an approach did not lead to any performance gain, since the computation grain of one call to the f evaluation can be too fine to offset the task creation and management costs, and since the tree traversal cost is much lower than the cost of all evaluations.

2) Parallel memory allocations: Each new leaf requires to handle a memory space, which is accessed for storing the center coordinates and the f-value. In the sequential SOO implementation, the numerous calls to the dynamic memory allocator are efficiently processed. However, synchronizations are required within the system memory allocator to support concurrent memory allocation requests issued by multiple threads; which can have a negative impact. We thus manage to improve the PE version by rewriting the memory allocations.

The new obtained version, called $PE\text{-}alloc$, allocates memory *by blocks* for the new leaves. Each thread keeps track of its own blocks in a linked list. While its current block is not full, a thread can use the space within this block for its next leaf. When the current block is full, the thread allocates a new one. The size of each new block doubles in order to obtain a logarithmic complexity. All the variants presented in the remainder will include this memory allocation scheme.

3) Parallel heap operations: Each divided cell must ultimately be extracted from its level, and its children inserted in the next one. Extraction and insertion in levels are heap operations. In the previous versions, PE and $PE\text{-}alloc$, heap operations are performed sequentially. This could lead to a performance bottleneck due to Amdahl’s law. Indeed, despite a logarithmic complexity, heap operations can have a non-negligible cost for levels with numerous leaves. We thus consider their parallel executions, along with the parallel function evaluations, in a new version called $PE\text{-}heap$.

Extracting and inserting leaves concurrently in parallel may however lead to inconsistencies in the heap structures if not handled properly. In fact, there is a risk of invalidating a heap if two threads modify it at the same time. Besides, we can remark that the order of heap operations matters. Let us consider two consecutive tree levels represented by heaps H_i and H_{i+1} both containing a leaf chosen during the traversal. The leaves selected in H_i and H_{i+1} are thus respectively $best(H_i)$ and $best(H_{i+1})$. The required heap operations on H_{i+1} are the extraction of $best(H_{i+1})$ and the insertion of the new sub-cells created with respect to $best(H_i)$. Hence, the new sub-cells should not be added in H_{i+1} before $best(H_{i+1})$ is removed; since otherwise, the content of H_{i+1} would change before extracting the originally selected leaf. As such, when one needs to extract and insert some leaves at a same level/heap, the extraction has to be completed first. This is illustrated in Figure 2.

To manage these issues in our $PE\text{-}heap$ version, we proceed as follows. Using K threads, the array $\mathcal{S}$ (containing the indexes of levels whose leaves must be split) is divided into K slices, as in the PE and $PE\text{-}alloc$ versions. Each thread $\mathcal{T}_k$, assigned one slice $\mathcal{S}^k$, has to: divide the selected leaves $best(S^k_n)_{n\in\mathbb{N}}$, remove them from the corresponding heaps, and insert their newly created children in their respective heaps. Hence, a thread browses its slice in increasing index order.

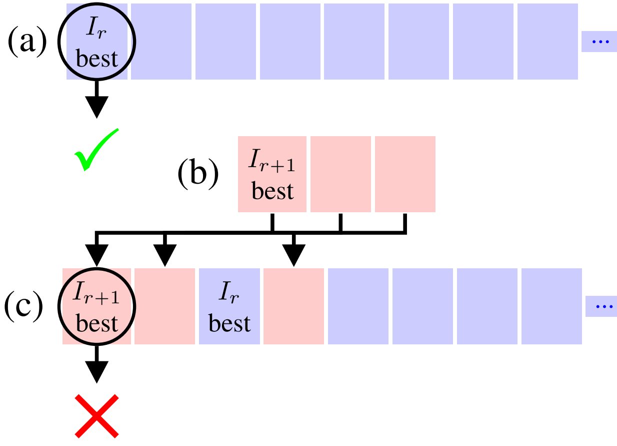

Fig. 2: Illustration of the side-effect of inserting new leaves in a heap at a given level before extracting the best leaf at the same level. (a): the best leaf of the heap (in iteration I_r) has to be extracted. (b): new leaves (from the previous level), produced in iteration I_r, have to be inserted before moving to iteration I_{r+1}. (c): inserting the new leaves before extracting the best leaf of iteration I_r would lead to a different (wrong) best leaf being extracted.

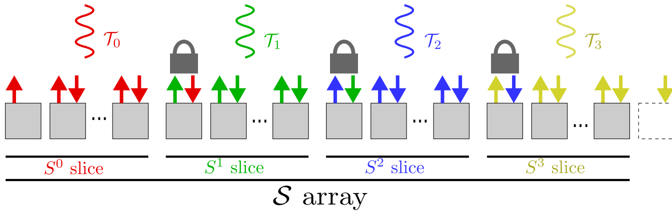

Fig. 3: *PE-heap* strategy using 4 threads, with a leaf divided at each level, and denoting: ↑: extraction; ↓: insertion; 🔒: lock; ⬚: new empty level.

Two situations have to be managed to ensure that extractions are performed before insertions. When the slice $\mathcal{S}^k$ contains both S_i and S_{i+1}, the same thread $\mathcal{T}_k$ dividing $best(H_{S_i})$ will have to divide $best(H_{S_{i+1}})$ as well. If $S_i + 1 = S_{i+1}$, then $\mathcal{T}_k$ simply extracts $best(H_{S_i+1})$ before inserting the children of $best(H_{S_i})$ in H_{S_i+1}. In the other case where S_{i+1} belongs to $\mathcal{S}^{k+1}$, insertion in $H_{S_{i+1}}$ must not occur before $\mathcal{T}_{k+1}$ has extracted $best(H_{S_{i+1}})$. For this purpose, we rely on a lock on $H_{S_{i+1}}$. Since this happens between two adjacent slices, we use one lock on the first heap of each slice, i.e. on heaps $H_{S_0^k}$ for $0 < k < K$, excluding $H_{S_0^0}$ since there is no leaf to divide above the corresponding level. This is illustrated in Figure 3.

At the beginning of each parallel region (i.e. after each traversal), the lock on level $H_{S_0^k}$ is acquired by the thread $\mathcal{T}_k$. $\mathcal{T}_k$ starts by extracting $best(H_{S_0^k})$ and release the lock right after. When the thread $\mathcal{T}_{k-1}$ has to insert the sub-cells of its last division in $H_{S_0^k}$, it will first acquire the lock on level $H_{S_0^k}$. Hence, we ensure that $\mathcal{T}_{k-1}$ will insert in $H_{S_0^k}$ only after $best(H_{S_0^k})$ has been extracted. Moreover, it is very unlikely that threads will waste time waiting on locks. Indeed, the threads originally holding a lock will release it at the start of the parallel region, while those who have to acquire it for insertions, will only do so after completing the divisions of their slice. In practice, we implement locks with lightweight OpenMP atomic operations on specific flags, using active waiting when the lock is already acquired by another thread. This active waiting is relevant because, as described above, there will probably be no waiting.

To summarize, after a traversal is performed, the *PE-heap* version processes in parallel both the leaf divisions and the heap operations. This *PE-heap* version is thus more complex to set up compared to the *PE* one, but it manages all operations in parallel apart from the tree traversal.

B. Parallel traversals

1) Motivation and Rational: All previous strategies are designed to handle an obvious level of parallelism in SOO, namely, processing the evaluations in parallel. Consequently, the underlying parallelism is limited by the number of evaluations *at each* traversal, which may be about several hundred in practice. Considering modern high-end CPUs with several dozens of cores, this may be too few to balance the computation load among all threads. Moreover, the tree traversal is performed in sequential in the *PE*-based strategies. Although the traversal cost is much lower than the cost of function evaluations, when considering a large number of CPU cores, this sequential tree traversal may limit the best achievable parallel speedup due to Amdahl's law. Therefore, we propose to leverage another, less obvious, parallelism level which includes the tree traversals.

A single tree traversal cannot be executed in parallel since one needs to update the $v_{\min}$ value from the H_i heap at level i, before continuing the traversal with heap H_{i+1} at level $i+1$. Besides, each iteration updates the tree. This means that the traversal at iteration I_r cannot be *completed* before iteration I_{r-1} is over. Nevertheless, the traversal at iteration I_r starts from the root of the tree downward. To process a level i in iteration I_r, one *only* needs the knowledge of the values in the upper heaps $H_x, 0 \leqslant x \leqslant i$, as updated by the previous iteration I_{r-1}. Thus, provided that the values in the heaps $H_x, 0 \leqslant x \leqslant i$, are up to date, a thread could start a new traversal to process the tree from the root down to level H_i independently of the other threads. Such a thread would not be able to progress after a level H_i until the tree is updated for the remaining levels $j > i$. Hence, we would like these updates to be performed as soon as possible (by an other thread).

In the sequential SOO algorithm, the tree levels are updated at each iteration by first performing a full traversal, and only then dividing the selected leaves and updating the heaps. In order to update the tree levels in as soon as possible, we handle a traversal in a slightly different manner. We determine as usual whether the leaf $best(H_j)$ at level H_j should be removed and divided. But then, we update H_j *immediately*, by extracting a selected leaf and/or by dividing the leaf from level $j-1$ and inserting its sub-cells in H_j, and *without* waiting for the current traversal to be fully completed. In other words,

Algorithm 2: PT strategy pseudocode with K threads

```
1  H_0 ← { root cell representing the whole domain }
2  value(root) ← f(center of root)
3  Parallel region for each thread T_k, 0 ⩽ k ⩽ K − 1
4      r ← k
5      while budget not exhausted do
6          v_min ← +∞
7          ℓ_next ← no leaf
8          for i ∈ {0, . . . , tree height} at iteration r do
9              wait (if required) until T_{k−1 mod K} updates H_i
10             ℓ_current ← ℓ_next
11             ℓ* ← best(H_i)
12             if value(ℓ*) ⩽ v_min then
13                 v_min ← value(ℓ*)
14                 ℓ_next ← ℓ*
15                 extract best(H_i)
16             else
17                 ℓ_next ← no leaf
18             if a leaf is stored in ℓ_current then
19                 split ℓ_current into ℓ1, ℓ2, ℓ3
20                 value(ℓ1) ← f(center of ℓ1)
21                 value(ℓ2) ← value(ℓ_current)
22                 value(ℓ3) ← f(center of ℓ3)
23                 insert ℓ1, ℓ2, ℓ3 in H_i
24         r ← r + K
```

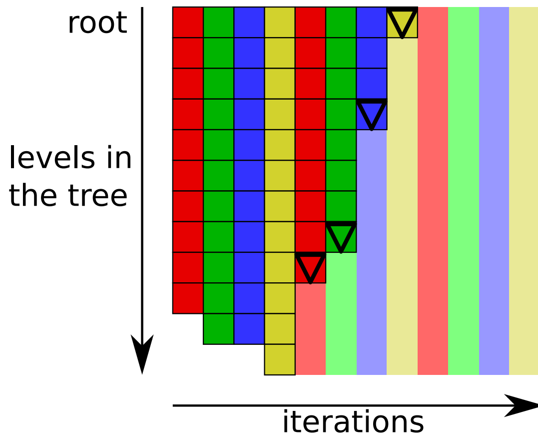

Fig. 4: PT strategy using 4 threads, denoting ▼: a thread; ■: a heap updated to the corresponding iteration; ■: a heap yet to be processed.

we embed the division of leaves (along with their side effects on the heaps) within a traversal, so that divisions are processed as soon as they are determined. Note that such an *embedded* traversal does *not* change the operations performed by SOO. It simply considers them in a different, but consistent, order. This shall then allow us to perform not just a single traversal, but *multiple* traversals, consistently in parallel.

2) The PT *version:* Based on the previous considerations, we propose a new parallel strategy, called "Parallel Traversal" (PT), which enables multiple parallel traversals of the tree. Its pseudocode is provided in Algorithm 2. The main idea is to distribute all SOO iterations among threads using a round-robin load balancing. Using K threads, the thread $\mathcal{T}_k$ will process iterations $I_{k+n\times K}$ for $n \in \mathbb{N}$; each one corresponding to an embedded traversal. The thread $\mathcal{T}_k$ processing an assigned iteration I_r updates the heap H_i while maintaining a *local variable* $v_{\min}$. This variable is only used locally by the thread for the current iteration it is processing. To ensure heap consistency, before processing H_i, the thread $\mathcal{T}_k$ has to check that the (previous) iteration I_{r-1} has been processed up to H_i (by a previous thread), so as to ensure that H_i is ready for the current iteration. If this is not the case, $\mathcal{T}_k$ simply waits until H_i is ready. When $\mathcal{T}_k$ terminates processing I_r, it starts processing the next unprocessed iteration, which is I_{r+K}, if enough budget is available. An example of the distribution of iterations among the threads, and of their parallel processing according to the PT strategy is shown in Fig. 4.

Contrary to the PE-based strategies where a new OpenMP parallel region is created at each iteration, the PT strategy uses only one OpenMP parallel region. Within this parallel region, only two thread synchronizations are required: (i) a first one to ensure for each thread $\mathcal{T}_k$ that its "previous" thread $\mathcal{T}_{(k-1) \bmod K}$ has already processed the current heap, and (ii) a second one to ensure that the budget will not be exceeded in parallel when continuing a traversal. Apart from these two synchronizations, the threads are always busy.

a) Heap synchronization: Regarding the OpenMP implementation of the first synchronization, we rely on atomic operations to ensure that thread $\mathcal{T}_k$ does not process a given heap H_i before its previous thread ($\mathcal{T}_{k-1 \bmod K}$) has updated it. Each thread maintains the tree level index and the iteration index it is currently processing. Atomic writes are used by each thread to update these indexes when moving to the next level or to the next iteration. Atomic reads enable each thread to safely access the indexes of its previous thread. The *seq_cst* memory-order clause is used with all these atomic operations for memory coherency. When changing an iteration index, which requires changing the level index too because the thread is going back at the root, the level index is always modified before the iteration index. When reading, the iteration index is always accessed before the level index. Such ordering of read/write operations is mandatory to avoid that a thread overestimate the progress made by another one. Threads that cannot progress more in an iteration simply wait actively. To avoid using atomic read operations at each level, each thread retrieves the level and iterations indexes (i, r) of its previous thread, and continues (without any atomic read operation) until it reaches level i. Once this level is reached, atomic reads will again be used to check the new level and iteration indexes of the previous thread. To avoid the false sharing of cache lines among the coherent caches of the CPU cores [20], the level and iteration indexes of distinct threads are not stored contiguously in memory, but rather on distinct cache lines.

b) Budget synchronization: Regarding the second required synchronization, it guarantees that the different threads will consume the exact same budget as in sequential SOO, and that the exact same tree will be produced. Hence, each thread has to know if there is enough budget left for a full traversal,

or if it should process a partial traversal. We thus make the first thread $\mathcal{T}_0$ estimate, for each group of I_r iterations (with $nK \leqslant r < (n+1)K, n \in \mathbb{N}$), whether there is enough budget left to completely process K iterations in the most budget-consuming scenario, i.e. when at each iteration one leaf is selected in each heap. Such an estimation can be obtained by a routine computation using the current tree height. If the budget is sufficient, then each of the K threads performs a full new iteration I_r. Otherwise, $\mathcal{T}_k$ starts iteration I_r by retrieving the number i_{cmpl} of completed iterations and the number of evaluations consumed by completed iterations. Then, $\mathcal{T}_k$ computes the maximum number of evaluations that could still be required by all threads processing a previous iteration $I_{i_{cmpl}<s<r}$. This allows $\mathcal{T}_k$ to determine the number of leaves that it can divide without exceeding the budget. If $\mathcal{T}_k$ cannot ensure that its next divisions would also have been performed by the sequential SOO algorithm (i.e. without exceeding the budget), $\mathcal{T}_k$ waits for its previous threads to progress. If the budget is fully exhausted, then all the threads terminate. This synchronization scheme requires atomic operations on several variables: the height of the tree, the number of completed iterations, the number of already performed evaluations, and a flag indicating whether $\mathcal{T}_0$ determined that the iterations of the current group of iterations can be fully processed.

3) Traversal swapping (PT-$swap$): In the PT strategy, all the basic steps of SOO are executed in parallel (i.e., the evaluations, heap operations, and the traversals). However, a possible limiting factor lies in the required synchronizations. The possible underlying waiting time for two consecutive threads can only occurs when a thread $\mathcal{T}_k$ starts processing the heap level H_i (at iteration I_{r+1}) and its previous thread ($\mathcal{T}_{k-1 \bmod K}$) has not finished processing H_i (at iteration I_r). Besides the fact the threads do not run synchronously on the MIMD architecture of multicore processors, this could typically arise because the number of leaf divisions varies from one iteration to another, and/or the cost of heap operations depends on the heap content. To minimize waiting times, we notice that the progress of a traversal in Algorithm 2, at a given iteration I_r for levels $j > i$, depends only on the computation of $v_{\min}$ at level i, and on the value of the leaf $best(H_i)$ (if this leaf is chosen at level i). The complete update of heap H_i at level i (i.e. the leaf extraction, the two evaluations and the new leaf insertions) is only required for the next traversal at iteration I_{r+1}. There is thus no need to wait for the complete update of heap H_i to start processing the other heaps $H_{j>i}$.

Consequently, we propose a new version, PT-$swap$, where threads can swap their traversals as illustrated on Figure 5. PT-$swap$ extends the PT strategy as follows. Whenever a thread $\mathcal{T}_q$ (in red in Figure 5) starts some "heavy" computations (extracting a leaf, or dividing a leaf and inserting the sub-cells) at a level i for the I_r traversal, $\mathcal{T}_q$ first computes and makes available $v_{\min}$, and possibly a chosen leaf from H_i, to a "next" thread (i.e. a thread closer to the root that may be blocked by $\mathcal{T}_q$). If a thread $\mathcal{T}_k$ (in green in Figure 5) arrives at level i before $\mathcal{T}_q$ has finished processing H_i, thread $\mathcal{T}_k$ does *not* wait anymore. Instead, it steals the traversal I_r from $\mathcal{T}_q$

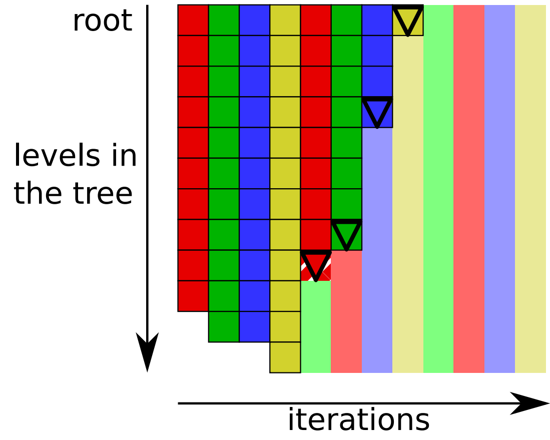

Fig. 5: PT-$swap$ strategy using 4 threads, denoting ▼: a thread; ■: a heap updated to the corresponding iteration; ▨: heap currently processed by a blocking thread; ■: a heap yet to be processed.

and processes it. When $\mathcal{T}_q$ terminates processing heap H_i, $\mathcal{T}_q$ proceeds as usual with its I_r traversal for levels $j > i$, if no other thread had stolen its current traversal. Otherwise, $\mathcal{T}_q$ proceeds with the traversal I_{r+1}, originally held by $\mathcal{T}_k$, and continues it for the levels $j \geqslant i$. We allow such a swapping only once for a given thread $\mathcal{T}_q$ at a given heap H_i for a given iteration I_r: If a third thread arrives at level i after $\mathcal{T}_k$ has stolen the traversal of $\mathcal{T}_q$, it waits until $\mathcal{T}_q$ updates H_i for I_r.

Setting up the swap mechanism needs some locks, which we implement with OpenMP atomic operations. Each thread $\mathcal{T}_q$ has its own lock which it releases to indicate that it starts the heavier computations for the I_r iteration, and that I_r can be swapped. A thread $\mathcal{T}_k$, processing an iteration I_{r+1}, and waiting for $\mathcal{T}_q$, can then acquire the lock, exchange variables related to the two traversals, and release the lock back to let $\mathcal{T}_q$ proceed with I_{r+1}. Moreover, swapping of two traversals implies to carefully exchange a number of variables between the two underlying threads. In addition to $v_{\min}$ and the chosen leaf, all variables required to perform a traversal must be swapped, as well as the variables used for the budget. The thread ordering is also impacted by a swap. More precisely, we use a specific data structure to indicate the "previous" thread $\mathcal{T}_q$ of each thread $\mathcal{T}_k$ (i.e. $\mathcal{T}_q$ "precedes" $\mathcal{T}_k$ and $\mathcal{T}_k$ must possibly wait for $\mathcal{T}_q$). This data structure also provides access to the current level index of $\mathcal{T}_q$, as well as to the iteration index, and to the swap lock among other variables. When $\mathcal{T}_k$ and $\mathcal{T}_q$ swap, their ordering is also updated with respect to the tree and to the other threads in this data structure. In Figure 5, if $\mathcal{T}_k$ (depicted in green) precedes $\mathcal{T}_c$ (depicted in blue) before the swap, then, after a swap with $\mathcal{T}_q$, $\mathcal{T}_k$ precedes $\mathcal{T}_q$ (in red) and $\mathcal{T}_q$ precedes $\mathcal{T}_c$. If a swapped thread had the specific role of $\mathcal{T}_0$ regarding the budget consumption (see Section III-B1), this responsibility must be swapped as well.

To summarize, PT-$swap$ is intended to reduce the waiting times among consecutive threads, and comes with a careful implementation of thread synchronization to enable swapping.

TABLE I: Parallel features within each SOO parallel version

Feature	*PE*	*PE-alloc*	*PE-heap*	*PT*	*PT-swap*
Evaluations in parallel	**yes**	**yes**	**yes**	**yes**	**yes**
Allocation by blocks	no	**yes**	**yes**	**yes**	**yes**
Parallel heap operations	no	no	**yes**	**yes**	**yes**
Traversals in parallel	no	no	no	**yes**	**yes**
Swapping traversals	no	no	no	no	**yes**

IV. Experimental results

In this section, we discuss the performance of the different proposed parallel SOO variants. Our experiments are conducted on a high-end 128-core compute node using a number of benchmark functions as detailed in the following.

A. Experimental setup

All algorithms are written in C, compiled with version 10.2.1 of GCC. We used OpenMP's parallel loops and atomic operations for our multi-thread implementations. The five competing parallel SOO variants are: PE, PE-$alloc$, PE-$heap$, PT, and PT-$swap$. Notice that each variant builds upon the previous one by adding a new parallel feature, as summarized in Table I, so as to allow us to fairly evaluate the designed variants in an incremental manner.

For benchmarking purposes, we use the COmparing Continuous Optimizers (COCO) [21] implementation of the Black-Box Optimization Benchmarking (BBOB)[1] functions [18]. BBOB is a state-of-the-art test suite, used in particular in a reference workshop held yearly in the well-established Genetic and Evolutionary Computation Conference (GECCO). More than 200 algorithms were already benchmarked within this framework. BBOB provides a set of 24 continuous functions exposing different properties, and believed to represent a broad range of optimization problems that one may encounter in practice. These functions are organized into five groups of increasing difficulty, and are available in multiple dimensions.

Regarding the hardware, we used Grid'5000, a HPC platform (https://www.grid5000.fr) providing access to various clusters of compute nodes. We ran our experiments on one node of the Sirius cluster, based in Lyon. This node has 128 physical CPU cores spread among two 64-core AMD EPYC 7742 CPUs. The cores are grouped into 8 Non-Uniform Memory Access (NUMA) domains of 16 cores each. We depict this architecture in Figure 6, using the `lstopo` command from the Hardware Locality (`hwloc`) package [22]. Each physical CPU core offers 2-way SMT (Simultaneous multi-threading). For the thread placement, we used the OpenMP options `OMP_PLACES=cores` and `OMP_PROC_BIND=close` at execution. These options prevent the threads from moving from one core to another, and more importantly from one NUMA domain to another. They also favor data locality within the memory hierarchy between threads with consecutive numbers, which can be beneficial for the PE-$heap$, PT and PT-$swap$ strategies. The parallel speedups presented in the following are all based on the execution times of the reference sequential algorithm presented in Section II. The turbo mode,

[1] See also: https://numbbo.github.io/data-archive/bbob/

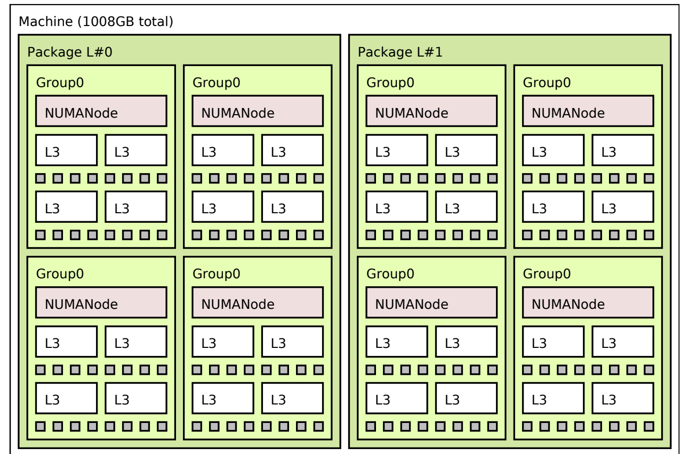

Fig. 6: Architecture of the Sirius node of Grid'5000 composed of two 64-core AMD EPYC 7742 CPUs. Each ■ denotes one of the 128 physical CPU cores. L3 caches sizes are 16MB, NUMA nodes ones are 126GB.

which favors executions with few cores, is deactivated in order to have non-biased parallel speedups.

A note on SIMD computing: We rely on the compiler to (possibly) vectorize the COCO code of each f function, and benefit from the AVX2 SIMD units of the AMD EPYC 7742 CPUs. We also considered the vectorization of multiple calls to the same f function (using one call per SIMD lane): this failed due to several features (assertions, memory allocations, use of multiple function pointers) within the COCO code.

B. Results and discussion

1) Speedups: We start our analysis by reporting in Table II the speedups obtained when running the different parallel variants using 128 threads (on 128 physical cores) and a budget of 40×10^7 calls to the function evaluation, for each of the 24 BBOB functions in dimension 40.

Overall, one can clearly see that the mean speedup improves significantly over the different designed versions. More precisely, the PE version, which focuses on merely parallelizing the function evaluations, leads to a very low average speedup of 3.6. By avoiding multiple individual memory allocations in parallel, the PE-$alloc$ version is able to increase the average speedup to 10.1. The PE-$heap$ version extends PE-$alloc$ by processing the heap operations in parallel; which increases the average speedup to 22.6. This is however still not efficient for 128 cores, and shows that the PE-based strategies fail to offer significant speedups on a large number of cores.

On the contrary, the gain in parallel speedup is much more important when moving to the PT version, which allows us to obtain an average speedup of 84.1. In fact, this version handles both the evaluations and the iterations performed by the SOO algorithm entirely in parallel. In particular, processing iterations entirely on their dedicated threads removes the cost of performing the underlying tree traversals sequentially, as well as the need of starting and stopping the threads and

TABLE II: Speedups for the parallel versions with 128 threads on functions of dimension 40 with 40×10^7 evaluations (the best value of each row is in bold type)

function	*PE*	*PE-alloc*	*PE-heap*	*PT*	*PT-swap*
1	0.4	1.8	4.8	**33.8**	26.5
2	2.3	10.8	25.3	97.2	**98.1**
3	3.0	12.9	31.0	104.0	**104.2**
4	2.3	10.5	26.1	91.8	**97.7**
5	0.4	3.3	9.0	**50.9**	46.1
6	1.9	6.6	17.1	**78.2**	73.8
7	3.7	9.2	14.8	84.4	**89.5**
8	0.6	2.2	6.4	**40.0**	34.9
9	1.8	5.3	13.2	**70.9**	69.4
10	5.9	13.2	27.9	98.5	**102.5**
11	4.1	12.1	26.5	92.9	**98.8**
12	2.7	10.1	26.6	97.4	**99.1**
13	2.0	5.5	14.3	**71.7**	71.1
14	3.0	7.9	19.6	**87.7**	86.8
15	6.5	16.7	36.8	106.4	**111.1**
16	7.7	17.4	35.7	103.1	**110.0**
17	5.2	12.1	26.4	95.7	**100.1**
18	5.1	12.0	24.8	92.2	**99.0**
19	2.3	5.9	14.4	**74.9**	72.8
20	2.5	8.5	21.4	89.7	**91.8**
21	4.9	16.4	36.0	91.2	**110.9**
22	1.2	6.0	16.4	75.9	**77.3**
23	13.5	27.0	47.0	106.2	**117.1**
24	2.8	8.5	19.9	83.0	**87.8**
mean	3.6	10.1	22.6	84.1	**86.5**
median	2.7	9.6	23.1	90.5	**94.8**
max	13.5	27.0	47.0	106.4	**117.1**

distributing the required divisions among them. Looking at the PT-$swap$ version, we found that it can improve over the PT version for a fairly large number of experimented functions, though with some exceptions. In fact, this version can reduce the cost of thread synchronization by enabling two consecutive threads to swap their traversals. This can offer performance gains up to 22% with respect to the PT version, and parallel speedups up to 117.1 for 128 threads, e.g., see function 23 in Table II. We also notice that PT-$swap$ systematically leads to a performance gain for all functions where the PT speedup is at least 90. For some functions with an already low PT speedup, such as 1, 5, and 8, PT-$swap$ can however lead to lower performance results. This is clearly because swapping thread traversals is not for free. However, the median speedup is still in favor of PT-$swap$, which indicates that this version is of special interest.

From Table II, we can also see that for each version, the obtained speedups can vary significantly depending on the experimented function. We found that this is tightly related to the computation grain size, that is, to the time required to call one f function. In Figure 7, we show a scatter plot rendering the correlation between the parallel speedup and the sequential execution time for the 24 test functions using various dimensions D, ranging from very low, to relatively large. We clearly see that the higher the sequential execution time, the higher its computation grain size, and the better the parallel speedup. This can also explain the discrepancies of the effect of PT-$swap$ on the speedups. In fact, for functions with relatively large computation grains, the evaluation lasts long enough for the swap to be shorter than waiting for a thread to complete the evaluation. Hence, for such functions, PT-$swap$ performs better and provides a non-negligible gain compared to the PT version.

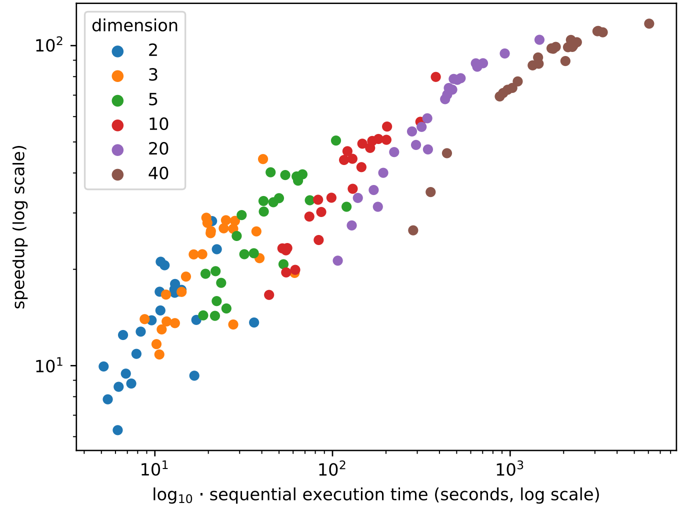

Fig. 7: PT-$swap$ speedups with 128 threads versus sequential execution times for the 24 test functions, using various dimensions $D \in \{2, 3, 5, 10, 20, 40\}$, and a budget of $D \times 10^7$ function evaluations.

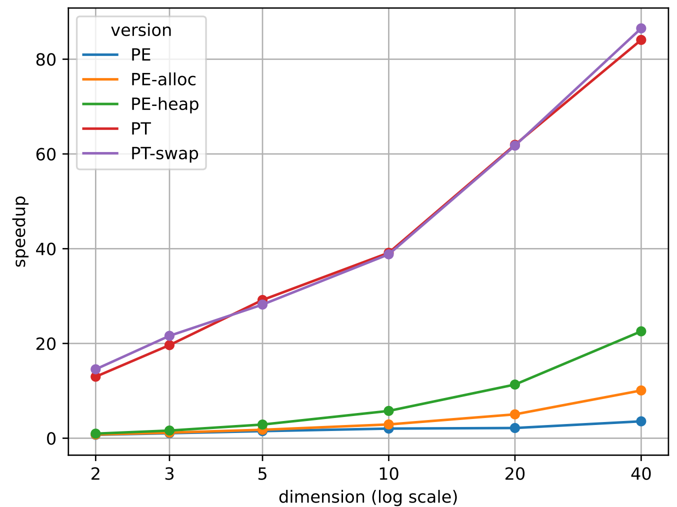

Fig. 8: Averaged speedups versus dimension with 128 threads over functions, with $dimension \times 10^7$ evaluations.

To go deeper in the impact of the computation grain on performance, we show in Figure 8 the obtained speedups as a function of problem dimension D for the different versions. We can clearly see that the speedups increase with the dimension, independently of the considered parallel version. This is clearly to be attributed to the fact that the computation grain increases as well. Interestingly, the results in Figure 8 allows us to better appreciate the performance gap between the PE

versions and the PT ones, as a function of problem dimension. On the one hand, the PT versions are able to scale relatively well with the problem dimension, whereas the PE versions have more difficulties. On the other hand, the PT versions achieve significantly better speedups independently of problem dimension. In fact, the gain of the average speedup of PT-$swap$ compared to the average speedup of PE-$heap$ (the best PE version) reaches respectively $3.8\times$, $5.5\times$, $6.7\times$, $9.8\times$, $13.4\times$, and $15.1\times$ for dimensions 40, 20, 10, 5, 3, and 2.

2) Scalability: We conclude our analysis by studying the average speedups obtained for an increasing number of threads, including some settings where the SMT feature (i.e., simultaneous multithreading) is used. This is shown in Figures 9 and 10 respectively for dimensions 40 and 100.

When the SMT is *not* considered, i.e. we rely on 1 thread per physical CPU core, the versions based on parallel traversals (PT and PT-$swap$) scale clearly much better with the number of threads compared to the versions based on parallel evaluations (PE, PE-$alloc$ and PE-$heap$). Besides, the more threads, the larger the performance gap between the PT-based versions and the PE-based ones.

When considering SMT usage, i.e. when using 2 threads per physical CPU core, the speedup decreases systematically for the PE-based versions for both considered dimensions. The situation is seemingly different for the PT-based versions. In fact, the speedup decreases only for dimension 40 when using the two CPUs (256 threads on 128 cores). Actually, this performance drop for the PT-based versions is likely not due to the SMT feature itself, but rather to a too fine computation grain. This is first indicated by the SMT performance gain within one CPU (128 threads on 64 cores: see Figure 9) for dimension 40. This is then confirmed by the SMT performance gain on two CPUs for dimension 100 (see Figure 10) for the PT-based versions, the SMT feature still leading to a performance drop for the PE-based versions. Notice in particular the superiority of the PT-$swap$ version at the largest scales with SMT. For example, the average speedup of PT-$swap$ using 256 threads is 132.4 in dimension 100, which also shows the SMT benefit on 128 cores. The maximum speedup of PT-$swap$, obtained for function 12, even reaches 174.8. Therefore, we can conclude that the parallel design of the PT-based versions is extremely accurate to achieve parallel efficiency and scalability provided that the computation grain is large enough.

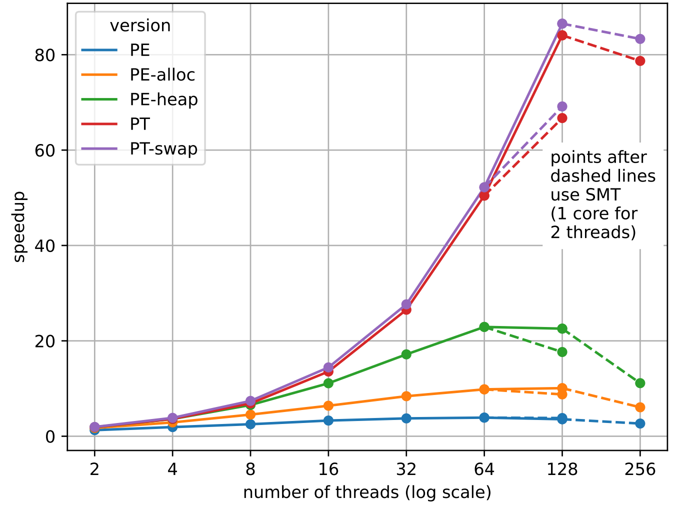

Fig. 9: Averaged speedups versus number of threads over functions of dimension 40, with 40×10^7 evaluations.

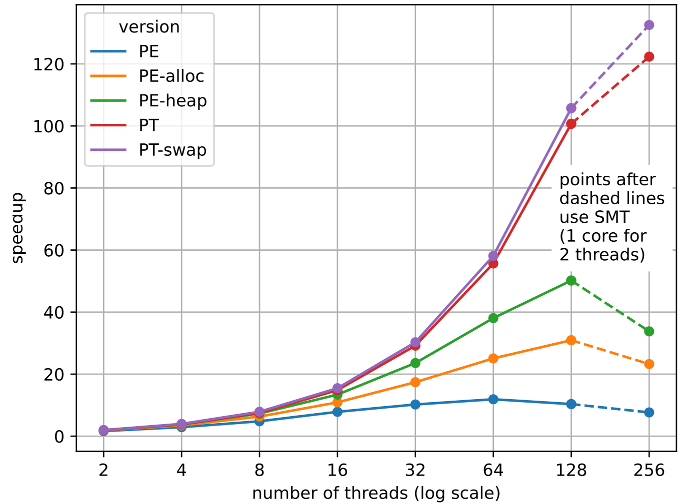

Fig. 10: Averaged speedups versus number of threads over functions of dimension 100, with 100×10^6 evaluations.

V. Conclusion

In this paper, we investigated different parallel versions[2] of the SOO (Simultaneous Optimistic Optimization) algorithm for blackbox optimization. We first considered a naive version focusing on the parallel processing of the independent function evaluations. Even with some improvements regarding parallel memory allocations and parallel heap operations, this version leads to low speedups on a shared-memory architecture with dozens of CPU cores, and fails to scale with the number of threads or with the dimension. We have then leveraged a second, less obvious, parallelism level to fully parallelize SOO based on its tree traversals, and allowed the threads to swap their traversals in parallel at the aim of increasing parallel efficiency. The so-obtained traversal-based strategies can offer (in average over the experimented functions) a $3.2\times$ performance gain over the evaluation-based strategies. The traversal-based strategies can also benefit from the SMT feature of the CPU cores, providing parallel speedups up to 174.8 on 128 cores. Besides, it is worth-noticing that all investigated strategies eventually perform the exact same operations as the sequential SOO algorithm. Hence, our parallel versions output exactly the same solution, and therefore provide the same theoretical quality guarantees as sequential SOO.

[2]Our source code is freely available on demand.

As research perspectives, this work opens a number of interesting questions. For example, we could compare our parallel implementations of SOO to other parallel global optimizers, such as a version of MCS [23] or GLOBAL [24] that distributes the search space across threads.

Furthermore, it would be interesting to leverage our techniques in a distributed-memory environment, where the challenge would lie in the efficient handling of the inter-node communications. Besides, instead of following the exact same search space exploration scheme than SOO, one may wonder whether relaxing such a requirement would not lead to improved parallel versions. One challenging idea would be to take inspiration from SOO in order to design alternative *fully asynchronous* strategies, eventually leading to a parallel SOO-like algorithm scaling on a large number of nodes with dozens of CPU cores each.

VI. Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

The authors thank the Hauts-de-France region for partly funding David Redon's PhD thesis.

References

[1] J. Larson, M. Menickelly, and S. M. Wild, "Derivative-free optimization methods," *Acta Numerica*, vol. 28, pp. 287–404, 2019.
[2] A. E. Eiben, J. E. Smith *et al.*, *Introduction to evolutionary computing*. Springer, 2003, vol. 53.
[3] R. Munos, "Optimistic optimization of a deterministic function without the knowledge of its smoothness," in *25th Annual Conference on Neural Information Processing Systems*, 2011, pp. 783–791.
[4] ——, "From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning," *Found. Trends Mach. Learn.*, vol. 7, no. 1, pp. 1–129, 2014.
[5] P. Preux, R. Munos, and M. Valko, "Bandits attack function optimization," in *IEEE CEC*, 2014, pp. 2245–2252.
[6] B. Derbel and P. Preux, "Simultaneous optimistic optimization on the noiseless BBOB testbed," in *IEEE CEC*, 2015, pp. 2010–2017.
[7] Z. Wang, B. Shakibi, L. Jin, and N. de Freitas, "Bayesian multi-scale optimistic optimization," in *Proceedings of the 17th Int. Conference on Artificial Intelligence and Statistics, AISTATS*, 2014, pp. 1005–1014.
[8] H. Qian and Y. Yu, "Scaling simultaneous optimistic optimization for high-dimensional non-convex functions with low effective dimensions," in *AAAI Conference on Artificial Intelligence*, 2016, pp. 2000–2006.
[9] L. Busoniu, A. Daniels, R. Munos, and R. Babuska, "Optimistic planning for continuous-action deterministic systems," in *IEEE Symp. on Adaptive Dynamic Programming and Reinforcement Learning*, 2013, pp. 69–76.
[10] M. Valko, A. Carpentier, and R. Munos, "Stochastic simultaneous optimistic optimization," in *ICML*, 2013, pp. 19–27.
[11] L. Busoniu and I. Morarescu, "Consensus for agents with general dynamics using optimistic optimization," in *IEEE Conference on Decision and Control, CDC*, 2013, pp. 6735–6740.
[12] B. Derbel, A. Liefooghe, S. Vérel, H. E. Aguirre, and K. Tanaka, "New features for continuous exploratory landscape analysis based on the SOO tree," in *ACM FOGA*, 2019, pp. 72–86.
[13] T. Vu and B. Derbel, "Parallel branch-and-bound in multi-core multi-CPU multi-GPU heterogeneous environments," *FGCS*, vol. 56, pp. 95–109, 2016.
[14] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.
[15] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *ACM KDD*, 2016, pp. 785–794.
[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "LightGBM: A highly efficient gradient boosting decision tree," in *NIPS*, 2017, pp. 3146–3154.
[17] "Top500 list, november 2021," 2021.
[18] N. Hansen, S. Finck, R. Ros, and A. Auger, "Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions," INRIA, Research Report RR-6829, 2009.
[19] OpenMP Architecture Review Board, "OpenMP Application Program Interface, V 5.0," 2018.
[20] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Sixth Edition*, 2017.
[21] N. Hansen, T. Tusar, O. Mersmann, A. Auger, and D. Brockhoff, "COCO: the experimental procedure," *CoRR*, vol. abs/1603.08776, 2016.
[22] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *Euromicro Conf. on Para., Dist. and Net.-based Proc.* IEEE, 2010, pp. 180–186.
[23] W. Huyer and A. Neumaier, "Global optimization by multilevel coordinate search," *J. Glob. Optim.*, vol. 14, no. 4, pp. 331–355, 1999.
[24] T. Csendes, "Nonlinear parameter estimation by global optimization - efficiency and reliability," *Acta Cybern.*, vol. 8, no. 4, pp. 361–372, 1988.